Theoretical Informatics and Applications Will be set by the publisher
Informatique Théorique et Applications

A FORMALISATION OF THE MYHILL-NERODE THEOREM BASED ON REGULAR EXPRESSIONS [*]

Chunhan Wu[1], Xingyuan Zhang[1] and Christian Urban[2, 3]

Abstract. There are numerous textbooks on regular languages. Nearly all of them introduce the subject by describing finite automata and only mentioning on the side a connection with regular expressions. Unfortunately, automata are difficult to formalise in HOL-based theorem provers. The reason is that they need to be represented as graphs, matrices or functions, none of which are inductive datatypes. Also convenient operations for disjoint unions of graphs, matrices and functions are not easily formalisiable in HOL. In contrast, regular expressions can be defined conveniently as a datatype and a corresponding reasoning infrastructure comes for free. We show in this paper that a central result from formal language theory—the Myhill-Nerode Theorem—can be recreated using only regular expressions. From this theorem many closure properties of regular languages follow.

1991 Mathematics Subject Classification. 68Q45.

1. Introduction

Regular languages are an important and well-understood subject in Computer Science, with many beautiful theorems and many useful algorithms. There is a wide range of textbooks on this subject, many of which are aimed at students and contain very detailed 'pencil-and-paper' proofs (e.g. [15, 16]). It seems natural to exercise theorem provers by formalising the theorems and by verifying formally the algorithms.

A popular choice for a theorem prover would be one based on Higher-Order Logic (HOL), for example HOL4, HOLlight or Isabelle/HOL. For the development presented in

Keywords and phrases: Myhill-Nerode theorem, regular expressions, Isabelle theorem prover

[*] *This is a revised and expanded version of [26].*

[1] PLA University of Science and Technology, China

[2] TU Munich, Germany

[3] corresponding author

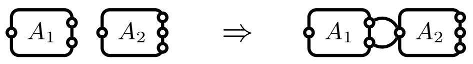
this paper we will use the Isabelle/HOL. HOL is a predicate calculus that allows quantification over predicate variables. Its type system is based on Church's Simple Theory of Types [8]. Although many mathematical concepts can be conveniently expressed in HOL, there are some limitations that hurt badly when attempting a simple-minded formalisation of regular languages in it.

The typical approach (for example [15, 16]) to regular languages is to introduce finite deterministic automata and then define everything in terms of them. For example, a regular language is normally defined as:

Definition 1.1. A language A is *regular*, provided there is a finite deterministic automaton that recognises all strings of A.

This approach has many benefits. Among them is the fact that it is easy to convince oneself that regular languages are closed under complementation: one just has to exchange the accepting and non-accepting states in the corresponding automaton to obtain an automaton for the complement language. The problem, however, lies with formalising such reasoning in a HOL-based theorem prover. Automata are built up from states and transitions that need to be represented as graphs, matrices or functions, none of which can be defined as an inductive datatype.

In case of graphs and matrices, this means we have to build our own reasoning infrastructure for them, as neither Isabelle/HOL nor HOL4 nor HOLlight support them with libraries. Even worse, reasoning about graphs and matrices can be a real hassle in HOL-based theorem provers, because we have to be able to combine automata. Consider for example the operation of sequencing two automata, say A_1 and A_2, by connecting the accepting states of A_1 to the initial state of A_2:



On 'paper' we can define the corresponding graph in terms of the disjoint union of the state nodes. Unfortunately in HOL, the standard definition for disjoint union, namely

$$A_1 \uplus A_2 \overset{def}{=} \{(1, x) \mid x \in A_1\} \cup \{(2, y) \mid y \in A_2\} \tag{1}$$

changes the type—the disjoint union is not a set, but a set of pairs. Using this definition for disjoint union means we do not have a single type for the states of automata. As a result we will not be able to define a regular language as one for which there exists an automaton that recognises all its strings (Definition 1.1). This is because we cannot make a definition in HOL that is only polymorphic in the state type, but not in the predicate for regularity; and there is no type quantification available in HOL (unlike in Coq, for example).[1]

An alternative, which provides us with a single type for states of automata, is to give every state node an identity, for example a natural number, and then be careful to rename these identities apart whenever connecting two automata. This results in clunky proofs establishing that properties are invariant under renaming. Similarly, connecting two automata represented as matrices results in very adhoc constructions, which are not pleasant to reason about.

[1]Slind already pointed out this problem in an email to the HOL4 mailing list on 21st April 2005.

Functions are much better supported in Isabelle/HOL, but they still lead to similar problems as with graphs. Composing, for example, two non-deterministic automata in parallel requires also the formalisation of disjoint unions. Nipkow [18] dismisses for this the option of using identities, because it leads according to him to "messy proofs". Since he does not need to define what regular languages are, Nipkow opts for a variant of (1) using bit lists, but writes

> *"All lemmas appear obvious given a picture of the composition of automata... Yet their proofs require a painful amount of detail."*

and

> *"If the reader finds the above treatment in terms of bit lists revoltingly concrete, I cannot disagree. A more abstract approach is clearly desirable."*

Moreover, it is not so clear how to conveniently impose a finiteness condition upon functions in order to represent *finite* automata. The best is probably to resort to more advanced reasoning frameworks, such as *locales* or *type classes*, which are *not* available in all HOL-based theorem provers.

Because of these problems to do with representing automata, there seems to be no substantial formalisation of automata theory and regular languages carried out in HOL-based theorem provers. Nipkow [18] establishes the link between regular expressions and automata in the context of lexing. Berghofer and Reiter [5] formalise automata working over bit strings in the context of Presburger arithmetic. The only larger formalisations of automata theory are carried out in Nuprl [9] and in Coq, e.g. [1, 11].

Also, one might consider automata as just convenient 'vehicles' for establishing properties about regular languages. However, paper proofs about automata often involve subtle side-conditions which are easily overlooked, but which make formal reasoning rather painful. For example Kozen's proof of the Myhill-Nerode Theorem requires that automata do not have inaccessible states [16]. Another subtle side-condition is completeness of automata, that is automata need to have total transition functions and at most one 'sink' state from which there is no connection to a final state (Brzozowski mentions this side-condition in the context of state complexity of automata [7]). Such side-conditions mean that if we define a regular language as one for which there exists *a* finite automaton that recognises all its strings (see Definition 1.1), then we need a lemma which ensures that another equivalent one can be found satisfying the side-condition, and also need to make sure our operations on automata preserve them. Unfortunately, such 'little' and 'obvious' lemmas make formalisations of automata theory hair-pulling experiences.

In this paper, we will not attempt to formalise automata theory in Isabelle/HOL nor will we attempt to formalise automata proofs from the literature, but take a different approach to regular languages than is usually taken. Instead of defining a regular language as one where there exists an automaton that recognises all its strings, we define a regular language as:

Definition 1.2. A language A is *regular*, provided there is a regular expression that matches all strings of A.

And then 'forget' automata. The reason is that regular expressions, unlike graphs, matrices and functions, can be easily defined as an inductive datatype. A reasoning infrastructure (like induction and recursion) comes for free in HOL. Moreover, no side-conditions will be needed for regular expressions, like we need for automata. This convenience of regular expressions has recently been exploited in HOL4 with a formalisation of regular expression matching based on derivatives [21] and with an equivalence checker for regular expressions in Isabelle/HOL [17]. The main purpose of this paper is to show that a central result about regular languages—the Myhill-Nerode Theorem—can be recreated by only using regular expressions. This theorem gives necessary and sufficient conditions for when a language is regular. As a corollary of this theorem we can easily establish the usual closure properties, including complementation, for regular languages. We use the Continuation Lemma [23], which is also a corollary of the Myhill-Nerode Theorem, for establishing the non-regularity of the language $a^n b^n$.

Contributions: There is an extensive literature on regular languages. To our best knowledge, our proof of the Myhill-Nerode Theorem is the first that is based on regular expressions, only. The part of this theorem stating that finitely many partitions imply regularity of the language is proved by an argument about solving equational systems. This argument appears to be folklore. For the other part, we give two proofs: one direct proof using certain tagging-functions, and another indirect proof using Antimirov's partial derivatives [2]. Again to our best knowledge, the tagging-functions have not been used before for establishing the Myhill-Nerode Theorem. Derivatives of regular expressions have been used recently quite widely in the literature; partial derivatives, in contrast, attract much less attention. However, partial derivatives are more suitable in the context of the Myhill-Nerode Theorem, since it is easier to establish formally their finiteness result. We are not aware of any proof that uses either of them for proving the Myhill-Nerode Theorem.

2. Preliminaries

Strings in Isabelle/HOL are lists of characters with the *empty string* being represented by the empty list, written $[]$. We assume there are only finitely many different characters. *Languages* are sets of strings. The language containing all strings is written in Isabelle/HOL as *UNIV*. The concatenation of two languages is written $A \cdot B$ and a language raised to the power n is written A^n. They are defined as usual

$$\begin{array}{lcl} A \cdot B & \overset{def}{=} & \{s_1 @ s_2 \mid s_1 \in A \wedge s_2 \in B\} \\ A^{0} & \overset{def}{=} & \{[]\} \\ A^{n+1} & \overset{def}{=} & A \cdot A^n \end{array}$$

where @ is the list-append operation. The Kleene-star of a language A is defined as the union over all powers, namely $A^\star = (\bigcup_n A^n)$. In the paper we will make use of the following properties of these constructions.

Proposition 2.1.
(i) $A^\star = A \cdot A^\star \cup \{[]\}$
(ii) *If* $[] \notin A$ *and* $s \in A^{n+1}$ *then* $n < \textit{length}\ s$.
(iii) $B \cdot (\bigcup_n A^n) = (\bigcup_n B \cdot A^n)$
(iv) *If* $x \in A^\star$ *and* $x \neq []$ *then there exists an* x_p *and* x_s *with* $x = x_p \mathbin{@} x_s$ *and* $x_p \neq []$ *such that* $x_p \in A$ *and* $x_s \in A^\star$.

In (ii) we use the notation *length s* for the length of a string; this property states that if $[] \notin A$ then the lengths of the strings in A^{n+1} must be longer than n. Property (iv) states that a non-empty string in $A^\star$ can always be split up into a non-empty prefix belonging to A and the rest being in $A^\star$. We omit the proofs for these properties, but invite the reader to consult our formalisation.[2]

The notation in Isabelle/HOL for the quotient of a language A according to an equivalence relation $\approx$ is $A /\!/ \approx$. We will write $[\![x]\!]_\approx$ for the equivalence class defined as $\{y \mid y \approx x\}$, and have $x \approx y$ if and only if $[\![x]\!]_\approx = [\![y]\!]_\approx$.

Central to our proof will be the solution of equational systems involving equivalence classes of languages. For this we will use Arden's Lemma (see for example [24, Page 100]), which solves equations of the form $X = A \cdot X \cup B$ provided $[] \notin A$. However we will need the following 'reversed' version of Arden's Lemma ('reversed' in the sense of changing the order of $A \cdot X$ to $X \cdot A$).

Lemma 2.2 (Reversed Arden's Lemma)**.**
If $[] \notin A$ *then* $X = X \cdot A \cup B$ *if and only if* $X = B \cdot A^\star$.

Proof. For the right-to-left direction we assume $X = B \cdot A^\star$ and show that $X = X \cdot A \cup B$ holds. From Property 2.1(i) we have $A^\star = A \cdot A^\star \cup \{[]\}$, which is equal to $A^\star = A^\star \cdot A \cup \{[]\}$. Adding B to both sides gives $B \cdot A^\star = B \cdot (A^\star \cdot A \cup \{[]\})$, whose right-hand side is equal to $(B \cdot A^\star) \cdot A \cup B$. Applying the assumed equation completes this direction.

For the other direction we assume $X = X \cdot A \cup B$. By a simple induction on n, we can establish the property

$$(*) \qquad X = X \cdot A^{n+1} \cup (\bigcup_{m \le n} B \cdot A^m)$$

Using this property we can show that $B \cdot A^n \subseteq X$ holds for all n. From this we can infer $B \cdot A^\star \subseteq X$ using the definition of $\star$. For the inclusion in the other direction we assume a string s with length k is an element in X. Since $[] \notin A$ we know by Property 2.1(ii) that $s \notin X \cdot A^{k+1}$ since its length is only k (the strings in $X \cdot A^{k+1}$ are all longer). From $(*)$ it follows then that s must be an element in $\bigcup_{m \le k} B \cdot A^m$. This in turn implies that s is in $\bigcup_n B \cdot A^n$. Using Property 2.1(iii) this is equal to $B \cdot A^\star$, as we needed to show. □

Regular expressions are defined as the inductive datatype

[2]Available in the Archive of Formal Proofs at http://afp.sourceforge.net/devel-entries/Myhill-Nerode.shtml [27].

$$\begin{array}{rcl} r & ::= & \mathit{ZERO} \\ & \mid & \mathit{ONE} \\ & \mid & \mathit{ATOM}\ c \\ & \mid & \mathit{TIMES}\ r\ r \\ & \mid & \mathit{PLUS}\ r\ r \\ & \mid & \mathit{STAR}\ r \end{array}$$

and the language matched by a regular expression is defined as

$$\begin{array}{rcl} \mathcal{L}(\mathit{ZERO}) & \overset{def}{=} & \{\} \\ \mathcal{L}(\mathit{ONE}) & \overset{def}{=} & \{[]\} \\ \mathcal{L}(\mathit{ATOM}\ c) & \overset{def}{=} & \{[c]\} \\ \mathcal{L}(\mathit{PLUS}\ r_1\ r_2) & \overset{def}{=} & \mathcal{L}(r_1) \cup \mathcal{L}(r_2) \\ \mathcal{L}(\mathit{TIMES}\ r_1\ r_2) & \overset{def}{=} & \mathcal{L}(r_1) \cdot \mathcal{L}(r_2) \\ \mathcal{L}(\mathit{STAR}\ r) & \overset{def}{=} & \mathcal{L}(r)^{\star} \end{array}$$

Given a finite set of regular expressions *rs*, we will make use of the operation of generating a regular expression that matches the union of all languages of *rs*. We only need to know the existence of such a regular expression and therefore we use Isabelle/HOL's *fold_graph* and Hilbert's ε to define $+rs$. This operation, roughly speaking, folds *PLUS* over the set *rs* with *ZERO* for the empty set. We can prove that for a finite set *rs*

$$\mathcal{L}(+rs) = \bigcup\ (\mathcal{L}\ `\ rs) \tag{2}$$

holds, whereby $\mathcal{L}\ `\ rs$ stands for the image of the set *rs* under function $\mathcal{L}$ defined as

$$\mathcal{L}\ `\ rs \overset{def}{=} \{\mathcal{L}(r) \mid r \in rs\}$$

In what follows we shall use this convenient short-hand notation for images of sets also with other functions.

3. The Myhill-Nerode Theorem, First Part

The key definition in the Myhill-Nerode Theorem is the *Myhill-Nerode Relation*, which states that w.r.t. a language two strings are related, provided there is no distinguishing extension in this language. This can be defined as a tertiary relation.

Definition 3.1 (Myhill-Nerode Relation)**.** Given a language A, two strings x and y are Myhill-Nerode related provided

$$x \approx_A y \overset{def}{=} \forall z.\ (x\ @\ z \in A) = (y\ @\ z \in A)$$

It is easy to see that $\approx_A$ is an equivalence relation, which partitions the set of all strings, *UNIV*, into a set of disjoint equivalence classes. To illustrate this quotient construction, let us give a simple example: consider the regular language containing just the string $[c]$. The relation $\approx_{\{[c]\}}$ partitions *UNIV* into three equivalence classes X_1, X_2 and X_3 as follows

$$\begin{array}{l} X_1 = \{[]\} \\ X_2 = \{[c]\} \\ X_3 = \mathit{UNIV} - \{[], [c]\} \end{array}$$

One direction of the Myhill-Nerode Theorem establishes that if there are finitely many equivalence classes, like in the example above, then the language is regular. In our setting we therefore have to show:

Theorem 3.2. *If finite* $(\mathit{UNIV} /\!\!/ \approx_A)$ *then regular* A.

To prove this theorem, we first define the set *finals A* as those equivalence classes from $\mathit{UNIV} /\!\!/ \approx_A$ that contain strings of A, namely

$$\mathit{finals}\ A \overset{def}{=} \{[\![s]\!]_{\approx_A} \mid s \in A\} \tag{3}$$

In our running example, X_2 is the only equivalence class in *finals* $\{[c]\}$. It is straightforward to show that in general

$$A = \bigcup \mathit{finals}\ A \qquad\qquad \mathit{finals}\ A \subseteq \mathit{UNIV} /\!\!/ \approx_A \tag{4}$$

hold. Therefore if we know that there exists a regular expression for every equivalence class in *finals A* (which by assumption must be a finite set), then we can use $+$ to obtain a regular expression that matches every string in A.

Our proof of Theorem 3.2 relies on a method that can calculate a regular expression for *every* equivalence class, not just the ones in *finals A*. We first define the notion of *one-character-transition* between two equivalence classes

$$Y \overset{c}{\Longmapsto} X \overset{def}{=} Y \cdot \{[c]\} \subseteq X \tag{5}$$

which means that if we concatenate the character c to the end of all strings in the equivalence class Y, we obtain a subset of X. Note that we do not define an automaton here, we merely relate two sets (with the help of a character). In our concrete example we have $X_1 \overset{c}{\Longmapsto} X_2$, $X_1 \overset{d_i}{\Longmapsto} X_3$ with d_i being any other character than c, and $X_3 \overset{c_j}{\Longmapsto} X_3$ for any character c_j.

Next we construct an *initial equational system* that contains an equation for each equivalence class. We first give an informal description of this construction. Suppose we have the equivalence classes $X_1, \ldots, X_n$, there must be one and only one that contains the empty string $[]$ (since equivalence classes are disjoint). Let us assume $[] \in X_1$. We build the following equational system

$$\begin{array}{rcl} X_1 & = & (Y_{11}, \mathit{ATOM}\ c_{11}) + \ldots + (Y_{1p}, \mathit{ATOM}\ c_{1p}) + \lambda(\mathit{ONE}) \\ X_2 & = & (Y_{21}, \mathit{ATOM}\ c_{21}) + \ldots + (Y_{2o}, \mathit{ATOM}\ c_{2o}) \\ & \vdots & \\ X_n & = & (Y_{n1}, \mathit{ATOM}\ c_{n1}) + \ldots + (Y_{nq}, \mathit{ATOM}\ c_{nq}) \end{array}$$

where the terms $(Y_{ij}, \mathit{ATOM}\ c_{ij})$ stand for all transitions $Y_{ij} \overset{c_{ij}}{\Longmapsto} X_i$. There can only be finitely many terms of the form $(Y_{ij}, \mathit{ATOM}\ c_{ij})$ in a right-hand side since by assumption there are only finitely many equivalence classes and only finitely many characters. The term $\lambda(\mathit{ONE})$ in the first equation acts as a marker for the initial state, that is the equivalence class containing the empty string [].[3] In our running example we have the initial equational system

$$\begin{aligned} X_1 &= \lambda(\mathit{ONE}) \\ X_2 &= (X_1, \mathit{ATOM}\ c) \\ X_3 &= (X_1, \mathit{ATOM}\ d_1) + \ldots + (X_1, \mathit{ATOM}\ d_n) \\ &\qquad + (X_3, \mathit{ATOM}\ c_1) + \ldots + (X_3, \mathit{ATOM}\ c_m) \end{aligned} \tag{6}$$

where $d_1 \ldots d_n$ is the sequence of all characters but not containing c, and $c_1 \ldots c_m$ is the sequence of all characters.

Overloading the function $\mathcal{L}$ for the two kinds of terms in the equational system, we have

$$\mathcal{L}(Y, r) \overset{\mathit{def}}{=} Y \cdot \mathcal{L}(r) \qquad \mathcal{L}(\lambda(r)) \overset{\mathit{def}}{=} \mathcal{L}(r)$$

and we can prove for $X_{2..n}$ that the following equations

$$X_i = \mathcal{L}(Y_{i1}, \mathit{ATOM}\ c_{i1}) \cup \ldots \cup \mathcal{L}(Y_{iq}, \mathit{ATOM}\ c_{iq}). \tag{7}$$

hold. Similarly for X_1 we can show the following equation

$$X_1 = \mathcal{L}(Y_{11}, \mathit{ATOM}\ c_{11}) \cup \ldots \cup \mathcal{L}(Y_{1p}, \mathit{ATOM}\ c_{1p}) \cup \mathcal{L}(\lambda(\mathit{ONE})). \tag{8}$$

holds. The reason for adding the λ-marker to our initial equational system is to obtain this equation: it only holds with the marker, since none of the other terms contain the empty string. The point of the initial equational system is that solving it means we will be able to extract a regular expression for every equivalence class.

Our representation for the equations in Isabelle/HOL are pairs, where the first component is an equivalence class (a set of strings) and the second component is a set of terms. Given a set of equivalence classes *CS*, our initial equational system *Init CS* is thus formally defined as

$$\begin{aligned} \mathit{Init\_rhs}\ \mathit{CS}\ X &\overset{\mathit{def}}{=} \mathit{if}\ [] \in X \\ &\qquad \mathit{then}\ \{(Y, \mathit{ATOM}\ c) \mid Y \in \mathit{CS} \wedge Y \overset{c}{\Longmapsto} X\} \cup \{\lambda(\mathit{ONE})\} \\ &\qquad \mathit{else}\ \{(Y, \mathit{ATOM}\ c) \mid Y \in \mathit{CS} \wedge Y \overset{c}{\Longmapsto} X\} \\ \mathit{Init}\ \mathit{CS} &\overset{\mathit{def}}{=} \{(X, \mathit{Init\_rhs}\ \mathit{CS}\ X) \mid X \in \mathit{CS}\} \end{aligned} \tag{9}$$

Because we use sets of terms for representing the right-hand sides of equations, we can prove (7) and (8) more concisely as

Lemma 3.3. *If* $(X, \mathit{rhs}) \in \mathit{Init}\ (\mathit{UNIV} /\!/ \approx_A)$ *then* $X = \bigcup\ \mathcal{L}\ `\ \mathit{rhs}$.

[3]Note that we mark, roughly speaking, the single 'initial' state in the equational system, which is different from the method by Brzozowski [6], where he marks the 'terminal' states. We are forced to set up the equational system in our way, because the Myhill-Nerode Relation determines the 'direction' of the transitions—the successor 'state' of an equivalence class *Y* can be reached by adding a character to the end of *Y*. This is also the reason why we have to use our reversed version of Arden's Lemma.

Our proof of Theorem 3.2 will proceed by transforming the initial equational system into one in *solved form* maintaining the invariant in Lemma 3.3. From the solved form we will be able to read off the regular expressions.

In order to transform an equational system into solved form, we have two operations: one that takes an equation of the form $X = \mathit{rhs}$ and removes any recursive occurrences of X in the *rhs* using our variant of Arden's Lemma. The other operation takes an equation $X = \mathit{rhs}$ and substitutes X throughout the rest of the equational system adjusting the remaining regular expressions appropriately. To define this adjustment we define the *append-operation* taking a term and a regular expression as argument

$$\begin{aligned}(Y, r_2) \triangleleft r_1 &\stackrel{\mathit{def}}{=} (Y, \mathit{TIMES}\ r_2\ r_1)\\ \lambda(r_2) \triangleleft r_1 &\stackrel{\mathit{def}}{=} \lambda(\mathit{TIMES}\ r_2\ r_1)\end{aligned}$$

We lift this operation to entire right-hand sides of equations, written as $\mathit{rhs} \triangleleft r$. With this we can define the *arden-operation* for an equation of the form $X = \mathit{rhs}$ as:

$$\begin{aligned}\mathit{Arden}\ X\ \mathit{rhs} \stackrel{\mathit{def}}{=}\ & \mathit{let}\\ & \quad \mathit{rhs}' = \mathit{rhs} - \{(X, r) \mid (X, r) \in \mathit{rhs}\}\\ & \quad\ \ r' = \mathit{STAR}\ (+\{r \mid (X, r) \in \mathit{rhs}\})\\ & \mathit{in}\ \mathit{rhs}' \triangleleft r'\end{aligned} \tag{10}$$

In this definition, we first delete all terms of the form (X, r) from *rhs*; then we calculate the combined regular expressions for all r coming from the deleted (X, r), and take the *STAR* of it; finally we append this regular expression to rhs'. If we apply this operation to the right-hand side of X_3 in (6), we obtain the equation:

$$\begin{aligned}X_3 = (X_1, \mathit{TIMES}\ (\mathit{ATOM}\ d_1)\ (\mathit{STAR}\ +\{\mathit{ATOM}\ c_1, \ldots, \mathit{ATOM}\ c_m\})) + \ldots\\ \ldots + (X_1, \mathit{TIMES}\ (\mathit{ATOM}\ d_n)\ (\mathit{STAR}\ +\{\mathit{ATOM}\ c_1, \ldots, \mathit{ATOM}\ c_m\}))\end{aligned}$$

That means we eliminated the recursive occurrence of X_3 on the right-hand side. Note we used the abbreviation $+\{\mathit{ATOM}\ c_1, \ldots, \mathit{ATOM}\ c_m\}$ to stand for a regular expression that matches with every character. In our algorithm we are only interested in the existence of such a regular expression and do not specify it any further.

It can be easily seen that the *Arden*-operation mimics Arden's Lemma on the level of equations. To ensure the non-emptiness condition of Arden's Lemma we say that a right-hand side is *ardenable* provided

$$\mathit{ardenable}\ \mathit{rhs} \stackrel{\mathit{def}}{=} \forall\, Y\, r.\ (Y, r) \in \mathit{rhs} \longrightarrow [] \notin \mathcal{L}(r)$$

This allows us to prove a version of Arden's Lemma on the level of equations.

Lemma 3.4. *Given an equation $X = \mathit{rhs}$. If $X = \bigcup \mathcal{L}$ ‘ rhs, ardenable rhs, and finite rhs, then $X = \bigcup \mathcal{L}$ ‘ $(\mathit{Arden}\ X\ \mathit{rhs})$.*

Our *ardenable* condition is slightly stronger than needed for applying Arden's Lemma, but we can still ensure that it holds throughout our algorithm of transforming equations into solved form. The *substitution-operation* takes an equation of the form $X = \mathit{xrhs}$ and substitutes it into the right-hand side *rhs*.

$$\begin{array}{rcl} \mathit{Subst\ rhs\ X\ xrhs} & \overset{\mathit{def}}{=} & \mathit{let} \\ & & \quad \mathit{rhs}' = \mathit{rhs} - \{(X, r) \mid (X, r) \in \mathit{rhs}\} \\ & & \quad r' = +\{r \mid (X, r) \in \mathit{rhs}\} \\ & & \mathit{in}\ \mathit{rhs}' \cup (\mathit{xrhs} \triangleleft r') \end{array}$$

We again delete first all occurrences of (X, r) in *rhs*; we then calculate the regular expression corresponding to the deleted terms; finally we append this regular expression to *xrhs* and union it up with *rhs*′. When we use the substitution operation we will arrange it so that *xrhs* does not contain any occurrence of *X*. For example substituting the first equation in (6) into the right-hand side of the second, thus eliminating the equivalence class X_1, gives us the equation

$$X_2 = \lambda(\mathit{TIMES\ ONE\ (ATOM\ c)}) \tag{11}$$

With these two operations in place, we can define the operation that removes one equation from an equational systems *ES*. The operation *Subst_all* substitutes an equation $X = \mathit{xrhs}$ throughout an equational system *ES*; *Remove* then completely removes such an equation from *ES* by substituting it to the rest of the equational system, but first eliminating all recursive occurrences of *X* by applying *Arden* to *xrhs*.

$$\begin{array}{rcl} \mathit{Subst\_all\ ES\ X\ xrhs} & \overset{\mathit{def}}{=} & \{(Y, \mathit{Subst\ yrhs\ X\ xrhs}) \mid (Y, \mathit{yrhs}) \in \mathit{ES}\} \\ \mathit{Remove\ ES\ X\ xrhs} & \overset{\mathit{def}}{=} & \mathit{Subst\_all}\ (\mathit{ES} - \{(X, \mathit{xrhs})\})\ X\ (\mathit{Arden\ X\ xrhs}) \end{array}$$

Finally, we can define how an equational system should be solved. For this we will need to iterate the process of eliminating equations until only one equation will be left in the system. However, we do not just want to have any equation as being the last one, but the one involving the equivalence class for which we want to calculate the regular expression. Let us suppose this equivalence class is *X*. Since *X* is the one to be solved, in every iteration step we have to pick an equation to be eliminated that is different from *X*. In this way *X* is kept to the final step. The choice is implemented using Hilbert's choice operator, written *SOME* in the definition below.

$$\begin{array}{rcl} \mathit{Iter\ X\ ES} & \overset{\mathit{def}}{=} & \mathit{let} \\ & & \quad (Y, \mathit{yrhs}) = \mathit{SOME}\ (Y, \mathit{yrhs}).\ (Y, \mathit{yrhs}) \in \mathit{ES} \wedge X \neq Y \\ & & \mathit{in\ Remove\ ES\ Y\ yrhs} \end{array}$$

The last definition we need applies *Iter* over and over until a condition *Cond* is *not* satisfied anymore. This condition states that there are more than one equation left in the equational system *ES*. To solve an equational system we use Isabelle/HOL's *while*-operator as follows:

$$\mathit{Solve\ X\ ES} \overset{\mathit{def}}{=} \mathit{while\ Cond\ (Iter\ X)\ ES}$$

We are not concerned here with the definition of this operator (see Berghofer and Nipkow [4] for example), but note that we eliminate in each *Iter*-step a single equation, and therefore have a well-founded termination order by taking the cardinality of the equational system *ES*. This enables us to prove properties about our definition of *Solve* when we 'call' it with the equivalence class *X* and the initial equational system $\mathit{Init}\ (\mathit{UNIV} /\!/ \approx_A)$ from (9) using the principle:

$$\frac{\begin{array}{l} \textit{invariant}\ (\textit{Init}\ (\textit{UNIV} /\!/ \approx_A)) \\ \forall \textit{ES}.\ \textit{invariant ES} \wedge \textit{Cond ES} \longrightarrow \textit{invariant}\ (\textit{Iter X ES}) \\ \forall \textit{ES}.\ \textit{invariant ES} \wedge \textit{Cond ES} \longrightarrow \textit{card}\ (\textit{Iter X ES}) < \textit{card ES} \\ \forall \textit{ES}.\ \textit{invariant ES} \wedge \neg\ \textit{Cond ES} \longrightarrow \textit{P ES} \end{array}}{P\ (\textit{Solve X}\ (\textit{Init}\ (\textit{UNIV} /\!/ \approx_A)))} \tag{12}$$

This principle states that given an invariant (which we will specify below) we can prove a property P involving *Solve*. For this we have to discharge the following proof obligations: first the initial equational system satisfies the invariant; second the iteration step *Iter* preserves the invariant as long as the condition *Cond* holds; third *Iter* decreases the termination order, and fourth that once the condition does not hold anymore then the property P must hold.

The property P in our proof will state that $\textit{Solve X}\ (\textit{Init}\ (\textit{UNIV} /\!/ \approx_A))$ returns with a single equation $X = \textit{xrhs}$ for some *xrhs*, and that this equational system still satisfies the invariant. In order to get the proof through, the invariant is composed of the following six properties:

$$\begin{array}{lcll} \textit{invariant ES} & \overset{\textit{def}}{=} & \textit{finite ES} & (\textit{finiteness}) \\ & \wedge & \forall (X, \textit{rhs}) \in \textit{ES}.\ \textit{finite rhs} & (\textit{finiteness rhs}) \\ & \wedge & \forall (X, \textit{rhs}) \in \textit{ES}.\ X = \bigcup \mathcal{L}\ `\ \textit{rhs} & (\textit{soundness}) \\ & \wedge & \forall X\ \textit{rhs}\ \textit{rhs}'.\ (X, \textit{rhs}) \in \textit{ES} \wedge (X, \textit{rhs}') \in \textit{ES} \longrightarrow \textit{rhs} = \textit{rhs}' & \\ & & & (\textit{distinctness}) \\ & \wedge & \forall (X, \textit{rhs}) \in \textit{ES}.\ \textit{ardenable rhs} & (\textit{ardenable}) \\ & \wedge & \forall (X, \textit{rhs}) \in \textit{ES}.\ \textit{rhss rhs} \subseteq \textit{lhss ES} & (\textit{validity}) \end{array}$$

The first two ensure that the equational system is always finite (number of equations and number of terms in each equation); the third makes sure the 'meaning' of the equations is preserved under our transformations. The other properties are a bit more technical, but are needed to get our proof through. Distinctness states that every equation in the system is distinct. *Ardenable* ensures that we can always apply the *Arden* operation. The last property states that every *rhs* can only contain equivalence classes for which there is an equation. Therefore *lhss* is just the set containing the first components of an equational system, while *rhss* collects all equivalence classes X in the terms of the form (X, r). That means formally $\textit{lhss ES} \overset{\textit{def}}{=} \{X \mid (X, \textit{rhs}) \in \textit{ES}\}$ and $\textit{rhss rhs} \overset{\textit{def}}{=} \{X \mid (X, r) \in \textit{rhs}\}$.

It is straightforward to prove that the initial equational system satisfies the invariant.

Lemma 3.5. *If* $\textit{finite}\ (\textit{UNIV} /\!/ \approx_A)$ *then* $\textit{invariant}\ (\textit{Init}\ (\textit{UNIV} /\!/ \approx_A))$.

Proof. Finiteness is given by the assumption and the way how we set up the initial equational system. Soundness is proved in Lemma 3.3. Distinctness follows from the fact that the equivalence classes are disjoint. The *ardenable* property also follows from the setup of the initial equational system, as does validity. □

Next we show that *Iter* preserves the invariant.

Lemma 3.6. *If* $\textit{invariant ES}$ *and* $(X, \textit{rhs}) \in \textit{ES}$ *and* $\textit{Cond ES}$ *then* $\textit{invariant}\ (\textit{Iter X ES})$.

Proof. The argument boils down to choosing an equation $Y = \mathit{yrhs}$ to be eliminated and to show that $\mathit{Subst\_all}\ (\mathit{ES} - \{(Y, \mathit{yrhs})\})\ Y\ (\mathit{Arden}\ Y\ \mathit{yrhs})$ preserves the invariant. We prove this as follows:

$$\forall\ \mathit{ES}.\quad \mathit{invariant}\ (\mathit{ES} \cup \{(Y, \mathit{yrhs})\}) \text{ implies } \mathit{invariant}\ (\mathit{Subst\_all}\ \mathit{ES}\ Y\ (\mathit{Arden}\ Y\ \mathit{yrhs}))$$

Finiteness is straightforward, as the *Subst* and *Arden* operations keep the equational system finite. These operations also preserve soundness and distinctness (we proved soundness for *Arden* in Lemma 3.4). The property *ardenable* is clearly preserved because the append-operation cannot make a regular expression to match the empty string. Validity is given because *Arden* removes an equivalence class from *yrhs* and then *Subst_all* removes Y from the equational system. Having proved the implication above, we can instantiate *ES* with $\mathit{ES} - \{(Y, \mathit{yrhs})\}$ which matches with our proof-obligation of *Subst_all*. Since $\mathit{ES} = \mathit{ES} - \{(Y, \mathit{yrhs})\} \cup \{(Y, \mathit{yrhs})\}$, we can use the assumption to complete the proof. □

We also need the fact that *Iter* decreases the termination measure.

Lemma 3.7. *If invariant ES and* $(X, \mathit{rhs}) \in \mathit{ES}$ *and Cond ES then* $\mathit{card}\ (\mathit{Iter}\ X\ \mathit{ES}) < \mathit{card}\ \mathit{ES}$.

Proof. By assumption we know that *ES* is finite and has more than one element. Therefore there must be an element $(Y, \mathit{yrhs}) \in \mathit{ES}$ with $(Y, \mathit{yrhs}) \neq (X, \mathit{rhs})$. Using the distinctness property we can infer that $Y \neq X$. We further know that *Remove ES Y yrhs* removes the equation $Y = \mathit{yrhs}$ from the system, and therefore the cardinality of *Iter* strictly decreases. □

This brings us to our property we want to establish for *Solve*.

Lemma 3.8. *If finite* $(\mathit{UNIV} /\!/ \approx_A)$ *and* $X \in \mathit{UNIV} /\!/ \approx_A$ *then there exists a rhs such that* $\mathit{Solve}\ X\ (\mathit{Init}\ (\mathit{UNIV} /\!/ \approx_A)) = \{(X, \mathit{rhs})\}$ *and invariant* $\{(X, \mathit{rhs})\}$.

Proof. In order to prove this lemma using (12), we have to use a slightly stronger invariant since Lemma 3.6 and 3.7 have the precondition that $(X, \mathit{rhs}) \in \mathit{ES}$ for some *rhs*. This precondition is needed in order to choose in the *Iter*-step an equation that is not $X = \mathit{rhs}$. Therefore our invariant cannot be just *invariant ES*, but must be $\mathit{invariant}\ \mathit{ES} \wedge (\exists\, \mathit{rhs}.\ (X, \mathit{rhs}) \in \mathit{ES})$. By assumption $X \in \mathit{UNIV} /\!/ \approx_A$ and Lemma 3.5, the more general invariant holds for the initial equational system. This is premise 1 of (12). Premise 2 is given by Lemma 3.6 and the fact that *Iter* might modify the *rhs* in the equation $X = \mathit{rhs}$, but does not remove it. Premise 3 of (12) is by Lemma 3.7. Now in premise 4 we like to show that there exists a *rhs* such that $\mathit{ES} = \{(X, \mathit{rhs})\}$ and that $\mathit{invariant}\ \{(X, \mathit{rhs})\}$ holds, provided the condition *Cond* does not holds. By the stronger invariant we know there exists such a *rhs* with $(X, \mathit{rhs}) \in \mathit{ES}$. Because *Cond* is not true, we know the cardinality of *ES* is *1*. This means *ES* must actually be the set $\{(X, \mathit{rhs})\}$, for which the invariant holds. This allows us to conclude that $\mathit{Solve}\ X\ (\mathit{Init}\ (\mathit{UNIV} /\!/ \approx_A)) = \{(X, \mathit{rhs})\}$ and $\mathit{invariant}\ \{(X, \mathit{rhs})\}$ hold, as needed. □

With this lemma in place we can show that for every equivalence class in $\mathit{UNIV} /\!/ \approx_A$ there exists a regular expression.

Lemma 3.9. *If finite* $(UNIV /\!/ \approx_A)$ *and* $X \in UNIV /\!/ \approx_A$ *then regular* X.

Proof. By the preceding lemma, we know that there exists a *rhs* such that *Solve* X (*Init* $(UNIV /\!/ \approx_A)$) returns the equation $X = rhs$, and that the invariant holds for this equation. That means we know $X = \bigcup \mathcal{L}$ ‘ *rhs*. We further know that this is equal to $\bigcup \mathcal{L}$ ‘ (*Arden X rhs*) using the properties of the invariant and Lemma 3.4. Using the validity property for the equation $X = rhs$, we can infer that *rhss rhs* $\subseteq \{X\}$ and because the *Arden* operation removes that X from *rhs*, that *rhss* (*Arden X rhs*) $= \{\}$. This means the right-hand side *Arden X rhs* can only consist of terms of the form $\lambda(r)$. So we can collect those (finitely many) regular expressions *rs* and have $X = \mathcal{L}(+rs)$. With this we can conclude the proof. □

Lemma 3.9 allows us to finally give a proof for the first direction of the Myhill-Nerode Theorem.

Proof of Theorem 3.2. By Lemma 3.9 we know that there exists a regular expression for every equivalence class in $UNIV /\!/ \approx_A$. Since *finals A* is a subset of $UNIV /\!/ \approx_A$, we also know that for every equivalence class in *finals A* there exists a regular expression. Moreover by assumption we know that *finals A* must be finite, and therefore there must be a finite set of regular expressions *rs* such that $\bigcup$*finals A* $= \mathcal{L}(+rs)$. Since the left-hand side is equal to A, we can use $+rs$ as the regular expression that is needed in the theorem. □

Note that our algorithm for solving equational systems provides also a method for calculating a regular expression for the complement of a regular language: if we combine all regular expressions corresponding to equivalence classes not in *finals A*, then we obtain a regular expression for the complement language $\overline{A}$. This is similar to the usual construction of a ‘complement automaton’.

4. Myhill-Nerode, Second Part

In this section we will give a proof for establishing the second part of the Myhill-Nerode Theorem. It can be formulated in our setting as follows:

Theorem 4.1. *Given r is a regular expression, then finite* $(UNIV /\!/ \approx_{\mathcal{L}(r)})$.

The proof will be by induction on the structure of *r*. It turns out the base cases are straightforward.

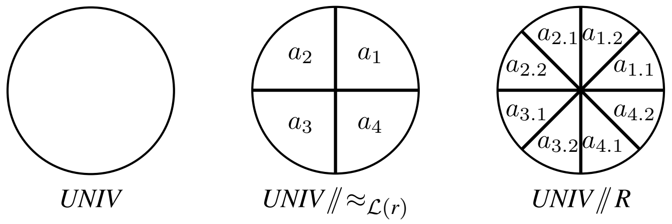
Base Cases. The cases for *ZERO*, *ONE* and *ATOM* are routine, because we can easily establish that

$$\begin{array}{l} UNIV /\!/ \approx_{\{\}} = \{UNIV\} \\ UNIV /\!/ \approx_{\{[]\}} \subseteq \{\{[]\}, UNIV - \{[]\}\} \\ UNIV /\!/ \approx_{\{[c]\}} \subseteq \{\{[]\}, \{[c]\}, UNIV - \{[], [c]\}\} \end{array}$$

hold, which shows that $UNIV /\!/ \approx_{\mathcal{L}(r)}$ must be finite. □

Much more interesting, however, are the inductive cases. They seem hard to be solved directly. The reader is invited to try.

In order to see how our proof proceeds consider the following suggestive picture taken from Constable et al [9]:



(13)

The relation $\approx_{\mathcal{L}(r)}$ partitions the set of all strings, *UNIV*, into some equivalence classes. To show that there are only finitely many of them, it suffices to show in each induction step that another relation, say R, has finitely many equivalence classes and refines $\approx_{\mathcal{L}(r)}$.

Definition 4.2. A relation R_1 *refines* R_2 provided $R_1 \subseteq R_2$.

For constructing R, we will rely on some *tagging-functions* defined over strings. Given the inductive hypothesis, it will be easy to prove that the *range* of these tagging-functions is finite. The range of a function f is defined as

$$\mathit{range}\, f \overset{def}{=} f \text{ ` } \mathit{UNIV}$$

that means we take the image of f w.r.t. all elements in the domain. With this we will be able to infer that the tagging-functions, seen as relations, give rise to finitely many equivalence classes. Finally we will show that the tagging-relations are more refined than $\approx_{\mathcal{L}(r)}$, which implies that *UNIV* $/\!\!/ \approx_{\mathcal{L}(r)}$ must also be finite. We formally define the notion of a *tagging-relation* as follows.

Definition 4.3 (Tagging-Relation)**.** Given a tagging-function *tag*, then two strings x and y are *tag-related* provided

$$x \approxeq_{tag} y \overset{def}{=} \mathit{tag}\; x = \mathit{tag}\; y\ .$$

In order to establish finiteness of a set A, we shall use the following powerful principle from Isabelle/HOL's library.

$$\mathit{If\ finite}\ (f \text{ ` } A)\ \ \mathit{and}\ \ \mathit{inj\_on}\, f\, A\ \ \mathit{then\ finite}\, A. \tag{14}$$

It states that if an image of a set under an injective function f (injective over this set) is finite, then the set A itself must be finite. We can use it to establish the following two lemmas.

Lemma 4.4. *If finite* $(\mathit{range\ tag})$ *then finite* $(\mathit{UNIV} /\!\!/ \approxeq_{tag})$.

Proof. We set in (14), f to be $X \mapsto \mathit{tag}$ ` $X$. We have *range f* to be a subset of *Pow* $(\mathit{range\ tag})$, which we know must be finite by assumption. Now $f$ ` *UNIV* $/\!\!/ \approxeq_{tag}$ is a subset of *range f*, and so also finite. Injectivity amounts to showing that $X = Y$ under the assumptions that $X, Y \in \mathit{UNIV} /\!\!/ \approxeq_{tag}$ and $f\, X = f\, Y$. From the assumptions we obtain $x \in X$ and $y \in Y$ with $\mathit{tag}\; x = \mathit{tag}\; y$. Since x and y are tag-related, this in turn means that the equivalence classes X and Y must be equal. Therefore (14) allows us to conclude with *finite* $(UNIV /\!/ \approxeq_{tag})$. $\square$

Lemma 4.5. *Given two equivalence relations R_1 and R_2, whereby R_1 refines R_2. If finite $(UNIV /\!/ R_1)$ then finite $(UNIV /\!/ R_2)$.*

Proof. We prove this lemma again using (14). This time we set f to be $X \mapsto \{[\![x]\!]_{R_1} \mid x \in X\}$. It is easy to see that *finite* $(f \text{ ` } UNIV /\!/ R_2)$ because it is a subset of *Pow* $(UNIV /\!/ R_1)$, which must be finite by assumption. What remains to be shown is that f is injective on $UNIV /\!/ R_2$. This is equivalent to showing that two equivalence classes, say X and Y, in $UNIV /\!/ R_2$ are equal, provided $f\, X = f\, Y$. For $X = Y$ to be equal, we have to find two elements $x \in X$ and $y \in Y$ such that they are R_2 related. We know there exists a $x \in X$ with $X = [\![x]\!]_{R_2}$. From the latter fact we can infer that $[\![x]\!]_{R_1} \in f\, X$ and further $[\![x]\!]_{R_1} \in f\, Y$. This means we can obtain a y such that $[\![x]\!]_{R_1} = [\![y]\!]_{R_1}$ holds. Consequently x and y are R_1-related. Since by assumption R_1 refines R_2, they must also be R_2-related, as we need to show. $\square$

Chaining Lemma 4.4 and 4.5 together, means in order to show that $UNIV /\!/ \approx_{\mathcal{L}(r)}$ is finite, we have to construct a tagging-function whose range can be shown to be finite and whose tagging-relation refines $\approx_{\mathcal{L}(r)}$. Let us attempt the *PLUS*-case first. We take as tagging-function

$$+tag\ A\ B\ x \stackrel{def}{=} ([\![x]\!]_{\approx_A}, [\![x]\!]_{\approx_B})$$

where A and B are some arbitrary languages. The reason for this choice is that we need to establish that $\approxeq_{+tag\ A\ B}$ refines $\approx_{A\,\cup\,B}$. This amounts to showing $x \approx_A y$ or $x \approx_B y$ under the assumption $x \approxeq_{+tag\ A\ B}\ y$. As we shall see, this definition will provide us with just the right assumptions in order to get the proof through.

PLUS-Case. We can show in general, if *finite* $(UNIV /\!/ \approx_A)$ and *finite* $(UNIV /\!/ \approx_B)$ then *finite* $(UNIV /\!/ \approx_A \times UNIV /\!/ \approx_B)$ holds. The range of $+tag\ A\ B$ is a subset of this product set—so finite. For the refinement proof-obligation, we know that $([\![x]\!]_{\approx_A}, [\![x]\!]_{\approx_B}) = ([\![y]\!]_{\approx_A}, [\![y]\!]_{\approx_B})$ holds by assumption. Then clearly either $x \approx_A y$ or $x \approx_B y$, as we needed to show. Finally we can discharge this case by setting A to $\mathcal{L}(r_1)$ and B to $\mathcal{L}(r_2)$. $\square$

The *TIMES*-case is slightly more complicated. We first prove the following lemma, which will aid the proof about refinement.

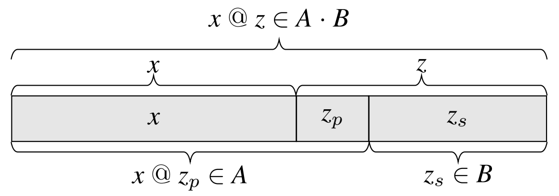
Lemma 4.6. *The relation $\approxeq_{tag}$ refines $\approx_A$, provided for all strings x, y and z we have that $x \approxeq_{tag}\ y$ and $x\ @\ z \in A$ imply $y\ @\ z \in A$.*

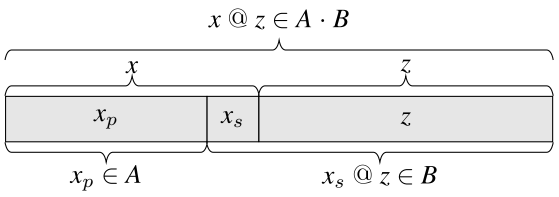
We therefore can analyse how the strings $x\ @\ z$ are in the language A and then construct an appropriate tagging-function to infer that $y\ @\ z$ are also in A. For this we will use the notion of the set of all possible *partitions* of a string:

$$Partitions\ x \stackrel{def}{=} \{(x_p, x_s) \mid x_p\ @\ x_s = x\} \tag{15}$$

If we know that $(x_p, x_s) \in Partitions\ x$, we will refer to x_p as the *prefix* of the string x, and respectively to x_s as the *suffix*.

Now assuming $x @ z \in A \cdot B$, there are only two possible ways of how to 'split' this string to be in $A \cdot B$:





Either x and a prefix of z is in A and the rest in B (first picture) or there is a prefix of x in A and the rest is in B (second picture). In both cases we have to show that $y @ z \in A \cdot B$. The first case we will only go through if we know that $x \approx_A y$ holds $(*)$. Because then we can infer from $x @ z_p \in A$ that $y @ z_p \in A$ holds for all z_p. In the second case we only know that x_p and x_s is one possible partition of the string x. We have to know that both x_p and the corresponding partition y_p are in A, and that x_s is 'B-related' to y_s $(**)$. From the latter fact we can infer that $y_s @ z \in B$. This will solve the second case. Taking the two requirements, $(*)$ and $(**)$, together we define the tagging-function in the *TIMES*-case as:

$$\times tag\ A\ B \overset{def}{=} ([\![x]\!]_{\approx A}, \{[\![x_s]\!]_{\approx B} \mid x_p \in A \wedge (x_p, x_s) \in Partitions\ x\})$$

Note that we have to make the assumption for all suffixes x_s, since we do not know anything about how the string x is partitioned. With this definition in place, let us prove the *TIMES*-case.

TIMES-Case. If *finite* $(UNIV /\!/ \approx_A)$ and *finite* $(UNIV /\!/ \approx_B)$ then *finite* $(UNIV /\!/ \approx_A \times Pow\ (UNIV /\!/ \approx_B))$ holds. The range of $\times tag\ A\ B$ is a subset of this product set, and therefore finite. For the refinement of $\approx_{A \cdot B}$ and $\approxeq_{\times tag\ A\ B}$, we have by Lemma 4.6

$$\times tag\ A\ B\ x = \times tag\ A\ B\ y$$

and $x @ z \in A \cdot B$, and have to establish $y @ z \in A \cdot B$. As shown in the pictures above, there are two cases to be considered. First, there exists a z_p and z_s such that $x @ z_p \in A$ and $z_s \in B$. By the assumption about $\times tag\ A\ B$ we have $[\![x]\!]_{\approx_A} = [\![y]\!]_{\approx_A}$ and thus $x \approx_A y$. Hence by the Myhill-Nerode Relation $y @ z_p \in A$ holds. Using $z_s \in B$, we can conclude in this case with $y @ z \in A \cdot B$ (recall $z_p @ z_s = z$).

Second there exists a partition x_p and x_s with $x_p \in A$ and $x_s @ z \in B$. We therefore have

$$[\![x_s]\!]_{\approx B} \in \{[\![x_s]\!]_{\approx B} \mid x_p \in A \wedge (x_p, x_s) \in Partitions\ x\}$$

and by the assumption about $\times tag\ A\ B$ also

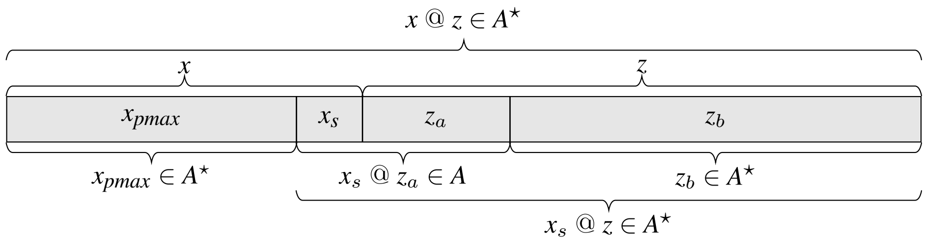
$$[\![x_s]\!]_{\approx B} \in \{[\![y_s]\!]_{\approx B} \mid y_p \in A \wedge (y_p, y_s) \in \mathit{Partitions}\ y\}$$

This means there must be a partition y_p and y_s such that $y_p \in A$ and $[\![x_s]\!]_{\approx_B} = [\![y_s]\!]_{\approx_B}$. Unfolding the Myhill-Nerode Relation and together with the facts that $x_p \in A$ and $x_s @ z \in B$, we obtain $y_p \in A$ and $y_s @ z \in B$, as needed in this case. We again can complete the *TIMES*-case by setting A to $\mathcal{L}(r_1)$ and B to $\mathcal{L}(r_2)$. □

The case for *STAR* is similar to *TIMES*, but poses a few extra challenges. To deal with them, we define first the notion of a *string prefix* and a *strict string prefix*:

$$x \leq y \overset{def}{=} \exists z.\ y = x @ z$$
$$x < y \overset{def}{=} x \leq y \wedge x \neq y$$

When analysing the case of $x @ z$ being an element in $A^\star$ and x is not the empty string, we have the following picture:



We can find a strict prefix x_p of x such that $x_p \in A^\star$, $x_p < x$ and the rest $x_s @ z \in A^\star$. For example the empty string [] would do (recall $x \neq []$). There are potentially many such prefixes, but there can only be finitely many of them (the string x is finite). Let us therefore choose the longest one and call it x_{pmax}. Now for the rest of the string $x_s @ z$ we know it is in $A^\star$ and cannot be the empty string. By Property 2.1(iv), we can separate this string into two parts, say a and b, such that $a \neq []$, $a \in A$ and $b \in A^\star$. Now a must be strictly longer than x_s, otherwise x_{pmax} is not the longest prefix. That means a 'overlaps' with z, splitting it into two components z_a and z_b. For this we know that $x_s @ z_a \in A$ and $z_b \in A^\star$. To cut a story short, we have divided $x @ z \in A^\star$ such that we have a string a with $a \in A$ that lies just on the 'border' of x and z. This string is $x_s @ z_a$.

In order to show that $x @ z \in A^\star$ implies $y @ z \in A^\star$, we use the following tagging-function:

$$\star tag\ A\ x \overset{def}{=} \{[\![x_s]\!]_{\approx A} \mid x_p < x \wedge x_p \in A^\star \wedge (x_s, x_p) \in \mathit{Partitions}\ x\}$$

STAR-Case. If *finite* $(\mathit{UNIV} /\!/ \approx_A)$ then *finite* $(\mathit{Pow}\ (\mathit{UNIV} /\!/ \approx_A))$ holds. The range of $\star tag$ A is a subset of this set, and therefore finite. Again we have to show under the assumption $x \approx_{\star tag\ A} y$ that $x @ z \in A^\star$ implies $y @ z \in A^\star$.

We first need to consider the case that x is the empty string. From the assumption about strict prefixes in $\approx_{\star tag\ A}$, we can infer y is the empty string and then clearly have $y @ z \in A^\star$. In case x is not the empty string, we can divide the string $x @ z$ as shown in the picture above. By the tagging-function and the facts $x_{pmax} \in A^\star$ and $x_{pmax} < x$, we have

$$[\![x_s]\!]_{\approx A} \in \{[\![x_s]\!]_{\approx A} \mid x_{pmax} < x \wedge x_{pmax} \in A^\star \wedge (x_{pmax}, x_s) \in \mathit{Partitions}\ x\}$$

which by assumption is equal to

$$[\![x_s]\!]_{\approx A} \in \{[\![y_s]\!]_{\approx A} \mid y_p < y \wedge y_p \in A^\star \wedge (y_p, y_s) \in \mathit{Partitions}\ y\}$$

From this we know there exist a partition y_p and y_s with $y_p \in A^\star$ and also $x_s \approx_A y_s$. Unfolding the Myhill-Nerode Relation we know $y_s \mathbin{@} z_a \in A$. We also know that $z_b \in A^\star$. Therefore $y_p \mathbin{@} (y_s \mathbin{@} z_a) \mathbin{@} z_b \in A^\star$, which means $y \mathbin{@} z \in A^\star$. The last step is to set A to $\mathcal{L}(r)$ and thus complete the proof. □

5. Second Part proved using Partial Derivatives

As we have seen in the previous section, in order to establish the second direction of the Myhill-Nerode Theorem, it is sufficient to find a more refined relation than $\approx_{\mathcal{L}(r)}$ for which we can show that there are only finitely many equivalence classes. So far we showed this directly by induction on r using tagging-functions. However, there is also an indirect method to come up with such a refined relation by using derivatives of regular expressions [6].

Assume the following two definitions for the *left-quotient* of a language, which we write as $\mathit{Der}\ c\ A$ and $\mathit{Ders}\ s\ A$ where c is a character and s a string, respectively:

$$\begin{array}{rcl} \mathit{Der}\ c\ A & \stackrel{\mathit{def}}{=} & \{s \mid [c] \mathbin{@} s \in A\} \\ \mathit{Ders}\ s\ A & \stackrel{\mathit{def}}{=} & \{s' \mid s \mathbin{@} s' \in A\} \end{array}$$

In order to aid readability, we shall make use of the following abbreviation

$$\mathit{Derss}\ s\ \mathit{As} \stackrel{\mathit{def}}{=} \bigcup \mathit{Ders}\ s\ `\ \mathit{As}$$

where we apply the left-quotient to a set of languages and then combine the results. Clearly we have the following equivalence between the Myhill-Nerode Relation (Definition 3.1) and left-quotients

$$x \approx_A y \quad \text{if and only if} \quad \mathit{Ders}\ x\ A = \mathit{Ders}\ y\ A \tag{16}$$

It is also straightforward to establish the following properties of left-quotients

$$\begin{array}{lcl} \mathit{Der}\ a\ \{\} & = & \{\} \\ \mathit{Der}\ a\ \{[]\} & = & \{\} \\ \mathit{Der}\ a\ \{[b]\} & = & \mathit{if}\ a = b\ \mathit{then}\ \{[]\}\ \mathit{else}\ \{\} \\ \mathit{Der}\ a\ (A \cup B) & = & \mathit{Der}\ a\ A \cup \mathit{Der}\ a\ B \\ \mathit{Der}\ c\ (A \cdot B) & = & (\mathit{Der}\ c\ A) \cdot B \cup (\mathit{if}\ [] \in A\ \mathit{then}\ \mathit{Der}\ c\ B\ \mathit{else}\ \{\}) \\ \mathit{Der}\ c\ (A^\star) & = & (\mathit{Der}\ c\ A) \cdot A^\star \\ \mathit{Ders}\ []\ A & = & A \\ \mathit{Ders}\ (c :: s)\ A & = & \mathit{Ders}\ s\ (\mathit{Der}\ c\ A) \end{array} \tag{17}$$

Note that in the last equation we use the list-cons operator written _ :: _. The only interesting case is the case of $A^\star$ where we use Property 2.1(i) in order to infer that $\mathit{Der}\ c\ (A^\star) = \mathit{Der}\ c\ (A \cdot A^\star)$. We can then complete the proof by using the fifth equation and noting that $\mathit{Der}\ c\ (A^\star) \subseteq (\mathit{Der}\ c\ A) \cdot A^\star$ provided $[] \in A$.

Brzozowski observed that the left-quotients for languages of regular expressions can be calculated directly using the notion of *derivatives of a regular expression* [6]. We define this notion in Isabelle/HOL as follows:

$$\begin{array}{lcl}
\mathit{der}\ c\ (\mathit{ZERO}) & \stackrel{\mathit{def}}{=} & \mathit{ZERO}\\
\mathit{der}\ c\ (\mathit{ONE}) & \stackrel{\mathit{def}}{=} & \mathit{ZERO}\\
\mathit{der}\ c\ (\mathit{ATOM}\ d) & \stackrel{\mathit{def}}{=} & \mathit{if}\ c = d\ \mathit{then}\ \mathit{ONE}\ \mathit{else}\ \mathit{ZERO}\\
\mathit{der}\ c\ (\mathit{PLUS}\ r_1\ r_2) & \stackrel{\mathit{def}}{=} & \mathit{PLUS}\ (\mathit{der}\ c\ r_1)\ (\mathit{der}\ c\ r_2)\\
\mathit{der}\ c\ (\mathit{TIMES}\ r_1\ r_2) & \stackrel{\mathit{def}}{=} & \mathit{if}\ \delta(r_1)\ \mathit{then}\ \mathit{PLUS}\ (\mathit{TIMES}\ (\mathit{der}\ c\ r_1)\ r_2)\ (\mathit{der}\ c\ r_2)\\
 & & \qquad\qquad \mathit{else}\ \mathit{TIMES}\ (\mathit{der}\ c\ r_1)\ r_2\\
\mathit{der}\ c\ (\mathit{STAR}\ r) & \stackrel{\mathit{def}}{=} & \mathit{TIMES}\ (\mathit{der}\ c\ r)\ (\mathit{STAR}\ r)\\
\mathit{ders}\ []\ r & \stackrel{\mathit{def}}{=} & r\\
\mathit{ders}\ (c :: s)\ r & \stackrel{\mathit{def}}{=} & \mathit{ders}\ s\ (\mathit{der}\ c\ r)
\end{array}$$

The last two clauses extend derivatives from characters to strings. The boolean function $\delta(r)$ needed in the *TIMES*-case tests whether a regular expression can recognise the empty string. It can be defined as follows.

$$\begin{array}{lcl}
\delta(\mathit{ZERO}) & \stackrel{\mathit{def}}{=} & \mathit{False}\\
\delta(\mathit{ONE}) & \stackrel{\mathit{def}}{=} & \mathit{True}\\
\delta(\mathit{ATOM}\ c) & \stackrel{\mathit{def}}{=} & \mathit{False}\\
\delta(\mathit{PLUS}\ r_1\ r_2) & \stackrel{\mathit{def}}{=} & \delta(r_1) \vee \delta(r_2)\\
\delta(\mathit{TIMES}\ r_1\ r_2) & \stackrel{\mathit{def}}{=} & \delta(r_1) \wedge \delta(r_2)\\
\delta(\mathit{STAR}\ r) & \stackrel{\mathit{def}}{=} & \mathit{True}
\end{array}$$

By induction on the regular expression r, respectively on the string s, one can easily show that left-quotients and derivatives of regular expressions relate as follows (see for example [24]):

$$\begin{array}{c}
\mathit{Der}\ c\ (\mathcal{L}(r)) = \mathcal{L}(\mathit{der}\ c\ r)\\
\mathit{Ders}\ s\ (\mathcal{L}(r)) = \mathcal{L}(\mathit{ders}\ s\ r)
\end{array} \tag{18}$$

The importance of this fact in the context of the Myhill-Nerode Theorem is that we can use (16) and (18) in order to establish that

$$x \approx_{\mathcal{L}(r)} y \quad \text{if and only if} \quad \mathcal{L}(\mathit{ders}\ x\ r) = \mathcal{L}(\mathit{ders}\ y\ r).$$

holds and hence

$$x \approx_{\mathcal{L}(r)} y \quad \text{provided} \quad \mathit{ders}\ x\ r = \mathit{ders}\ y\ r \tag{19}$$

This means the right-hand side (seen as a relation) refines the Myhill-Nerode Relation. Consequently, we can use $\approxeq_{(\lambda x.\ \mathit{ders}\ x\ r)}$ as a tagging-relation. However, in order to be useful for the second part of the Myhill-Nerode Theorem, we have to be able to establish that for the corresponding language there are only finitely many derivatives—thus ensuring that there are only finitely many equivalence classes. Unfortunately, this is not true in general. Sakarovitch gives an example where a regular expression has infinitely many derivatives w.r.t. the language $(ab)^\star \cup (ab)^\star a$, which is formally written in our notation as $\{[a,b]\}^\star \cup (\{[a,b]\}^\star \cdot \{[a]\})$ (see [24, Page 141]).

What Brzozowski [6] established is that for every language there *are* only finitely 'dissimilar' derivatives for a regular expression. Two regular expressions are said to be *similar* provided they can be identified using the using the *ACI*-identities:

$$\begin{array}{ll} (A) & \mathit{PLUS}\ (\mathit{PLUS}\ r_1\ r_2)\ r_3 \equiv \mathit{PLUS}\ r_1\ (\mathit{PLUS}\ r_2\ r_3) \\ (C) & \mathit{PLUS}\ r_1\ r_2 \equiv \mathit{PLUS}\ r_2\ r_1 \\ (I) & \mathit{PLUS}\ r\ r \equiv r \end{array} \tag{20}$$

Carrying this idea through, we must not consider the set of all derivatives, but the one modulo *ACI*. In principle, this can be done formally, but it is very painful in a theorem prover (since there is no direct characterisation of the set of dissimilar derivatives).

Fortunately, there is a much simpler approach using *partial derivatives*. They were introduced by Antimirov [2] and can be defined in Isabelle/HOL as follows:

$$\begin{array}{lcl} \mathit{pder}\ c\ (\mathit{ZERO}) & \stackrel{\mathit{def}}{=} & \{\} \\ \mathit{pder}\ c\ (\mathit{ONE}) & \stackrel{\mathit{def}}{=} & \{\} \\ \mathit{pder}\ c\ (\mathit{ATOM}\ d) & \stackrel{\mathit{def}}{=} & \mathit{if}\ c = d\ \mathit{then}\ \{\mathit{ONE}\}\ \mathit{else}\ \{\} \\ \mathit{pder}\ c\ (\mathit{PLUS}\ r_1\ r_2) & \stackrel{\mathit{def}}{=} & \mathit{pder}\ c\ r_1 \cup \mathit{pder}\ c\ r_2 \\ \mathit{pder}\ c\ (\mathit{TIMES}\ r_1\ r_2) & \stackrel{\mathit{def}}{=} & \mathit{if}\ \delta(r_1)\ \mathit{then}\ \mathit{TIMESS}\ (\mathit{pder}\ c\ r_1)\ r_2 \cup \mathit{pder}\ c\ r_2 \\ & & \qquad\qquad \mathit{else}\ \mathit{TIMESS}\ (\mathit{pder}\ c\ r_1)\ r_2 \\ \mathit{pder}\ c\ (\mathit{STAR}\ r) & \stackrel{\mathit{def}}{=} & \mathit{TIMESS}\ (\mathit{pder}\ c\ r)\ (\mathit{STAR}\ r) \\ \mathit{pders}\ []\ r & \stackrel{\mathit{def}}{=} & \{r\} \\ \mathit{pders}\ (c :: s)\ r & \stackrel{\mathit{def}}{=} & \bigcup\ (\mathit{pders}\ s)\ \text{`}\ (\mathit{pder}\ c\ r) \end{array}$$

Again the last two clauses extend partial derivatives from characters to strings. Unlike 'simple' derivatives, the functions for partial derivatives return sets of regular expressions. In the *TIMES* and *STAR* cases we therefore use the auxiliary definition

$$\mathit{TIMESS}\ rs\ r \stackrel{\mathit{def}}{=} \{\mathit{TIMES}\ r'\ r \mid r' \in rs\}$$

in order to 'sequence' a regular expression with a set of regular expressions. Note that in the last clause we first build the set of partial derivatives w.r.t the character c, then build the image of this set under the function *pders s* and finally 'union up' all resulting sets. It will be convenient to introduce for this the following abbreviation

$$\mathit{pderss}\ s\ rs \stackrel{\mathit{def}}{=} \bigcup \mathit{pders}\ s\ \text{`}\ rs$$

which simplifies the last clause of *pders* to

$$\mathit{pders}\ (c :: s)\ r \stackrel{\mathit{def}}{=} \mathit{pderss}\ s\ (\mathit{pder}\ c\ r)$$

Partial derivatives can be seen as having the *ACI*-identities already built in: taking the partial derivatives of the regular expressions in (20) gives us in each case equal sets. Antimirov [2] showed a similar result to (18) for partial derivatives, namely

$$\begin{array}{rl} (i) & \mathit{Der}\ c\ (\mathcal{L}(r)) = \bigcup \mathcal{L}\ \text{`}\ \mathit{pder}\ c\ r \\ (ii) & \mathit{Ders}\ s\ (\mathcal{L}(r)) = \bigcup \mathcal{L}\ \text{`}\ \mathit{pders}\ s\ r \end{array} \tag{21}$$

Proof. The first fact is by a simple induction on r. For the second we slightly modify Antimirov's proof by performing an induction on s where we generalise over all r. That means in the *cons*-case the induction hypothesis is

$$(IH) \quad \forall\, r.\; Ders\ s\ (\mathcal{L}(r)) = \bigcup \mathcal{L}\ \text{‘}\ pders\ s\ r$$

With this we can establish

$$\begin{array}{rcll} Ders\ (c :: s)\ (\mathcal{L}(r)) & = & Ders\ s\ (Der\ c\ (\mathcal{L}(r))) & \text{by def.} \\ & = & Ders\ s\ (\bigcup \mathcal{L}\ \text{‘}\ pder\ c\ r) & \text{by (21.}i\text{)} \\ & = & Derss\ s\ (\mathcal{L}\ \text{‘}\ pder\ c\ r) & \text{by def. of } Ders \\ & = & \bigcup \mathcal{L}\ \text{‘}\ pderss\ s\ (pder\ c\ r) & \text{by IH} \\ & = & \bigcup \mathcal{L}\ \text{‘}\ pders\ (c :: s)\ r & \text{by def.} \end{array}$$

Note that in order to apply the induction hypothesis in the fourth equation, we need the generalisation over all regular expressions r. The case for the empty string is routine and omitted. □

Taking (18) and (21) together gives the relationship between languages of derivatives and partial derivatives

$$\begin{array}{ll} (i) & \mathcal{L}(der\ c\ r) = \bigcup \mathcal{L}\ \text{‘}\ pder\ c\ r \\ (ii) & \mathcal{L}(ders\ s\ r) = \bigcup \mathcal{L}\ \text{‘}\ pders\ s\ r \end{array} \tag{22}$$

These two properties confirm the observation made earlier that by using sets, partial derivatives have the *ACI*-identities of derivatives already built in.

Antimirov also proved that for every language and every regular expression there are only finitely many partial derivatives, whereby the set of partial derivatives of r w.r.t. a language A is defined as

$$pdersl\ A\ r \stackrel{def}{=} \bigcup_{x\in A} pders\ x\ r \tag{23}$$

Theorem 5.1 (Antimirov [2])**.** *For every language A and every regular expression r, finite* (*pdersl A r*).

Antimirov's proof first establishes this theorem for the language $UNIV^+$, which is the set of all non-empty strings. For this he proves:

$$\begin{array}{l} pdersl\ UNIV^+\ (ZERO) = \{\} \\ pdersl\ UNIV^+\ (ONE) = \{\} \\ pdersl\ UNIV^+\ (ATOM\ c) = \{ONE\} \\ pdersl\ UNIV^+\ (PLUS\ r_1\ r_2) = pdersl\ UNIV^+\ r_1 \cup pdersl\ UNIV^+\ r_2 \\ pdersl\ UNIV^+\ (TIMES\ r_1\ r_2) \subseteq TIMESS\ (pdersl\ UNIV^+\ r_1)\ r_2 \cup pdersl\ UNIV^+\ r_2 \\ pdersl\ UNIV^+\ (STAR\ r) \subseteq TIMESS\ (pdersl\ UNIV^+\ r)\ (STAR\ r) \end{array} \tag{24}$$

from which one can deduce by induction on r that

$$finite\ (pdersl\ UNIV^+\ r)$$

holds. Now Antimirov's theorem follows because

$$\textit{pdersl UNIV } r = \textit{pders } [] \; r \cup \textit{pdersl UNIV}^{+} \; r$$

and for all languages A, *pdersl* A r is a subset of *pdersl UNIV* r. Since we follow Antimirov's proof quite closely in our formalisation (only the last two cases of (24) involve some non-routine induction arguments), we omit the details.

Let us now return to our proof for the second direction in the Myhill-Nerode Theorem. The point of the above calculations is to use $\approxeq_{(\lambda x.\, \textit{pders}\; x\; r)}$ as tagging-relation.

Proof of Theorem 4.1 (second version). Using (16) and (21) we can easily infer that

$$x \approx_{\mathcal{L}(r)} y \quad \text{provided} \quad \textit{pders}\; x\; r = \textit{pders}\; y\; r$$

which means the tagging-relation $\approxeq_{(\lambda x.\, \textit{pders}\; x\; r)}$ refines $\approx_{\mathcal{L}(r)}$. So we know by Lemma 4.5, *finite* $(\textit{UNIV} /\!/ \approx_{\mathcal{L}(r)})$ holds if *finite* $(\textit{UNIV} /\!/ \approxeq_{(\lambda x.\, \textit{pders}\; x\; r)})$. In order to establish the latter, we can use Lemma 4.4 and show that the range of the tagging-function $\lambda x.\, \textit{pders}\; x\; r$ is finite. For this recall Definition 23, which gives us that

$$\textit{pdersl UNIV } r \stackrel{\textit{def}}{=} \bigcup_{x} \textit{pders}\; x\; r$$

Now the range of $\lambda x.\, \textit{pders}\; x\; r$ is a subset of *Pow* (*pdersl UNIV* r), which we know is finite by Theorem 5.1. Consequently there are only finitely many equivalence classes of $\approxeq_{(\lambda x.\, \textit{pders}\; x\; r)}$. This relation refines $\approx_{\mathcal{L}(r)}$, and therefore we can again conclude the second part of the Myhill-Nerode Theorem. □

6. Closure Properties of Regular Languages

The beauty of regular languages is that they are closed under many set operations. Closure under union, concatenation and Kleene-star are trivial to establish given our definition of regularity (recall Definition 1.2). More interesting in our setting is the closure under complement, because it seems difficult to construct a regular expression for the complement language by direct means. However the existence of such a regular expression can now be easily proved using both parts of the Myhill-Nerode Theorem, since

$$s_1 \approx_A s_2 \text{ if and only if } s_1 \approx_{\overline{A}} s_2$$

holds for any strings s_1 and s_2. Therefore A and the complement language $\overline{A}$ give rise to the same partitions. So if one is finite, the other is too, and vice versa. As noted earlier, our algorithm for solving equational systems actually calculates a regular expression for the complement language. Calculating such a regular expression via automata using the standard method would be quite involved. It includes the steps: regular expression $\Rightarrow$ non-deterministic automaton $\Rightarrow$ deterministic automaton $\Rightarrow$ complement automaton $\Rightarrow$ regular expression. Clearly not something you want to formalise in a theorem prover in which it is cumbersome to reason about automata.

Once closure under complement is established, closure under intersection and set difference is also easy, because

$$A \cap B = \overline{(\overline{A} \cup \overline{B})}$$
$$A - B = \overline{(\overline{A} \cup B)}$$

Since all finite languages are regular, then by closure under complement also all co-finite languages. Closure of regular languages under reversal, that is

$$A^{-1} \overset{def}{=} \{s^{-1} \mid s \in A\}$$

can be shown with the help of the following operation defined recursively over regular expressions

$$\begin{aligned}
Rev\ (ZERO) &\overset{def}{=} ZERO\\
Rev\ (ONE) &\overset{def}{=} ONE\\
Rev\ (ATOM\ c) &\overset{def}{=} ATOM\ c\\
Rev\ (PLUS\ r_1\ r_2) &\overset{def}{=} PLUS\ (Rev\ r_1)\ (Rev\ r_2)\\
Rev\ (TIMES\ r_1\ r_2) &\overset{def}{=} TIMES\ (Rev\ r_2)\ (Rev\ r_1)\\
Rev\ (STAR\ r) &\overset{def}{=} STAR\ (Rev\ r)
\end{aligned}$$

For this operation we can show

$$(\mathcal{L}(r))^{-1} = \mathcal{L}(Rev\ r)$$

from which closure under reversal of regular languages follows.

A perhaps surprising fact is that regular languages are closed under any left-quotient. Define

$$Dersl\ B\ A \overset{def}{=} \bigcup_{x \in B} Ders\ x\ A$$

and assume B is any language and A is regular, then $Dersl\ B\ A$ is regular. To see this consider the following argument using partial derivatives: From A being regular we know there exists a regular expression r such that $A = \mathcal{L}(r)$. We also know that $pdersl\ B\ r$ is finite for every language B and regular expression r (recall Theorem 5.1). By definition and (21) we have

$$Dersl\ B\ (\mathcal{L}(r)) = \bigcup \mathcal{L}\ `\ pdersl\ B\ r \tag{25}$$

Since there are only finitely many regular expressions in $pdersl\ B\ r$, we know by (2) that there exists a regular expression so that the right-hand side of (25) is equal to the language $\mathcal{L}(+(pdersl\ B\ r))$. Thus the regular expression $+(pdersl\ B\ r)$ verifies that $Dersl\ B\ A$ is regular.

Even more surprising is the fact that for *every* language A, the language consisting of all substrings of A is regular [13] (see also [10, 25]). A *substring* can be obtained by striking out zero or more characters from a string. This can be defined inductively in Isabelle/HOL by the following three rules:

$$\frac{}{[] \preceq x} \qquad \frac{x \preceq y}{x \preceq c :: y} \qquad \frac{x \preceq y}{c :: x \preceq c :: y}$$

It is straightforward to prove that $\preceq$ is a partial order. Now define the *language of substrings* and *superstrings* of a language A respectively as

$$\begin{aligned}
Sub\ A &\overset{def}{=} \{x \mid \exists y{\in}A.\ x \preceq y\}\\
Sup\ A &\overset{def}{=} \{x \mid \exists y{\in}A.\ y \preceq x\}
\end{aligned}$$

We like to establish

Theorem 6.1 (Haines [13])**.** *For every language A, the languages* (i) *Sub A and* (ii) *Sup A are regular.*

Our proof follows the one given in [25, Pages 92–95], except that we use Higman's Lemma, which is already proved in the Isabelle/HOL library [3].[4] Higman's Lemma allows us to infer that every language A of antichains, satisfying

$$\forall x, y \in A.\, x \neq y \longrightarrow x \not\preceq y \wedge y \not\preceq x \tag{26}$$

is finite.

The first step in our proof of Theorem 6.1 is to establish the following simple properties for *Sup*

$$\begin{array}{lcl} \mathit{Sup}\ \{\} & \stackrel{\mathit{def}}{=} & \{\} \\ \mathit{Sup}\ \{[]\} & \stackrel{\mathit{def}}{=} & \mathit{UNIV} \\ \mathit{Sup}\ \{[c]\} & \stackrel{\mathit{def}}{=} & \mathit{UNIV} \cdot \{[c]\} \cdot \mathit{UNIV} \\ \mathit{Sup}\ (A \cup B) & \stackrel{\mathit{def}}{=} & \mathit{Sup}\ A \cup \mathit{Sup}\ B \\ \mathit{Sup}\ (A \cdot B) & \stackrel{\mathit{def}}{=} & \mathit{Sup}\ A \cdot \mathit{Sup}\ B \\ \mathit{Sup}\ (A^\star) & \stackrel{\mathit{def}}{=} & \mathit{UNIV} \end{array} \tag{27}$$

whereby the last equation follows from the fact that $A^\star$ contains the empty string. With these properties at our disposal we can establish the lemma

Lemma 6.2. *If A is regular, then also Sup A.*

Proof. Since our alphabet is finite, we have a regular expression, written *ALL*, that matches every string. Using this regular expression we can inductively define the operation $r\uparrow$

$$\begin{array}{lcl} (\mathit{ZERO})\uparrow & \stackrel{\mathit{def}}{=} & \mathit{ZERO} \\ (\mathit{ONE})\uparrow & \stackrel{\mathit{def}}{=} & \mathit{ALL} \\ (\mathit{ATOM}\ c)\uparrow & \stackrel{\mathit{def}}{=} & \mathit{TIMES}\ \mathit{ALL}\ (\mathit{TIMES}\ (\mathit{ATOM}\ c)\ \mathit{ALL}) \\ (\mathit{PLUS}\ r_1\ r_2)\uparrow & \stackrel{\mathit{def}}{=} & \mathit{PLUS}\ (r_1)\uparrow\ (r_2)\uparrow \\ (\mathit{TIMES}\ r_1\ r_2)\uparrow & \stackrel{\mathit{def}}{=} & \mathit{TIMES}\ (r_1)\uparrow\ (r_2)\uparrow \\ (\mathit{STAR}\ r)\uparrow & \stackrel{\mathit{def}}{=} & \mathit{ALL} \end{array}$$

and use (27) to establish that $\mathcal{L}((r)\uparrow) = \mathit{Sup}\ (\mathcal{L}(r))$ holds. This shows that *Sup A* is regular, provided A is. □

Now we can prove the main lemma w.r.t. *Sup*, namely

Lemma 6.3. *For every language A, there exists a finite language M such that*

$$\mathit{Sup}\ M = \mathit{Sup}\ A\ .$$

[4]Unfortunately, Berghofer's formalisation of Higman's Lemma is restricted to 2-letter alphabets, which means also our formalisation of Theorem 6.1 is 'tainted' with this constraint. However our methodology is applicable to any alphabet of finite size.

Proof. For M we take the set of all minimal elements of A. An element x is said to be *minimal* in A provided

$$min_A\ x \overset{def}{=} \forall y{\in}A.\ y \preceq x \longrightarrow x \preceq y$$

By Higman's Lemma (26) we know that $M \overset{def}{=} \{x \in A \mid min_A\ x\}$ is finite, since every minimal element is incomparable, except with itself. It is also straightforward to show that $Sup\ M \subseteq Sup\ A$. For the other direction we have $x \in Sup\ A$. From this we obtain a y such that $y \in A$ and $y \preceq x$. Since we have that the relation $\{(y, x) \mid y \preceq x \wedge x \neq y\}$ is well-founded, there must be a minimal element z such that $z \in A$ and $z \preceq y$, and hence by transitivity also $z \preceq x$ (here we deviate from the argument given in [25], because Isabelle/HOL provides already an extensive infrastructure for reasoning about well-foundedness). Since z is minimal and an element in A, we also know that z is in M. From this together with $z \preceq x$, we can infer that x is in $Sup\ M$, as required. □

This lemma allows us to establish the second part of Theorem 6.1.

Proof of the Second Part of Theorem 6.1. Given any language A, by Lemma 6.3 we know there exists a finite, and thus regular, language M. We further have $Sup\ M = Sup\ A$, which establishes the second part. □

In order to establish the first part of this theorem, we use the property proved in [25], namely that

$$\overline{Sub\ A} = Sup\ (\overline{Sub\ A}) \tag{28}$$

holds. Now the first part of Theorem 6.1 is a simple consequence of the second part.

Proof of the First Part of Theorem 6.1. By the second part, we know the right-hand side of (28) is regular, which means $\overline{Sub\ A}$ is regular. But since we established already that regularity is preserved under complement, also $Sub\ A$ must be regular. □

Finally we like to show that the Myhill-Nerode Theorem is also convenient for establishing the non-regularity of languages. For this we use the following version of the Continuation Lemma (see for example [23]).

Lemma 6.4 (Continuation Lemma). *If a language A is regular and a set B is infinite, then there exist two distinct strings x and y in B such that $x \approx_A y$.*

This lemma can be easily deduced from the Myhill-Nerode Theorem and the Pigeonhole Principle: Since A is regular, there can be only finitely many equivalence classes. Hence an infinite set must contain at least two strings that are in the same equivalence class, that is they need to be related by the Myhill-Nerode Relation.

Using this lemma, it is straightforward to establish that the language $A \overset{def}{=} \bigcup_n a^n\ @\ b^n$ is not regular (a^n stands for the strings consisting of n times the character a; similarly for b^n). For this consider the infinite set $B \overset{def}{=} \bigcup_n a^n$.

Lemma 6.5. *No two distinct strings in set B are Myhill-Nerode related by language A.*

Proof. After unfolding the definition of B, we need to establish that given $i \neq j$, the strings a^i and a^j are not Myhill-Nerode related by A. That means we have to show that $\forall z.\ a^i\ @\ z \in A = a^j\ @\ z \in A$ leads to a contradiction. Let us take b^i for z. Then we know $a^i\ @\ b^i \in A$. But since $i \neq j$, $a^j\ @\ b^i \notin A$. Therefore a^i and a^j cannot be Myhill-Nerode related by A, and we are done. $\square$

To conclude the proof of non-regularity for the language A, the Continuation Lemma and the lemma above lead to a contradiction assuming A is regular. Therefore the language A is not regular, as we wanted to show.

7. Conclusion and Related Work

In this paper we took the view that a regular language is one where there exists a regular expression that matches all of its strings. Regular expressions can conveniently be defined as a datatype in HOL-based theorem provers. For us it was therefore interesting to find out how far we can push this point of view. We have established in Isabelle/HOL both directions of the Myhill-Nerode Theorem.

Theorem 7.1 (The Myhill-Nerode Theorem)**.**
A language A is regular if and only if finite $(UNIV\,/\!/\!\approx_A)$.

Having formalised this theorem means we pushed our point of view quite far. Using this theorem we can obviously prove when a language is *not* regular—by establishing that it has infinitely many equivalence classes generated by the Myhill-Nerode Relation (this is usually the purpose of the Pumping Lemma [16]). We can also use it to establish the standard textbook results about closure properties of regular languages. Interesting is the case of closure under complement, because it seems difficult to construct a regular expression for the complement language by direct means. However the existence of such a regular expression can be easily proved using the Myhill-Nerode Theorem.

Our insistence on regular expressions for proving the Myhill-Nerode Theorem arose from the limitations of HOL, which is the logic underlying the popular theorem provers HOL4, HOLlight and Isabelle/HOL. In order to guarantee consistency, formalisations in HOL can only extend the logic with definitions that introduce a new concept in terms of already existing notions. A convenient definition for automata (based on graphs) uses a polymorphic type for the state nodes. This allows us to use the standard operation for disjoint union whenever we need to compose two automata. Unfortunately, we cannot use such a polymorphic definition in HOL as part of the definition for regularity of a language (a predicate over sets of strings). Consider for example the following attempt:

$$is\_regular\ A \overset{def}{=} \exists M(\alpha).\ is\_dfa\ (M) \wedge \mathcal{L}(M) = A$$

In this definifion, the definiens is polymorphic in the type of the automata M (indicated by dependency on the type-variable α), but the definiendum *is_regular* is not. Such definitions are excluded from HOL, because they can lead easily to inconsistencies (see [22] for a simple example). Also HOL does not contain type-quantifiers which would allow us to get rid of the polymorphism by quantifying over the type-variable α. Therefore when defining regularity in terms of automata, the only natural way out in HOL is to resort to state nodes with an identity, for example a natural number. Unfortunatly, the consequence is that we have to be careful when combining two automata so that there is no clash between two such states. This makes formalisations quite fiddly and rather unpleasant. Regular expressions proved much more convenient for reasoning in HOL since they can be defined as inductive datatype and a reasoning infrastructure comes for free. The definition of regularity in terms of regular expressions poses no problem at all for HOL. We showed in this paper that they can be used for establishing the central result in regular language theory—the Myhill-Nerode Theorem.

While regular expressions are convenient, they have some limitations. One is that there seems to be no method of calculating a minimal regular expression (for example in terms of length) for a regular language, like there is for automata. On the other hand, efficient regular expression matching, without using automata, poses no problem [20]. For an implementation of a simple regular expression matcher, whose correctness has been formally established, we refer the reader to Owens and Slind [21]. In our opinion, their formalisation is considerably slicker than for example the approach to regular expression matching taken in [14] and [28].

Our proof of the first direction is very much inspired by *Brzozowski's algebraic method* used to convert a finite automaton to a regular expression [6]. The close connection can be seen by considering the equivalence classes as the states of the minimal automaton for the regular language. However there are some subtle differences. Because our equivalence classes (or correspondingly states) arise from the Myhill-Nerode Relation, the most natural choice is to characterise each state with the set of strings starting from the initial state leading up to that state. Usually, however, the states are characterised as the strings starting from that state leading to the terminal states. The first choice has consequences about how the initial equational system is set up. We have the λ-term on our 'initial state', while Brzozowski has it on the terminal states. This means we also need to reverse the direction of Arden's Lemma. We have not found anything in the 'pencil-and-paper-reasoning' literature about our way of proving the first direction of the Myhill-Nerode Theorem, but it appears to be folklore.

We presented two proofs for the second direction of the Myhill-Nerode Theorem. One direct proof using tagging-functions and another using partial derivatives. This part of our work is where our method using regular expressions shines, because we can completely side-step the standard argument [16] where automata need to be composed. However, it is also the direction where we had to spend most of the 'conceptual' time, as our first proof based on tagging-functions is new for establishing the Myhill-Nerode Theorem. All standard proofs of this direction proceed by arguments over automata.

The indirect proof for the second direction arose from our interest in Brzozowski's derivatives for regular expression matching. While Brzozowski already established that there are only finitely many dissimilar derivatives for every regular expression, this result is not as straightforward to formalise in a theorem prover as one might wish. The reason is that the set of dissimilar derivatives is not defined inductively, but in terms of an ACI-equivalence relation. This difficulty prevented for example Krauss and Nipkow to prove termination of their equivalence checker for regular expressions [17]. Their checker is based on Brzozowski's derivatives and for their argument the lack of a formal proof of termination is not crucial (it merely lets them "sleep better" [17]). We expect that their development simplifies by using partial derivatives, instead of derivatives, and that the termination of the algorithm can be formally established (the main ingredient is Theorem 5.1). However, since partial derivatives use sets of regular expressions, one needs to carefully analyse whether the resulting algorithm is still executable. Given the existing infrastructure for executable sets in Isabelle/HOL [12], it should.

Our formalisation of the Myhill-Nerode Theorem consists of 780 lines of Isabelle/Isar code for the first direction and 460 for the second (the one based on tagging-functions), plus around 300 lines of standard material about regular languages. The formalisation of derivatives and partial derivatives shown in Section 5 consists of 390 lines of code. The closure properties in Section 6 (except Theorem 6.1) can be established in 190 lines of code. The Continuation Lemma and the non-regularity of $a^n\ b^n$ require 70 lines of code. The algorithm for solving equational systems, which we used in the first direction, is conceptually relatively simple. Still the use of sets over which the algorithm operates means it is not as easy to formalise as one might hope. However, it seems sets cannot be avoided since the 'input' of the algorithm consists of equivalence classes and we cannot see how to reformulate the theory so that we can use lists or matrices. Lists would be much easier to reason about, since we can define functions over them by recursion. For sets we have to use set-comprehensions, which is slightly unwieldy. Matrices would allow us to use the slick formalisation by Nipkow of the Gauss-Jordan algorithm [19].

While our formalisation might appear large, it should be seen in the context of the work done by Constable at al [9] who formalised the Myhill-Nerode Theorem in Nuprl using automata. They write that their four-member team needed something on the magnitude of 18 months for their formalisation. It is hard to gauge the size of a formalisation in Nurpl, but from what is shown in the Nuprl Math Library about their development it seems substantially larger than ours. We attribute this to our use of regular expressions, which meant we did not need to 'fight' the theorem prover. Also, Filliâtre reports that his formalisation in Coq of automata theory and Kleene's theorem is "rather big" [11]. More recently, Almeida et al reported about another formalisation of regular languages in Coq [1]. Their main result is the correctness of Mirkin's construction of an automaton from a regular expression using partial derivatives. This took approximately 10600 lines of code. In terms of time, the estimate for our formalisation is that we needed approximately 3 months and this included the time to find our proof arguments. Unlike Constable et al, who were able to follow the Myhill-Nerode proof from [15], we had to find our own arguments. So for us the formalisation was not the bottleneck. The code of our formalisation can be found in the Archive of Formal Proofs at http://afp.sourceforge.net/devel-entries/Myhill-Nerode.shtml [27].

Acknowledgements: We are grateful for the comments we received from Larry Paulson. Tobias Nipkow made us aware of the properties in Theorem 6.1 and Tjark Weber helped us with proving them.

References

[1] J. B. Almeida, N. Moriera, D. Pereira, and S. M. de Sousa. Partial Derivative Automata Formalized in Coq. In *Proc. of the 15th International Conference on Implementation and Application of Automata*, volume 6482 of *LNCS*, pages 59–68, 2010.

[2] V. Antimirov. Partial Derivatives of Regular Expressions and Finite Automata Constructions. *Theoretical Computer Science*, 155:291–319, 1995.

[3] S. Berghofer. A Constructive Proof of Higman's Lemma in Isabelle. In *In Proc. of the Workshop on Types*, volume 3085 of *LNCS*, pages 66–82, 2003.

[4] S. Berghofer and T. Nipkow. Executing Higher Order Logic. In *Proc. of the International Workshop on Types for Proofs and Programs*, volume 2277 of *LNCS*, pages 24–40, 2002.

[5] S. Berghofer and M. Reiter. Formalizing the Logic-Automaton Connection. In *Proc. of the 22nd International Conference on Theorem Proving in Higher Order Logics*, volume 5674 of *LNCS*, pages 147–163, 2009.

[6] J. A. Brzozowski. Derivatives of Regular Expressions. *Journal of the ACM*, 11(4):481–494, 1964.

[7] J. A. Brzozowski. Quotient Complexity of Regular Languages. *Journal of Automata, Languages and Combinatorics*, 15(1/2):71–89, 2010.

[8] A. Church. A Formulation of the Simple Theory of Types. *Journal of Symbolic Logic*, 5(2):56–68, 1940.

[9] R. L. Constable, P. B. Jackson, P. Naumov, and J. C. Uribe. Constructively Formalizing Automata Theory. In *Proof, Language, and Interaction*, pages 213–238. MIT Press, 2000.

[10] S. A. Fenner, W. I. Gasarch, and B. Postow. The Complexity of Finding SUBSEQ(A). *Theory of Computing Systems*, 45(3):577–612, 2009.

[11] J.-C. Filliâtre. Finite Automata Theory in Coq: A Constructive Proof of Kleene's Theorem. Research Report 97–04, LIP - ENS Lyon, 1997.

[12] F. Haftmann. *Code Generation from Specifications in Higher-Order Logic*. PhD thesis, Technical University of Munich, 2009.

[13] L. H. Haines. On Free Monoids Partially Ordered by Embedding. *Journal of Combinatorial Theory*, 6:94–98, 1969.

[14] R. Harper. Proof-Directed Debugging. *Journal of Functional Programming*, 9(4):463–469, 1999.

[15] J. E. Hopcroft and J. D. Ullman. *Formal Languages and Their Relation to Automata*. Addison-Wesley, 1969.

[16] D. Kozen. *Automata and Computability*. Springer Verlag, 1997.

[17] A. Krauss and T. Nipkow. Proof Pearl: Regular Expression Equivalence and Relation Algebra. To appear in Journal of Automated Reasoning, 2011.

[18] T. Nipkow. Verified Lexical Analysis. In *Proc. of the 11th International Conference on Theorem Proving in Higher Order Logics*, volume 1479 of *LNCS*, pages 1–15, 1998.

[19] T. Nipkow. Gauss-Jordan Elimination for Matrices Represented as Functions. In G. Klein, T. Nipkow, and L. Paulson, editors, *The Archive of Formal Proofs*. http://afp.sourceforge.net/entries/Gauss-Jordan-Elim-Fun.shtml, 2011. Formal proof development.

[20] S. Owens, J. Reppy, and A. Turon. Regular-Expression Derivatives Re-Examined. *Journal of Functional Programming*, 19(2):173–190, 2009.

[21] S. Owens and K. Slind. Adapting Functional Programs to Higher Order Logic. *Higher-Order and Symbolic Computation*, 21(4):377–409, 2008.

[22] A. M. Pitts. *Syntax and Semantics*. Part of the documentation for the HOL4 system.

[23] A. L. Rosenberg. A Big Ideas Approach to the Theory of Computation. Course notes for CMPSCI 401 at the University of Massachusetts, 2006.

[24] J. Sakarovitch. *Elements of Automata Theory*. Cambridge University Press, 2009.

[25] J. Shallit. *A Second Course in Formal Languages and Automata Theory*. Cambridge University Press, 2008.

[26] C. Wu, X. Zhang, and C. Urban. A Formalisation of the Myhill-Nerode Theorem based on Regular Expressions (Proof Pearl). In *Proc. of the 2nd International Conference on Interactive Theorem Proving*, volume 6898 of *LNCS*, pages 341–356, 2011.

[27] C. Wu, X. Zhang, and C. Urban. The Myhill-Nerode Theorem based on Regular Expressions. In G. Klein, T. Nipkow, and L. Paulson, editors, *The Archive of Formal Proofs*. http://afp.sourceforge.net/devel-entries/Myhill-Nerode.shtml, August 2011. Formal proof development.

[28] K. Yi. Educational Pearl: 'Proof-Directed Debugging' Revisited for a First-Order Version. *Journal of Functional Programming*, 16(6):663–670, 2006.

Communicated by (The editor will be set by the publisher).
(The dates will be set by the publisher).