Under consideration for publication in J. Functional Programming

Verified Decision Procedures for MSO on Words Based on Derivatives of Regular Expressions

DMITRIY TRAYTEL
ETH Zürich, Switzerland
and
TOBIAS NIPKOW
Technische Universität München, Germany

Abstract

Monadic second-order logic on finite words (MSO) is a decidable yet expressive logic into which many decision problems can be encoded. Since MSO formulas correspond to regular languages, equivalence of MSO formulas can be reduced to the equivalence of some regular structures (e.g. automata). This paper presents a verified functional decision procedure for MSO formulas that is not based on automata but on regular expressions. Functional languages are ideally suited for this task: regular expressions are data types and functions on them are defined by pattern matching and recursion and are verified by structural induction.

Decision procedures for regular expression equivalence have been formalized before, usually based on Brzozowski derivatives. Yet, for a straightforward embedding of MSO formulas into regular expressions an extension of regular expressions with a projection operation is required. We prove total correctness and completeness of an equivalence checker for regular expressions extended in that way. We also define a language-preserving translation of formulas into regular expressions with respect to two different semantics of MSO. Our results have been formalized and verified in the theorem prover Isabelle. Using Isabelle's code generation facility, this yields purely functional, formally verified programs that decide equivalence of MSO formulas.

1 Introduction

Many decision procedures for logical theories are based on the famous logic-automaton connection. That is, they reduce the decision problem for some logical theory to a decidable question about some class of automata. Automata are usually implemented with the help of imperative data structures for efficiency reasons.

In functional languages, automata are not an ideal abstraction because they are graphs rather than trees. In contrast, regular expressions are perfect for functional languages and they are equally expressive. In fact, Brzozowski (1964) showed how automata-based algorithms can be recast as recursive algebraic manipulations of regular expressions. His *derivatives* can be seen as a way of simulating automaton states with regular expressions and computing the next-state function symbolically.

Recently Brzozowski's derivatives were discovered by functional programmers and theorem provers. Owens *et al.* (2009) realized that regular expressions and their derivatives fit perfectly with data types and recursive functions. Their paper explores regular expression

matching based directly on regular expressions rather than automata. Fischer *et al.* (2010) also explore regular expression matching, but by means of marked regular expressions rather than derivatives. Slightly later, the interactive theorem proving community woke up to the beauty of derivatives, too. This resulted in four papers about verified decision procedures for the equivalence of regular expressions based on derivatives and on marked regular expressions (see Related Work below). In one of these four papers, Coquand and Siles (2011) state that "A more ambitious project will be to use this work for writing a decision procedure for WS1S", a monadic second-order logic. Our paper does just that (and more).

Monadic second-order logic on finite words (MSO) is a decidable yet expressive logic into which many decision problems can be encoded (Thomas, 1997). MSO allows only monadic predicates but quantification both over numbers and finite sets of numbers. Two closely related but subtly different semantics can be found in the literature. One of the two, WS1S—the Weak monadic Second-order logic of 1 Successor, is based on arithmetic. The other, M2L(Str) (Henriksen *et al.*, 1995), is more closely related to formal languages. There seems to be some disagreement as to which semantics is the more appropriate one for verification purposes (Klarlund, 1999; Ayari & Basin, 2000). Hence we cover both.

Essentially, MSO formulas describe regular languages. Therefore MSO formulas can be decided by translating them into automata. This is the basis of the highly successful MONA tool (Elgaard *et al.*, 1998) for deciding WS1S. MONA's success is due to its (in practical terms) highly efficient implementation and to the ease with which very different verification problems can be encoded in monadic second-order logic, for example Presburger arithmetic and Hoare logic for pointer programs.

The contribution of this paper is the presentation of the first purely functional decision procedures for two interpretations of MSO based on derivatives of regular expressions. These decision procedures have been verified in Isabelle/HOL and we sketch their correctness proofs. We are not aware of any previous decision procedure for MSO based on regular expressions (as opposed to automata), let alone a verified program.

It is instructive to compare our decision procedure for WS1S with MONA. MONA is a highly tuned implementation using cache-conscious data structures including a BDD-based automaton representation. Ours is a (by comparison tiny) purely functional program that operates on regular expressions and can only cope with small examples. MONA is not verified (and the prospect of doing so is daunting), whereas our code is.

In this paper we distinguish ordinary regular expressions that contain only concatenation, union, and iteration from *extended* regular expressions that also provide complement and intersection. The rest of the paper is organized as follows. Section 2 gives an overview of related work. Section 3 introduces some basic notations. Sections 4 and 5 constitute the main contribution of our paper—the first shows how to decide equivalence of extended regular expressions with an additional projection operation, the second reduces equivalence of MSO formulas to equivalence of exactly those regular expressions with respect to both semantics, M2L and WS1S. In total this yields a decision procedure for MSO on words. A short case study of the decision procedure is given in Section 6.

This paper is an extended and revised version of the homonymous ICFP 2013 functional pearl (Traytel & Nipkow, 2013). The new contributions are mostly actual changes to the decision procedure aiming to improve on both, performance and presentation:

- Picking up an idea from Owens *et al.* (2009), we change the semantics of atomic regular expressions to represent sets of alphabet letters rather than single letters (Section 4.4).
- Based on earlier experimental results (Nipkow & Traytel, 2014) we change the back-end decision procedure for regular expression equivalence to use partial derivatives instead of Brzozowski derivatives (Section 4.5).
- We improve the translations of M2L(Str) formulas to regular expressions by removing redundancies and expand on the previously omitted implementation of the translation of WS1S formulas to regular expressions (Section 5.4).
- A reevaluation of the performance shows a sizable improvement, yet still far away from competing with MONA (Section 6).

While the paper is intended to be self contained with respect to the presented functional program deciding equivalence of MSO formulas, we deliberately give only rough intuitions instead of detailed proofs. The proofs are where they truly belong: in the publicly available formalization (Traytel & Nipkow, 2014).

2 Related Work

Brzozowski (1964) introduced the notion of derivatives of extended regular expressions and Ginzburg (1967) employed them in an algorithm for deciding language equivalence that we essentially are using here. Antimirov (1996) devised the related notion of partial derivatives of ordinary regular expressions. Caron *et al.* (2011) extended partial derivatives to extended regular expressions. The concept of derivatives as means to compute the next state symbolically goes beyond regular expressions—as witnessed by libraries for parsing developed by Danielsson (2010) in Agda and by Might *et al.* (2011) in Lisp using lazily evaluated variations of Brzozowski derivatives for parser combinators. Furthermore, Kozen (2008) lifted derivatives to expressions of Kleene algebra with tests.

MONA was linked to Isabelle by Basin and Friedrich (2000) and to PVS by Owre and Rueß (2000). In both cases, MONA is used as a trusted oracle for deciding formulas in the respective theorem prover.

Now we discuss work on verified decision procedures for regular expressions. The first verified equivalence checker for regular expressions was published by Braibant and Pous (2010). They worked with automata, not regular expressions, their theory was large and their algorithm efficient. In response, Krauss and Nipkow (2012) gave a much simpler partial correctness proof for an equivalence checker for regular expressions based on derivatives. Coquand and Siles (2011) showed total correctness of their equivalence checker for extended regular expressions based on derivatives. Asperti (2012) presented an equivalence checker for regular expressions via marked regular expressions (as previously used by Fischer *et al.*(2010)) and showed total correctness. Moreira *et al.* (2012) presented an equivalence checker for regular expressions based on partial derivatives and showed its total correctness. Recently, we have devised a general framework that unifies the different approaches based on derivatives, partial derivatives, and marked regular expressions under one roof (Nipkow & Traytel, 2014). Berghofer and Reiter (2009) formalized a decision procedure for Presburger arithmetic via automata in Isabelle/HOL.

Outside of the application area of equivalence checking, Wu *et al.* (2014) benefited from the inductive structure of regular expressions to formally verify the Myhill-Nerode theorem.

3 Preliminaries

Although we formalized everything in this paper in the theorem prover Isabelle/HOL (Nipkow *et al.*, 2002; Nipkow & Klein, 2014), no knowledge of theorem provers or Isabelle/HOL is required because we employ mostly ordinary mathematical notation in our presentation. Some specific notations are summarized below.

The symbol $\mathbb{B}$ represents the type of Booleans, where $\top$ and $\bot$ represent true and false. The type of sets and the type of lists over some type τ are written τ set and τ list. In general, type constructors follow their arguments. The letters α and β represent type variables. The notation $t :: \tau$ means that term t has type τ.

Many of our functions are curried. In some cases we write the first argument as an index: instead of $f\ a\ b$ we write $f_a(b)$ (in preference to just $f_a\ b$). The projection functions on pairs are called fst and snd. The image of a function f over a set S is written $f \bullet S$.

Lists are built up from the empty list [] via the infix # operator that prepends an element x to a list xs: $x \# xs$. Two lists are concatenated with the infix @ operator. Accessing the nth element of a list xs is denoted by $xs[n]$; the indexing is zero-based. The length of the list xs is written $|xs|$.

Finite words as in formal language theory are modeled as finite lists, i.e. type α list. The empty word is the empty list. As is customary, concatenation of two words u and v is denoted by their juxtaposition uv; similarly for a single letter a of the alphabet and a word w: aw. That is, the operators # and @ remain implicit (for words, not for arbitrary lists).

4 Extended Regular Expressions

In Section 5, MSO formulas are 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.

Decision procedures for equivalence of regular expression have been formalized earlier in theorem provers. Here, we extend the existing formalization and the soundness proof in Isabelle/HOL by Krauss and Nipkow (2012) with negation and intersection operation on regular expressions, as well as with a nonstandard projection operation. Additionally, we provide proofs of termination and completeness.

4.1 Syntax and Semantics

Regular expressions extended with intersection and complement allow us to encode Boolean operators on formulas in a straightforward fashion. A further operation—the *projection* Π—plays the crucial role of encoding existential quantifiers. These Π-*extended* regular expressions (to distinguish them from mere extended regular expressions) are defined as a recursive data type α RE, where α is the type of the underlying alphabet. In conventional

concrete syntax, $\alpha\ \mathsf{RE}$ is defined by the grammar

$$\begin{array}{rcccccc} r & = & \mathbf{0} & | & \mathbf{1} & | & a \\ & | & r+s & | & r\cdot s & | & r^* \\ & | & r\cap s & | & \neg\, r & | & \Pi\, r \end{array}$$

where $r, s :: \alpha\ \mathsf{RE}$ and $a :: \alpha$. 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 $\pi :: \Sigma_{n+1} \to \Sigma_n$[1] that translates between the different alphabets. In our application, n will represent the number of free variables of the translated MSO 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 as the recursive predicate $\mathsf{wf} :: \mathbb{N} \to \alpha\ \mathsf{RE} \to \mathbb{B}$.

$$\begin{array}{lll} \mathsf{wf}_n(\mathbf{0}) & = \top & \mathsf{wf}_n(\mathbf{1}) \quad = \top \\ \mathsf{wf}_n(a) & = a \in \Sigma_n & \mathsf{wf}_n(r+s) = \mathsf{wf}_n(r) \wedge \mathsf{wf}_n(s) \\ \mathsf{wf}_n(r\cdot s) & = \mathsf{wf}_n(r) \wedge \mathsf{wf}_n(s) & \mathsf{wf}_n(r^*) \quad = \mathsf{wf}_n(r) \\ \mathsf{wf}_n(r\cap s) & = \mathsf{wf}_n(r) \wedge \mathsf{wf}_n(s) & \mathsf{wf}_n(\neg\, r) \quad = \mathsf{wf}_n(r) \\ \mathsf{wf}_n(\Pi\, r) & = \mathsf{wf}_{n+1}(r) & \end{array}$$

We call a regular expression r *n-wellformed* if $\mathsf{wf}_n(r)$ holds.

The *language* $\mathscr{L} :: \mathbb{N} \to \alpha\ \mathsf{RE} \to (\alpha\ \mathsf{list})\ \mathsf{set}$ 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^*.

$$\begin{array}{lll} \mathscr{L}_n(\mathbf{0}) & = \{\} & \mathscr{L}_n(\mathbf{1}) \quad = \{[]\} \\ \mathscr{L}_n(a) & = \{a\} & \mathscr{L}_n(r+s) = \mathscr{L}_n(r) \cup \mathscr{L}_n(s) \\ \mathscr{L}_n(r\cdot s) & = \mathscr{L}_n(r) \cdot \mathscr{L}_n(s) & \mathscr{L}_n(r^*) \quad = \mathscr{L}_n(r)^* \\ \mathscr{L}_n(r\cap s) & = \mathscr{L}_n(r) \cap \mathscr{L}_n(s) & \mathscr{L}_n(\neg\, r) \quad = \Sigma_n^* \setminus \mathscr{L}_n(r) \\ \mathscr{L}_n(\Pi\, r) & = \mathsf{map}\,\pi \bullet \mathscr{L}_{n+1}(r) & \end{array}$$

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 $\mathscr{L}_n(\neg\, r) = \Sigma_n^* \setminus \mathscr{L}_n(r)$.

The definition $\mathscr{L}_n(\Pi\, r) = \mathsf{map}\,\pi \bullet \mathscr{L}_{n+1}(r)$ 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:

[1] Due to Isabelle's lack of dependent types the actual type of π is $\alpha \to \alpha$. The more refined dependent type $\Sigma_{n+1} \to \Sigma_n$ is realized via Isabelle's tool for modeling parameterized systems with additional assumptions: locales (Ballarin, 2006). A locale fixes parameters and states assumptions about them. Hence, we use the locale assumption $\pi \bullet \Sigma_{n+1} \subseteq \Sigma_n$ to relate locale parameters π and Σ.

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. Though this is a good intuition, the actual encoding of formulas later on will be slightly more complicated. Fortunately we can ignore these complications for now by working with arbitrary but fixed Σ_n and π in the current section. Specific instantiations for them are given in Section 5.

4.2 Deciding Language Equivalence

Now we turn our attention to deciding equivalence of Π-extended regular expressions. The key concepts required for this are nullability and derivatives. We call a regular expression *nullable* if its language contains the empty word $[]$. Nullability can be easily checked syntactically by the following recursive function $\varepsilon :: \alpha\ \mathsf{RE} \to \mathbb{B}$.

$$\begin{array}{llll} \varepsilon(\mathbf{0}) & = \bot & \varepsilon(\mathbf{1}) & = \top \\ \varepsilon(a) & = \bot & \varepsilon(r+s) & = \varepsilon(r) \vee \varepsilon(s) \\ \varepsilon(r \cdot s) & = \varepsilon(r) \wedge \varepsilon(s) & \varepsilon(r^*) & = \top \\ \varepsilon(r \cap s) & = \varepsilon(r) \wedge \varepsilon(s) & \varepsilon(\neg\, r) & = \neg \varepsilon(r) \\ \varepsilon(\Pi\, r) & = \varepsilon(r) & & \end{array}$$

The characteristic property—$\varepsilon(r)$ iff $[] \in \mathscr{L}_n(r)$ for any regular expression r and $n :: \mathbb{N}$—follows by structural induction on r.

The second key concept—the *derivative* of a regular expression $\mathscr{D} :: \alpha \to \alpha\ \mathsf{RE} \to \alpha\ \mathsf{RE}$ and its lifting to words $\mathscr{D}^* :: \alpha\ \mathsf{list} \to \alpha\ \mathsf{RE} \to \alpha\ \mathsf{RE}$—semantically corresponds to left quotients of regular languages with respect to a fixed letter or word. Just as before, the recursive definition is purely syntactic and the semantic correspondence is established by a straightforward structural induction.

$$\begin{array}{lll} \mathscr{D}_b(\mathbf{0}) & = \mathbf{0} & \mathscr{D}_b(\mathbf{1}) = \mathbf{0} \\ \mathscr{D}_b(a) & = \text{if } a = b \text{ then } \mathbf{1} \text{ else } \mathbf{0} & \mathscr{D}_b(r+s) = \mathscr{D}_b(r) + \mathscr{D}_b(s) \\ \mathscr{D}_b(r \cdot s) & = & \mathscr{D}_b(r^*) = \mathscr{D}_b(r) \cdot r^* \\ \quad \text{if } \varepsilon(r) \text{ then } \mathscr{D}_b(r) \cdot s + \mathscr{D}_b(s) & & \\ \quad \text{else } \mathscr{D}_b(r) \cdot s & & \\ \mathscr{D}_b(r \cap s) = \mathscr{D}_b(r) \cap \mathscr{D}_b(s) & & \mathscr{D}_b(\neg\, r) = \neg\, \mathscr{D}_b(r) \\ \mathscr{D}_b(\Pi\, r) & = \Pi \left(\bigoplus_{c \in \pi^{-} b} \mathscr{D}_c(r) \right) & \\ \mathscr{D}^*_{[]}(r) & = r & \mathscr{D}^*_{bw}(r) = \mathscr{D}^*_w(\mathscr{D}_b(r)) \end{array}$$

Lemma 1

Assume $b \in \Sigma_n$, $v \in \Sigma_n^*$ and let r be an n-wellformed regular expression. Then $\mathscr{L}_n(\mathscr{D}_b(r)) = \{w \mid bw \in \mathscr{L}_n(r)\}$ and $\mathsf{wf}_n(\mathscr{D}_b(r))$, and consequently $\mathscr{L}_n(\mathscr{D}^*_v(r)) = \{w \mid vw \in \mathscr{L}_n(r)\}$ and $\mathsf{wf}_n(\mathscr{D}^*_v(r))$.

The projection case introduced some new syntax that deserves some explanation. The preimage π^- applied to a letter $b \in \Sigma_n$ denotes the set $\{c \in \Sigma_{n+1} \mid \pi\, c = b\}$. 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 $\mathbf{0}$.

Derivatives of extended regular expressions were introduced by Brzozowski (1964) fifty years ago. Our contribution is the extension of the concept to handle the projection operation. Since the projection acts homomorphically on words, it is clear that the derivative of $\Pi\, r$ with respect to a letter b can be expressed as a projection of derivatives of r. The concrete definition is a consequence of the following identity of left quotients for $b \in \Sigma_n$ and $A \subseteq \Sigma_{n+1}^*$:

$$\{w \mid bw \in \mathsf{map}\,\pi \bullet A\} = \mathsf{map}\,\pi \bullet \bigcup_{c \in \pi^- b} \{w \mid cw \in A\}$$

Although we completely avoid automata in the formalization, a derivative with respect to the letter b can be seen as a transition labeled by b in a deterministic automaton, the states of which are labeled by regular expressions. The automaton accepting the language of a regular expression r can be thus constructed iteratively by exploring all derivatives of r and defining exactly those states as accepting, which are labeled by a nullable regular expression. However, the set $\{\mathscr{D}_w^*(r) \mid w :: \alpha\ \mathsf{list}\}$ of states reachable in this manner is infinite in general. To obtain a finite automaton, the states must be partitioned into classes of regular expressions that are *ACI-equivalent*, i.e. syntactically equal modulo associativity, commutativity and idempotence of the $+$-constructor or more formally related by the following inductively defined congruence $\sim$.

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

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

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

Brzozowski showed that the number of $\sim$-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—indeed the verification of Theorem 2 constitutes the most intricate proof in the present work.

Theorem 2
$\{\langle \mathscr{D}_w^*(r) \rangle \mid w :: \alpha\ \mathsf{list}\}$ is finite for any regular expression r.

The function $\langle - \rangle :: \alpha\ \mathsf{RE} \to \alpha\ \mathsf{RE}$ is the ACI normalization function, which maps ACI-equivalent regular expressions to the same representative, i.e., defines a particular executable choice of representatives of $\sim$-equivalence classes. It is defined by means of a normalizing constructor $\oplus :: \alpha\ \mathsf{RE} \to \alpha\ \mathsf{RE} \to \alpha\ \mathsf{RE}$ and an arbitrary linear order $\preceq$ on

regular expressions. The equations for $\oplus$ are matched sequentially.

$$\begin{array}{llll} \langle \mathbf{0} \rangle & = \mathbf{0} & \langle \mathbf{1} \rangle & = \mathbf{1} \\ \langle a \rangle & = a & \langle r + s \rangle & = \langle r \rangle \oplus \langle s \rangle \\ \langle r \cdot s \rangle & = \langle r \rangle \cdot \langle s \rangle & \langle r^* \rangle & = \langle r \rangle^* \\ \langle r \cap s \rangle & = \langle r \rangle \cap \langle s \rangle & \langle \neg\, r \rangle & = \neg \langle r \rangle \\ \langle \Pi\, r \rangle & = \Pi \langle r \rangle & & \end{array}$$

$$\begin{array}{lcl} (r + s) \oplus t & = & r \oplus (s \oplus t) \\ r \oplus (s + t) & = & \text{if } r = s \text{ then } s + t \\ & & \text{else if } r \preceq s \text{ then } r + (s + t) \\ & & \quad \text{else } s + (r \oplus t) \\ r \oplus s & = & \text{if } r = s \text{ then } r \\ & & \text{else if } r \preceq s \text{ then } r + s \\ & & \quad \text{else } s + r \end{array}$$

When proving Theorem 2 by induction, on a high-level most cases follow Brzozowski's original proof (1964). 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 $\mathscr{D}^*_w(r)$ 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 $\mathscr{D}^*_w(r)$ modulo ACI.

The above proof sketch is very informal. The corresponding formal proof is technically more challenging, e.g., we need to define precisely in which way $\mathscr{D}^*_w(r)$ 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 $\preceq$ and duplicated summands are removed. From this, further later on useful properties of $\langle - \rangle$ can be derived:

Lemma 3
Let r be a regular expression, $n :: \mathbb{N}$ and $b \in \Sigma_n$. Then $\mathscr{L}_n \langle r \rangle = \mathscr{L}_n(r)$, $\langle \langle r \rangle \rangle = \langle r \rangle$ and $\langle \mathscr{D}_b \langle r \rangle \rangle = \langle \mathscr{D}_b(r) \rangle$.

So far, ACI normalization only connects Brzozowski derivatives to deterministic finite automata. Furthermore, it will ensure termination of our decision procedure even without ever entering the world of automata. Instead we follow Rutten (1998), who gives an alternative view on deterministic automata as coalgebras. In the coalgebraic setting the function $\lambda r.\ (\varepsilon(r), \lambda b.\ \mathscr{D}_b(r)) :: \alpha\ \mathsf{RE} \to \mathbb{B} \times (\alpha \to \alpha\ \mathsf{RE})$ is a D-coalgebra for the functor $D(S) = \mathbb{B} \times (\alpha \to S)$. The final coalgebra of D exists and corresponds exactly to the set of all languages. Therefore, we obtain the powerful coinduction principle, reducing language equality to bisimilarity. We phrase this general theorem instantiated to our concrete setting. The formalized proof itself does not require any category theory; it resembles the reasoning in § 4 of Rutten (1998).

Theorem 4 (*Coinduction*)

Let $R :: (\alpha\ \mathsf{RE} \times \alpha\ \mathsf{RE})\ \mathsf{set}$ be a relation, such that for all $(r, s) \in R$ we have:

1. $\mathsf{wf}_n(r) \wedge \mathsf{wf}_n(s)$;
2. $\varepsilon(r) \leftrightarrow \varepsilon(s)$;
3. $(\langle \mathscr{D}_b(r) \rangle, \langle \mathscr{D}_b(s) \rangle) \in R$ for all $b \in \Sigma_n$.

Then for all $(r, s) \in R$, $\mathscr{L}_n(r) = \mathscr{L}_n(s)$ holds.

From Lemma 1 and Lemma 3, we know that the relation

$$\mathscr{B} = \{(\langle \mathscr{D}^*_w(r) \rangle, \langle \mathscr{D}^*_w(s) \rangle) \mid w \in \Sigma^*_n\}$$

contains $(\langle r \rangle, \langle s \rangle)$ and fulfills the assumptions 1 and 3 of the coinduction theorem, assuming that r and s are both n-wellformed. Moreover, using Theorem 2 it follows that this relation is finite. Thus, checking assumption 2 for every pair of this finite relation is sufficient to prove language equality of r and s by coinduction. We obtain the following abstract specification of a language equivalence checking algorithm.

Theorem 5
Let r and s be n-wellformed regular expressions. Then $\mathscr{L}_n(r) = \mathscr{L}_n(s)$ iff we have $\varepsilon(r') \leftrightarrow \varepsilon(s')$ for all $(r', s') \in \mathscr{B}$.

4.3 Executable Algorithm from a Theorem

Our goal is not only to prove some abstract theorems about a decision procedure, but also to extract executable code in some functional programming language (e.g. Standard ML, Haskell, OCaml) using the code generation facility of Isabelle/HOL (Haftmann & Nipkow, 2010). Theorem 5 is not enough to do so: it contains a set comprehension ranging over the infinite set Σ^*_n, which is not executable as such. We need to instruct the system how to enumerate $\mathscr{B}$.

We start with the pair $(\langle r \rangle, \langle s \rangle)$ and compute its pairwise derivatives for all letters of the alphabet. For the computed pairs of regular expressions we proceed by computing their derivatives and so on. This of course does not terminate. However, if we stop our exploration at pairs that we have seen before it does, since we are exploring a finite set.

In more detail, we use a worklist algorithm that iteratively adds not yet inspected pairs of regular expressions while exhausting words of increasing length until no new pairs are generated. Saturation is reached by means of the executable combinator $\mathsf{while} :: (\alpha \to \mathbb{B}) \to (\alpha \to \alpha) \to \alpha \to \alpha\ \mathsf{option}$ from the Isabelle/HOL library. The option type $\alpha\ \mathsf{option}$ has two constructors $\mathsf{None} :: \alpha\ \mathsf{option}$ and $\mathsf{Some} :: \alpha \to \alpha\ \mathsf{option}$. Some lifts elements from the base type α to the option type, while None is usually used to indicate some exceptional behavior. The definition of while

$$\mathsf{while}\ b\ c\ s\ =\ \mathsf{if}\ \exists k.\neg b(c^k(s))\ \mathsf{then}\ \mathsf{Some}\ (c^{\mathsf{Least}\ k.\neg b(c^k(s))}(s))\ \mathsf{else}\ \mathsf{None}$$

is not executable, but the following key lemma is:

$$\mathsf{while}\ b\ c\ s\ =\ \mathsf{if}\ b\ s\ \mathsf{then}\ \mathsf{while}\ b\ c\ (c\ s)\ \mathsf{else}\ \mathsf{Some}\ s$$

The code generated from this 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 prove termination if we can show that the result is $\neq$ None.

In our case, the state s of the while loop consists of a worklist $ws :: (\alpha\ \mathsf{RE} \times \alpha\ \mathsf{RE})\ \mathsf{list}$ of unprocessed pairs of regular expressions together with a set $N :: (\gamma \times \gamma)\ \mathsf{set}$ of already seen pairs modulo a normalization function $\mathsf{norm} :: \alpha\ \mathsf{RE} \to \gamma$. This normalization function (which is a parameter of our setup) is applied to already ACI-normalized expressions, to syntactically identify further language equivalent expressions. This makes the bisimulation relation that must be exhausted smaller, thus saturation is reached faster. The range type of the normalization is not fixed, but we require a notion of languages $\mathscr{L}^\gamma :: \mathbb{N} \to \gamma \to (\alpha\ \mathsf{list})\ \mathsf{set}$ to be available for it, such that $\mathscr{L}^\gamma_n(\mathsf{norm}\ r) = \mathscr{L}_n(r)$ holds. In the simplest case norm can be the identity function and $\mathscr{L}^\gamma = \mathscr{L}$. More interesting is a function on regular expressions that eliminates $\mathbf{0}$ from unions, concatenations and intersections and $\mathbf{1}$ from concatenations. Other regular structures such as automata or different kinds of regular expressions as instantiations for γ might enable even more sophisticated simplifications.

We define the arguments to the while combinator $\mathsf{b} :: (\alpha\ \mathsf{RE} \times \alpha\ \mathsf{RE})\ \mathsf{list} \times (\gamma \times \gamma)\ \mathsf{set} \to \mathbb{B}$ and $\mathsf{c} :: \mathbb{N} \to (\alpha\ \mathsf{RE} \times \alpha\ \mathsf{RE})\ \mathsf{list} \times (\gamma \times \gamma)\ \mathsf{set} \to (\alpha\ \mathsf{RE} \times \alpha\ \mathsf{RE})\ \mathsf{list} \times (\gamma \times \gamma)\ \mathsf{set}$.

$$
\begin{array}{l}
\mathsf{b}\ ([], \_) \quad = \ \bot \\
\mathsf{b}\ ((r,s)\ \#\ \_, \_) \ = \ \varepsilon(r) \leftrightarrow \varepsilon(s) \\
\\
\mathsf{c}_n\ ((r,s)\ \#\ ws, N) \ = \\
\quad \mathsf{let} \\
\qquad ss = \mathsf{map}\ (\lambda b. \\
\qquad\quad \mathsf{let} \\
\qquad\qquad r' = \langle \mathscr{D}_b(r) \rangle; \quad s' = \langle \mathscr{D}_b(s) \rangle \\
\qquad\quad \mathsf{in}\ ((r', s'), (\mathsf{norm}\ r', \mathsf{norm}\ s'))) \ \Sigma_n; \\
\qquad new = \mathsf{remdups}\ \mathsf{snd}\ (\mathsf{filter}\ (\lambda(\_, rs).\ rs \notin N)\ ss) \\
\quad \mathsf{in}\ (ws\ @\ \mathsf{map}\ \mathsf{fst}\ new, \mathsf{set}\ (\mathsf{map}\ \mathsf{snd}\ new) \cup N)
\end{array}
$$

The function $\mathsf{set} :: \alpha\ \mathsf{list} \to \alpha\ \mathsf{set}$ maps a list to the set of its elements, $\mathsf{filter} :: (\alpha \to \mathbb{B}) \to \alpha\ \mathsf{list} \to \alpha\ \mathsf{list}$ removes elements that do not fulfill the given predicate, while $\mathsf{remdups} :: (\alpha \to \beta) \to \alpha\ \mathsf{list} \to \alpha\ \mathsf{list}$ is used to keep the worklist as small as possible. $\mathsf{remdups}\ f\ xs$ removes duplicates from xs modulo the function f, e.g. $\mathsf{remdups}\ \mathsf{snd}\ [(\mathbf{0},\mathbf{0}), (\mathbf{1},\mathbf{0})] = [(\mathbf{1},\mathbf{0})]$ (which element is actually kept is irrelevant; the result $[(\mathbf{0},\mathbf{0})]$ would also be valid).

Finally, a wellformedness check completes the now executable algorithm $\mathsf{eqv}^{\mathsf{RE}} :: \mathbb{N} \to \alpha\ \mathsf{RE} \to \alpha\ \mathsf{RE} \to \mathbb{B}$.

$$
\begin{array}{l}
\mathsf{eqv}^{\mathsf{RE}}_n\ r\ s \ = \\
\quad \mathsf{wf}_n(r) \wedge \mathsf{wf}_n(s) \wedge \\
\quad (\mathsf{case\ while\ b\ c}_n\ ([(\langle r \rangle, \langle s \rangle)], \{(\mathsf{norm}\langle r \rangle, \mathsf{norm}\langle s \rangle)\})\ \mathsf{of} \\
\qquad\quad \mathsf{Some}\ ([], \_) \Rightarrow \top \\
\quad \mid\ \mathsf{Some}\ (\_\ \#\ \_, \_) \Rightarrow \bot)
\end{array}
$$

The termination of $\mathsf{eqv}^{\mathsf{RE}}$ for any input is guaranteed by two facts: (1) all recursively defined functions in Isabelle/HOL terminate by their definitional principle (either primitive or wellfounded recursion) and (2) the termination of while follows from Theorem 2 and the fact that the set N of already seen pairs in the state is a subset of $(\lambda(r,s).\ (\mathsf{norm}\ r, \mathsf{norm}\ s)) \bullet \{(\langle \mathscr{D}^*_w(r) \rangle, \langle \mathscr{D}^*_w(s) \rangle) \mid w \in \Sigma^*_n\}$.

Theorem 6 (*Termination*)
Let r and s be n-wellformed regular expressions. Then

$$\mathsf{while}\ \mathsf{b}\ \mathsf{c}_n\,([[(\langle r\rangle,\langle s\rangle)]],\{(\mathsf{norm}\langle r\rangle,\mathsf{norm}\langle s\rangle)\}) \neq \mathsf{None}.$$

Function $\mathsf{eqv}^{\mathsf{RE}}$ deserves the name decision procedure since it constitutes a refinement of the algorithm abstractly stated in Theorem 5, and is therefore sound and complete. The refinement follows from proving the following predicate being an invariant for the states (ws, N) of the while-loop given two initial n-wellformed regular expressions r and s:

$$\begin{aligned}
&\mathsf{inv}\ (ws, N) = \\
&\quad (\forall (r', s') \in \mathsf{set}\ ws.\ (\mathsf{norm}\ r', \mathsf{norm}\ s') \in N) \wedge \\
&\quad (\forall (r', s') \in N.\ \exists w \in \Sigma_n^*.\ \mathscr{D}_w^*(r) = r' \wedge \mathscr{D}_w^*(r) = s') \wedge \\
&\quad (\forall (r', s') \in N \setminus ((\lambda(r, s).\ (\mathsf{norm}\ r, \mathsf{norm}\ s)) \bullet (\mathsf{set}\ ws)).\ \varepsilon(r') \leftrightarrow \varepsilon(s') \wedge \\
&\qquad (\forall a \in \Sigma_n.\ (\mathsf{norm}\ (\mathscr{D}_a(r')), \mathsf{norm}\ (\mathscr{D}_a(s'))) \in N))
\end{aligned}$$

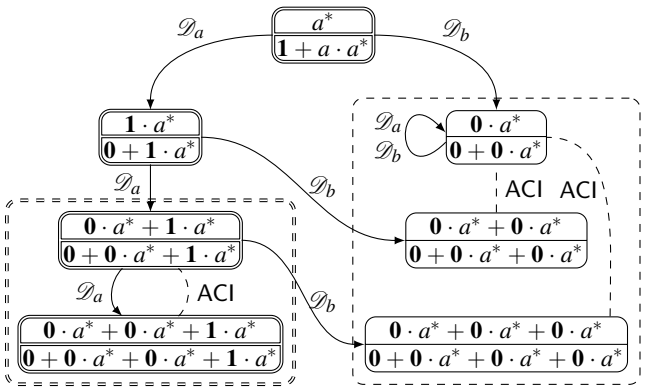
For an execution of $\mathsf{eqv}^{\mathsf{RE}}$, either ws is eventually emptied—in which case the last conjunct of inv corresponds to N being a bisimulation modulo norm—or the test b fails for pair in ws yielding a counterexample to language equivalence using the first two conjuncts of inv.

Theorem 7 (*Soundness*)
Let r and s be regular expressions such that $\mathsf{eqv}_n^{\mathsf{RE}}\, r\ s$. Then $\mathscr{L}_n(r) = \mathscr{L}_n(s)$.

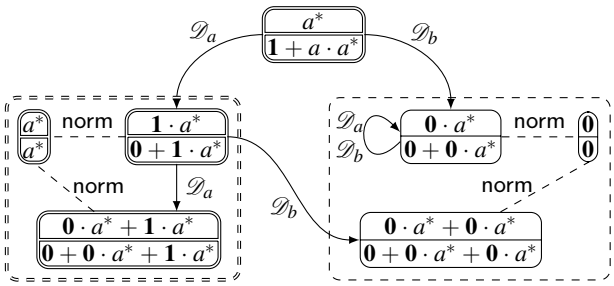
Theorem 8 (*Completeness*)
Let r and s be n-wellformed regular expressions such that $\mathscr{L}_n(r) = \mathscr{L}_n(s)$. Then $\mathsf{eqv}_n^{\mathsf{RE}}\, r\ s$.

Let us observe the decision procedure at work by looking at the regular expressions a^* and $\mathbf{1} + a \cdot a^*$ for some $a \in \Sigma_n = \{a, b\}$ for some n. For presentation purposes, the correspondence of derivatives to automata is useful. Figure 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 plain ACI normalization, while those of automaton (b) are modulo a stronger normalization function, making the automaton smaller. Transitions correspond to pairwise derivatives and doubled margins denote states for which the associated pairs of regular expressions are pairwise nullable. Both automata are the result of our decision procedure performing a breadth-first exploration starting with the initially given pair and ignoring states that are in the equivalence class of already visited states. The absence of pairs (r, s) for which r is nullable and s is not nullable (or vice versa) proves the equivalence of all pairs in the automaton, including the pair $(a^*, \mathbf{1} + a \cdot a^*)$.

Let us mention two obvious performance deficits of our algorithm. First, the ACI normalization is sorting the summands in expressions basically using bubble-sort. Using a set data structure instead of binary sums would improve this to merge-sort and is certainly desirable. Second, 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 (Bonchi & Pous, 2013; Pous, 2015). However, when verifying an algorithm one has to settle for a solution somewhere in between of efficiency and simplicity.



(a) norm is identity function



(b) norm unfolds $\mathbf{0} + r = r$, $\mathbf{0} \cdot r = \mathbf{0}$ and $\mathbf{1} \cdot r = r$

Fig. 1. Checking the equivalence of a^* and $\mathbf{1} + a \cdot a^*$ for $\Sigma_n = \{a, b\}$

4.4 Atoms with More Structure

Owens *et al.* (2009) 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 $(\alpha\ \mathsf{set})\ \mathsf{RE}$ instead of $\alpha\ \mathsf{RE}$ (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 $\alpha\ \mathsf{RE}$, the regular expressions over the alphabet type α on which the algorithm operates will be of type $\beta\ \mathsf{RE}$, where the relationship between α and the new atoms β is given by a function $\mathsf{mem}^{\mathsf{A}} :: \beta \to \alpha \to \mathbb{B}$. The new semantics $\mathscr{L} :: \mathbb{N} \to \beta\ \mathsf{RE} \to (\alpha\ \mathsf{list})\ \mathsf{set}$ of such regular expressions is defined just as the old $\mathscr{L}$ except for the atom case. A similar adjustment is required for the new derivative $\mathscr{D} :: \alpha \to \beta\ \mathsf{RE} \to \beta\ \mathsf{RE}$.

$$\mathscr{L}_n(b) = \{a \mid \mathsf{mem}^{\mathsf{A}}\ b\ a\} \qquad \mathscr{D}_a(b) = \mathsf{if}\ \mathsf{mem}^{\mathsf{A}}\ b\ a\ \mathsf{then}\ \mathbf{1}\ \mathsf{else}\ \mathbf{0}$$

Furthermore, the function $\mathsf{wf}^{\mathsf{A}} :: \mathit{nat} \to \beta \to \mathbb{B}$ is used to detect whether a β-atom is wellformed. The wellformedness check for regular expressions $\mathsf{wf} :: \mathit{nat} \to \beta\ \mathsf{RE} \to \mathbb{B}$ will use wf^{A} in the atom case: $\mathsf{wf}_n(b) = \mathsf{wf}^{\mathsf{A}}\ n\ b$. The functions $\mathsf{mem}^{\mathsf{A}}$ 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.* (2009), one would instantiate β with α set and define $\mathsf{mem}^{\mathsf{A}}\ (B :: \alpha\ \mathsf{set})\ a \leftrightarrow (a \in B)$ and $\mathsf{wf}^{\mathsf{A}}\ n\ B \leftrightarrow (\forall a \in B.\ a \in \Sigma_n)$.

The benefit of the abstract formulation is the fact that β can be instantiated with a set representation tailored to the particularities of the used regular expressions. In our case, the regular expressions are translated MSO formulas and a few very particular sets of letters arise from the translation. Therefore, in Section 5 we will define a data type $\alpha\ \mathsf{atom}$ matching exactly those particularities and instantiate β, $\mathsf{mem}^{\mathsf{A}}$, and wf^{A} accordingly.

4.5 Alternatives to Brzozowski Derivatives

The example from Figure 1 shows that the choice of the normalization is crucial for the size of the bisimulation relation. In prior work (Nipkow & Traytel, 2014) we show that partial derivatives[2] of ordinary regular expressions can be represented by a composition $\langle\!\langle - \rangle\!\rangle \circ \mathscr{D}_a$ where $\langle\!\langle - \rangle\!\rangle$ is a particular normalization function defined using smart constructors and observe that $\langle\!\langle - \rangle\!\rangle$ 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 $\langle\!\langle - \rangle\!\rangle$ to Π-extended regular expressions as follows. The equations for the smart constructors $\boxplus$, $\boxdot$, $\boxcap$, $\boxminus$, and $\boxPi$ are matched sequentially.

$$
\begin{aligned}
\langle\!\langle \mathbf{0} \rangle\!\rangle &= \mathbf{0} \\
\langle\!\langle \mathbf{1} \rangle\!\rangle &= \mathbf{1} \\
\langle\!\langle a \rangle\!\rangle &= a \\
\langle\!\langle r + s \rangle\!\rangle &= \langle\!\langle r \rangle\!\rangle \boxplus \langle\!\langle s \rangle\!\rangle \\
\langle\!\langle r \cdot s \rangle\!\rangle &= \langle\!\langle r \rangle\!\rangle \boxdot s \\
\langle\!\langle r^* \rangle\!\rangle &= r^* \\
\langle\!\langle r \cap s \rangle\!\rangle &= \langle\!\langle r \rangle\!\rangle \boxcap \langle\!\langle s \rangle\!\rangle \\
\langle\!\langle \neg\, r \rangle\!\rangle &= \boxminus \langle\!\langle r \rangle\!\rangle \\
\langle\!\langle \Pi\, r \rangle\!\rangle &= \boxPi \langle\!\langle r \rangle\!\rangle
\end{aligned}
$$

$$
\begin{aligned}
\mathbf{0} \boxdot r &= \mathbf{0} \\
\mathbf{1} \boxdot r &= r \\
(r + s) \boxdot t &= (r \boxdot s) \boxplus (s \boxdot t) \\
r \boxdot s &= s \cdot t
\end{aligned}
$$

$$
\begin{aligned}
\boxminus (r + s) &= (\boxminus r) \boxcap (\boxminus s) \\
\boxminus (r \cap s) &= (\boxminus r) \boxplus (\boxminus s) \\
\boxminus (\neg\, r) &= r \\
\boxminus r &= \neg\, r
\end{aligned}
$$

$$
\begin{aligned}
\boxPi\, \mathbf{0} &= \mathbf{0} \\
\boxPi\, \mathbf{1} &= \mathbf{1} \\
\boxPi\, (r + s) &= (\boxPi\, r) \boxplus (\boxPi\, s) \\
\boxPi\, r &= \Pi\, r
\end{aligned}
$$

$$
\begin{aligned}
\mathbf{0} \boxplus r &= r \\
r \boxplus \mathbf{0} &= r \\
(r + s) \boxplus t &= r \boxplus (s \boxplus t) \\
r \boxplus (s + t) &= \text{if } r = s \text{ then } s + t \\
&\quad \text{else if } r \preceq s \text{ then } r + (s + t) \\
&\quad\quad \text{else } s + (r \boxplus t) \\
r \boxplus s &= \text{if } r = s \text{ then } r \\
&\quad \text{else if } r \preceq s \text{ then } r + s \\
&\quad\quad \text{else } s + r
\end{aligned}
$$

$$
\begin{aligned}
\mathbf{0} \boxcap r &= \mathbf{0} \\
r \boxcap \mathbf{0} &= \mathbf{0} \\
(\neg\, \mathbf{0}) \boxcap r &= r \\
r \boxcap (\neg\, \mathbf{0}) &= r \\
(r + s) \boxcap t &= (r \boxcap t) \boxplus (s \boxcap t) \\
r \boxcap (s + t) &= (r \boxcap s) \boxplus (r \boxcap t) \\
(r \cap s) \boxcap t &= r \boxcap (s \boxcap t) \\
r \boxcap (s \cap t) &= \text{if } r = s \text{ then } s \cap t \\
&\quad \text{else if } r \preceq s \text{ then } r \cap (s \cap t) \\
&\quad\quad \text{else } s \cap (r \boxcap t) \\
r \boxcap s &= \text{if } r = s \text{ then } r \\
&\quad \text{else if } r \preceq s \text{ then } r \cap s \\
&\quad\quad \text{else } s \cap r
\end{aligned}
$$

[2] Partial derivatives (Antimirov, 1996) refine Brzozowski derivatives by splitting the derivation result at some $+$-constructors into a finite set of regular expressions. Partial derivatives correspond to nondeterministic automata in the same way derivatives correspond to deterministic ones.

It is worth noticing that $\langle\!\langle - \rangle\!\rangle$ does not descend recursively into right hand side of concatenation and into iteration. Also, $\boxdot$ 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.* (2011), 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.* (2011) 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.

Lemma 9
Let r be an n-wellformed regular expression. Then $\mathsf{wf}_n\langle\!\langle r \rangle\!\rangle$ and $\mathscr{L}_n\langle\!\langle r \rangle\!\rangle = \mathscr{L}_n(r)$.

We remark that the normalization $\langle\!\langle - \rangle\!\rangle$ does not enjoy nice algebraic properties. The source of the problem is that our smart constructor $\boxdot$ is not idempotent. To see this, assuming $a \preceq b \preceq a \cap b$, we calculate: $(a+b) \boxdot (a+b) = a + b + (a \cap b)$. 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 (Caron *et al.*, 2011). (Those operations suffer from the same limitations.) However, the performance when using this normalization in practice seems reasonable and our attempts in changing the normalization function to make $\boxdot$ idempotent (for example by giving up distributivity of $\cap$ over $+$ or by adding more equality checks in the definition of $\boxdot$) resulted in a perceivable decrease in performance.

Nevertheless, an interesting question is whether one can find a fast normalization function that decides equivalence under the following inductively defined equivalence relation $\approx$, which is modeled after what the normalization function $\langle\!\langle - \rangle\!\rangle$ attempts (but fails) to equate. Note that, unlike $\sim$ (ACI), the relation $\approx$ is only an equivalence, not a congruence. Not being able to find such a normalization we leave this question as future work.

$$\begin{array}{cccc}
\mathbf{0} + r \approx r & r + \mathbf{0} \approx r & \mathbf{0} \cdot r \approx \mathbf{0} & \mathbf{1} \cdot r \approx r \\
(\neg\,\mathbf{0}) \cap r \approx r & r \cap (\neg\,\mathbf{0}) \approx r & \mathbf{0} \cap r \approx \mathbf{0} & r \cap \mathbf{0} \approx \mathbf{0} \\
r + (s+t) \approx (r+s) + t & & r + s \approx s + r & r + r \approx r \\
r \cap (s \cap t) \approx (r \cap s) \cap t & & r \cap s \approx s \cap r & r \cap r \approx r \\
r \cap (s+t) \approx (r \cap s) + (r \cap t) & & (r+s) \cap t \approx (r \cap t) + (s \cap t) & \\
(r+s) \cdot t \approx (r \cdot t) + (s \cdot t) & & \neg\,(\neg\, r) \approx r & \\
\neg\,(r+s) \approx (\neg\, r) \cap (\neg\, s) & & \neg\,(r \cap s) \approx (\neg\, r) + (\neg\, s) & \\
\Pi\,(r+s) \approx \Pi\, r + \Pi\, s & & \Pi\,\mathbf{0} \approx \mathbf{0} & \Pi\,\mathbf{1} \approx \mathbf{1} \\
r \approx r & \dfrac{r \approx s}{s \approx r} & \dfrac{r \approx s \quad s \approx t}{r \approx t} & \dfrac{r_1 \approx s_1 \quad r_2 \approx s_2}{r_1 + r_2 \approx s_1 + s_2} \\
\dfrac{r_1 \approx s_1 \quad r_2 \approx s_2}{r_1 \cap r_2 \approx s_1 \cap s_2} & \dfrac{r_1 \approx s_1}{r_1 \cdot t \approx s_1 \cdot t} & \dfrac{r \approx s}{\neg\, r \approx \neg\, s} & \dfrac{r \approx s}{\Pi\, r \approx \Pi\, s}
\end{array}$$

Another promising alternative to Brzozowski derivatives is the data type $\alpha\ \mathsf{RE}^{\mathsf{op}}$ of *dual regular expressions* (Okhotin, 2005). The data type is obtained by modifying $\alpha\ \mathsf{RE}$ as following: drop the negation and intersection constructors and add to every remaining n-ary constructor $\circ$ a Boolean flag b with the following semantics $\mathscr{L}^{\mathsf{op}} :: \mathbb{N} \to \alpha\ \mathsf{RE}^{\mathsf{op}} \to \mathbb{B}$.

$$\mathscr{L}_n^{\mathsf{op}}(\circ\, b\, r_1\ \cdots r_n) = \mathscr{L}_n(\text{if } b \text{ then } \circ\, r_1\ \cdots r_n \text{ else } \neg\,(\circ\,(\neg\, r_1)\ \cdots (\neg\, r_n)))$$

In the formalization (Traytel & Nipkow, 2014) we define wellformedness, derivatives and some ad hoc normalization on $\alpha\ \mathsf{RE}^{\mathsf{op}}$ and generalize the bisimulation construction to work on $\alpha\ \mathsf{RE}^{\mathsf{op}}$ as well. The evaluation will show that the decision procedure we obtain by using dual regular expression performs better for the WS1S semantics of MSO, but worse for the M2L semantics—a phenomenon for which we do not have an explanation yet.

5 MSO on Finite Words

Logics on finite words consider formulas in the context of a formal word, with variables representing positions in the word. In the first-order logic on words a variable always denotes a single position, while in monadic second-order logic on finite words (MSO) variables come in two flavors: first-order variables for single positions and second-order variables for finite sets of positions.

In the next subsections we first define the syntax of formulas and give them a semantics that is related to formal languages: M2L(Str). The second semantics, WS1S, is introduced as a relaxation of M2L (we drop the "(Str)" from now on). Both semantics are equally expressive and deciding both is of nonelementary complexity. The benefits and drawbacks of the two semantics are discussed elsewhere (Klarlund, 1999; Ayari & Basin, 2000).

5.1 Syntax and M2L Semantics

MSO formulas are syntactically represented by the recursive data type $\alpha\ \Phi$ using de Bruijn indices for variable bindings. Terms of $\alpha\ \Phi$ are generated by the grammar

$$\varphi = \mathsf{Q}\ a\ m \mid m_1 < m_2 \mid m \in M \mid \varphi \wedge \psi \mid \varphi \vee \psi \mid \neg\, \varphi \mid \exists\, \varphi \mid \mathbf{\exists}\, \varphi$$

where $\varphi, \psi :: \alpha\ \Phi$, $m, m_1, m_2, M :: \mathbb{N}$ and $a :: \alpha$. Lower-case variables m, m_1, m_2 denote first-order variables, M denotes a second order variable. The atomic formula $\mathsf{Q}\ a\ m$ requires the letter of the word at the position represented by variable m to be a; the constructors $<$ and $\in$ compare positions; Boolean operators are interpreted as usual.

The bold existential quantifier $\mathbf{\exists}$ binds second-order variables, $\exists$ binds first-order variables. Occurrences of bound variables represented as de Bruijn indices refer to their binders by counting the number of nested existential quantifier between the binder and the occurrence. For example, the formula $\exists\, (\mathsf{Q}\ a\ 0 \wedge (\mathbf{\exists}\ 1 \in 0))$ translates to $\exists x.(\mathsf{Q}\ a\ x \wedge (\exists X.\ x \in X))$ when using names. The first 0 in the nameless formula refers to the outermost first-order quantifier. Inside of the inner second-order quantifier, index 1 refers to the outermost quantifier and index 0 to the inner quantifier. The nameless representation simplifies reasoning by implicitly capturing α-equivalence of formulas. On the downside, de Bruijn indices are less readable and must be manipulated with care.

Formulas may have free variables. The functions $\mathscr{V}_1 :: \alpha\ \Phi \to \mathbb{N}\ \mathsf{set}$ and $\mathscr{V}_2 :: \alpha\ \Phi \to \mathbb{N}\ \mathsf{set}$ collect the free first-order and second-order variables:

$$\begin{array}{lll|lll}
\mathscr{V}_1(\mathsf{Q}\ a\ m) & = \{m\} & & \mathscr{V}_2(\mathsf{Q}\ a\ m) & = \{\} \\
\mathscr{V}_1(m_1 < m_2) & = \{m_1, m_2\} & & \mathscr{V}_2(m_1 < m_2) & = \{\} \\
\mathscr{V}_1(m \in M) & = \{m\} & & \mathscr{V}_2(m \in M) & = \{M\} \\
\mathscr{V}_1(\varphi \wedge \psi) & = \mathscr{V}_1(\varphi) \cup \mathscr{V}_1(\psi) & & \mathscr{V}_2(\varphi \wedge \psi) & = \mathscr{V}_2(\varphi) \cup \mathscr{V}_2(\psi) \\
\mathscr{V}_1(\varphi \vee \psi) & = \mathscr{V}_1(\varphi) \cup \mathscr{V}_1(\psi) & & \mathscr{V}_2(\varphi \vee \psi) & = \mathscr{V}_2(\varphi) \cup \mathscr{V}_2(\psi) \\
\mathscr{V}_1(\neg\, \varphi) & = \mathscr{V}_1(\varphi) & & \mathscr{V}_2(\neg\, \varphi) & = \mathscr{V}_2(\varphi) \\
\mathscr{V}_1(\exists\, \varphi) & = \lfloor \mathscr{V}_1(\varphi) \backslash \{0\} \rfloor & & \mathscr{V}_2(\exists\, \varphi) & = \lfloor \mathscr{V}_2(\varphi) \rfloor \\
\mathscr{V}_1(\mathbf{\exists}\, \varphi) & = \lfloor \mathscr{V}_1(\varphi) \rfloor & & \mathscr{V}_2(\mathbf{\exists}\, \varphi) & = \lfloor \mathscr{V}_2(\varphi) \backslash \{0\} \rfloor
\end{array}$$

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 $\alpha\ \Phi$ 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 $\mathsf{wf}^\Phi :: \mathbb{N} \to \alpha\ \Phi \to \mathbb{B}$ as $\mathsf{wf}^\Phi_n(\varphi) = (\mathscr{V}_1(\varphi) \cap \mathscr{V}_2(\varphi) = \{\}) \wedge \mathsf{pre\_wf}^\Phi_n(\varphi)$ and call a formula φ *n-wellformed* if $\mathsf{wf}^\Phi_n(\varphi)$ holds. The recursively defined predicate $\mathsf{pre\_wf}^\Phi :: \mathbb{N} \to \alpha\ \Phi \to \mathbb{B}$ is used for further assumptions on the structure of n-wellformed formulas, which will simplify our proofs:

$$\begin{array}{lll}
\mathsf{pre_wf}^\Phi_n(\mathsf{Q}\ a\ m) & = a \in \Sigma \wedge m < n \\
\mathsf{pre_wf}^\Phi_n(m_1 < m_2) & = m_1 < n \wedge m_2 < n \\
\mathsf{pre_wf}^\Phi_n(m \in M) & = m < n \wedge M < n \\
\mathsf{pre_wf}^\Phi_n(\varphi \wedge \psi) & = \mathsf{pre_wf}^\Phi_n(\varphi) \wedge \mathsf{pre_wf}^\Phi_n(\psi) \\
\mathsf{pre_wf}^\Phi_n(\varphi \vee \psi) & = \mathsf{pre_wf}^\Phi_n(\varphi) \wedge \mathsf{pre_wf}^\Phi_n(\psi) \\
\mathsf{pre_wf}^\Phi_n(\neg\, \varphi) & = \mathsf{pre_wf}^\Phi_n(\varphi) \\
\mathsf{pre_wf}^\Phi_n(\exists\, \varphi) & = \mathsf{pre_wf}^\Phi_{n+1}(\varphi) \wedge 0 \in \mathscr{V}_1(\varphi) \wedge 0 \notin \mathscr{V}_2(\varphi) \\
\mathsf{pre_wf}^\Phi_n(\mathbf{\exists}\, \varphi) & = \mathsf{pre_wf}^\Phi_{n+1}(\varphi) \wedge 0 \notin \mathscr{V}_1(\varphi) \wedge 0 \in \mathscr{V}_2(\varphi)
\end{array}$$

$\mathsf{pre\_wf}^\Phi_n(\varphi)$ ensures that the index of every free variable in φ is below n and the values of type α come from a fixed alphabet Σ. Note that Σ is really just a fixed set of letters of type α, independent of any n and is a parameter of our setup. Moreover, $\mathsf{pre\_wf}^\Phi$ 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.

An *interpretation* of an MSO formula is a pair of a word $w :: \alpha\ \mathsf{list}$ from Σ^* and an assignment $\mathscr{I} :: (\mathbb{N} + \mathbb{N}\ \mathsf{set})\ \mathsf{list}$ for free variables. The latter essentially consists of two functions with finite domain: one from first-order variables to positions and the other from second-order variables to sets of positions. We represent those two functions by a list, once again benefiting from de Bruijn indices—the value lookup for a variable with de Bruijn index i corresponds to inspecting the assignment $\mathscr{I}$ at position i, i.e. $\mathscr{I}[i]$. The range of $\mathscr{I}$ is a sum type, denoting the disjoint union of its two argument types. The sum type has two constructors $\mathsf{Inl} :: \alpha \to \alpha + \beta$ and $\mathsf{Inr} :: \beta \to \alpha + \beta$, such that for a first-order variable m

there is a position p with $\mathscr{I}[m] = \mathsf{Inl}\ p$ and for a second-order variable M there is a finite set of positions P with $\mathscr{I}[M] = \mathsf{Inr}\ P$.

An interpretation that *satisfies* a formula is called a model. Satisfiability for M2L, denoted by infix $\vDash :: \alpha\ \mathsf{list} \times (\mathbb{N} + \mathbb{N}\ \mathsf{set})\ \mathsf{list} \to \alpha\ \Phi \to \mathbb{B}$, is defined recursively on $\alpha\ \Phi$. To simplify the notation, the constructors Inl and Inr are stripped implicitly in the definition.

$$
\begin{aligned}
(w, \mathscr{I}) \vDash \mathsf{Q}\ a\ m &\leftrightarrow w[\mathscr{I}[m]] = a \\
(w, \mathscr{I}) \vDash m_1 < m_2 &\leftrightarrow \mathscr{I}[m_1] < \mathscr{I}[m_2] \\
(w, \mathscr{I}) \vDash m \in M &\leftrightarrow \mathscr{I}[m] \in \mathscr{I}[M] \\
(w, \mathscr{I}) \vDash \varphi \wedge \psi &\leftrightarrow (w, \mathscr{I}) \vDash \varphi \wedge (w, \mathscr{I}) \vDash \psi \\
(w, \mathscr{I}) \vDash \varphi \vee \psi &\leftrightarrow (w, \mathscr{I}) \vDash \varphi \vee (w, \mathscr{I}) \vDash \psi \\
(w, \mathscr{I}) \vDash \neg\, \varphi &\leftrightarrow (w, \mathscr{I}) \nvDash \varphi \\
(w, \mathscr{I}) \vDash \exists\, \varphi &\leftrightarrow \exists p \in \{0, \ldots, |w|-1\}.\ (w, \mathsf{Inl}\ p\ \#\ \mathscr{I}) \vDash \varphi \\
(w, \mathscr{I}) \vDash \mathsf{\exists}\, \varphi &\leftrightarrow \exists P \subseteq \{0, \ldots, |w|-1\}.\ (w, \mathsf{Inr}\ P\ \#\ \mathscr{I}) \vDash \varphi
\end{aligned}
$$

For the definition to make sense, $\mathscr{I}$ must correctly map first-order variables to positions (i.e. $\mathscr{I}[m] = \mathsf{Inl}\ p$) and second-order variables to sets of positions (i.e. $\mathscr{I}[M] = \mathsf{Inr}\ P$). Furthermore, all positions in $\mathscr{I}$ should be below the length of the word, and for technical reasons the word should not be empty. We formalize these assumptions by the predicate $\mathsf{wf}^{\mathsf{M2L}} :: \alpha\ \Phi \to \alpha\ \mathsf{list} \times (\mathbb{N} + \mathbb{N}\ \mathsf{set})\ \mathsf{list} \to \mathbb{B}$ and call an interpretation *M2L-wellformed* for φ if $\mathsf{wf}^{\mathsf{M2L}}_{\varphi}(w, \mathscr{I})$ holds:

$$
\begin{aligned}
\mathsf{wf}^{\mathsf{M2L}}_{\varphi}(w, \mathscr{I}) = {} & w \neq [\,] \wedge w \in \Sigma^* \wedge \\
& \forall\, \mathsf{Inl}\ p \in \mathsf{set}\ \mathscr{I}.\ p < |w| \wedge \\
& \forall\, \mathsf{Inr}\ P \in \mathsf{set}\ \mathscr{I}.\ (\forall p \in P.\ p < |w|) \wedge \\
& \forall m \in \mathscr{V}_1(\varphi).\ (\exists p.\ \mathscr{I}[m] = \mathsf{Inl}\ p) \wedge \\
& \forall M \in \mathscr{V}_2(\varphi).\ (\exists P.\ \mathscr{I}[M] = \mathsf{Inr}\ P)
\end{aligned}
$$

5.2 WS1S Semantics

In an M2L-wellformed model, positions are restricted by the length of the word. This is the key difference compared to WS1S. In WS1S no a priori restrictions on the variable ranges are made, although all second-order variables still represent finite sets. The subtle difference is illustrated by the formula $\mathsf{\exists}\ (\forall\, 0 \in 1)$ (with names: $\exists X. \forall x.\ x \in X$), where $\forall\, \varphi$ is just an abbreviation for $\neg\, \exists\, \neg\, \varphi$. In the M2L semantics $\mathsf{\exists}\ (\forall\, 0 \in 1)$ is satisfied by all wellformed interpretations—the witness set for the outer existential quantifier is for a wellformed interpretation $(w, \mathscr{I})$ just the set $\{0, \ldots, |w|-1\}$. In contrast, in WS1S, there is no finite set which contains all arbitrarily large positions, thus $\mathsf{\exists}\ (\forall\, 0 \in 1)$ is unsatisfiable.

Formally, satisfiability for WS1S, denoted by infix $\vDash^{\mathsf{e}} :: \alpha\ \mathsf{list} \times (\mathbb{N} + \mathbb{N}\ \mathsf{set})\ \mathsf{list} \to \alpha\ \Phi \to \mathbb{B}$, is defined just as for M2L (replacing $\vDash$ by $\vDash^{\mathsf{e}}$) except for the following equations.

$$
\begin{aligned}
(w, \mathscr{I}) \vDash^{\mathsf{e}} \mathsf{Q}\ a\ m &\leftrightarrow (\text{if } \mathscr{I}[m] < |w| \text{ then } w[\mathscr{I}[m]] \text{ else } \mathsf{z}) = a \\
(w, \mathscr{I}) \vDash^{\mathsf{e}} \exists\, \varphi &\leftrightarrow \exists p.\ (w, \mathsf{Inl}\ p\ \#\ \mathscr{I}) \vDash^{\mathsf{e}} \varphi \\
(w, \mathscr{I}) \vDash^{\mathsf{e}} \mathsf{\exists}\, \varphi &\leftrightarrow \exists P.\ (w, \mathsf{Inr}\ P\ \#\ \mathscr{I}) \vDash^{\mathsf{e}} \varphi \wedge \mathsf{finite}\ P
\end{aligned}
$$

Here, z is a distinguished letter from Σ. WS1S as defined in the literature does not handle the $\mathsf{Q}\ a\ m$ case at all, usually interpreting formulas only with respect to the assignment $\mathscr{I}$. In order to be able to use the same syntax and the same type of interpretations for both semantics, we have made the above choice. This also allows us to translate $\mathsf{Q}\ a\ m$ into the same regular expression irrespective of the intended semantics.

Besides the mentioned relaxation of *WS1S-wellformedness* regarding variable ranges, the empty word also does not impose technical complications as in M2L. Therefore, the predicate $\mathsf{wf}^{\mathsf{WS1S}} :: \alpha\ \Phi \to \alpha\ \mathsf{list} \times (\mathbb{N} + \mathbb{N}\ \mathsf{set})\ \mathsf{list} \to \mathbb{B}$ is defined as follows.

$$\begin{aligned}\mathsf{wf}^{\mathsf{WS1S}}_{\varphi}(w, \mathscr{I}) = w \in \Sigma^* \wedge\\ \forall \mathsf{Inr}\ P \in \mathsf{set}\ \mathscr{I}.\ \mathsf{finite}\ P \wedge\\ \forall m \in \mathscr{V}_1(\varphi).\ (\exists p.\ \mathscr{I}[m] = \mathsf{Inl}\ p) \wedge\\ \forall M \in \mathscr{V}_2(\varphi).\ (\exists P.\ \mathscr{I}[M] = \mathsf{Inr}\ P)\end{aligned}$$

5.3 Encoding Interpretations as Words

Formulas are equivalent if they have the same set of wellformed models. To relate equivalent formulas with language equivalent regular expressions, the set of wellformed models must be represented as a formal language by encoding interpretations as words. As before, we cover the encoding of the M2L semantics first.

To simplify the formalization, we choose a very simple encoding using Boolean vectors. For an interpretation $(w, \mathscr{I})$, we associate with every position p in the word w a Boolean vector bs of length $|\mathscr{I}|$, such that $bs[m] = \top$ iff the mth variable in $\mathscr{I}$ is first-order and its value is p or it is second-order and its value contains p. For example, for $\Sigma = \{a, b\}$ the interpretation $(w, \mathscr{I}) = (aba, \mathsf{Inl}\ 0\ \#\ \mathsf{Inr}\ \{1,2\}\ \#\ \mathsf{Inl}\ 2\ \#\ [])$ can be written in two dimensions as follows:

	a	b	a
$\mathsf{Inl}\ 0$	$\top$	$\bot$	$\bot$
$\mathsf{Inr}\ \{1,2\}$	$\bot$	$\top$	$\top$
$\mathsf{Inl}\ 2$	$\bot$	$\bot$	$\top$

In the first row, the value $\top$ is placed only in the first column because the first variable of $\mathscr{I}$ is the first-order position 0. In general, the columns correspond to the Boolean vectors associated with positions in the word, while every row corresponds to one variable. For first-order variables there must be exactly one $\top$ per row. The first row encodes the value of the most recently bound variable. Now, we consider every column as a letter of a new alphabet, which is the underlying alphabet $\Sigma_n = \Sigma \times \mathbb{B}^n$ of regular expressions of Section 4. This transformation of interpretations into words over Σ_n is performed by the function $\mathsf{enc}^{\mathsf{M2L}} :: \alpha\ \mathsf{list} \times (\mathbb{N} + \mathbb{N}\ \mathsf{set})\ \mathsf{list} \to (\alpha \times \mathbb{B}\ \mathsf{list})\ \mathsf{list}$; we omit its obvious definition.

Furthermore, the second parameter $\pi :: \Sigma_{n+1} \to \Sigma_n$ of our decision procedure for regular expressions can now be instantiated as the function that maps $(a, b\ \#\ bs)$ to (a, bs). 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.

Finally, the *M2L-language* $\mathscr{L}^{\mathsf{M2L}} :: \mathbb{N} \to \alpha\ \Phi \to (\alpha \times \mathbb{B}\ \mathsf{list})\ \mathsf{set}$ of an MSO formula is the set of encodings of its wellformed models, i.e. $\mathscr{L}^{\mathsf{M2L}}_n(\varphi) = \{\mathsf{enc}^{\mathsf{M2L}}(w, \mathscr{I}) \mid \mathsf{wf}^{\mathsf{M2L}}_\varphi(w, \mathscr{I}) \wedge |\mathscr{I}| = n \wedge (w, \mathscr{I}) \vDash \varphi\}$.

Concerning WS1S, the encoding is slightly more complicated due to the following observation: Interpretations $(w, \mathscr{I})$ and $(w\mathsf{z}^n, \mathscr{I})$ for all $n :: \mathbb{N}$ behave the same when considering satisfiability and wellformedness with respect to a formula (z^n denotes n-fold repetition of the letter z as a word). That suggests that the example interpretation $(w, \mathscr{I}) = (aba,\ \mathsf{Inl}\ 0\ \#\ \mathsf{Inr}\ \{1,2\}\ \#\ \mathsf{Inl}\ 2\ \#\ [])$ from above can be encoded as

	a	b	a	z^m
$\mathsf{Inl}\ 0$	$\top$	$\bot$	$\bot$	$\bot^m$
$\mathsf{Inr}\ \{1,2\}$	$\bot$	$\top$	$\top$	$\bot^m$
$\mathsf{Inl}\ 2$	$\bot$	$\bot$	$\top$	$\bot^m$

for every $m :: \mathbb{N}$. Hence, the a single WS1S interpretation is translated into a countably infinite set of words by a function $\mathsf{enc}^{\mathsf{WS1S}} :: \alpha\ \mathsf{list} \times (\mathbb{N} + \mathbb{N}\ \mathsf{set})\ \mathsf{list} \to (\alpha \times \mathbb{B}\ \mathsf{list})\ \mathsf{list}\ \mathsf{set}$; we again omit its formal definition. Accordingly, the *WS1S-language* $\mathscr{L}^{\mathsf{WS1S}} :: \mathbb{N} \to \alpha\ \Phi \to (\alpha \times \mathbb{B}\ \mathsf{list})\ \mathsf{set}$ of an MSO formula is defined as the union of all encodings of its well-formed models: $\mathscr{L}^{\mathsf{WS1S}}_n(\varphi) = \bigcup\{\mathsf{enc}^{\mathsf{WS1S}}(w, \mathscr{I}) \mid \mathsf{wf}^{\mathsf{WS1S}}_\varphi(w, \mathscr{I}) \wedge |\mathscr{I}| = n \wedge (w, \mathscr{I}) \models \varphi\}$.

5.4 From M2L Formulas to Regular Expressions

We have fixed the underlying alphabet type $\alpha \times \mathbb{B}\ \mathsf{list}$ of the language of a formula. In principle, we could start translating formulas of type $\alpha\ \Phi$ into regular expressions of type $(\alpha \times \mathbb{B}\ \mathsf{list})\ \mathsf{RE}$. However, the abstraction for atoms introduced in Section 4.4 caters for a more efficient encoding of formulas. We define the data type $\alpha\ \mathsf{atom}$ as

$$atm = \mathsf{A}\ a\_bs \mid \mathsf{AQ}\ m\ a \mid \mathsf{ANth}\ m\ b \mid \mathsf{ANth}_2\ m\ M$$

where $atm :: \alpha\ \mathsf{atom}$, $a\_bs :: \alpha \times \mathbb{B}\ \mathsf{list}$, $m, m_1, m_2, M :: \mathbb{N}$, $a :: \alpha$, and $b :: \mathbb{B}$. Each constructor of $\alpha\ \mathsf{atom}$ represents a set of elements of type $\alpha \times \mathbb{B}\ \mathsf{list}$. The constructor A represents the singleton set containing the constructor's argument, $\mathsf{AQ}\ m\ a$ encodes all pairs whose first element is a and whose second element (a Boolean vector) has $\top$ at index m. Both, this informal description as well as the constructor name should indicate that $\mathsf{AQ}\ m\ a$ is closely related to the formula $\mathsf{Q}\ a\ m$. The remaining two constructors have a similar purpose, being related to the other base cases of the formula type. Let us make this precise by instantiating the two parameters $\mathsf{mem}^{\mathsf{A}} :: \alpha\ \mathsf{atom} \to \alpha \times \mathbb{B}\ \mathsf{list} \to \mathbb{B}$ and $\mathsf{wf}^{\mathsf{A}} :: \mathbb{N} \to \alpha\ \mathsf{atom} \to \mathbb{B}$.

$$\begin{array}{llll}
\mathsf{mem}^{\mathsf{A}}\ (\mathsf{A}\ a_bs)\ a_bs' & \leftrightarrow a_bs = a_bs' & \mathsf{wf}^{\mathsf{A}}\ n\ (\mathsf{A}\ a_bs) & \leftrightarrow a_bs \in \Sigma_n \\
\mathsf{mem}^{\mathsf{A}}\ (\mathsf{AQ}\ m\ a)\ (a', bs) & \leftrightarrow a = a' \wedge bs[m] & \mathsf{wf}^{\mathsf{A}}\ n\ (\mathsf{AQ}\ m\ a) & \leftrightarrow a \in \Sigma \wedge m < n \\
\mathsf{mem}^{\mathsf{A}}\ (\mathsf{ANth}\ m\ b)\ (_, bs) & \leftrightarrow bs[m] = b & \mathsf{wf}^{\mathsf{A}}\ n\ (\mathsf{ANth}\ m\ b) & \leftrightarrow m < n \\
\mathsf{mem}^{\mathsf{A}}\ (\mathsf{ANth}_2\ m\ M)\ (_, bs) & \leftrightarrow bs[m] \wedge bs[M] & \mathsf{wf}^{\mathsf{A}}\ n\ (\mathsf{ANth}_2\ m\ M) & \leftrightarrow m < n \wedge M < n
\end{array}$$

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

$\mathsf{mkRE}^{\mathsf{M2L}} :: \mathbb{N} \to \alpha\ \Phi \to (\alpha\ \mathsf{atom})\ \mathsf{RE}$.

$$\begin{array}{lcl}
\mathsf{mkRE}_n^{\mathsf{M2L}}(\mathsf{Q}\ a\ m) & = & \neg\,\mathbf{0} \cdot \mathsf{AQ}\ m\ a \cdot \neg\,\mathbf{0} \\
\mathsf{mkRE}_n^{\mathsf{M2L}}(m_1 < m_2) & = & \neg\,\mathbf{0} \cdot \mathsf{ANth}\ m_1\ \top \cdot \neg\,\mathbf{0} \cdot \mathsf{ANth}\ m_2\ \top \cdot \neg\,\mathbf{0} \\
\mathsf{mkRE}_n^{\mathsf{M2L}}(m \in M) & = & \neg\,\mathbf{0} \cdot \mathsf{ANth}_2\ m\ M \cdot \neg\,\mathbf{0} \\
\mathsf{mkRE}_n^{\mathsf{M2L}}(\varphi \wedge \psi) & = & \mathsf{mkRE}_n^{\mathsf{M2L}}(\varphi) \cap \mathsf{mkRE}_n^{\mathsf{M2L}}(\psi) \\
\mathsf{mkRE}_n^{\mathsf{M2L}}(\varphi \vee \psi) & = & \mathsf{mkRE}_n^{\mathsf{M2L}}(\varphi) + \mathsf{mkRE}_n^{\mathsf{M2L}}(\psi) \\
\mathsf{mkRE}_n^{\mathsf{M2L}}(\neg\,\varphi) & = & \neg\,\mathsf{mkRE}_n^{\mathsf{M2L}}(\varphi) \\
\mathsf{mkRE}_n^{\mathsf{M2L}}(\exists\,\varphi) & = & \Pi\left(\mathsf{mkRE}_{n+1}^{\mathsf{M2L}}(\varphi) \cap \mathsf{WF}_{n+1}\{0\}\right) \\
\mathsf{mkRE}_n^{\mathsf{M2L}}(\mathbf{\exists}\,\varphi) & = & \Pi\left(\mathsf{mkRE}_{n+1}^{\mathsf{M2L}}(\varphi)\right)
\end{array}$$

At first, we ignore the function WF that is used in the case of the first-order quantifier. The natural number parameter of $\mathsf{mkRE}^{\mathsf{M2L}}$ indicates the number for free variables for the processed formula. The parameter is increased when entering recursively the scope of an existential quantifier.

The intuition behind the translation is demonstrated by the case $\mathsf{Q}\ a\ m$. We fix a well-formed model $(w, \mathscr{I})$ of $\mathsf{Q}\ a\ m$. This model must satisfy $w[\mathscr{I}[m]] = a$, or equivalently the fact that there exists a Boolean vector bs of length n such that $\mathsf{enc}^{\mathsf{M2L}}(w, \mathscr{I})[\mathscr{I}[m]] = (a, bs)$ and $bs[m] = \top$. Therefore, the letter at position $\mathscr{I}[m]$ of $\mathsf{enc}^{\mathsf{M2L}}(w, \mathscr{I})$ is matched by the "middle" part $\mathsf{AQ}\ m\ a$ of $\mathsf{mkRE}_n^{\mathsf{M2L}}(\mathsf{Q}\ a\ m)$, while the subexpressions $\neg\,\mathbf{0}$ (whose language is Σ_n^*) match the first $\mathscr{I}[m]$ and the last $n - \mathscr{I}[m]$ letters of $\mathsf{enc}^{\mathsf{M2L}}(w, \mathscr{I})$.

Conversely, if we fix a word from $\mathsf{mkRE}_n^{\mathsf{M2L}}(\mathsf{Q}\ a\ m)$, it will be equal to an encoding of an interpretation that satisfies $\mathsf{Q}\ a\ m$ by a similar argument. However, the interpretation might be not wellformed for $\mathsf{Q}\ a\ m$. This happens because the regular expression $\mathsf{mkRE}_n^{\mathsf{M2L}}(\mathsf{Q}\ a\ m)$ 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 $\neg\,\mathbf{0}$ 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 $\mathsf{WF} :: \mathbb{N} \to \mathbb{N}\ \mathsf{set} \to (\alpha\ \mathsf{atom})\ \mathsf{RE}$ function.

$$\mathsf{WF}_n(X) \;=\; \bigcap_{m \in X} (\mathsf{ANth}\ m\ \bot)^* \cdot \mathsf{ANth}\ m\ \top \cdot (\mathsf{ANth}\ m\ \bot)^*$$

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

Lemma 10
Let φ be an n-wellformed formula. Then

- $\mathscr{L}_n(\mathsf{WF}_n(\mathscr{V}_1(\varphi))) \setminus \{[]\} = \{\mathsf{enc}^{\mathsf{M2L}}(w, \mathscr{I}) \mid \mathsf{wf}_\varphi^{\mathsf{M2L}}(w, \mathscr{I}) \wedge |\mathscr{I}| = n\}$, and
- $\mathscr{L}_n(\mathsf{WF}_n(\mathscr{V}_1(\varphi))) = \{\mathsf{enc}^{\mathsf{WS1S}}(w, \mathscr{I}) \mid \mathsf{wf}_\varphi^{\mathsf{WS1S}}(w, \mathscr{I}) \wedge |\mathscr{I}| = n\}$.

Using WF in every case of the recursive definition of $\mathsf{mkRE}^{\mathsf{M2L}}$ 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.

MSO formulas interpreted in WS1S are translated into regular expressions by means of the function $\mathsf{mkRE}^{\mathsf{WS1S}} :: \mathbb{N} \to \alpha\ \Phi \to (\alpha\ \mathsf{atom})\ \mathsf{RE}$.

The definition of $\mathsf{mkRE}^{\mathsf{WS1S}}$ coincides with the one of $\mathsf{mkRE}^{\mathsf{M2L}}$ except for the existential quantifier cases:

$$\mathsf{mkRE}_n^{\mathsf{WS1S}}(\exists\, \varphi) = \mathscr{Q}\ (\mathsf{z}, \bot^n)\ \left(\Pi\left(\mathsf{mkRE}_{n+1}^{\mathsf{WS1S}}(\varphi) \cap \mathsf{WF}_{n+1}\{0\}\right)\right)$$
$$\mathsf{mkRE}_n^{\mathsf{WS1S}}(\mathbf{\exists}\, \varphi) = \mathscr{Q}\ (\mathsf{z}, \bot^n)\ \left(\Pi\left(\mathsf{mkRE}_{n+1}^{\mathsf{WS1S}}(\varphi)\right)\right)$$

The regular operation $\mathscr{Q} :: \alpha \times \mathbb{B}\ \mathsf{list} \to (\alpha\ \mathsf{atom})\ \mathsf{RE} \to (\alpha\ \mathsf{atom})\ \mathsf{RE}$ reestablishes the invariant of having all words terminated with a suffix $(\mathsf{z}, \bot^n)^m$ for every $m :: \mathbb{N}$ in the WS1S language encoding of a formula as required by definition of $\mathsf{enc}^{\mathsf{WS1S}}$ (this invariant might be violated by the projection). More precisely, the following language identity holds for an n-wellformed regular expression r:

$$\mathscr{L}_n(\mathscr{Q}\ a\ r) = \left\{ xa^m \mid \exists l.\ xa^l \in \mathscr{L}_n(r) \right\}$$

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

$$\begin{array}{lll}
\mathscr{D}_b^{\leftarrow}(\mathbf{0}) & = \mathbf{0} & \mathscr{D}_b^{\leftarrow}(\mathbf{1}) = \mathbf{0} \\
\mathscr{D}_b^{\leftarrow}(a) & = \text{if } a = b \text{ then } \mathbf{1} \text{ else } \mathbf{0} & \mathscr{D}_b^{\leftarrow}(r+s) = \mathscr{D}_b^{\leftarrow}(r) + \mathscr{D}_b^{\leftarrow}(s) \\
\mathscr{D}_b^{\leftarrow}(r \cdot s) & = & \mathscr{D}_b^{\leftarrow}(r^*) = r^* \cdot \mathscr{D}_b^{\leftarrow}(r) \\
\quad \text{if } \varepsilon(s) \text{ then } r \cdot \mathscr{D}_b^{\leftarrow}(s) + \mathscr{D}_b^{\leftarrow}(r) & & \\
\quad \text{else } r \cdot \mathscr{D}_b^{\leftarrow}(s) & & \\
\mathscr{D}_b^{\leftarrow}(r \cap s) = \mathscr{D}_b^{\leftarrow}(r) \cap \mathscr{D}_b^{\leftarrow}(s) & & \mathscr{D}_b^{\leftarrow}(\neg\, r) = \neg\, \mathscr{D}_b^{\leftarrow}(r) \\
\mathscr{D}_b^{\leftarrow}(\Pi\, r) & = \Pi\left(\bigoplus_{c \in \pi^{-} b} \mathscr{D}_c^{\leftarrow}(r)\right) &
\end{array}$$

Repeated derivation is implemented using the while combinator. The state over which the combinator iterates is of type $\mathbb{B} \times \alpha\ \mathsf{RE}\ \mathsf{list}$. The Boolean component simply indicates whether the loop should be executed once more, while the list contains all the derivatives computed so far in reversed order (i.e. 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 dual of Theorem 2 for ACI-equivalent "right derivatives". After exiting the while loop, the operation $\mathscr{Q}$ unions all the expressions computed so far, yielding an operation whose language is $\left\{ x \mid \exists l.\ xa^l \in \mathscr{L}_n(r) \right\}$. To obtain the desired semantics the

iteration of a (lifted to the α atom type by A) is concatenated to the union.

$$
\begin{aligned}
&\mathsf{b}^{\leftarrow}\ (\mathit{continue}, \_) = \mathit{continue}\\
&\mathsf{c}_a^{\leftarrow}\ (\_, rs)\ =\\
&\quad \text{let } s = \langle \mathscr{D}_a^{\leftarrow}\ (\mathsf{head}\ rs)\rangle\\
&\quad \text{in if } s \in \mathsf{set}\ rs \text{ then } (\bot, rs) \text{ else } (\top, s\ \#\ rs)\\
&\\
&\mathscr{Q}\ a\ r\ =\\
&\quad \text{let } R = \mathsf{case\ while\ b}^{\leftarrow}\ \mathsf{c}_a^{\leftarrow}\ (\top, \langle r\rangle) \text{ of } \mathsf{Some}\ (\_, rs) \Rightarrow \mathsf{set}\ rs\\
&\quad \text{in } \left(\bigoplus_{r\in R} r\right) \cdot (\mathsf{A}\ a)^*
\end{aligned}
$$

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

Theorem 11
Let φ be an n-wellformed formula. Then

- $\mathscr{L}_n^{\mathsf{M2L}}(\varphi) = \mathscr{L}_n(\mathsf{mkRE}_n^{\mathsf{M2L}}(\varphi) \cap \mathsf{WF}_n(\varphi)) \setminus \{[]\}$, and
- $\mathscr{L}_n^{\mathsf{WS1S}}(\varphi) = \mathscr{L}_n(\mathsf{mkRE}_n^{\mathsf{WS1S}}(\varphi) \cap \mathsf{WF}_n(\varphi))$.

The proof is by structural induction on φ. Above we have seen the argument for the base case $\mathsf{Q}\ a\ m$, other base cases follow similarly. The cases $\exists\,\varphi$ and $\mathsf{\exists}\,\varphi$ 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 $\mathsf{enc}^{\mathsf{M2L}}$ (and $\mathsf{enc}^{\mathsf{WS1S}}$) do 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 (sets of words). This is again established by structural induction on formulas for both semantics.

Lemma 12
Let $(w_1, \mathscr{I}_1)$ and $(w_2, \mathscr{I}_2)$ be two M2L-wellformed interpretations for a formula φ such that $\mathsf{enc}^{\mathsf{M2L}}(w_1, \mathscr{I}_1) = \mathsf{enc}^{\mathsf{M2L}}(w_2, \mathscr{I}_2)$. Then $(w_1, \mathscr{I}_1) \models \varphi \leftrightarrow (w_2, \mathscr{I}_2) \models \varphi$.
Let $(w_1, \mathscr{I}_1)$ and $(w_2, \mathscr{I}_2)$ be two WS1S-wellformed interpretations for a formula φ such that $\mathsf{enc}^{\mathsf{WS1S}}(w_1, \mathscr{I}_1) = \mathsf{enc}^{\mathsf{WS1S}}(w_2, \mathscr{I}_2)$. Then $(w_1, \mathscr{I}_1) \models^{\mathsf{o}} \varphi \leftrightarrow (w_2, \mathscr{I}_2) \models^{\mathsf{o}} \varphi$.

5.5 Deciding Language Equivalence of Formulas

The algorithms $\mathsf{eqv}^{\mathsf{M2L}} :: \mathbb{N} \to \alpha\ \Phi \to \alpha\ \Phi \to \mathbb{B}$ and $\mathsf{eqv}^{\mathsf{WS1S}} :: \mathbb{N} \to \alpha\ \Phi \to \alpha\ \Phi \to \mathbb{B}$ that decide language equivalence of MSO formulas check wellformedness of the input formulas, translate the formulas into regular expressions and let $\mathsf{eqv}^{\mathsf{RE}}$ do the work:

$$\mathsf{eqv}_n^{\mathsf{M2L}}\,\varphi\,\psi\ = \mathsf{wf}_n^{\Phi}(\varphi \vee \psi) \wedge \mathsf{eqv}_n^{\mathsf{RE}}\,(\mathsf{mkRE}_n^{\mathsf{M2L}}(\varphi) + \mathbf{1})\,(\mathsf{mkRE}_n^{\mathsf{M2L}}(\psi) + \mathbf{1})$$

$$\mathsf{eqv}_n^{\mathsf{WS1S}}\,\varphi\,\psi\ = \mathsf{wf}_n^{\Phi}(\varphi \vee \psi) \wedge \mathsf{eqv}_n^{\mathsf{RE}}\,(\mathsf{mkRE}_n^{\mathsf{WS1S}}(\varphi))\,(\mathsf{mkRE}_n^{\mathsf{WS1S}}(\psi))$$

Note that wellformedness is checked on the disjunction of both formulas to ensure that they agree on free variables (i.e. no first-order free variable of φ is used as a second-order free

variable in ψ and vice versa). Further, 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 Theorem 11 does not give us any information whether the empty word is contained in the output of $\mathsf{mkRE}^{\mathsf{M2L}}$ or not.

Termination of $\mathsf{eqv}^{\mathsf{RE}}$ is ensured by Theorem 6 and the definition principle of primitive recursion for wf^{Φ}, $\mathsf{mkRE}^{\mathsf{M2L}}$ and $\mathsf{mkRE}^{\mathsf{WS1S}}$. Soundness and completeness follow easily from Theorems 7, 8 and 11.

Theorem 13 (*Soundness*)
Let φ and ψ be MSO formulas.

- If $\mathsf{eqv}_n^{\mathsf{M2L}}\,\varphi\,\psi$, then $\mathscr{L}_n^{\mathsf{M2L}}(\varphi) = \mathscr{L}_n^{\mathsf{M2L}}(\psi)$.
- If $\mathsf{eqv}_n^{\mathsf{WS1S}}\,\varphi\,\psi$, then $\mathscr{L}_n^{\mathsf{WS1S}}(\varphi) = \mathscr{L}_n^{\mathsf{WS1S}}(\psi)$.

Theorem 14 (*Completeness*)
Let $\varphi \vee \psi$ be an n-wellformed MSO formula.

- If $\mathscr{L}_n^{\mathsf{M2L}}(\varphi) = \mathscr{L}_n^{\mathsf{M2L}}(\psi)$, then $\mathsf{eqv}_n^{\mathsf{M2L}}\,\varphi\,\psi$.
- If $\mathscr{L}_n^{\mathsf{WS1S}}(\varphi) = \mathscr{L}_n^{\mathsf{WS1S}}(\psi)$, then $\mathsf{eqv}_n^{\mathsf{WS1S}}\,\varphi\,\psi$.

As a sanity check let us apply our translation for M2L to the formula $\varphi = \mathsf{\exists}\ (\forall\ 0 \in 1)$ (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 $\Sigma_0 = \Sigma \times \mathbb{B}^0$ for some base alphabet Σ. E.g. we can take $\Sigma = \{a\}$ and write the unique element of Σ_0 as $\hat{a}$. The function $\langle\!\langle \mathsf{mkRE}_0^{\mathsf{M2L}}(\varphi) \rangle\!\rangle$ translates φ to the accepting Π-extended regular expression $\Pi\,\mathsf{r}$ over Σ_0 where r is an abbreviation:

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

Derivatives of $\Pi\,\mathsf{r}$ by words of the form $\hat{a}^n$ for $n > 0$ are all $\approx$-equivalent to a single (also accepting) expression. More precisely, for all $w \in \Sigma_0^* \setminus \{[]\}$ we have:

$$\mathscr{D}_w^*(\Pi\,\mathsf{r}) = \Pi\,\mathsf{r} + \Pi\,(\mathsf{r} \cap \neg\,\Pi\,((\mathsf{ANth}\ 0\ \bot^*) \cap \neg\,(\neg\,\mathbf{0} \cdot \mathsf{ANth}_2\ 0\ 1 \cdot \neg\,\mathbf{0})))$$

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).

6 Application: 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.

In order to create larger formulas, it is helpful to introduce some syntactic abbreviations. We define the unsatisfiable formula $\bot$ as $\exists\,0 < 0$ and the valid formula $\top$ as $\neg\,\bot$. Now, checking that a formula is valid amounts to checking its equivalence to $\top$. We call the function that performs this check Thm. Implication $\varphi \to \psi$ is defined as $(\neg\,\varphi) \vee \psi$ and universal quantification $\forall\,\varphi$ as before as $\neg\,\exists\,\neg\,\varphi$. Next, we introduce temporal logical operators *always* $\Box P :: \mathbb{N} \to \alpha\ \Phi$ and *eventually* $\Diamond P :: \mathbb{N} \to \alpha\ \Phi$ depending on $P :: \mathbb{N} \to \alpha\ \Phi$—a formula parameterized by a single variable 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.

$$\begin{array}{lcl} \Box P\,t & = & \forall\,(\neg\,t+1<0\rightarrow P\,0) \\ \Diamond P\,t & = & \exists\,(\neg\,t+1<0\wedge P\,0) \\ (P\Rightarrow Q)\,t & = & P\,t\rightarrow Q\,t \\ (P\mathbin{\textcircled{\scriptsize$\vee$}}Q)\,t & = & P\,t\vee Q\,t \end{array}$$

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.

Formulas of linear temporal logic also contain atomic predicates for which the interpretation must specify at which points in time they are true. This information can be encoded in two ways, which we compare in the following.

The first possibility is to encode atomic predicates in the word of the interpretation. This is done by identifying Σ with the powerset $\mathscr{P}$ of atomic predicates. For every point in time, that is for every position in the word, the letter is the set of predicates that are true at this point. Using this encoding we can prove the validity of the following closed formulas over the alphabet $\mathscr{P}\{P\} = \{\{P\},\{\}\}$ automatically within a few milliseconds.

$$\forall\,(\Box(\mathsf{Q}\,\{P\})\Rightarrow\Diamond(\mathsf{Q}\,\{P\}))\;0$$
$$\forall\,(\Box(\mathsf{Q}\,\{P\})\Rightarrow\Box\Diamond(\mathsf{Q}\,\{P\}))\;0$$

Alternatively, a free second-order variable can be used to encode an atomic predicate directly. The variable denotes the set of points in time for which the atomic predicate holds. The alphabet Σ can then be trivial, i.e. $\Sigma = \{a\}$ for an arbitrary a. Using this encoding the above two formulas correspond to

$$\forall\,(\Box(\lambda t.\,t\in 2)\Rightarrow\Diamond(\lambda t.\,t\in 2))\;0$$
$$\forall\,(\Box(\lambda t.\,t\in 2)\Rightarrow\Box\Diamond(\lambda t.\,t\in 3))\;0$$

Both formulas have one free second-order variable 0 that is lifted when passing two or three quantifiers. The generated algorithm shows the equivalence to $\top$ again within milliseconds.

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 implication for both representations of atomic predicates as shown in Figure 2. When the number of predicates n is increased, the size of φ_n grows exponentially: to express that a predicate P holds at some position we need the disjunction of all atoms containing P. In contrast, the size of ψ_n grows linearly. The complexity of ψ_n is hidden in the number of variables and therefore in its encoding—the latter also grows exponentially with increasing n.

Both, φ_i and ψ_i are theorems under both semantics. The running times of the decision procedure Thm in seconds are summarized in Figures 3 and 4 (column Thm, first number refers to the M2L semantics, the second to WS1S). Thereby, ψ_1, ψ_2 and ψ_3 were processed over $\Sigma = \{a\}$, φ_1 was processed over $\Sigma = \mathscr{P}\{P\}$, φ_2 over $\Sigma = \mathscr{P}\{P_1,P_2\}$ and finally φ_3 over $\Sigma = \mathscr{P}\{P_1,P_2,P_3\}$. The column "ICFP 2013" recapitulates the running times from the earlier unoptimized version of this procedure (Traytel & Nipkow, 2013).

$$\varphi_1 = \forall\,(\Box(\mathsf{Q}\,\{P\}) \Rightarrow \Box(\mathsf{Q}\,\{P\}))\ 0$$
$$\varphi_2 = \forall\,(\Box(\mathsf{Q}\,\{P_1\} \mathbin{\textcircled{\scriptsize V}} \mathsf{Q}\,\{P_1,P_2\} \Rightarrow \mathsf{Q}\,\{P_2\} \mathbin{\textcircled{\scriptsize V}} \mathsf{Q}\,\{P_1,P_2\}) \Rightarrow \Box(\mathsf{Q}\,\{P_1\} \mathbin{\textcircled{\scriptsize V}} \mathsf{Q}\,\{P_1,P_2\}) \Rightarrow \Box(\mathsf{Q}\,\{P_2\} \mathbin{\textcircled{\scriptsize V}} \mathsf{Q}\,\{P_1,P_2\}))\ 0$$
$$\begin{aligned}\varphi_3 = \forall\,(&\Box(\mathsf{Q}\,\{P_1\} \mathbin{\textcircled{\scriptsize V}} \mathsf{Q}\,\{P_1,P_2\} \mathbin{\textcircled{\scriptsize V}} \mathsf{Q}\,\{P_1,P_3\} \mathbin{\textcircled{\scriptsize V}} \mathsf{Q}\,\{P_1,P_2,P_3\} \Rightarrow\\ &\mathsf{Q}\,\{P_2\} \mathbin{\textcircled{\scriptsize V}} \mathsf{Q}\,\{P_1,P_2\} \mathbin{\textcircled{\scriptsize V}} \mathsf{Q}\,\{P_2,P_3\} \mathbin{\textcircled{\scriptsize V}} \mathsf{Q}\,\{P_1,P_2,P_3\} \Rightarrow\\ &\mathsf{Q}\,\{P_3\} \mathbin{\textcircled{\scriptsize V}} \mathsf{Q}\,\{P_1,P_3\} \mathbin{\textcircled{\scriptsize V}} \mathsf{Q}\,\{P_2,P_3\} \mathbin{\textcircled{\scriptsize V}} \mathsf{Q}\,\{P_1,P_2,P_3\}) \Rightarrow\\ &\Box(\mathsf{Q}\,\{P_1\} \mathbin{\textcircled{\scriptsize V}} \mathsf{Q}\,\{P_1,P_2\} \mathbin{\textcircled{\scriptsize V}} \mathsf{Q}\,\{P_1,P_3\} \mathbin{\textcircled{\scriptsize V}} \mathsf{Q}\,\{P_1,P_2,P_3\}) \Rightarrow\\ &\Box(\mathsf{Q}\,\{P_2\} \mathbin{\textcircled{\scriptsize V}} \mathsf{Q}\,\{P_1,P_2\} \mathbin{\textcircled{\scriptsize V}} \mathsf{Q}\,\{P_2,P_3\} \mathbin{\textcircled{\scriptsize V}} \mathsf{Q}\,\{P_1,P_2,P_3\}) \Rightarrow\\ &\Box(\mathsf{Q}\,\{P_3\} \mathbin{\textcircled{\scriptsize V}} \mathsf{Q}\,\{P_1,P_3\} \mathbin{\textcircled{\scriptsize V}} \mathsf{Q}\,\{P_2,P_3\} \mathbin{\textcircled{\scriptsize V}} \mathsf{Q}\,\{P_1,P_2,P_3\}))\ 0\end{aligned}$$
$$\psi_1 = \forall\,(\Box(\lambda t.t \in 2) \Rightarrow \Box(\lambda t.t \in 2))\ 0$$
$$\psi_2 = \forall\,(\Box(\lambda t.t \in 2 \to t \in 3) \Rightarrow \Box(\lambda t.t \in 2) \Rightarrow \Box(\lambda t.t \in 3))\ 0$$
$$\psi_3 = \forall\,(\Box(\lambda t.t \in 2 \to t \in 3 \to t \in 4) \Rightarrow \Box(\lambda t.t \in 2) \Rightarrow \Box(\lambda t.t \in 3) \Rightarrow \Box(\lambda t.t \in 4))\ 0$$

Fig. 2. Definition of φ_n and ψ_n

n	size	ICFP 2013	Thm	$\mathsf{Thm}_{\mathsf{interm}}$	$\mathsf{Thm}_{\mathsf{dual}}$
1	54/ 31907	0/–	0/230.9	0/ 17.1	0/ 0.4
2	100/ 60045	2/–	0.7/ –	0/711.6	0/21.1
3	192/114857	4860/–	964.4/ –	30.4/ –	31.7/ –

Fig. 3. Benchmarks for φ_n (under M2L/WS1S semantics)

n	size	ICFP 2013	Thm	$\mathsf{Thm}_{\mathsf{interm}}$	$\mathsf{Thm}_{\mathsf{dual}}$
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/ –

Fig. 4. Benchmarks for ψ_n (under M2L/WS1S semantics)

Figures 3 and 4 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 $\mathscr{Q}$ 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: $\mathsf{Thm}_{\mathsf{interm}}$ and $\mathsf{Thm}_{\mathsf{dual}}$. One remaining source of inefficiency in Thm is the fact that, although it constructs a bisimulation modulo $\langle\!\langle - \rangle\!\rangle$, the intermediate expressions on which the derivatives are computed are only ACI-normalized (i.e. they might contain redundant subexpressions like $\mathbf{0} \cap r$ and the derivative would needlessly recurse in r). The algorithm $\mathsf{Thm}_{\mathsf{interm}}$ addresses this inefficiency by normalizing the intermediate expressions with $\langle\!\langle - \rangle\!\rangle$. This intermediate normalization might seem harmless, but it is not clear anymore that the number of derivatives interspersed with normalization is finite[3]. We were not able to prove finiteness of such derivatives interspersed with $\langle\!\langle - \rangle\!\rangle$ (although we conjecture that it holds). However, we have proved that under the condition that $\mathsf{Thm}_{\mathsf{interm}}$ terminates, its output—namely $\top$ if the input formula is valid, $\bot$ otherwise—is correct.

[3] For example, using the terminating normalization function that does the same ACI simplifications as $\langle - \rangle$, but additionally soundly rewrites $\mathbf{1} \cdot a^*$ 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^*.

The algorithm $\mathsf{Thm}_{\mathsf{dual}}$ is similar to $\mathsf{Thm}_{\mathsf{interm}}$ in the respect that it normalizes intermediate expressions. Therefore, we again guarantee only partial correctness. Unlike $\mathsf{Thm}_{\mathsf{interm}}$, $\mathsf{Thm}_{\mathsf{dual}}$ works with dual regular expressions (Section 4.5). It 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 (Haftmann *et al.*, 2013). 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 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.

7 Conclusion

We have presented functional programs that decide equivalence of MSO formulas for two different semantics in Isabelle/HOL. They come with formal proofs of termination, soundness and completeness. The programs operate by translating formulas into Π-extended regular expressions and deciding the language equivalence of the latter using Brzozowski derivatives. Although formalized in Isabelle/HOL's functional programming language, we can automatically generate code from them in different functional target languages. The development amounts to roughly 500 lines of functional programs and 6500 lines of proofs, of which 3000 lines are devoted to deciding equivalence of Π-extended regular expressions. The functional programs are completely contained in this paper. The Isabelle scripts are publicly available (Traytel & Nipkow, 2014).

Our work can be continued in two dimensions. First, our algorithm still offers much room for optimization. Especially, the inefficient formalization of $\mathscr{Q}$ should be revised. Second, several related decidable logics can be formalized and verified using similar technology. A related logic is MSO on infinite words (also called S1S). S1S formulas can be translated into ω-regular expressions representing ω-regular languages. A verified decision procedure for deciding equivalence of ω-regular expressions without constructing ω-automata is an interesting challenge. A similarly ambitious goal is to move from words to trees (or even from ω-words to ω-trees) and decide equivalence of MSO formulas on (in)finite trees (or alternatively (W)S2S formulas) by translating them into (ω-)regular tree expressions.

Acknowledgments We thank Alexander Krauss for inspiring discussions, Jasmin Blanchette for numerous comments on the presentation, and several anonymous reviewers for a wealth of comments and questions that helped to clarify certain fine points. While carrying out this work, Traytel was affiliated with TU München and supported by the doctorate program 1480 (PUMA) of the Deutsche Forschungsgemeinschaft (DFG).

References

Antimirov, Valentin. (1996). Partial derivatives of regular expressions and finite automaton constructions. *Theor. Comput. Sci.*, **155**(2), 291–319.

Asperti, Andrea. (2012). A compact proof of decidability for regular expression equivalence. *Pages 283–298 of:* Beringer, L., & Felty, A. (eds), *Proc. Int. Conf. Interactive Theorem Proving, ITP 2012*. Lect. Notes Comput. Sci., vol. 7406. Springer.

Ayari, Abdelwaheb, & Basin, David. (2000). Bounded model construction for monadic second-order logics. *Pages 99–112 of:* Emerson, E. A., & Sistla, A. P. (eds), *Proc. Int. Conf. Computer Aided Verification, CAV 2000*. Lect. Notes Comput. Sci., vol. 1855. Springer.

Ballarin, Clemens. (2006). Interpretation of locales in Isabelle: Theories and proof contexts. *Pages 31–43 of:* Borwein, J. M., & Farmer, W. M. (eds), *Proc. Int. Conf. Mathematical Knowledge Management, MKM 2006*. Lect. Notes Comput. Sci., vol. 4108. Springer.

Basin, David, & Friedrich, Stefan. (2000). Combining WS1S and HOL. *Pages 39–56 of:* Gabbay, D.M., & de Rijke, M. (eds), *Frontiers of Combining Systems 2*. Studies in Logic and Computation, vol. 7. Research Studies Press.

Berghofer, Stefan, & Reiter, Markus. (2009). Formalizing the logic-automaton connection. *Pages 147–163 of:* Berghofer, S., Nipkow, T., Urban, C., & Wenzel, M. (eds), *Proc. Int. Conf. Theorem Proving in Higher Order Logics, TPHOLs 2009*. Lect. Notes Comput. Sci., vol. 5674. Springer.

Bonchi, Filippo, & Pous, Damien. (2013). Checking NFA equivalence with bisimulations up to congruence. *Pages 457–468 of:* Giacobazzi, Roberto, & Cousot, Radhia (eds), *Proc. Int. Symp. principles of programming languages, POPL 2013*. ACM.

Braibant, Thomas, & Pous, Damien. (2010). An efficient Coq tactic for deciding Kleene algebras. *Pages 163–178 of:* Kaufmann, M., & Paulson, L. (eds), *Proc. Int. Conf. Interactive Theorem Proving, ITP 2010*. Lect. Notes Comput. Sci., vol. 6172. Springer.

Brzozowski, Janusz A. (1964). Derivatives of regular expressions. *J. ACM*, **11**(4), 481–494.

Caron, Pascal, Champarnaud, Jean-Marc, & Mignot, Ludovic. (2011). Partial derivatives of an extended regular expression. *Pages 179–191 of:* Dediu, A.-H., Inenaga, S., & Martín-Vide, C. (eds), *Proc. Int. Conf. Language and Automata Theory and Applications, LATA 2011*. Lect. Notes Comput. Sci., vol. 6638. Springer.

Coquand, Thierry, & Siles, Vincent. (2011). A decision procedure for regular expression equivalence in type theory. *Pages 119–134 of:* Jouannaud, J.-P., & Shao, Z. (eds), *Proc. Int. Conf. Certified Programs and Proofs, CPP 2011*. Lect. Notes Comput. Sci., vol. 7086. Springer.

Danielsson, Nils Anders. (2010). Total parser combinators. *Pages 285–296 of:* Hudak, P., & Weirich, S. (eds), *Proc. Int. Conf. Functional Programming, ICFP 2010*. ACM.

Elgaard, Jacob, Klarlund, Nils, & Møller, Anders. (1998). MONA 1.x: new techniques for WS1S and WS2S. *Pages 516–520 of:* Hu, A. J., & Vardi, M. Y. (eds), *Proc. Int. Conf. Computer Aided Verification, CAV 1998*. Lect. Notes Comput. Sci., vol. 1427. Springer.

Fischer, Sebastian, Huch, Frank, & Wilke, Thomas. (2010). A play on regular expressions: functional pearl. *Pages 357–368 of:* Hudak, P., & Weirich, S. (eds), *Proc. Int. Conf. Functional Programming, ICFP 2010*. ACM.

Ginzburg, A. (1967). A procedure for checking equality of regular expressions. *J. ACM*, **14**(2), 355–362.

Haftmann, Florian, & Nipkow, Tobias. (2010). Code generation via higher-order rewrite systems. *Pages 103–117 of:* Blume, M., Kobayashi, N., & Vidal, G. (eds), *Proc. Int. Symp. Functional and Logic Programming, FLOPS 2010*. Lect. Notes Comput. Sci., vol. 6009. Springer.

Haftmann, Florian, Krauss, Alexander, Kunčar, Ondřej, & Nipkow, Tobias. (2013). Data refinement in Isabelle/HOL. *Pages 100–115 of:* Blazy, S., Paulin-Mohring, C., & Pichardie, D. (eds), *Proc. Int. Conf. Interactive Theorem Proving, ITP 2013*. Lect. Notes Comput. Sci., vol. 7998. Springer.

Henriksen, Jesper G., Jensen, Jakob L., Jørgensen, Michael E., Klarlund, Nils, Paige, Robert, Rauhe, Theis, & Sandholm, Anders. (1995). MONA: Monadic second-order logic in practice. *Pages 89–110 of:* Brinksma, E., Cleaveland, R., Larsen, K., Margaria, T., & Steffen, B. (eds), *Proc. Int. Workshop Tools and Algorithms for the Construction and Analysis of Systems, TACAS 1995*. Lect. Notes Comput. Sci., vol. 1019. Springer.

Klarlund, Nils. (1999). A theory of restrictions for logics and automata. *Pages 406–417 of:* Halbwachs, N., & Peled, D. (eds), *Proc. Int. Conf. Computer Aided Verification, CAV 1999*. Lect. Notes Comput. Sci., vol. 1633. Springer.

Kozen, Dexter. 2008 (March). *On the coalgebraic theory of Kleene algebra with tests*. Tech. rept. `http://hdl.handle.net/1813/10173`. Computing and Information Science, Cornell University.

Krauss, Alexander, & Nipkow, Tobias. (2012). Proof pearl: Regular expression equivalence and relation algebra. *J. Automated Reasoning*, **49**(1), 95–106. published online March 2011.

Might, Matthew, Darais, David, & Spiewak, Daniel. (2011). Parsing with derivatives: A functional pearl. *Pages 189–195 of:* Chakravarty, M. M. T., Hu, Z., & Danvy, O. (eds), *Proc. Int. Conf. Functional Programming, ICFP 2011*. ACM.

Moreira, Nelma, Pereira, David, & de Sousa, Simão Melo. (2012). Deciding regular expressions (in-)equivalence in Coq. *Pages 98–113 of:* Kahl, W., & Griffin, T. (eds), *Relational and algebraic methods in computer science, RAMiCS 2012*. Lect. Notes Comput. Sci., vol. 7560. Springer.

Nipkow, Tobias, & Klein, Gerwin. (2014). *Concrete Semantics: With Isabelle/HOL*. Springer. `http://www.in.tum.de/~nipkow/Concrete-Semantics`.

Nipkow, Tobias, & Traytel, Dmitriy. (2014). Unified decision procedures for regular expression equivalence. *Pages 450–466 of:* Klein, G., & Gamboa, R. (eds), *Proc. Int. Conf. Interactive Theorem Proving, ITP 2014*. Lect. Notes Comput. Sci., vol. 8558. Springer.

Nipkow, Tobias, Paulson, Lawrence, & Wenzel, Markus. (2002). *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. Lect. Notes Comput. Sci., vol. 2283. Springer.

Okhotin, Alexander. (2005). The dual of concatenation. *Theor. Comput. Sci.*, **345**(2-3), 425–447.

Owens, Scott, Reppy, John H., & Turon, Aaron. (2009). Regular-expression derivatives re-examined. *J. Funct. Program.*, **19**(2), 173–190.

Owre, Sam, & Rueß, Harald. (2000). Integrating WS1S with PVS. *Pages 548–551 of:* Emerson, E. A., & Sistla, A. P. (eds), *Proc. Int. Conf. Computer Aided Verification, CAV 2000*. Lect. Notes Comput. Sci., vol. 1855. Springer.

Pous, Damien. (2015). Symbolic algorithms for language equivalence and Kleene algebra with test. *Pages 357–368 of:* Walker, David (ed), *Proc. Int. Symp. Principles of Programming Languages, POPL 2015*. ACM.

Rutten, Jan J. M. M. (1998). Automata and coinduction (an exercise in coalgebra). *Pages 194–218 of:* Sangiorgi, D., & de Simone, R. (eds), *Proc. Int. Conf. Concurrency Theory, CONCUR 1998*. Lect. Notes Comput. Sci., vol. 1466. Springer.

Thomas, Wolfgang. (1997). Languages, automata, and logic. *Pages 389–455 of:* Rozenberg, G., & Salomaa, A. (eds), *Handbook of Formal Languages*. Springer.

Traytel, Dmitriy, & Nipkow, Tobias. (2013). Verified decision procedures for MSO on words based on derivatives of regular expressions. *Pages 3–12 of:* Morrisett, G., & Uustalu, T. (eds), *Proc. Int. Conf. Functional Programming, ICFP 2013*. ACM.

Traytel, Dmitriy, & Nipkow, Tobias. (2014). Decision procedures for MSO on words based on derivatives of regular expressions. Klein, G., Nipkow, T., & Paulson, L. (eds), *Archive of Formal Proofs*. `http://afp.sf.net/entries/MSO_Regex_Equivalence.shtml`, Formal proof development.

Wu, Chunhan, Zhang, Xingyuan, & Urban, Christian. (2014). A formalisation of the Myhill-Nerode theorem based on regular expressions. *J. Automated Reasoning,*, **52**(4), 451–480.