A Formalisation of the Myhill-Nerode Theorem based on Regular Expressions (Proof Pearl)

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

[1] PLA University of Science and Technology, China
[2] TU Munich, Germany

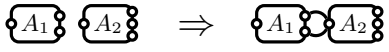
Abstract. There are numerous textbooks on regular languages. Nearly all of them introduce the subject by describing finite automata and only mentioning on the side a connection with regular expressions. Unfortunately, automata are difficult to formalise in HOL-based theorem provers. The reason is that they need to be represented as graphs, matrices or functions, none of which are inductive datatypes. Also convenient operations for disjoint unions of graphs 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.

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. [7]). It seems natural to exercise theorem provers by formalising the theorems and by verifying formally the algorithms.

There is however a problem: the typical approach to regular languages is to introduce finite automata and then define everything in terms of them. For example, a regular language is normally defined as one whose strings are recognised by a finite deterministic automaton. 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, in our case Isabelle/HOL. 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. 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

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

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 automata and hence will not be able to state certain properties about *all* automata, since there is no type quantification available in HOL (unlike in Coq, for example). An alternative, which provides us with a single type for 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.

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 [9] dismisses for this the option of using identities, because it leads according to him to "messy proofs". He 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 [9] establishes the link between regular expressions and automata in the context of lexing. Berghofer and Reiter [2] formalise automata working over bit strings in the context of Presburger arithmetic. The only larger formalisations of automata theory are carried out in Nuprl [4] and in Coq [5].

In this paper, we will not attempt to formalise automata theory in Isabelle/HOL, but take a different approach to regular languages. Instead of defining a regular language as one where there exists an automaton that recognises all strings of the language, we define a regular language as:

Definition 1. *A language A is* regular*, provided there is a regular expression that matches all strings of A.*

The reason is that regular expressions, unlike graphs, matrices and functions, can be easily defined as inductive datatype. Consequently a corresponding reasoning infrastructure comes for free. This has recently been exploited in HOL4 with a formalisation of regular expression matching based on derivatives [11] and with an equivalence

checker for regular expressions in Isabelle/HOL [8]. The 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.

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. We prove the part of this theorem stating that a regular expression has only finitely many partitions using certain tagging-functions. Again to our best knowledge, these tagging-functions have not been used before to establish 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 $[]$. *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

$$A \cdot B \stackrel{def}{=} \{s_1 @ s_2 \mid s_1 \in A \wedge s_2 \in B\} \qquad A^0 \stackrel{def}{=} \{[]\} \qquad A^{n+1} \stackrel{def}{=} A \cdot A^n$$

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

Proposition 1.
(i) $A^\star = \{[]\} \cup A \cdot A^\star$
(ii) If $[] \notin A$ *and* $s \in A^{n+1}$ *then* $n < |s|$.
(iii) $B \cdot (\bigcup_n A^n) = (\bigcup_n B \cdot A^n)$

In (ii) we use the notation $|s|$ 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. We omit the proofs for these properties, but invite the reader to consult our formalisation.[3]

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 $\{y \mid y \approx x\}$.

Central to our proof will be the solution of equational systems involving equivalence classes of languages. For this we will use Arden's Lemma [3], which solves equations of the form $X = A \cdot X \cup B$ provided $[] \notin A$. However we will need the following 'reverse' version of Arden's Lemma ('reverse' in the sense of changing the order of $A \cdot X$ to $X \cdot A$).

[3] Available at http://www4.in.tum.de/~urbanc/regexp.html

Lemma 1 (Reverse 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 Prop. 1(i) we have $A^\star = \{[]\} \cup A \cdot A^\star$, which is equal to $A^\star = \{[]\} \cup A^\star \cdot A$. Adding B to both sides gives $B \cdot A^\star = B \cdot (\{[]\} \cup A^\star \cdot A)$, whose right-hand side is equal to $(B \cdot A^\star) \cdot A \cup B$. This 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

$$(*) \qquad X = X \cdot A^{n+1} \cup (\bigcup_{m \in \{0..n\}} B \cdot A^m)$$

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 Prop. 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 $\bigcup_{m \in \{0..k\}} B \cdot A^m$. This in turn implies that s is in $\bigcup_n B \cdot A^n$. Using Prop. 1(iii) this is equal to $B \cdot A^\star$, as we needed to show. □

Regular expressions are defined as the inductive datatype

$$r ::= NULL \mid EMPTY \mid CHAR\ c \mid SEQ\ r\ r \mid ALT\ r\ r \mid STAR\ r$$

and the language matched by a regular expression is defined as

$$\begin{array}{rcl@{\qquad}rcl}
\mathcal{L}(NULL) & \stackrel{def}{=} & \varnothing & \mathcal{L}(SEQ\ r_1\ r_2) & \stackrel{def}{=} & \mathcal{L}(r_1) \cdot \mathcal{L}(r_2) \\
\mathcal{L}(EMPTY) & \stackrel{def}{=} & \{[]\} & \mathcal{L}(ALT\ r_1\ r_2) & \stackrel{def}{=} & \mathcal{L}(r_1) \cup \mathcal{L}(r_2) \\
\mathcal{L}(CHAR\ c) & \stackrel{def}{=} & \{[c]\} & \mathcal{L}(STAR\ r) & \stackrel{def}{=} & \mathcal{L}(r)^\star
\end{array}$$

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 ALT over the set rs with $NULL$ 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 $\mathcal{L}$.

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

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

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

$$X_1 = \{[]\} \qquad X_2 = \{[c]\} \qquad X_3 = \mathit{UNIV} - \{[], [c]\}$$

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 1. *If finite* $(\mathit{UNIV} /\!\!/ \approx_A)$ *then* $\exists r.\ A = \mathcal{L}(r)$.

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

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

In our running example, X_2 is the only equivalence class in *finals* $\{[c]\}$. It is straightforward to show that in general $A = \bigcup \mathit{finals}\ A$ and $\mathit{finals}\ A \subseteq \mathit{UNIV} /\!\!/ \approx_A$ 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 Thm. 1 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

$$Y \stackrel{c}{\Longrightarrow} X \stackrel{def}{=} Y \cdot \{[c]\} \subseteq X \tag{4}$$

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 $X_1 \stackrel{c}{\Longrightarrow} X_2$, $X_1 \stackrel{d}{\Longrightarrow} X_3$ with d being any other character than c, and $X_3 \stackrel{d}{\Longrightarrow} X_3$ for any d.

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

$$\begin{array}{l} X_1 = (Y_{11}, \mathit{CHAR}\ c_{11}) + \ldots + (Y_{1p}, \mathit{CHAR}\ c_{1p}) + \lambda(\mathit{EMPTY}) \\ X_2 = (Y_{21}, \mathit{CHAR}\ c_{21}) + \ldots + (Y_{2o}, \mathit{CHAR}\ c_{2o}) \\ \quad \vdots \\ X_n = (Y_{n1}, \mathit{CHAR}\ c_{n1}) + \ldots + (Y_{nq}, \mathit{CHAR}\ c_{nq}) \end{array}$$

where the terms $(Y_{ij}, \mathit{CHAR}\ c_{ij})$ stand for all transitions $Y_{ij} \stackrel{c_{ij}}{\Longrightarrow} X_i$. There can only be finitely many terms of the form $(Y_{ij}, \mathit{CHAR}\ c_{ij})$ in a right-hand side since by assumption there are only finitely many equivalence classes and only finitely many characters. The term $\lambda(\mathit{EMPTY})$ in the first equation acts as a marker for the initial state, that is the equivalence class containing $[]$.[4] Overloading the function $\mathcal{L}$ for the two kinds of terms in the equational system, we have

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

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

$$X_i = \mathcal{L}(Y_{i1}, \mathit{CHAR}\ c_{i1}) \cup \ldots \cup \mathcal{L}(Y_{iq}, \mathit{CHAR}\ c_{iq}). \tag{5}$$

hold. Similarly for X_1 we can show the following equation

$$X_1 = \mathcal{L}(Y_{11}, \mathit{CHAR}\ c_{11}) \cup \ldots \cup \mathcal{L}(Y_{1p}, \mathit{CHAR}\ c_{1p}) \cup \mathcal{L}(\lambda(\mathit{EMPTY})). \tag{6}$$

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

$$\begin{array}{rcl} \mathit{Init\_rhs}\ \mathit{CS}\ X & \stackrel{\mathit{def}}{=} & \mathit{if}\ [] \in X \\ & & \mathit{then}\ \{(Y, \mathit{CHAR}\ c) \mid Y \in \mathit{CS} \wedge Y \stackrel{c}{\Longrightarrow} X\} \cup \{\lambda(\mathit{EMPTY})\} \\ & & \mathit{else}\ \{(Y, \mathit{CHAR}\ c) \mid Y \in \mathit{CS} \wedge Y \stackrel{c}{\Longrightarrow} X\} \\ \mathit{Init}\ \mathit{CS} & \stackrel{\mathit{def}}{=} & \{(X, \mathit{Init\_rhs}\ \mathit{CS}\ X) \mid X \in \mathit{CS}\} \end{array} \tag{7}$$

Because we use sets of terms for representing the right-hand sides of equations, we can prove (5) and (6) more concisely as

Lemma 2. *If* $(X, \mathit{rhs}) \in \mathit{Init}\ (\mathit{UNIV} /\!\!/ \approx_A)$ *then* $X = \bigcup\ \mathcal{L}\ `\ \mathit{rhs}$.

Our proof of Thm. 1 will proceed by transforming the initial equational system into one in *solved form* maintaining the invariant in Lem. 2. 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

[4] Note that we mark, roughly speaking, the single 'initial' state in the equational system, which is different from the method by Brzozowski [3], 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 reverse version of Arden's Lemma.

an equation $X = rhs$ 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

$$(Y, r_2) \triangleleft r_1 \stackrel{def}{=} (Y, SEQ\ r_2\ r_1) \qquad \lambda(r_2) \triangleleft r_1 \stackrel{def}{=} \lambda(SEQ\ r_2\ r_1)$$

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

$$\begin{array}{lll} Arden\ X\ rhs & \stackrel{def}{=} & let \\ & & \quad rhs' = rhs - \{(X, r) \mid (X, r) \in rhs\} \\ & & \quad\ r' = STAR\ (\biguplus\{r \mid (X, r) \in rhs\}) \\ & & in\ \ rhs' \triangleleft r' \end{array} \tag{8}$$

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'. It can be easily seen that this 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

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

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

Lemma 3. *Given an equation $X = rhs$. If $X = \bigcup \mathcal{L}\ `\ rhs$, ardenable rhs, and finite rhs, then $X = \bigcup \mathcal{L}\ `\ (Arden\ X\ rhs)$.*

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

$$\begin{array}{lll} Subst\ rhs\ X\ xrhs & \stackrel{def}{=} & let \\ & & \quad rhs' = rhs - \{(X, r) \mid (X, r) \in rhs\} \\ & & \quad\ r' = \biguplus\{r \mid (X, r) \in rhs\} \\ & & in\ \ rhs' \cup (xrhs \triangleleft r') \end{array}$$

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.

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 $X = xrhs$ 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*.

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

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.

$$\begin{array}{l} \mathit{Iter}\ X\ \mathit{ES} \stackrel{\mathit{def}}{=} \mathit{let} \\ \qquad (Y, \mathit{yrhs}) = \mathit{SOME}\ (Y, \mathit{yrhs}).\ (Y, \mathit{yrhs}) \in \mathit{ES} \wedge X \neq Y \\ \mathit{in}\ \ \mathit{Remove}\ \mathit{ES}\ Y\ \mathit{yrhs} \end{array}$$

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:

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

We are not concerned here with the definition of this operator (see Berghofer and Nipkow [1]), 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 (7) using the principle:

$$\frac{\begin{array}{l} \mathit{invariant}\ (\mathit{Init}\ (\mathit{UNIV} /\!\!/ \approx_A)) \\ \forall\, \mathit{ES}.\ \mathit{invariant}\ \mathit{ES} \wedge \mathit{Cond}\ \mathit{ES} \longrightarrow \mathit{invariant}\ (\mathit{Iter}\ X\ \mathit{ES}) \\ \forall\, \mathit{ES}.\ \mathit{invariant}\ \mathit{ES} \wedge \mathit{Cond}\ \mathit{ES} \longrightarrow \mathit{card}\ (\mathit{Iter}\ X\ \mathit{ES}) < \mathit{card}\ \mathit{ES} \\ \forall\, \mathit{ES}.\ \mathit{invariant}\ \mathit{ES} \wedge \neg\, \mathit{Cond}\ \mathit{ES} \longrightarrow P\ \mathit{ES} \end{array}}{P\ (\mathit{Solve}\ X\ (\mathit{Init}\ (\mathit{UNIV} /\!\!/ \approx_A)))} \tag{9}$$

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 $\mathit{Solve}\ X\ (\mathit{Init}\ (\mathit{UNIV} /\!\!/ \approx_A))$ returns with a single equation $X = \mathit{xrhs}$ 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:

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

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 $\mathit{lhss}\ ES \stackrel{def}{=} \{X \mid (X, \mathit{rhs}) \in ES\}$ and $\mathit{rhss}\ \mathit{rhs} \stackrel{def}{=} \{X \mid (X, r) \in \mathit{rhs}\}$.

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

Lemma 4. *If finite* $(\mathit{UNIV} /\!\!/ \approx_A)$ *then invariant* $(\mathit{Init}\ (\mathit{UNIV} /\!\!/ \approx_A))$.

Proof. Finiteness is given by the assumption and the way how we set up the initial equational system. Soundness is proved in Lem. 2. 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 5. *If invariant ES and* $(X, \mathit{rhs}) \in ES$ *and Cond ES then invariant* $(\mathit{Iter}\ X\ ES)$.

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

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

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 Lem. 3). 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 $ES - \{(Y, \mathit{yrhs})\}$ which matches with our proof-obligation of *Subst_all*. Since $ES = ES - \{(Y, \mathit{yrhs})\} \cup \{(Y, \mathit{yrhs})\}$, we can use the assumption to complete the proof. □

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

Lemma 6. *If invariant ES and* $(X, rhs) \in ES$ *and Cond ES then card* $(Iter\ X\ ES) <$ *card ES*.

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

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

Lemma 7. *If finite* $(UNIV /\!\!/ \approx_A)$ *and* $X \in UNIV /\!\!/ \approx_A$ *then there exists a rhs such that Solve X* $(Init\ (UNIV /\!\!/ \approx_A)) = \{(X, rhs)\}$ *and invariant* $\{(X, rhs)\}$.

Proof. In order to prove this lemma using (9), we have to use a slightly stronger invariant since Lem. 5 and 6 have the precondition that $(X, rhs) \in ES$ for some *rhs*. This precondition is needed in order to choose in the *Iter*-step an equation that is not $X = rhs$. Therefore our invariant cannot be just *invariant ES*, but must be *invariant ES* $\wedge$ $(\exists\, rhs.$ $(X, rhs) \in ES)$. By assumption $X \in UNIV /\!\!/ \approx_A$ and Lem. 4, the more general invariant holds for the initial equational system. This is premise 1 of (9). Premise 2 is given by Lem. 5 and the fact that *Iter* might modify the *rhs* in the equation $X = rhs$, but does not remove it. Premise 3 of (9) is by Lem. 6. Now in premise 4 we like to show that there exists a *rhs* such that $ES = \{(X, rhs)\}$ and that *invariant* $\{(X, rhs)\}$ holds, provided the condition *Cond* does not holds. By the stronger invariant we know there exists such a *rhs* with $(X, rhs) \in ES$. Because *Cond* is not true, we know the cardinality of *ES* is *1*. This means *ES* must actually be the set $\{(X, rhs)\}$, for which the invariant holds. This allows us to conclude that *Solve X* $(Init\ (UNIV /\!\!/ \approx_A)) = \{(X, rhs)\}$ and *invariant* $\{(X, rhs)\}$ hold, as needed. □

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

Lemma 8. *If finite* $(UNIV /\!\!/ \approx_A)$ *and* $X \in UNIV /\!\!/ \approx_A$ *then* $\exists\, r.\ X = \mathcal{L}(r)$.

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 Lem. 3. 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 $rhss\ (Arden\ X\ rhs) = \varnothing$. 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 $X = \mathcal{L}(\sum rs)$. With this we can conclude the proof. □

Lem. 8 allows us to finally give a proof for the first direction of the Myhill-Nerode theorem.

Proof (of Thm. 1). By Lem. 8 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 $\bigcup finals\ A = \mathcal{L}(\uplus rs)$. Since the left-hand side is equal to A, we can use $\uplus rs$ as the regular expression that is needed in the theorem. □

4 Myhill-Nerode, Second Part

We will prove in this section the second part of the Myhill-Nerode theorem. It can be formulated in our setting as follows:

Theorem 2. *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.

Proof (Base Cases). The cases for *NULL*, *EMPTY* and *CHAR* are routine, because we can easily establish that

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

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

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

Our proof 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 $range\ f \stackrel{def}{=} f\ `\ UNIV$). With this we will be able to infer that the tagging-functions, seen as relations, give rise to finitely many equivalence classes of *UNIV*. 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 (a relation R_1 is said to *refine* R_2 provided $R_1 \subseteq R_2$). We formally define the notion of a *tagging-relation* as follows.

Definition 3 (Tagging-Relation). *Given a tagging-function tag, then two strings x and y are* tag-related *provided*

$$x =tag= y \stackrel{def}{=} tag\ x = tag\ y\ .$$

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

$$If\ finite\ (f\ `\ A)\ \ and\ \ inj\_on\ f\ A\ \ then\ \ finite\ A. \tag{10}$$

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 9. *If finite* $(range\ tag)$ *then finite* $(UNIV /\!\!/ =tag=)$.

Proof. We set in (10), f to be $X \mapsto tag\ `\ X$. We have *range* f to be a subset of *Pow* $(range\ tag)$, which we know must be finite by assumption. Now $f\ (UNIV /\!\!/ =tag=)$ is a subset of *range* f, and so also finite. Injectivity amounts to showing that $X = Y$ under the assumptions that $X, Y \in UNIV /\!\!/ =tag=$ and $f\,X = f\,Y$. From the assumptions we can obtain $x \in X$ and $y \in Y$ with $tag\ x = tag\ y$. Since x and y are tag-related, this in turn means that the equivalence classes X and Y must be equal. □

Lemma 10. *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 (10). This time we set f to be $X \mapsto \{[\![x]\!]_{R_1} \mid x \in X\}$. It is easy to see that *finite* $(f\ `\ UNIV /\!\!/ R_2)$ because it is a subset of *Pow* $(UNIV /\!\!/ R_1)$, which is 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. □

Chaining Lem. 9 and 10 together, means in order to show that $UNIV /\!\!/ \approx_{\mathcal{L}(r)}$ is finite, we have to find a tagging-function whose range can be shown to be finite and whose tagging-relation refines $\approx_{\mathcal{L}(r)}$. Let us attempt the *ALT*-case first.

Proof (ALT-Case). We take as tagging-function

$$tag_{ALT}\ A\ B\ x \stackrel{def}{=} ([\![x]\!]_{\approx A}, [\![x]\!]_{\approx B})$$

where A and B are some arbitrary languages. 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 $tag_{ALT}\ A\ B$ is a subset of this product set—so finite. It remains to be shown that $=tag_{ALT}\ A\ B=$ refines $\approx_{A \cup B}$. This amounts to showing

$$tag_{ALT}\ A\ B\ x = tag_{ALT}\ A\ B\ y \longrightarrow x \approx_{A \cup B} y$$

which by unfolding the Myhill-Nerode relation is identical to

$$\forall z.\ tag_{ALT}\ A\ B\ x = tag_{ALT}\ A\ B\ y \wedge x \mathbin{@} z \in A \cup B \longrightarrow y \mathbin{@} z \in A \cup B \qquad (11)$$

since both $=tag_{ALT}\ A\ B=$ and $\approx_{A \cup B}$ are symmetric. To solve (11) we just have to unfold the definition of the tagging-function and analyse in which set, A or B, the string $x \mathbin{@} z$ is. The definition of the tagging-function will give us in each case the information to infer that $y \mathbin{@} z \in A \cup B$. Finally we can discharge this case by setting A to $\mathcal{L}(r_1)$ and B to $\mathcal{L}(r_2)$. □

The pattern in (11) is repeated for the other two cases. Unfortunately, they are slightly more complicated. In the *SEQ*-case we essentially have to be able to infer that

$$\ldots x \mathbin{@} z \in A \cdot B \longrightarrow y \mathbin{@} z \in A \cdot B$$

using the information given by the appropriate tagging-function. The complication is to find out what the possible splits of $x \mathbin{@} z$ are to be in $A \cdot B$ (this was easy in case of $A \cup B$). To deal with this complication we define the notions of *string prefixes*

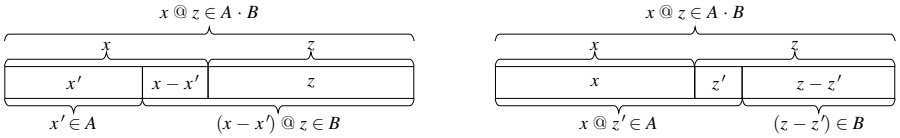
$$x \leq y \stackrel{def}{=} \exists z.\, y = x \mathbin{@} z \qquad x < y \stackrel{def}{=} x \leq y \wedge x \neq y$$

and *string subtraction*:

$$[] - y \stackrel{def}{=} [] \qquad x - [] \stackrel{def}{=} x \qquad cx - dy \stackrel{def}{=} \textit{if } c = d \textit{ then } x - y \textit{ else } cx$$

where c and d are characters, and x and y are strings.

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



Either there is a prefix of x in A and the rest is in B (first picture), or x and a prefix of z is in A and the rest in B (second picture). In both cases we have to show that $y \mathbin{@} z \in A \cdot B$. For this we use the following tagging-function

$$tag_{SEQ}\ A\ B\ x \stackrel{def}{=} (\llbracket x \rrbracket_{\approx_A}, \{\llbracket (x - x') \rrbracket_{\approx_B} \mid x' \leq x \wedge x' \in A\})$$

with the idea that in the first split we have to make sure that $(x - x') \mathbin{@} z$ is in the language B.

Proof (SEQ-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 $tag_{SEQ}\ A\ B$ is a subset of this product set, and therefore finite. We have to show injectivity of this tagging-function as

$$\forall z.\, tag_{SEQ}\ A\ B\ x = tag_{SEQ}\ A\ B\ y \wedge x \mathbin{@} z \in A \cdot B \longrightarrow y \mathbin{@} z \in A \cdot B$$

There are two cases to be considered (see pictures above). First, there exists a x' such that $x' \in A$, $x' \leq x$ and $(x - x') \mathbin{@} z \in B$ hold. We therefore have

$$\llbracket (x - x') \rrbracket_{\approx_B} \in \{\llbracket (x - x') \rrbracket_{\approx_B} \mid x' \leq x \wedge x' \in A\}$$

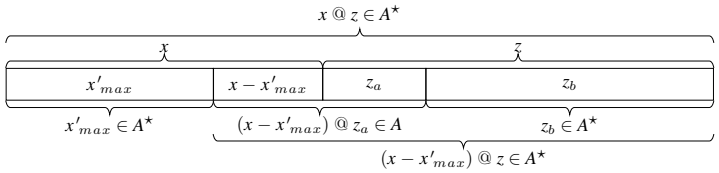
and by the assumption about $tag_{SEQ}\ A\ B$ also

$$\llbracket (x - x') \rrbracket_{\approx_B} \in \{\llbracket (y - y') \rrbracket_{\approx_B} \mid y' \leq y \wedge y' \in A\}$$

That means there must be a y' such that $y' \in A$ and $\llbracket (x - x') \rrbracket_{\approx_B} = \llbracket (y - y') \rrbracket_{\approx_B}$. This equality means that $x - x' \approx_B y - y'$ holds. Unfolding the Myhill-Nerode relation and together with the fact that $(x - x') \mathbin{@} z \in B$, we have $(y - y') \mathbin{@} z \in B$. We already know $y' \in A$, therefore $y \mathbin{@} z \in A \cdot B$, as needed in this case.

Second, there exists a z' such that $x \mathbin{@} z' \in A$ and $z - z' \in B$. By the assumption about $tag_{SEQ}\ A\ B$ we have $\llbracket x \rrbracket_{\approx_A} = \llbracket y \rrbracket_{\approx_A}$ and thus $x \approx_A y$. Which means by the Myhill-Nerode relation that $y \mathbin{@} z' \in A$ holds. Using $z - z' \in B$, we can conclude also in this case with $y \mathbin{@} z \in A \cdot B$. We again can complete the *SEQ*-case by setting A to $\mathcal{L}(r_1)$ and B to $\mathcal{L}(r_2)$. □

The case for *STAR* is similar to *SEQ*, but poses a few extra challenges. When we analyse the case that $x \mathbin{@} z$ is an element in $A^\star$ and x is not the empty string, we have the following picture:



We can find a strict prefix x' of x such that $x' \in A^\star$, $x' < x$ and the rest $(x - x') \mathbin{@} z \in A^\star$. For example the empty string [] would do. 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 ${x'}_{max}$. Now for the rest of the string $(x - {x'}_{max}) \mathbin{@} z$ we know it is in $A^\star$. By definition of $A^\star$, we can separate this string into two parts, say a and b, such that $a \in A$ and $b \in A^\star$. Now a must be strictly longer than $x - {x'}_{max}$, otherwise ${x'}_{max}$ 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 $(x - {x'}_{max}) \mathbin{@} z_a \in A$ and $z_b \in A^\star$. To cut a story short, we have divided $x \mathbin{@} z \in A^\star$ such that we have a string a with $a \in A$ that lies just on the 'border' of x and z. This string is $(x - {x'}_{max}) \mathbin{@} z_a$.

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

$$tag_{STAR}\, A\, x \stackrel{def}{=} \{[\![(x - x')]\!]_{\approx_A} \mid x' < x \wedge x' \in A^\star\}$$

Proof (STAR-Case). If *finite* $(UNIV /\!\!/ \approx_A)$ then *finite* $(Pow\ (UNIV /\!\!/ \approx_A))$ holds. The range of $tag_{STAR}\, A$ is a subset of this set, and therefore finite. Again we have to show injectivity of this tagging-function as

$$\forall z.\ tag_{STAR}\, A\, x = tag_{STAR}\, A\, y \wedge x \mathbin{@} z \in A^\star \longrightarrow y \mathbin{@} z \in A^\star$$

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

$$[\![(x - {x'}_{max})]\!]_{\approx_A} \in \{[\![(x - x')]\!]_{\approx_A} \mid x' < x \wedge x' \in A^\star\}$$

which by assumption is equal to

$$[\![(x - {x'}_{max})]\!]_{\approx_A} \in \{[\![(y - y')]\!]_{\approx_A} \mid y' < y \wedge y' \in A^\star\}$$

and we know that we have a $y' \in A^\star$ and $y' < y$ and also know $x - {x'}_{max} \approx_A y - y'$. Unfolding the Myhill-Nerode relation we know $(y - y') \mathbin{@} z_a \in A$. We also know that $z_b \in A^\star$. Therefore $y' \mathbin{@} ((y - y') \mathbin{@} z_a) \mathbin{@} z_b \in A^\star$, which means $y \mathbin{@} z \in A^\star$. As the last step we have to set A to $\mathcal{L}(r)$ and complete the proof. □

5 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 3 (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 [7]). 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 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. Proving the existence of 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.

While regular expressions are convenient in formalisations, 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 [10]. For an implementation of a simple regular expression matcher, whose correctness has been formally established, we refer the reader to Owens and Slind [11].

Our formalisation consists of 780 lines of Isabelle/Isar code for the first direction and 460 for the second, plus around 300 lines of standard material about regular languages. While this might be seen as too large to count as a concise proof pearl, this should be seen in the context of the work done by Constable at al [4] 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. 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 proofs from [6], we had to find our own arguments. So for us the formalisation was not the bottleneck. 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. The code of ours can be found in the Mercurial Repository at http://www4.in.tum.de/~urbanc/regexp.html.

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 [3]. The close con-

nection 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. Since we identify equivalence classes with the states of the automaton, then 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 briefly considered using the method Brzozowski presented in the Appendix of [3] in order to prove the second direction of the Myhill-Nerode theorem. There he calculates the derivatives for regular expressions and shows that for every language there can be only finitely many of them (if regarded equal modulo ACI). We could have used as tagging-function the set of derivatives of a regular expression with respect to a language. Using the fact that two strings are Myhill-Nerode related whenever their derivative is the same, together with the fact that there are only finitely such derivatives would give us a similar argument as ours. However it seems not so easy to calculate the set of derivatives modulo ACI. Therefore we preferred our direct method of using tagging-functions. This is also where our method shines, because we can completely side-step the standard argument [7] where automata need to be composed, which as stated in the Introduction is not so easy to formalise in a HOL-based theorem prover. However, it is also the direction where we had to spend most of the 'conceptual' time, as our proof-argument based on tagging-functions is new for establishing the Myhill-Nerode theorem. All standard proofs of this direction use arguments over automata.

References

1. 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.
2. 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.
3. J. A. Brzozowski. Derivatives of Regular Expressions. *J. ACM*, 11:481–494, 1964.
4. 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.
5. J.-C. Filliâtre. Finite Automata Theory in Coq: A Constructive Proof of Kleene's Theorem. Research Report 97–04, LIP - ENS Lyon, 1997.
6. J. E. Hopcroft and J. D. Ullman. *Formal Languages and Their Relation to Automata*. Addison-Wesley, 1969.
7. D. Kozen. *Automata and Computability*. Springer Verlag, 1997.
8. A. Kraus and T. Nipkow. Proof Pearl: Regular Expression Equivalence and Relation Algebra. To appear in Journal of Automated Reasoning, 2011.
9. 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.
10. S. Owens, J. Reppy, and A. Turon. Regular-Expression Derivatives Re-Examined. *Journal of Functional Programming*, 19(2):173–190, 2009.
11. S. Owens and K. Slind. Adapting Functional Programs to Higher Order Logic. *Higher-Order and Symbolic Computation*, 21(4):377–409, 2008.