# Constructively Formalizing Automata Theory*

Robert L. Constable †
Paul B. Jackson ‡
Pavel Naumov†
Juan Uribe†

November 18, 1997

**Abstract**

We present a constructive formalization of the Myhill-Nerode theorem on the minimization of finite automata that follows the account in Hopcroft and Ullman's book *Formal Languages and Their Relation to Automata*. We chose to formalize this theorem because it illustrates many points critical to formalization of computational mathematics, especially the extraction of an important algorithm from a proof as a method of knowing that the algorithm is correct. It also gave us an opportunity to experiment with a constructive implementation of quotient sets.

We carried out the formalization in Nuprl, an interactive theorem prover based on constructive type theory. Nuprl borrows an implementation of the ML language from the LCF system of Milner, Gordon, and Wadsworth, and makes heavy use of the notion of *tactic* pioneered by Milner in LCF.

We are interested in the pedagogical value of electronic formal mathematical texts and have put our formalization on the World Wide Web. Readers are invited to judge whether the formalization adds value in comparison to a careful informal account.

**Key Words and Phrases:** automata, constructivity, congruence, equivalence relation, formal languages, foundational logic, LCF, logic, Martin-Löf semantics, Myhill-Nerode theorem, Nuprl, program extraction, propositions-as-types, quotient types, regular languages, state minimization, tactics, theorem prover, type theory.

*Supported in part by NSF grants CCR-9423687, DUE-955162, and ONR grant N00014-92-J-1764.

†Department of Computer Science, Cornell University
‡Laboratory for Foundations of Computer Science, University of Edinburgh

# 1 Introduction

## 1.1 Background

It is widely believed that we know how to formalize large tracts of *classical* mathematics — namely write in the style of Bourbaki [4] using some version of set theory and fill in all the details. Indeed, the *Journal of Formalized Mathematics* publishes results formalized in set theory and checked by the Mizar system. Despite this belief and the many formalizations accomplished, massive formalization is not a *fait accompli*; many challenges remain in such areas as the organization of large databases of mathematics and the raising of the level of automation.

In contrast, there is no general agreement on how to formalize *computational* mathematics. Even worse, few people appreciate that this is a significant new problem (see [5]). Our interest is in examining whether some kind of *constructive type theory* is an appropriate formalism.

There are two immediately-appealing aspects of constructive type theories. Firstly, they have built-in a functional programming language in which algorithms can be expressed. Secondly, propositions can be read as claiming the existence of functional programs and data, and if some proof can be given of a proposition, the corresponding programs and data can be automatically synthesized (or sometimes we say *extracted*) from the proof. For example, from the proof of $\forall x \in S.\ \exists y \in T.\ P(x, y)$, we can synthesize a functional program that, when given any element $s$ of type $S$ as data, can compute some $t$ of type $T$ such that proposition $P(s, t)$ holds. We sometimes call the programs and data that can be synthesized from proofs of some proposition the *computational content* of the proposition.

There is a tradeoff in gaining this extra expressivity: fewer truths can be proven in logics based on constructive type theories than in classical logic. For example, the propositions $P \Leftrightarrow \neg\neg P$, and $P \vee \neg P$ are no longer true for arbitrary propositions $P$. For introductory material on constructive type theory, consult [30] or [10].

The particular constructive type theory we are working with is similar to one of Martin-Löf's [24], and is implemented in the Nuprl proof development system [6, 20]. Nuprl provides an environment for assembling *theories* consisting of definitions, theorems, and proofs. It also has an interpreter for executing the functional programs that users write and that are synthesized from proofs.

Previous topics we have experimented with in Nuprl include elementary number theory [17], elementary analysis [9], and the algebra of polynomials [20].

## 1.2 Choice of Topic

Automata theory is an appealing topic for formalization because of its central role in computer science. Recently, we have been considering whether we could formalize a whole book on automata theory such as Hopcroft and Ullman's *Formal Languages and their Relation to Automata* [15]. Such a formalization could have significant pedagogical value: it could serve as a novel hypertext reference for students studying automata theory, and would stand as a corpus of familiar examples for any computer scientist interested in formalization techniques.

We report in this article on a preliminary step in this direction, namely, the formalization in Nuprl of the Myhill-Nerode theorem on the existence and uniqueness of minimum finite automata. We based the formalization on the presentation in the book cited above. We chose this theorem because it is one of the first significant theorems in the book, and because it involves computationally interesting constructions.

Automata theory was previously explored in Nuprl by Christoph Kreitz in 1986 [22]. In particular, he proved the pumping lemma for finite automata. We saw that we needed this lemma for our constructivization of the Myhill-Nerode theorem, and so reproved it using Nuprl's current tactic collection. We therefore could compare the currently achievable level of automation with that achievable in 1986.

An under-explored aspect of Nuprl's type theory is its novel quotient types (see Section 4.3). Jackson, for example, had experimented with these previously [20], but we still didn't have much experience with how best to reason efficiently with them. The heart of the Myhill-Nerode theorem involves a quotient construction, and so provided a good opportunity to gain more experience.

## 1.3 Value of the Formalization

Readers can evaluate the formal text on the Web that resulted from our formalization efforts. It is possible to directly judge whether our definitions are faithful to Hopcroft and Ullman's, whether the formal definitions help clarify the concepts, whether the proofs are sufficiently readable and informative, and whether the availability of detail about all proof steps is useful. The formal material also provided the underpinning for this article in that our definitions and proof summaries refer to the complete formal library. The material can be the foundation for other documents that explain the detailed proofs. We have not yet produced such documents for the automata library (but see Section 9), however, there are examples of this genre of writing in the Formal-Courseware section of the Nuprl home page. For example,

Stuart Allen has produced a hybrid style of formal and informal proof to accompany the basic theorems about functions proved by Paul Jackson and used extensively in this formalization.

There are other aspects and by-products of our formalization that cannot be directly evaluated by reading the formal text; they require experience with the system. In this category is the experience of *confidence* in the results that comes from learning to trust Nuprl. We know that reading results checked by both a human and a machine raises confidence in their correctness, similar to the added confidence gained by having a trusted colleague check a result.

Another value of the formalization is the interactivity provided by the underlying system. For example, Nuprl can show dependencies among theorems and definitions, and it can execute algorithms extracted from proofs. Users can experiment with alternate proofs and can observe the effect on the extracted programs.

In addition to the readable and highly reliable interactive formal text, the formalization has created an interesting *digital artifact.* The formal theory becomes an object that we can manipulate, measure, transform and explore. To experience these capabilities, one must learn to use a system like Nuprl.

## 1.4 Interpretations of the Mathematics

Nuprl's type theory can be interpreted in several ways. In the semantics given by Allen [1], all functions are computable. The type theory is therefore compatible with *recursive mathematics* in which all functions are given by Turing machines. Every theorem in Nuprl can be seen as a theorem of recursive mathematics, but the converse is not true; the type theory is sufficiently weak that non-classical results in recursive mathematics, such as that every function from $\mathbb{R}$ to $\mathbb{R}$ is continuous, are not provable.

Howe has given a set-theoretic interpretation of Nuprl's type theory that shows that every theorem provable in Nuprl can be read as a theorem of classical mathematics [19]. This interpretation includes all non-computational set-theoretic functions in the denotation of function types.

Having both classical and recursive interpretations makes Nuprl a suitable tool for formalizing constructive mathematics in the style advocated by Bishop [2].

## 1.5 Electronic Access to Formalization

The key ideas of the formalization are presented in this article in a self-contained way. To find out more, the reader is invited to browse the full

formalization on the World Wide Web. Start by visiting the Nuprl project's home page at URL

`http://www.cs.cornell.edu/Info/Projects/NuPrl/nuprl.html`

From there, the reader can access hypertext presentations of both the work presented in this article and other more recent work in automata theory.

Nuprl itself is free software that can be obtained from this web site. It runs on a freely-available version Allegro Common Lisp under Linux as well as free CMU Common Lisp under Unix.

### 1.6 Related work

A non-constructive set-theoretic formalization of minimization theorems for Moore and Mealy automata has been done in the Mizar system [21]. This closely follows a presentation in Denning, Dennis Qualitz [8].

Theorems asserting the equivalence between deterministic (DFA) and non-deterministic (NFA) finite state automata, and between NFAs with and without epsilon moves, were proven in Nqthm [31], and subsequently in PVS [28]. These formalizations were based on theorems 2.1 and 2.2 in [16]. Notably, a flaw was found in the textbook proof of theorem 2.2. The formation of DFA states from sets of NFA states was significantly more complicated in the constructive Nqthm proofs than the non-constructive PVS proofs. Part of the difficulty in the Nqthm proofs was in modelling finite sets using lists, and handling the equality of lists considered as sets. We wouldn't expect to have this difficulty in Nuprl because, as shown in [20], we can take advantage of Nuprl's quotient type to appropriately redefine the equality relation on lists.

Quotient types have been explored in the ECC constructive type theory by Hofmann [14], and probably most of the development presented in this article could be straightforwardly formalized in the LEGO mechanization of ECC [27]. One major difference between ECC and Nuprl is that, in ECC, both explicitly written and synthesized programs must be embellished with parts that are unimportant for computation, but necessary for proofs of correctness.

### 1.7 Outline

In Section 2 we present the basic ideas from Nuprl needed for this article. Section 3 defines the notion of a formal language, and Section 4 provides the preliminaries on automata. Section 5 proves the Myhill-Nerode theorem,

and Section 6 presents a corollary which makes explicit the construction and properties of the minimum automata introduced in the course of proving the Myhill-Nerode theorem. Section 7 discusses various issues that came up, Section 8 summarizes our results, and Section 9 presents our conclusions and outlines future work. Finally, Appendix A provides an index for notation.

All the material on languages and automata closely follows that in Chapter 1 and Sections 3.1 and 3.2 of Hopcroft and Ullman [15].

# 2 Type Theory Preliminaries

## 2.1 Basic Types

The integers $\mathbb{Z}$ are a primitive type of Nuprl. Defined subtypes of the integers include the bounded-below range $\{i\ldots\} == \{j : \mathbb{Z} \mid i \leq j\}$, the naturals $\mathbb{N} == \{0\ldots\}$, and the finite types $\mathbb{N}k == \{n : \mathbb{N} \mid n < k\}$. (In Nuprl notation, we use $==$ for definitional equality).

The Booleans $\mathbb{B}$ are defined type. For the purposes of this article the exact definition is unimportant. The canonical elements of $\mathbb{B}$ are *tt* and *ff* denoting true and false respectively. As explained in Section 2.6, boolean expressions and propositions are distinct. The prefix operation $\uparrow$ converts a boolean expression to a proposition.

## 2.2 Recursive Types

The only recursive type relevant here is the *list* type. Given any type $A$, the type $A$ *list* is the type of finite sequences of elements of type $A$. The empty list is *nil*. Lists are constructed using an infix "." constructor, often referred to as "cons". Given an element $a$ of type $A$ and a list $l$ of type $A$ *list*, cons forms a new list $a.l$. The functions *hd* (head) and *tl* (tail) take lists apart. They satisfy the equations $hd(a.l) = a$ and $tl(a.l) = l$. The boolean-valued function *null* tests whether a list is empty. The infix append operation @ joins two lists together.

## 2.3 Product Types

If $A$ and $B$ are types, then so is their *cartesian product*, $A \times B$. The elements of $A \times B$ are ordered pairs, $\langle a, b\rangle$ with $a \in A$ and $b \in B$. The product and pairing operation are assumed to associate to the right, so we write $A \times B \times C$ for $A \times (B \times C)$, and $\langle a, b, c\rangle$ for $\langle a, \langle b, c\rangle\rangle$.

The product type is a special case of a *dependent product type*, also known as a $\Sigma$ type. In this type, the type of the second component of pairs can depend on the first component of pairs.

## 2.4 Function Types

If $A$ and $B$ are types, then $A \to B$ denotes the type of all total computable functions from $A$ to $B$. The canonical elements of this type are lambda terms, $\lambda x.\, b$. Let $b[a/x]$ denote the substitution of the term $a$ for all free occurrences of $x$ in $b$. For $\lambda x.\, b$ to be a function from $A$ to $B$, its value $b[a/x]$ must be of type $B$ for all arguments $a$ of type $A$. If $f \in A \to B$ and $a \in A$, then $f\ a$ denotes the application of $f$ to argument $a$.

The function type is a special case of a *dependent function type*, also known as a $\Pi$ type. The type of an application of a function in a dependent function type can depend on the argument the function is applied to. There are no examples of dependent function types in this article.

## 2.5 Recursive Function Definitions

A recursive function definition in Nuprl is written $lhs ==_r rhs$, where $lhs$ is the function being defined, and $rhs$ may include instances of $lhs$ as subterms. For example, the list append operation @ can be defined with

$$u@v ==_r \mathit{if}\ \mathit{null}(u)\ \mathit{then}\ v\ \mathit{else}\ \mathit{hd}(u).(\mathit{tl}(u)@v).$$

Recursive functions are created using the $Y$ recursion combinator, which is definable since Nuprl's computation language is untyped. Immediately after introducing a recursive definition, we prove a well-formedness lemma showing that evaluation of the definition on arguments in specified types always terminates and gives a result in a specified type. The lemma for @ is

$$\vdash \forall A : \mathbb{U}.\ \forall u, v : A\ \mathit{list}.\ u@v \in A\ \mathit{list}.$$

## 2.6 Propositions and Universes

In so-called "classical" accounts of logic, a proposition has a truth value in $\mathbb{B}$, and propositions can be treated as boolean expressions. We are interested not only in the truth value of propositions, but also in their computational sense; how they can be seen as specifications for programs. To support this computational view, it is necessary for us to have a type $\mathbb{P}$ of propositions distinct from $\mathbb{B}$.

There are two distinguished atomic propositions, $\top$ the canonically true one and $\bot$ the canonically false one. Given propositions $P, Q$ we can form compounds in the usual way:

$P \wedge Q$ for "$P$ and $Q$ ",
$P \vee Q$ for "$P$ or $Q$",
$P \Rightarrow Q$ for "$P$ implies $Q$" also written "$P$ only if $Q$",
$P \Leftrightarrow Q$ for "$P$ if and only if $Q$" also written "$P$ iff $Q$".

Negation, $\neg P$, is defined as $P \Rightarrow \bot$.

A *propositional function* on a type $T$ is any map $P \in T \rightarrow \mathbb{P}$. Given such a $P$, then we can form the propositions:

$\forall x{:}T.\ P\ x$ "for all $x$ of type $A$, $P\ x$ holds,"
$\exists x{:}T.\ P\ x$ "for some $x$ of type $A$, $P\ x$ holds."

Associated with every type $T$ is the atomic equality relation $x = y$ *in* $T$. The definition of this equality is given with each type. Often, the *in* $T$ is dropped; it usually can be inferred from consideration of $x$ or $y$.

Types in Nuprl are members of *universe* types. Nuprl has a hierarchy of universe types to avoid the problem of a universe type being a member of itself. There happens also to be a corresponding hierarchy of proposition types. For the purposes of this article, it is sufficient that we use $\mathbb{U}$ to denote some typical universe type and $\mathbb{P}$ to denote some typical type of propositions. See [6] or [20] for fuller accounts of Nuprl's logic and universe types.

### 2.7 Subtypes

If $T$ is a type and $P \in T \rightarrow \mathbb{P}$ is a propositional function, then $\{x{:}A \mid P\ x\}$ denotes the type of all elements of $A$ satisfying $P$. Looking at this subset type from a constructive point of view, it's important to note that when we assume that we have some element $a$ in this type, we don't have any access to the computational content of the the proposition $P\ x$, even though we know it to be true. Further discussion of the subset type can be found in [6, 20, 25] as well as in Section 2.8.

### 2.8 Finiteness

A predicate asserting that a type $T$ is finite is

$$\mathit{Fin}(T) == \exists k{:}\mathbb{N}.\ \ \mathit{1\text{-}1\text{-}Corresp}(\mathbb{N}k; T),$$

where $\mathit{1\text{-}1\text{-}Corresp}(\mathbb{N}k; T)$ just when there there exist functions $f$ of type $\mathbb{N}k \rightarrow T$ and $g$ of type $T \rightarrow \mathbb{N}k$ that are mutual inverses.

Constructively, if we assume $\mathit{Fin}(T)$, we are assuming that $T$'s cardinality and the computable functions $f$ and $g$ are available for use. Likewise, if we are proving $\mathit{Fin}(T)$, we have to give $T$'s cardinality and produce suitable computable functions $f$ and $g$.

Because the predicate $Fin(T)$ has significant computational content, it is not that constructively useful to form the type of all finite types $\{T : \mathbb{U} | Fin(T)\}$ using the subset type; if we know some type is in this collection of finite types, we still have no way of finding out its size, or enumerating its contents.

# 3 Languages and their Representation

## 3.1 Alphabets and Languages

Hopcroft and Ullman begin their book with the question: What is a language? Their answer starts with a definition of an *alphabet*. They define an alphabet to be any finite set of *symbols*. The exact structure of symbols is unimportant, so we take an alphabet to be any type *Alph*, and we always assume $Fin(Alph)$. As noted in Section 2.8, a consequence of finiteness is that the equality relation on *Alph* is decidable.

In Hopcroft and Ullman we read that a *sentence* over an alphabet is any string of finite length composed of symbols from the alphabet. We use lists of type *Alph list* to represent strings over an alphabet *Alph*. We choose to reverse the order of alphabet symbols, so the string `abc` is represented by the list $\mathtt{c}.\mathtt{b}.\mathtt{a}.nil$.

Hopcroft and Ullman define a *language* to be a set of sentences over an alphabet. In Nuprl's type theory, though types superficially resemble sets, they are not as versatile. For example, one cannot take the union or intersection of two arbitrary types, and a type membership predicate can be awkward to reason with. So, instead of considering a language $L$ over an alphabet *Alph* to be a subtype of *Alph list*, we consider $L$ to be a propositional function over *Alph list*, that is, a function of type *Alph list* $\rightarrow \mathbb{P}$. When sets are represented in Nuprl's type theory as propositional functions over some common domain type, common set operations and predicates are straightforward to define and use.

We let $Language(Alph)$, the type of languages over alphabet *Alph*, be an abbreviation for *Alph list* $\rightarrow \mathbb{P}$.

We define two languages to be equal, written $L = M$, just when for all $x$ in *Alph list*, $L\ x \Leftrightarrow M\ x$.

## 3.2 Representations of Languages

Our definition of a language as a propositional function $L \in$ *Alph list* $\rightarrow \mathbb{P}$ captures the intuition that to know a language is to know the criteria for saying when a sentence is in it. To say $x$ is in the language $L$ is to know how

to prove $L\ x$. This agrees with Hopcroft and Ullman; they are concerned with certain special ways of knowing $L\ x$.

One especially simple kind of representation of $L$ arises when the proposition $L\ x$ is decidable, i.e. when there is a function $R_L \in \mathit{Alph\ list} \to \mathbb{B}$ such that

$$L\ x \text{ iff } \uparrow (R_L\ x).$$

We call the function $R_L$ a *language recognizer*, and the language in this case is said to be *decidable* or *recursive*.

# 4 Finite Automata

## 4.1 Definition

Hopcroft and Ullman define a *finite automaton* $M$ to be a system

$$\langle K, \mathit{Alph}, \delta, q_0, F \rangle$$

where $K$ is a finite nonempty set of *states*, $\mathit{Alph}$ is a finite *input alphabet*, $\delta$ is a mapping of $K \times \mathit{Alph}$ into $K$, $q_0$ *in* $K$ is the *initial state*, and $F \subseteq K$ is the set of *final states*.

In defining a finite automaton in Nuprl, we first assume that some type $\mathit{Alph}$ is given for an alphabet, and some type $\mathit{St}$ for the set of states. We assume both $\mathit{Alph}$ and $\mathit{St}$ are finite, though occasionally we relax these constraints when they are not necessary. An automaton $A$ is then a triple $\langle \delta(A), I(A), F(A) \rangle$, where the next state function $\delta(A)$ has type $\mathit{St} \to \mathit{Alph} \to \mathit{St}$, the initial state $I(A)$ is a member of $\mathit{St}$, and $F(A)$ is a function of type $\mathit{St} \to \mathbb{B}$ that returns $tt$ just when applied to final states. By defining $F(A)$ as a boolean-valued function, we ensure that we can compute when an automaton is in a final state. The type of all such automata is

$$\mathit{Automata}(\mathit{Alph}; \mathit{St}) == (\mathit{St} \to \mathit{Alph} \to \mathit{St}) \times \mathit{St} \times (\mathit{St} \to \mathbb{B}).$$

See Section 7.1 for a discussion of the difference between our and Hopcroft and Ullman's definition.

## 4.2 Semantics of Automata

Hopcroft and Ullman extend the automaton transition function $\delta$ to input strings with the recursive definition:

$$\hat{\delta}(q, \mathit{nil}) = q$$

$$\hat{\delta}(q, a.x) = \delta(\hat{\delta}(q, x), a),$$

where $a$ is a symbol in the alphabet $Alph$ and $x$ is a string over $Alph$. They define the language accepted by the automaton as

$$\{x \mid \hat{\delta}(q_0, x) \text{ is in } F\}.$$

We make analogous definitions in Nuprl. Let $A$ be an automaton of type $Automata(Alph; St)$, let $l$ be an input string in type $Alph\ list$, and let $s$ be a state in $St$. We define the recursive function

$$\delta'(A)(s;\ l) ==_r \mathit{if}\ null(l)\ \mathit{then}\ s\ \mathit{else}\ \delta(A)\ \ \delta'(A)(s;\ tl(l))\ \ hd(l),$$

which given $A$ in state $s$ to start, computes the new state of $A$ after input of $l$.

We then define $A(l)$ which computes the state of $A$ after input of $l$, starting in the initial state:

$$A(l) == \delta'(A)(I(A);\ l).$$

Using the final-state function $F(A)$, we define a language recognizer $L_b(A)$ for $A$ as

$$L_b(A) == \lambda l{:}\,Alph\ list.\ F(A)\ A(l).$$

The language accepted by $A$ is defined by a similar function which returns a proposition rather than a boolean. Using the $\uparrow$ function which converts a boolean to the corresponding proposition, $L(A)$, the language accepted by $A$, is defined as

$$L(A) == \lambda l{:}\,Alph\ list.\ \uparrow (L_b(A)\ l).$$

### 4.3 Equivalence Relations and Quotient Types

Prior to presenting the Myhill-Nerode theorem, Hopcroft and Ullman give a brief introduction to equivalence relations and how they partition the sets they are over into equivalence classes. They take a binary relation on a set $S$ to be a set of pairs of elements of $S$. As with representing languages (see Section 3.1), we find it more convenient to represent relations as characteristic functions: we consider a binary relation on a type $S$ to be a function of type $S \to S \to \mathbb{P}$ $(= S \to (S \to \mathbb{P}))$. To express that elements

$x$ and $y$ of type $S$ are related by a binary relation $R$ of type $S \to S \to \mathbb{P}$, we use both prefix application notation $R\ x\ y$ and infix notation $x\ R\ y$.

In the Myhill-Nerode theorem, an automaton is constructed that uses the equivalence classes of an equivalence relation as the states of an automaton. This is problematic constructively, because the equivalence classes in question have infinite size, and we would like to have finite representations of states on which we can define computable transition functions.

The obvious solution is to use some element of an equivalence class as a representative for the whole class. We do this with the help of Nuprl's quotient types. Given a type $S$ and an equivalence relation $E$ on $S$, the *quotient type* $S//E$ has the same members as $S$, but has as its associated equality relation the relation $E$ rather than the equality relation associated with $S$.

In Nuprl's type theory, for a function $f$ to be in a type $S \to T$, it must respect the equalities associated with $S$ and $T$. Specifically, if the equalities are $=_S$ and $=_T$ respectively, we have $f\ x =_T f\ y$ whenever $x =_S y$. If $E$ is an equivalence relation on $S$, and we want to show that $f$ also has type $S//E \to T$, we have to check that $f\ x =_T f\ y$ whenever $x\ E\ y$.

The quotient type $S//E$ behaves much like a type of the equivalence classes of $E$. Often when set-theoretically defining a function with a set of equivalence classes as domain, the function mentions representatives of equivalence classes, and it is necessary to check that the value of the function is independent of the particular choice of representatives. With the quotient type $S//E$ as domain of a function in Nuprl's type theory, the rules for function type inhabitation enforce a corresponding constraint.

In presentations of quotient types from Nuprl theories, we occasionally use the notation $x, y : S//(x\ E\ y)$ for the type $S//E$. This more verbose notation is useful when the primary notation for relation $E$ includes its arguments.

### 4.4 Finite Index Equivalence Relations

In set theory, an equivalence relation $E$ on a set $S$ is is said to be of *finite index* if $E$ has a finite number of equivalence classes.

In Nuprl's type theory, we express that an equivalence relation $E$ on type $S$ has finite index by saying $Fin(S//E)$, that is, the quotient type $S//E$ is in one-one correspondence with $\{0 \ldots k-1\}$ for some non-negative number $k$. This definition works because the functions defining the bijection between $S//E$ and $\{0 \ldots k-1\}$ must respect $E$. Note that when $S$ is infinite, is is possible for $S//E$ to be finite, even though $S$ and $S//E$ have the same elements.

### 4.5 Equivalence Relations on Strings

We introduce here a couple of definitions that are useful for stating the Myhill-Nerode theorem.

**Definition:** An equivalence relation $E$ on *Alph list* is called *extension invariant* [1] just when for all $x, y, z$ in *Alph list*

$$x\ E\ y \Rightarrow (z@x)\ E\ (z@y).$$

**Definition:** A language $L$ over alphabet *Alph induces* an equivalence relation $R(L)$ given by

$$x\ R(L)\ y\ \Leftrightarrow (\forall z : \mathit{Alph\ list}.\ z@x \in L \Leftrightarrow z@y \in L).$$

## 5 The Myhill-Nerode Theorem

We reproduce Hopcroft and Ullman's presentation of the Myhill-Nerode theorem in Section 5.1, and discuss its formalization in the following sections.

### 5.1 Hopcroft and Ullman Version

The statement and proof of the Myhill-Nerode theorem here is taken almost verbatim from [15]. A few changes have been made to make the notation more similar to that used in the formal development. The definitions of what it means for an equivalence relation to be *extension invariant* and of the equivalence relation *induced* by a language can be found in Section 4.4.

**Theorem 3.1.** The following three statements are equivalent:

(1) The set $L \subseteq$ *Alph list* is accepted by some finite automaton.

(2) $L$ is the union of some of the equivalence classes of an extension invariant equivalence relation of finite index.

(3) The equivalence relation on *Alph list* induced by $L$ is of finite index.

**Proof**
(1) $\Rightarrow$ (2).

[1] Hopcroft and Ullman have strings which are extended on the right and call such relations *right invariant*

Assume that $L$ is accepted by $M = (K, Alph, \delta, q_0, F)$. Let $R$ be the equivalence relation $x\ R\ y$ if and only if $\delta(q_0, x) = \delta(q_0, y)$. $R$ is extension invariant since, for any $z$, if $\delta(q_0, x) = \delta(q_0, y)$, then

$$\delta(q_0, z@x) = \delta(q_0, z@y).$$

The index of $R$ is finite since the index is at most the number of states in $K$. Furthermore, $L$ is the union of those equivalence classes which include an element $x$ such that $\delta(q_0, x)$ is in $F$.

(2) $\Rightarrow$ (3).
We show that any equivalence relation $R$ satisfying statement (2) is a refinement of the equivalence relation $R(L)$ induced by $L$; that is, every equivalence class of $R$ is entirely contained in some equivalence class of $R(L)$. Thus the index of $R(L)$ cannot be greater than the index of $R$ and so is finite. Assume that $x\ R\ y$. Then since $R$ is extension invariant, for each $z$ in $Alph\ list$, $z@x\ R\ z@y$, and thus $z@y$ is in $L$ if and only if $z@x$ is in $L$. Thus $x\ R(L)\ y$, and hence, the equivalence class of $x$ in $R$ is contained in the equivalence class of $x$ in $R(L)$. We conclude that each equivalence class of $R$ is contained within some equivalence class of $R(L)$.

(3) $\Rightarrow$ (1).
Assume that $x\ R(L)\ y$. Then for each $w$ and $z$ in $Alph\ list$, $z@w@x$ is in $L$ if and only if $z@w@y$ is in $L$. Thus $w@x\ R(L)\ w@y$, and $R(L)$ is extension invariant. Now let $K'$ be the finite set of equivalence classes of $R(L)$ and $[x]$ the element of $K'$ containing $x$. Define $\delta([x], a) = [x.a]$. The definition is consistent, since $R(L)$ is extension invariant. Let $q'_0 = [nil]$ and let $F' = \{[x] \mid x \in L\}$. The finite automaton $M' = (K', Alph, \delta', q'_0, F')$ accepts $L$ since $\delta'(q'_0, x) = [x]$, and thus $x$ is in $L(M')$ if and only if $[x]$ is in $F'$.
**Qed**

## 5.2 Formalizing (1) $\Rightarrow$ (2)

The formal statement of the theorem in Nuprl's notation is

$$\begin{array}{l}
\vdash \forall Alph{:}\mathbb{U}.\ \ \forall L{:}Language(Alph).\\
\quad Fin(Alph)\\
\quad \Rightarrow (\exists St{:}\mathbb{U}.\ \ \exists Auto{:}Automata(Alph; St).\ Fin(St) \wedge\ L = L(Auto))\\
\quad \Rightarrow (\exists R{:}Alph\ list\ \rightarrow Alph\ list \rightarrow \mathbb{P}\\
\qquad EquivRel(Alph\ list; R)\\
\qquad \wedge \exists g{:}Alph\ list//R \rightarrow \mathbb{B}\\
\qquad\quad Fin(Alph\ list//R)\\
\qquad\quad \wedge\ (\forall l{:}Alph\ list.\ L\ l \Leftrightarrow \uparrow(g\ l))\\
\qquad\quad \wedge\ (\forall x, y, z{:}Alph\ list.\ R\ x\ y \Rightarrow R\ (z@x)\ (z@y))).
\end{array}$$

An English rendering of this is:

- Let *Alph*, an alphabet, be a finite type and let $L$ be a language over *Alph*,
- assume there exists a finite type of states *St* and an automata *Auto* over *Alph* and *St* that accepts the language $L$,
- then there exists a binary relation $R$ on *Alph list* that is
  - an equivalence relation $(EquivRel(Alph\ list; R)\})$,
  - right invariant $(\forall x, y, z{:}\,Alph\ list.\ R\ x\ y \Rightarrow R\ (z@x)\ (z@y))$,
  - and of finite index $(Fin(Alph\ list//R))$,
- and there exists a boolean-valued function $g$ with domain *Alph list*$//R$ that returns boolean true ($tt$) exactly on strings in the language $L$ $(\exists g{:}\,Alph\ list//R \to \mathbb{B}.\ \forall l{:}\,Alph\ list.\ L\ l \Leftrightarrow \uparrow(g\ l))$.

The function $g$ here acts as the characteristic function for the set of equivalence classes of the relation $R$ whose union gives the language $L$. As remarked in Section 3.1, it is often more straightforward in Nuprl's type theory to represent sets as characteristic functions than as types. Note that the Nuprl quotient type *Alph list*$//R$ still contains elements of *Alph list* as members, so it is legitimate to pass the function $g$ an element $l$ of *Alph list* as an argument.

In requiring that $g$ be boolean ($\mathbb{B}$) valued rather than proposition ($\mathbb{P}$) valued, we are augmenting the statement (2) of the theorem with the requirement that membership in the language $L$ be decidable. This augmentation is necessary for the constructive proofs of the other parts of the theorem.

**Proof**

1. As with the Hopcroft and Ullman proof, $R\ x\ y$ is defined as $Auto(x) = Auto(y)$. Showing $R$ is an equivalence relation and is extension invariant is straightforward.

2. Finiteness of *Alph list*$//R$ is argued by noting that *Alph list*$//R$ is isomorphic to the set of accessible states, which is a subset of *St*.

   The finiteness argument is first carried out abstractly by proving the lemma

   $$\vdash \forall T, S{:}\mathbb{U}.\ \forall f{:}T \to S.\ Fin(S)\ \wedge\ (\forall s{:}S.\ Dec(\exists t{:}T.\ f\,t = s)) \Rightarrow Fin(x, y{:}T//(f\ x = f\ y))$$

   which is then instantiated with $T$ being *Alph list*, $S$ being *St*, and $f$ being the function $\lambda l.\ Auto(l)$.

   In using this lemma, the precondition

$\forall s{:}St.\ Dec(\exists t{:}Alph\ list.\ Auto(t) = s)$

has to be discharged. Read constructively, this precondition requires that, for any state $s$, it is possible to compute whether or not $s$ is accessible, and further, if $s$ is accessible, it must be possible to compute some string $t$ that, when input to the automaton, puts the automaton into state $s$.

The precondition is proven with the help of a corollary of the pumping lemma which states that in searching for a string that puts an automaton in a certain state, it is only necessary to try strings whose length is not greater than the number of states of the automaton.

3. We define $g$ on $Alph\ list//R$ to be $tt$ exactly when $F(Auto(x)) = tt$, i.e. $g\ x = F(Auto(x))$. That $g$ is functional *wrt* $R$ follows directly from the definition of $R$.

**Qed**

### 5.3 Formalizing (2) ⇒ (3)

Given a type $A$ representing an alphabet, and a language $L$ over $A$, the binary relation $R(L)$ induced by $L$ is defined as

$$R(L) == \lambda x, y.\ \forall z{:}A\ list.\ L\ z@x \Leftrightarrow L\ z@y$$

and has type $A\ list\ \to A\ list\ \to \mathbb{P}$. The display of parameter $A$ to $R(L)$ is suppressed, since $A$ can be inferred from considering the type of $L$. We establish straightforwardly that $R(L)$ is an equivalence relation.

The formal statement of (2) ⇒ (3) is:

$$\begin{aligned}
\vdash\ & \forall n{:}\{1...\}.\ \forall A{:}\mathbb{U}.\ \ \forall L{:}Language(A).\ \ \forall R{:}A\ list \to A\ list \to \mathbb{P}.\\
& \quad Fin(A)\\
& \Rightarrow EquivRel(A\ list; R)\\
& \Rightarrow \textit{1-1-Corresp}(\mathbb{N}n; A\ list//R)\\
& \Rightarrow (\forall x, y, z{:}A\ list.\ \ x\ R\ y \Rightarrow (z@x)\ R\ (z@y))\\
& \Rightarrow (\exists g{:}A\ list//R \to \mathbb{B}.\ \ \forall l{:}A\ list.\ \ L\ l \Leftrightarrow\uparrow (g\ l))\\
& \Rightarrow (\exists m{:}\mathbb{N}.\ \ \textit{1-1-Corresp}(\mathbb{N}m; A\ list//R(L)))\\
& \qquad \wedge(\forall l{:}A\ list.\ Dec(L\ l)).
\end{aligned}$$

An English reading is:

- Let the alphabet $A$ be a finite type,
- let R be a binary relation on $A\ list$ that is
  - an equivalence relation,

  - extensionally invariant,
  - and of finite index
    ($\mathit{1\text{-}1\text{-}Corresp}(\mathbb{N}n; A\ \mathit{list}//R)$ where $n$ is a positive integer),

- let $L$ be a language over $A$,
- assume $L$ is a union of equivalence classes of $R$ and is decidable
  $(\exists g : A\ \mathit{list}//R \to \mathbb{B}.\ \forall l : A\ \mathit{list}.\ L\ l \Leftrightarrow \uparrow (g\ l))$,
- then $R(L)$ is of finite index
  $(\exists m : \mathbb{N}.\ \mathit{1\text{-}1\text{-}Corresp}(\mathbb{N}m; A\ \mathit{list}//R(L)))$,
- and $L$ is decidable.

Note that here both statements (2) and (3) of Hopcroft and Ullman have been augmented with a requirement that membership in $L$ be decidable. The augmentation of (3) is necessary for the proof of (3) $\Rightarrow$ (1).

**Proof**
The argument that $R$ is a refinement of $R(L)$ follows the Hopcroft and Ullman argument and is completely straightforward.

To show that therefore the index of $R(L)$ is no larger than the index of $R$, we could instantiate a lemma of form

> **Quotient Index Lemma 1**. If $P$ and $Q$ are binary relations over a type $T$, and $P$ is a refinement of $Q$ ($x\ P\ y \Rightarrow x\ Q\ y$ for any $x$ and $y$), and the index of $T//P$ is some natural number $n$, then the index of $T//Q$ is some natural number $m$ such that $m \leq n$.

For this lemma to be constructive, a precondition requiring $S$ to be a decidable relation needs to be added.

Proving this lemma is tedious; it involves giving the explicit construction of a bijection between $\{0 \ldots m-1\}$ and $T//S$ given a bijection between $\{0 \ldots n-1\}$ and $T//R$. It turns out to be simpler to prove a lemma of form:

> **Quotient Index Lemma 2**. If $Q$ is a decidable binary relation over a type $T$, and the index of $T$ is some natural number $n$, then the index of $T//Q$ is some natural number $m$ such that $m \leq n$.

We instantiate the $T$ of this lemma with the type $\mathit{Alph\ list}//R$ and the $Q$ of this lemma with a binary relation $R'(g)$ which is similar to $R(L)$ in definition, but is defined over $\mathit{Alph\ list}//R$ rather than $\mathit{Alph\ list}$. The precondition that $\mathit{Alph\ list}//R$ is of finite index follows by assumption, and we get the

result that $(Alph\ list//R)//R'(g)$ is of finite index. That $Alph\ list//R(L)$ is of finite index trivially follows when we use the result that there is a one-one correspondence between $(Alph\ list//R)//R'(g)$ and $Alph\ list//R(L)$.

A remaining precondition of Quotient Index Lemma 2 is to show that $R'(g)$, or equivalently $R(L)$, is a decidable relation. This is not immediately obvious: Since $x\ R(L)\ y$ iff $z@x\ R\ z@y$ for every $z$, it seems that we have to try an infinite number of $z$ to compute if $x\ R(L)\ y$ true. (Note that we can test if $z@x\ R\ z@y$ since $R$ is decidable.) Again, the pumping lemma is of help; it shows that it is sufficient to only consider every $z$ of length up to the number of states of our automata $M$ which accepts $L$. Since $Alph$ is finite, there are only a finite number of $z$ to try.
**Qed**

### 5.4 Formalizing (3) ⇒ (1)

The formal statement of the theorem is:

$$\begin{aligned}\vdash\ & \forall Alph{:}\mathbb{U}.\ \forall L{:}Language(Alph)\\ & \quad Fin(Alph)\\ & \quad \Rightarrow (Fin(Alph\ list//R(L)) \wedge \forall l{:}Alph\ list.\ Dec(L\ l))\\ & \quad \Rightarrow \exists St{:}\mathbb{U}.\ \exists Auto{:}Automata(Alph;St).\ Fin(St) \wedge L = L(Auto).\end{aligned}$$

In English,

- Let the alphabet $Alph$ be a finite type, and let $L$ be a language over $Alph$,
- assume the relation $R(L)$ induced by $L$ is of finite index,
- assume membership of $L$ is decidable,
- then there is a finite type of states $St$, and an automaton $Auto$ over $Alph$ and $St$ that accepts $L$.

**Proof**
Checking $R(L)$ is extension invariant is straightforward.

For the type of states $St$ we take the quotient type $Alph\ list//R(L)$ instead of the set of equivalence classes of $R(L)$. Whereas Hopcroft and Ullman define the action of the automata in terms of equivalence classes, writing $\delta([x], a) = [a.x]$, here we use a function that works on representatives of equivalence classes. Specifically, given an element $x$ of $Alph\ list$ and an element $a$ of $Alph$, we define $\delta(Auto)\ x\ a$ to be the list $a.x$.

For the start state $I(Auto)$, we use the empty list $nil$, and for $F(Auto)$ we use a boolean-valued version of the characteristic function $L$ (remember that

we represent languages using characteristic functions rather than subtypes). A boolean-valued version of $L$ exists because we have as an assumption that $L$ is decidable.

In type-checking each of these components of $Auto$, we check that the definition of $Auto$ is consistent. For example, we check that $\delta(Auto)$ has type $Alph\ list//R(L) \rightarrow Alph \rightarrow Alph\ list//R(L)$. In checking this, we show that if $x\ R(L)\ y$, then $(\delta(Auto)\ x\ a)\ R(L)\ (\delta(Auto)\ y\ a)$. That is, $\delta(Auto)$ maps possibly-different representatives of some equivalence class of $R(L)$ to representatives of the same equivalence class of $R(L)$.
**Qed**

# 6 State Minimization

We discuss in this section a corollary to the Myhill-Nerode theorem that explicitly states the existence and uniqueness of a minimum finite automaton for any language accepted by some finite automaton.

## 6.1 Textbook Proof

The presentation here is taken almost verbatim from [15, p29]. The main change is to adopt the notation for strings used in the Nuprl development.

**Theorem 3.2.** The minimum state automaton accepting $L$ is unique up to an isomorphism (i.e., a renaming of the states) and is given by $M'$ of Theorem 3.1.

**Proof**
In the proof of Theorem 3.1 we saw that any $M = (K, Alph, \delta, q_0, F)$ accepting $L$ defines an equivalence relation which is a refinement of $R(L)$. Thus the number of states of $M$ is greater than or equal to the number of states of $M'$ of Theorem 3.1. If equality holds, then each of the states of $M$ can be identified with one of the states of $M'$. That is, let $q$ be a state of $M$. There must be some $x$ in $Alph\ list$, such that $\delta(q_0, x) = q$, otherwise $q$ could be removed from $K$, and a smaller automaton found. Identify $q$ with the state $\delta'(q_0', x)$ of $M'$. This identification will be consistent. If $\delta(q_0, x) = \delta(q_0, y) = q$, then, by Theorem 3.1, $x$ and $y$ are in the same equivalence class of $R$. Thus $\delta'(q_0', x) = \delta'(q_0', y)$.
**Qed**

## 6.2 Formalization of Minimization Theorem

First we make a few definitions. As earlier, let $Alph$ be an alphabet, $St$ be a type for states, and $Auto$ be some automaton over $Alph$ and $St$.

The type of states $MinSt(Auto)$ of the minimum automaton for the language accepted by $Auto$ is

$$MinSt(Auto) == Alph\ list // R(L(Auto))$$

and the minimum automaton itself is

$$MinAuto(Auto) == \langle(\lambda s, a.\ (a.s)),\ nil,\ L_b(Auto)\rangle.$$

We show that $MinAuto(Auto)$ has type $Automata(Alph; MinSt(Auto))$. These definitions make explicit the constructions implicit in our proof of the Myhill-Nerode theorem.

With the help of various auxiliary lemmas from the Myhill-Nerode development, we prove such theorems as that $MinSt(Auto)$ is a finite type and $MinAuto(Auto)$ accepts the same language as $Auto$.

We split our statement and proof of the minimization theorem into two parts. It is important to note here that the definitions of $MinSt(Auto)$ and $MinAuto(Auto)$ depend only on the language accepted by $Auto$, not on any particular structure of $Auto$. Without this observation, the two main statements will not be seen to claim what we intend them to claim. The two statements are

1. The statement that the minimum automaton really has the smallest number of states of any automata accepting the same language is

   $\vdash \forall Alph{:}\mathbb{U}.\ Fin(Alph) \Rightarrow$
   $\forall St{:}\mathbb{U}.\ Fin(St) \Rightarrow$
   $\forall Auto : Automata(Alph, St).$
   $|St| \geq |MinSt(Auto)|.$

   Here we use the definition

   $$|S| \geq |T| == \exists f : S \to T.\ Surj(S; T; f),$$

   that is, a type $S$ is at least as large as a type $T$ if there exists a surjective function from $S$ to $T$. When $S$ is non-empty and $T$ is empty, this predicate is false, whereas one would ideally want it to be true. We don't need to be concerned with this pathological case since types of states always include initial states.

   **Proof**
   Most of the argument here is already gone over in the Myhill-Nerode proof. In a few cases we have to prove some new intermediate lemmas that make various facts more explicit.
   **Qed**

2. Our statement that the minimum automata is isomorphic to any other is

$$\begin{array}{l}\vdash \forall Alph : \mathbb{U}.Fin(Alph) \Rightarrow \\ \quad \forall St : \mathbb{U}.Fin(St) \Rightarrow \\ \quad \forall Auto : Automata(Alph, St) \\ \quad 1\text{-}1\text{-}Corresp(St; MinSt(Auto)) \Rightarrow \\ \quad Auto \equiv MinAuto(Auto).\end{array}$$

Here we use the definition

$$\begin{array}{l}A1 \equiv A2 == \\ \quad \exists f : S1 \to S2. \\ \quad Bij(S1; S2; f) \\ \quad \wedge (\forall s : S1. \forall a : Alph.\ f\ (\delta(A1)\ s\ a) = \delta(A2)\ (f\ s)\ a) \\ \quad \wedge f\ I(A1) = I(A2) \\ \quad \wedge (\forall s : S1.\ F(A1)\ s = F(A2)\ (f\ s)\end{array}$$

to say that automata $A1$ and $A2$ are isomorphic. This definition follows the pattern of definitions of isomorphisms for algebraic structures. Hopcroft and Ullman omit the definition entirely, no doubt on the grounds that it is the obvious one to use. $Bij(S1; S2; f)$ is the proposition that function $f$ from type $S1$ to type $S2$ is a bijection. The definition of $\equiv$ takes $S1$, $S2$, and $Alph$ as parameters, but the display of these is suppressed because they can easily be inferred from consideration of the types of $A1$ and $A2$.

**Proof**
As with the Hopcroft and Ullman proof, we argue that we can assume without loss of generality that $Auto$ is connected. We then use our analogue of their construction of the identification function $f$ for the isomorphism. Hopcroft and Ullman state without proof that this identification is consistent. We need ourselves to fill in the tedious but routine steps of proof showing that the identification function has all the properties that make it an isomorphism.
**Qed**

# 7 Discussion

## 7.1 Structuring the definition of automata

Our parameterization of the type of automata by both an alphabet and a type of states is inelegant (See Section 4.1). Parameterization by an alphabet has its merits, but it is clear that the type of states ought to be paired with the transition function, the initial state, and the set of final states.

Constructively, a full specification of an automata also requires evidence that the state type is finite.

One solution is to have automata over a finite alphabet be tuples of form

$$\langle FinSt, \delta,\ I,\ F\rangle$$

where $\delta$, $I$, and $F$ are as before, and $FinSt$ is an element of a type of 'finite types', of four-tuples of form $\langle T, n, f, g\rangle$, where $T$ is a type, $n$ is the size of $T$, and $f$ and $g$ define an isomorphism between $T$ and $\mathbb{N}n$.

A similar solution involves writing the type of finite types as

$$T{:}\mathbb{U} \times Fin(T).$$

(This is the notation for Nuprl's dependent product type, sometimes called a $\Sigma$ type.) From the point of view of classical mathematics this is ill-formed, a proposition $Fin(T)$, is being used in a position where a type is expected. However, in constructive type theory, this is well-formed because propositions *are* types. Elements of $Fin(T)$ are tuples of form $\langle n, f, g, *\rangle$, and elements of $T{:}\mathbb{U} \times Fin(T)$ are tuples of form $\langle T, \langle n, f, g, *\rangle\rangle$.

The $*$ here is a term witnessing the proposition that $f$ and $g$ form an isomorphism. Such witnesses can form significant clutter, and there are standard techniques, for example using subset types, to define propositions carefully so that they have minimal or no such witnesses.

We avoided taking this approach, using $Fin(T)$ as a type, partly because of a wish to keep a straightforward classical reading. Perhaps though this is not important when so many of our concerns are with constructivity.

## 7.2 Use of quotient types

Due to the richness of Nuprl's type theory, type-checking is undecidable. In practice, heuristics help carry out most simpler type checking tasks completely automatically. However, Nuprl's quotient types introduce a new dimension of variability into the problem. Frequently we use a function with domain type $T$ where a function with domain type $T//E$ is expected, and we then repeatedly get proof obligations to show that the function respects $E$.

We realize that we need to introduce a discipline for use of quotient types, where, as much as possible, such problems are localized to the right-hand-side of definitions that are type-checked just once, and then always exploited in proofs with the help of characterizing lemmas, rather than definition expansion. We now have several similar proposals for such a discipline, but

didn't have the time to try one in this formalization. One key aspect of these proposals is that injections into quotient types are always explicitly tagged. This helps both the type checker in its type-inference, and the reader in understanding what terms in Nuprl's computation language are denoting.

For example, if $x$ is of type $T$ and $E$ is an equivalence relation over $T$, we might have the injection of $x$ into $T//E$ written as $[x]\{T//E\}$. For projections out of quotient types, we might have a projection operator written $qproj\{T//E\}(f)$ that takes a function $f$ of type $T \to S$, and turns it into a function of type $T//E \to S$. The type checking conditions for $qproj$ would include the requirement that $f$ be shown to respect $E$.

Using these injection and projection operators does not free us from checking that equivalence relations are respected, but it does make the location of those checks more predictable. Analogous operators are required when quotient types are implemented in strongly-typed type theories such as ECC [14], and when working with quotient structures in set theory.

### 7.3 Inadequacies in construction of the minimum automaton

A hard-to-understand definition in this formalization is that of the *MinAuto* function (see Section 6.2). There, the intended meaning of the state transition function mapping equivalence classes to equivalence classes is only apparent when we look at the type the function is supposed to have. If we write the transition function definition as

$$qproj\{MinSt(Auto)\}(\lambda x : Alph\ list.\ \lambda a : Alph.\ [a.x]\{MinSt(Auto)\}),$$

instead of

$$\lambda x,\ a.\ (a.x),$$

its meaning is more immediately evident. Here we have used the quotient type injection and projection operators described in Section 7.1 as well as type annotations on the lambda terms, again to help both readability and typechecking.

Another perhaps more serious defect of our construction of $MinAuto(Auto)$ is that it is computationally trivial. If we imagine applying $MinAuto(Auto)$ to some input string, then it does nothing more than copy that string, and pass it to $Auto$ to check if it should be accepted.

Creating a minimization function that actually does the work of computing a minimum automaton is not difficult, though we have not carried this out yet. We need to define a type of automata, $MinAuto'$ say, in which automata are

represented by finite data structures (integers, pairs, and lists, for example), not functions. The key is to exploit the function we can synthesize from the proof of $Fin(MinAuto(Auto))$. Given $Auto$ as argument, this function can compute the size $n$ of the minimum automaton accepting the language $Auto$ accepts and can provide mapping functions between $MinSt(Auto)$ and $\mathbb{N}n$. Using these mapping functions, we can construct a function that, when evaluated on argument $MinAuto(Auto)$, returns the finite data structures for a minimum automaton that accepts the same language as $Auto$.

### 7.4 Computational complexity of synthesized algorithms

With the proofs as we initially completed them, the time complexity of several extracted functions, including the size function described in Section 7.3, was exponential in the number of states. Aleksey Nogin at Cornell has recently reworked some of the proofs and introduced alternate auxiliary functions to reduce the complexity of the size function to a low-order polynomial. His work is viewable at the Nuprl web site (see Section 1.5). Folowing the approach described in Section 7.3, we should be able to extend Nogin's work so that we can synthesize an automata miniminization function of low polynomial time complexity.

## 8 Summary of Results

- We were successful in formalizing the Myhill-Nerode theorem in constructive type theory.
- We did not find errors in the statement or proof of the theorem in Hopcroft and Ullman. We did note Hopcroft and Ullman's elision of more-routine definitions and proofs. For example, they employ but do not define an isomorphism relation on automata, and they claim but do not prove that a mapping between the sets of states of two automata is an automata isomorphism.
- To make the Myhill-Nerode theorem constructively provable, we needed to add conditions on the decidability of language membership to two of the three equivalent propositions in its statement.

  Constructivity considerations when reasoning about finiteness forced us to consider how various automata properties can be computed. For example, by a combination of explicit introduction, and synthesis from appropriate constructive proofs, we introduced functions for

  - determining whether a state of an automaton is accessible, and, if so, what input string would put the automaton in that state,

– testing whether two states are equivalent.

Such functions can form the core of a function for carrying out the minimization procedure. Initially they had time complexity exponential in the number of states, but, in ongoing work, we have introduced alternate functions with low-order polynomial complexity.

# 9 Conclusions and Future Work

With this article and the accompanying online material, we have a presentation of a piece of mathematics that is completely precise and that can be viewed at differing levels of detail. We have argued that such presentations are superior to textbook only presentations, and we believe that we have begun to demonstrate this.

At Cornell we are currently experimenting with other examples of such formally-grounded explanations. We have already formalized other parts of Hopcroft and Ullman, including account of grammars and of nondeterministic automata. We judge that it would be possible to formalize Chapters 1–9 with our four person team in about eighteen months.

The collaboration methods we have learned would extend to larger teams. It would be especially interesting to collaborate with other theorem proving systems as Howe and his colleagues are doing with HOL and Nuprl [19, 18]. Much of a classical treatment of languages can easily be re-interpreted constructively. It would be especially fruitful to collaborate with teams using other constructive provers such as Alf, Coq, Lego, or Isabelle with its Martin-Löf-type-theory object logic. Although these provers are based on different formalizations of constructive mathematics, they all share the critical properties that computational notions can be expressed and that programs can be synthesized from proofs.

One weak point of our online presentation is the readability of proofs. We see no reason why online formal proofs should not be at least as clear as any informal proofs. Unlike many other provers, Nuprl maintains a proof tree datastructure that already assists us in generating comprehensible presentations of proofs. Ideas we are currently exploring to improve readability include the grouping of lower level tactic sequences under user supplied comments and the suppression of less-important proof branches. We are also following the work of the Centaur group to make proofs more readable [3, 29], and we expect to use the modularity feature of the Nuprl-Light refiner [13] to help us better structure theories.

# 10 Acknowledgments

We acknowledge the support granted by the National Science Foundation and the Office of Naval Research. We also thank Stuart Allen and Karl Crary for the discussions and input concerning this topic, Karla Consroe for help in preparing the document, and Aleksey Nogin for his work improving the efficiency of many critical proofs.

# A Index for Notation

The numbers on the right refer to pages where the notation is first introduced.

**Types**

| | | |
|---|---|---|
| $\{i\ldots\}$ | integers from $i$ upwards | 6 |
| $\{x{:}T \mid P_x\}$ | set type | 8 |
| $S \times T$ | product of types $S$ and $T$ | 6 |
| $S \to T$ | functions from type $S$ to type $T$ | 7 |
| $S\ list$ | list type | 6 |
| $T//E$ | quotient type | 12 |
| $x, y{:}T//x\ E\ y$ | long form for $T//E$ | 12 |
| $Automata(Alph; St)$ | automata over alphabet $Alph$ and states $St$ | 10 |
| $\mathbb{B}$ | booleans | 6 |
| $Language(Alph)$ | languages over alphabet $Alph$ | 9 |
| $MinSt(A)$ | states for $MinAuto(A)$ | 20 |
| $\mathbb{N}$ | naturals $\{0,\ 1, \ldots\}$ | 6 |
| $\mathbb{N}k$ | naturals $\{0\ldots k-1\}$ | 6 |
| $\mathbb{P}$ | propositions | 7 |
| $\mathbb{U}$ | universe of types | 8 |
| $\mathbb{Z}$ | integers | 6 |

**Predicates and Operators**

| | | |
|---|---|---|
| $f\ a$ | function application | 7 |
| $a.s$ | list building ("consing") | 6 |
| @ | list append | 6 |
| $[x]$ | equivalence class of $x$ | 14 |
| $\langle a, b\rangle$ | pairing | 6 |
| $==$ | definitional equality | 6 |
| $==_r$ | recursive function definition | 7 |
| $L = M$ | language equality | 9 |
| $\hat{\delta}$ | extended next-state function | 10 |
| $\uparrow b$ | asserts that boolean expression $b$ is true | 6 |

| | | |
|---|---|---|
| $\delta'$ | Nuprl version of $\hat{\delta}$ | 11 |
| $\delta(A)$ | next-state function of automaton $A$ | 10 |
| $\lambda x.\, b$ | lambda abstraction | 6 |
| $A(l)$ | state of automaton $A$ after input of $l$ | 11 |
| $Dec(P)$ | $P$ is decidable | 15 |
| $EquivRel(T; E)$ | $E$ is an equivalence relation over type $T$ | 15 |
| $F(A)$ | set of final states of automaton $A$ | 10 |
| $ff$ | boolean false | 6 |
| $Fin(T)$ | type $T$ is finite | 8 |
| $hd(l)$ | head of list $l$ | 6 |
| $I(A)$ | initial state of automaton $A$ | 10 |
| $L(A)$ | language accepted by automaton $A$ | 11 |
| $L_b(A)$ | boolean version of $L(A)$ | 11 |
| $MinAuto(A)$ | minimum automaton for $L(A)$ | 20 |
| $nil$ | empty list | 6 |
| $null(l)$ | function testing if list $l$ is empty | 6 |
| $1\text{-}1\text{-}Corresp(S; T)$ | 1-1 correspondence between types $S$ and $T$ | 8 |
| $R'(g)$ | like $R(L)$ but typing involves quotient type | 17 |
| $R(L)$ | relation induced by language $L$ | 16 |
| $tl(l)$ | tail of list $l$ | 6 |
| $tt$ | boolean true | 6 |

# References


[1] Stuart F. Allen. A Non-type-theoretic Definition of Martin-Löf's Types. In *Proc. of Second Symp. on Logic in Comp. Sci.*, pages 215–224. IEEE, June 1987.

[2] E. Bishop. *Foundations of Constructive Analysis.* McGraw Hill, NY, 1967.

[3] P. Borras, D. Clément, T. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. Centaur: the system. In *Software Engineering Notes*, volume 13(5). Third Symposium on Software Development Environments, 1988.

[4] N. Bourbaki. *Elements of Mathematics, Theory of Sets.* Addison-Wesley, Reading, MA, 1968.

[5] Robert L. Constable. Experience using type theory as a foundation for computer science. In *Proceedings of the Tenth Annual IEEE Symposium on Logic in Computer Science*, pages 266–279. LICS, June 1995.

[6] Robert L. Constable, Stuart F. Allen, H.M. Bromley, W.R. Cleaveland, J.F. Cremer, R.W. Harper, Douglas J. Howe, T.B. Knoblock, N.P.

Mendler, P. Panangaden, James T. Sasaki, and Scott F. Smith. *Implementing Mathematics with the Nuprl Development System.* Prentice-Hall, NJ, 1986.

[7] Thierry Coquand and G. Huet. The Calculus of Constructions. *Information and Computation*, 76:95–120, 1988.

[8] P. J. Denning, J. B. Dennis, and J. E. Qualitz. *Machines, Languages, and Computation.* Prentice-Hall, 1978.

[9] Max B. Forester. Formalizing constructive real analysis. Technical Report TR93-1382, Computer Science Dept., Cornell University, Ithaca, NY, 1993.

[10] J-Y. Girard, P. Taylor, and Y. Lafont. *Proofs and Types.* Cambridge Tracts in Computer Science, Vol. 7. Cambridge University Press, 1989.

[11] Michael Gordon and Tom Melham. *Introduction to HOL: a theorem proving environment for higher-order logic.* Cambridge University Press, 1993.

[12] Michael Gordon, Robin Milner, and Christopher Wadsworth. *Edinburgh LCF: a mechanized logic of computation,* Lecture Notes in Computer Science, Vol. 78. Springer-Verlag, NY, 1979.

[13] Jason J. Hickey. Nuprl-light: An implementation framework for higher-order logics. In *14th International Conference on Automated Deduction*, 1997.

[14] Martin Hofmann. *Extensional concepts in intensional type theory.* PhD thesis, University of Edinburgh, Laboratory for Foundations of Computer Science, 1994.

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

[16] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, Reading, Massachusetts, 1979.

[17] Douglas J. Howe. Implementing number theory: An experiment with Nuprl. In *8th International Conference on Automated Deduction*, volume 230 of *Lecture Notes in Computer Science*, pages 404–415. Springer-Verlag, July 1987.

[18] Douglas J. Howe. Importing mathematics from HOL into Nuprl. In J. von Wright, J. Grundy, and J. Harrison, editors, *Theorem Proving in Higher Order Logics*, volume 1125, of *LNCS*, pages 267–282. Springer-Verlag, Berlin, 1996.

[19] Douglas J. Howe. Semantic foundations for embedding HOL in Nuprl. In Martin Wirsing and Maurice Nivat, editors, *Algebraic Methodology and Software Technology*, volume 1101 of *LNCS*, pages 85–101. Springer-Verlag, Berlin, 1996.

[20] Paul B. Jackson. *Enhancing the Nuprl Proof Development System and Applying it to Computational Abstract Algebra.* PhD thesis, Cornell University, Ithaca, NY, January 1995.

[21] Miroslava Kaloper and Piotr Rudnicki. FSM_1: Minimization of finite state machines. *Journal of Formalized Mathematics*, 6, 1994. Electronically accessible from
`http://mizar.uw.bialystok.pl/JFM/` or
`http://web.cs.ualberta.ca/~piotr/Mizar/`.

[22] C. Kreitz. Constructive automata theory implemented with the Nuprl proof development system. Technical Report 86-779, Cornell University, Ithaca, New York, September 1986.

[23] Per Martin-Löf. Constructive mathematics and computer programming. In *Sixth International Congress for Logic, Methodology, and Philosophy of Science*, pages 153–75. North-Holland, Amsterdam, 1982.

[24] Per Martin-Löf. *Intuitionistic Type Theory, Studies in Proof Theory, Lecture Notes.* Bibliopolis, Napoli, 1984.

[25] B. Nordstrom, K. Petersson, and J. Smith. *Programming in Martin-Löf's Type Theory.* Oxford Sciences Publication, Oxford, 1990.

[26] L. C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science.* Springer-Verlag, 1994.

[27] Robert Pollack. *The Theory of LEGO:A Proof Checker for the Extended Calculus of Constructions.* PhD thesis, University of Edinburgh, Dept. of Computer Science, JCMaxwell Bldg, Mayfield Rd, Edinburgh EH9 3JZ, April 1995.

[28] N. Shankar. Rabin-Scott automata equivalence / flaw in Hopcroft-Ullman proof. Posting to PVS mail-list on 3 December 1995. Posting archived at
`http://www.csl.sri.com/pvs/mail-archive/pvslist95.txt.gz`.

[29] L. Théry, Y. Bertot, and G. Kahn. Real theorem provers deserve real user-interfaces. In *Software Engineering Notes*, volume 17(5), pages 120–129. 5th Symposium on Software Development Environments, 1992.

[30] S. Thompson. *Type Theory and Functional Programming.* Addison-Wesley, 1991.

[31] Debora Weber-Wulff. *Contributions to Mechanical Proofs of Correctness for Compiler Front-Ends.* PhD thesis, Institut für Informatik und Praktische Mathematik der Christian-Albrechts-Universität zu Kiel, April 1997.