Constructively Formalizing Automata Theory*

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

Cornell University

February 25, 1997

Abstract

This article and the World Wide Web library display of computer checked proofs is an experiment in the formalization of computational mathematics. [1] Readers are asked to judge whether this formalization adds value in comparison to a careful informal account. The topic is *state minimization* in finite automata theory. We follow the account in Hopcroft and Ullman's book *Formal Languages and Their Relation to Automata* where state minimization is a corollary to the Myhill/Nerode theorem. That book constitutes one of the most compact and elegant published accounts. It sets high standards against which to compare any formalization.

The Myhill/Nerode theorem was chosen 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 forces us to treat quotient sets computationally.

The theorem proving methodology used here is based on the concept of tactics pioneered by Robin Milner. The theorem prover we use is Nuprl ("new pearl") which, like its companion, HOL, is a descendent of the LCF system of Milner, Gordon and Wadsworth. It supports constructive reasoning and computation.

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

*Supported in part by NSF grants CCR-9244739,CCR-9148222.

[1] The library's url is www.cs.cornell.edu/Info/Projects/NuPrl/nuprl.html.

Contents

1 Introduction

1.1 Goals

It is widely believed that we know how to formalize large tracts of *classical* mathematics — namely write in the style of Bourbaki [3] 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. Indeed, the topic of state minimization of finite automata has been formalized in Mizar [15]. Despite this belief, and the many formalizations accomplished, massive formalization is not a *fait accompli*, and there are many research issues related to the formalization effort and its computerization (see [21]). Indeed, some doubt the appropriateness of set theory for expressing working mathematics [7].

In contrast, there is no general agreement on how to formalize *computational mathematics*[2]. This article is a contribution to understanding that task and exploring one approach to it. Our approach stresses that formalized computational mathematics can be useful in carrying out computations. One of our subgoals is to illustrate this utility in a particular way. We want to show that constructive proofs can be used to synthesize programs.

Specifically, we want to examine whether *constructive type theory* is a natural expression of the basic ideas of computational mathematics in the same sense that set theory is for "purely classical mathematics." We have explored this question for elementary number theory (num_thy_1a), for the algebra of polynomials ($general\_algebra$), for elementary analysis, and for elementary logic, as well as in other less systematic efforts. The type theory we use is based on Martin-Löf's semantics [19].

In this paper we examine these ideas in the setting of *basic automata theory*. There are several reasons for this choice.

1. The subject of formalization is closely allied with other subjects in computer science (such as programming languages and semantics, applied logic, automated deduction, problem solving environments, computer algebra systems, knowledge representation, and computing theory). Also automata theory is widely taught in computer science [16] and used in building systems. So we hope for *a large sympathetic audience* for the material we create.

2. One of the most basic theorems in finite automata theory, the Myhill/Nerode Theorem, illustrates beautifully the idea that algorithms can be extracted from constructive proofs; so it is a good test for our main subgoal.

[2] Even worse, few people appreciate that this is a significant new problem (see [4]).

3. The account of Myhill/Nerode in Hopcroft and Ullman's famous book [11] is constructive except for a few small points, one buried deep in the proof. The nonconstructive steps are easy to miss. We show how to make the proof entirely constructive with a trivial change in the theorem.

4. Automata theory appears to be well suited for expression in type theory. If our account is not convincing on this material, then our task will be harder than we imagine. Moreover, the formal account seems to be clarifying and helpful even in the case of one of the most compact and elegant informal expositions of automata theory. So our claim that formalism adds value is put to a good test.

5. The Myhill/Nerode theorem illustrates a phenomenon that nonspecialists are curious about. Why does formalization expand the work and the text by such large factors (at least a factor of 5 in this case and "under the surface" by 3 orders of magnitude)? Moreover, because the formalization of this theorem relies heavily on many results in list theory and a few in algebra, we can see the impact of a *knowledge base* on the formalization task. Because it required building new basic material about the quotient type, we see why formalization efforts are so laborious.

6. The existence of an earlier formalization of the pumping lemma from automata theory by Christoph Krietz in 1988 [17] in Nuprl 3 allows us to compare the progress made in the tactic collection from version 3 (1988) to version 4.2 (1995).

7. Finally, the formalization reveals some technical problems about how to formalize computational mathematics. The question involves the well-known methodology of *propositions-as-types*. We have found yet another complication in this principle (see also Allen [1]). The issue concerns the right equality relation on propositions. We will point out that our formalization is in some sense "pushing the envelope" of the methodology presently being used in the field.

1.2 Interpretations of the Mathematics

Even without formalization, expressing the ideas of Hopcroft and Ullman in type theory (especially Nuprl) opens the possibility for *new interpretations* of their mathematics. Their definitions refer to a fragment of set theory on which they informally define algorithms and procedures, but not in a systematic way. The first thing we show is how to treat computation systematically and foundationally with minor changes in their text.

Our presentation then enables a person to imagine that all of the mathematics is classical, as Howe's work illustrates [13]. It also allows the interpretation of *recursive mathematics* that all functions are given by "Turing machines" or Lisp programs. It also allows an Intuitionistic interpretation. One way to describe this style is to relate it to the work of Bishop [2] who showed that real, complex, and abstract analysis could be formalized in this *neutral* way.

1.3 Outline

In section 1 we present the basic ideas from Nuprl needed for this article. Surprisingly little is required, and we claim that this basic material is mostly as "readable" as the mathematical preliminaries in any undergraduate textbook at the level of Hopcroft and Ullman. Section 3 corresponds

to Hopcroft and Ullman's Chapter 1. We try to follow that account closely. Section 4 provides the preliminaries on automata, following Hopcroft and Ullman very closely. Section 5 proves the Myhill/Nerode Theorem. Section 6 discusses proofs of state minimization, filling in an omission in their proof, and simplifying, thereby showing an advantage of formalization. Section 7 discusses an issue of type theory raised by our work.

The key ideas of the formalization are presented here in a self-contained way, but the reader will understand the issues more thoroughly by reading either the Web library (www.cs.cornell.edu/Info/Projects/NuPrl/nuprl.html) or using the Nuprl system to read the actual libraries. Indeed, the article was originally written as an html document to accompany the actual on-line theorems. Various references are made to Nuprl libraries in the text. In the html version these were hot references (one could click on them to open the referenced files).

2 Type Theory Preliminaries

2.1 Basic Types

The integers $\mathbb{Z} = \{0, \pm 1, \pm 2, \ldots\}$ are a primitive type of Nuprl with primitive operations of

$$+ \ , \ - \ , \ \cdot \ , \ \div \ , \text{ rem (for remainder)}.$$

Equality, $x = y$ in $\mathbb{Z}$, and order, $x < y$, are also primitive. The natural numbers $\mathbb{N}$ are defined as $\{i{:}\mathbb{Z} \mid 0 \le i\}$, and the initial segments $\mathbb{N}k$ are $\{i{:}\mathbb{Z} \mid 0 \le i < k\}$. The segment $[i \ldots j] = \{x{:}\mathbb{Z} \mid i \le x \le j\}$ and $[i \ldots j^-] = \{x{:}\mathbb{Z} \mid i \le x < j\}$. So $\mathbb{N}k = [0 \ldots k^-]$. Basic facts about these types can be found in these libraries: *int_1*, *int_2*, *num_thy_1a*.

Given any type A, we can form the type of *lists* whose elements are all from A. This is called *A list*. The empty list is denoted nil regardless of the type A. The list_construction (or "consing") takes an element h of A and an *A list*, say t, and forms a new list denoted $h.t$. It adds the element h from A to the *head* of the list t.

The *append* operation on lists is critical in this article; it is denoted $x@y$ and is defined in the usual recursive way

$$nil@y = y$$
$$(h.t)@y = h.(t@y).$$

The Booleans, $\mathbb{B}$, consists of tt and ff denoting true and false. The normal if-then-else case selection is available along with the standard operations $\wedge_b, \vee_b, \Rightarrow_b$. We write the subscript "b" to distinguish these operations from the propositional connectives, $\wedge, \vee, \Rightarrow$, (see *bool_1*).

2.2 Cartesian Products

If A and B are types, then so is their *Cartesian product*, $A \times B$, whose elements are ordered pairs, $\langle a, b\rangle$ with $a \in A$ and $b \in B$. For example $\mathbb{Z} \times \mathbb{Z}$ consists of the points with integer co-ordinates in the plane. If $p \in A \times B$ then there are several common ways to denote the first and second components of the pair. Here are some of the common ways: first(p), 1of(p) or $p.1$ for the first, and second(p), 2of(p) or $p.2$ for the second. We have

$$\langle a, b\rangle.1 = a \text{ in } A \text{ and } \langle a, b\rangle.2 = b \text{ in } B.$$

An n-ary product, say $A \times B \times C$ is regarded as $A \times (B \times C)$. In general $A_1 \times \cdots \times A_n$ is $A_1 \times (A_2 \times \cdots \times A_n)$. Given $p \in A \times B \times C$, the 2nd component is $p.2.1$ and the 3rd is $p.2.2$. We'll see these selectors in the definition of an automaton (section 4.1).

2.3 Function Types

If A and B are types, then $A \rightarrow B$ denotes the type of all *computable* (total) functions from A to B. The canonical elements of this type are lambda terms, $\lambda(x.b)$. If we let $b[a/x]$ denote the *substitution* of the term a for all free occurrences of x in b, then we require of $\lambda(x.b)$ that $b[a/x] \in B$ for all terms a denoting elements of A. If $f \in A \rightarrow B$ and $a \in A$, then fa denotes the application of f to argument a. We know that $fa \in B$. See *fun_1*.

Recursive functions are defined in the style of ML. We use the form $lhs ==_r rhs$ to introduce a recursive definition, for example, $fact(n) ==_r n$ *if* $n = 0$ then 1 else $n * fact(n-1)$ *fi*. This invokes an ML tactic called *add_rec_def* with lhs, rhs as arguments. The tactic adds an abstraction named by the function name, e.g. *fact*. The abstraction is based on a recursion combinator such as Y. For example, $Y\lambda\ fact,\ n.\ if\ n = 0$ then 1 else $n * fact(n-1)$ *fi*) is the combinator added for factorial.

The abstraction is made invisible by various tactics that fold and unfold instances of the definiton, so the user need not be aware of the underlying λ-calculus foundations. In the automata library we use a recursive function to define the analogue of δ^* from HU. Informally, the definition is

$$\delta^* l ==_r \ \textit{if } null(l) \text{ then } I(DA) \text{ else } \delta(\delta^*(tl\ l))hd\ l\ \textit{fi}.$$

The actual definition is

$$DA(l) ==_r \ \textit{if } null(l) \text{ then } I(DA) \text{ else } \delta_{DA}(DA(tl\ l))hd\ l\ \textit{fi}.$$

2.4 Propositions and Universes

In so-called "classical" accounts of logic, a proposition has a truth value in $\mathbb{B}$. Consequently, propositions can be treated as boolean expressions. This boolean-valued account is more restrictive than the one we need in order to discuss computability issues, so we adopt a more abstract account of propositions. We want to consider both the *sense* and the *truth* of a proposition. In particular we are interested in their *computational sense*.

We will, of course, talk about the truth value of a proposition as well as its sense. The type of all *propositions* needed in this article is denoted $\mathbb{P}$. Nuprl can express "higher order logic" as well, in which case "larger" propositions are needed. See Jackson [14] or [5] for fuller accounts of higher order logic.

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:

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

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

$\forall x\!:\!A.P(x)$ "for all x of type A, $P(x)$ holds,"
$\exists x\!:\!A.P(x)$ "for some x of type A, $P(x)$ holds."

Also associated with every type A is the atomic *equality proposition*, $(x = y \text{ in } A)$. The definition of this equality is given with each type.

In classical logic the boolean value tt is considered the same as the true proposition $\top$ and ff is identified with $\bot$. But the proposition we associate with a boolean expression $bexp$ is this atomic equality: $bexp = tt$ in $\mathbb{B}$. Given $bexp$ we denote the corresponding proposition as True($bexp$). Clearly we know True(tt) iff $\top$ and True(ff) iff $\bot$. Sometimes we denote True($bexp$) by $\uparrow bexp$ for short. We call this up arrow "assert."

The types we need belong to a *universe*, $\mathbb{U}$.[3] If A and B are types then we have seen that $A \to B$ and $A\ list$ are also types.

2.5 Subtypes and Finiteness

We use a natural notion of subtype. If A is a type and $P\!:\!A \to \mathbb{P}$ is a propositional function, then $\{x\!:\!A \mid P(x)\}$ denotes the type of all elements of A satisfying P. To know that $a \in \{x\!:\!A \mid P(x)\}$ we must build the element a and find a proof of $P(a)$. There is a subtle computational point about these sets, namely a function f from $\{x\!:\!A \mid P(x)\}$ to B does not have access to a proof that $P(x)$ holds when calculating its value $f(x)$.[4]

A *finite type* is one which can be put into a 1-1 correspondence with $[1 \ldots n]$; its *cardinality* is n. We write $Fin(T)$ to mean that T is finite. This means *we can find* a number n and functions f and g such that

$$T \underset{g}{\overset{f}{\leftrightarrows}} [1 \ldots n]$$

such that f and g are *inverses* of each other; that is

$$\forall x\!:\!T.\ (f(g(x)) = x \text{ in } T)\ \ \&\ \ \forall i\!:\![1 \ldots n].\ (g(f(i)) = i \text{ in } \mathbb{Z}).$$

The definition of 1-1 correspondence is in *fun_1*, and finiteness is in *automata_2*.

Here is an important fact about finite types. We say that a type T is *discrete* iff there is a function $eq_T : T \times T \to \mathbb{B}$ such that $x = y$ in T iff $eq_T(x, y) = tt$, that is, T is discrete iff equality on T is decidable.

Fact: If T is finite, then it is discrete.

This is true because $[1 \ldots n]$ is discrete for any n; thus to decide $eq_T(x, y)$, ask whether $g(x) = g(y)$ in $\mathbb{N}$ for the function g witnessing $T's$ finiteness.

[3] We only need the universe of *small types* denoted simply $\mathbb{U}$. For a full discussion of universes, see Allen [1] as well as Jackson [14].

[4] A discussion of the constructive meaning of these types is beyond the scope of this work, but see [5, 14, 20].

2.6 Algebraic Structures and Dependent Types

In algebra and automata theory definitions are given using so-called (algebraic) *structures*. For example, a *monoid* is a type M together with a binary operation $f : M \times M \to M$ and an element, $e \in M$. The operation is associative and e is an identity. The monoid is the triple $\langle M, f, e \rangle$. The "signature" or type of this structure is $T : \mathbb{U} \times op : (T \times T \to T) \times i : T$.

This type is called a *dependent product* in Nuprl. The basic underlying form is $T : \mathbb{U} \times F(T)$ where F is a function from types to types, e.g. $F(T) = (T \times T \to T) \times T$.

We can explain the bound variables T, op, i in two ways. In Jackson's thesis these arise by iterating the binary dependent product construction as follows. Let $\mathbf{1}$ be a type with exactly one element, say $\{x : \mathbb{Z} \mid x = 1\}$. Then take $i : T \times \mathbf{1}$ as a type with T as a parameter. Call it $S_1(T)$. Next build $op : (T \times T \to T) \times S_1(T)$. Call this $S_2(T)$, finally build $T : \mathbb{U} \times S_2(T)$. We see that T, op, i are just the binding variables used in creating the product.

Another approach is to consider the type of names $\{op, i\}$ (a subtype of Atom in Nuprl) and define a function $S_2(T) : \{op, i\} \to \mathbb{U}$ where $S_2(T)(op) = (T \times T \to T)$ and $S_2(T)(i) = T$. Then the monoid signature is $T : \mathbb{U} \times S_2(T)$. This is the approach taken by Jason Hickey [10].

2.7 Reading Nuprl Proofs

Proofs in Nuprl are trees. The nodes of the tree consist of *sequents* and *justifications*. A sequent is a list of formulas, called hypotheses, paired with a single formula called the *goal*. The hypotheses are numbered $H_1, \ldots, H_n$. These sequents, also called *goals*, are displayed as

$$H_1, \ldots, H_n \vdash G.$$

The symbol, $\vdash$, called a *turnstile*, separates hypotheses from conclusions. A sequent is provable *iff* we can prove the goal G from the hypotheses H_i.

The justification component of a node gives a reason that the goal sequent follows from the subgoal sequents generated by the justification. Justifications are displayed as

by justification text.

A sequent, its justification, and subgoals constitute an individual *inference* in the proofs. Here is a schematic example:

$$1.P \Rightarrow Q \;\; 2.P \vdash Q \text{ by } D1$$

$$/ \quad \backslash$$

$$1.P \;\; 2.Q \vdash Q \text{ by Hyp2.} \qquad 1.P \vdash P \text{ by Hyp1.}$$

no subgoals no subgoals

In general an inference can look like

$$\bar{H}_0 \vdash G_0 \text{ by } J$$

$$/ \qquad | \qquad \backslash$$

$$\bar{H}_1 \vdash G_1 \quad \bar{H}_2 \vdash G_2 \cdots \bar{H}_n \vdash G_n$$

where the $\bar{H}_i$ are lists of formulas and G_i are single formulas.

Nuprl provides various tree traversal operations to facilitate "reading" a proof tree and modifying it. These proof trees are meant to be read with these tree walking operations. But we also want to print the trees. There are various schemes for doing this. We write vertical lines in the left margin to connect a subtree to its present goal. Here is how the first example would be printed.

1. $P \Rightarrow Q$
2. P

$\vdash Q$ by $D1$

$\vdots \ddots 1.P$

$\vdots \quad \vdash$ by Hyp1

$\vdots$

1. P
2. Q

$\vdash Q$ by Hyp2

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*. An *Alphabet* is any *finite* set of *symbols*. They consider only a countably infinite set from which all symbols will be drawn, and they leave *open* just what these symbols will be: "any countable number of additional symbols that the reader finds convenient may be added."

We adopt all the ingredients of this definition without needing to specify a countably infinite set. We simply require that an alphabet, $Alph$, is a finite type; to say this we first declare $Alph \in \mathbb{U}$. Since $\mathbb{U}$ is open, the definition is open. Then we require that $Alph$ be finite, postulating $Fin(Alph)$. One consequence of finiteness is that the equality relation on $Alph$ is decidable. This is true of any finite set as we noted in section 2.

In Hopcroft and Ullman we read this:

> A sentence over an Alphabet is any *string* of finite length composed of symbols from the *Alph*abet. Synonyms for sentence are *string* and word.

This definition is incomplete because they do not define *string*. We have to learn later what it really means. The lack of a fixed definition allows the authors to switch between equivalent notions

of list or array or string depending on their needs. We will note this later. Essentially they are introducing an *abstract* type without fixing the operations in advance.

They introduce these notations. If V is an *Alph*abet, then V^* is the set of all sentences on V. They include ε the empty sentence. V^+ is V^* without ε. A *language* is any *subset* of V^*. We use a concrete definition. A sentence for us is a list of elements from *Alpha*, that is, members of the type *Alph list*. The nil list is what we call the empty sentence. Example: if $Alph = \{0, 1\}$ then $Alph\ list = \{nil, (0), (1), (0\ 0), (0\ 1), (1\ 0), (1\ 1), (0\ 0\ 0), \ldots\}$.

A language is given by some condition for membership, say a predicate L that specifies when an element of *Alph list* belongs to the language. So a language is a *propositional function* over *Alph list*, namely an element of $Alph\ list \to \mathbb{P}$. We use Language($Alph$) to denote the type of *languages over Alph*. In the library definitions are called *abstractions* and have a tag A, as in:

A languages Language($Alph$)$= = Alph\ list \to \mathbb{P}$

In the library *Lang_1* we give many operations on languages:

union	$L \cup M$
intersection	$L \cap M$
complement	$\neg L$
product	$L \otimes M$
power	$L \uparrow n$
closure	$L \uparrow \infty$

We define equality of languages $L = M$ in Language($Alph$).

Hopcroft and Ullman raise these questions. How do we specify a language? Does there exist a finite representation for any language? They note that *Alph list* is countably infinite and hence Language($Alph$) is uncountable. So they conclude that there are many more languages than finite representations.

Our views are consistent with these, *but we allow other interpretations as well.* We say that a language is given by a propositional function, say L. (They say by a *set* but a set is not a finite representation.) One could consistently take the view that every function is given by an algorithm, and every algorithm is finitely representable. Hence all languages are finitely representable. This is the interpretation of so called *recursive mathematics.* All the work we present is consistent with this interpretation, as well, but as mentioned in the introduction we take the neutral view characteristic of "Bishop style" mathematics so all three views of the results are possible.

3.2 Procedures and Algorithms

Section 1.2 of Chapter 1 of Hopcroft and Ullman is concerned with *procedures* and *algorithms*. For us this is part of the basic type theory. Unlike in the case of set theory where computability need not be mentioned, in type theory computability is a basic concept. So we have covered these ideas already in section 1.

It is interesting that Hopcroft and Ullman rely on the concept of an *effective procedure* which is the same open-ended concept that we axiomatize in type theory. Only later, in Chapter 6, do they present Turing machines, a formalization of effective computability. Also, Hopcroft and Ullman consider the subject *metamathematically.* That is, they look at the mathematics from outside. For us that is like noticing properties of the underlying procedures. They do not talk about the

type or the meaning of the procedures only their computational behavior. This is mathematics as influenced by the great results of logic, a new 20th century mathematics.

3.3 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 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 is decidable, that is when there is a function R_L: $Alph$ list $\rightarrow \mathbb{B}$ such that

$$L(x) \text{ iff } R_L(x) = tt \text{ in } \mathbb{B}.$$

Such a language is called *decidable* or *recursive*.

Another way to represent a language L with a function is to provide an enumeration of L, that is a function $E_L \in \mathbb{N} \rightarrow Alph\ list$ such that

$$L(x) \text{ iff } \exists i{:}\mathbb{N}.\ (E_L(i) = x.)$$

The function E_L can also be said to represent L.

Given the function E_L an interesting procedure arises for specifying a language, the procedure is called a (real) *recognizer*. To specify L, we write a function

$$r_E : Alph\ list \rightarrow \mathbb{R}$$
$$\neg(r_E(x) = 0 \text{ in } \mathbb{R}) \text{ iff } x \in L$$
$$r_E(x) = \lambda(n.\text{ if } E_L(n) = x \text{ then } (\mu y \leq n.E_L(y) = x)^{-1} \text{ else } n^{-1}).$$

Hopcroft and Ullman go on to show that given a real recognizer, we can also define an enumerator. Basically we enumerate $L = \{x : Alph\ list \mid r(x) \neq 0 \text{ in } \mathbb{R}_R\}$. We do this uniformly only if the type is non-empty. Then given r, there is an operation $Enum(r)$ which produces a function from $\mathbb{N}$ onto L. Since we are interested mainly in automata and the Myhill-Nerode theorem of Chapter 3, we skip over Chapter 2 on Grammars although it would not be difficult to formalize all of the results there. (The only interesting result is Theorem 2.2—a context sensitive grammar is recursive.)

4 Finite Automata

4.1 Definition

Hopcroft and Ullman say: a *finite automaton* M over an alphabet $Alph$ is a system $(K, Alph, \delta, q_0, F)$ where K is a finite nonempty set of *states*, $Alph$ is a finite *input alphabet*, δ is a mapping of $K \times Alph$ into K, q_0 *in* K is the *initial state*, and $F \subseteq K$ is the set of *final states*.

In Nuprl this definition is formalized nearly verbatim. The "system" is just an element of a product type. We use the notation Automata($Alph$; $States$) to denote the type of all automata with input alphabet $Alph$ and states $States$. An automaton is a triple of transition, initial state and final states.

A automata

Automata($Alph$; $States$)		$== (States \to Alph \to States) \times States \times (States \to \mathbb{B})$
A *DA_act*	δa	$== a.1$ (the first component of a)
A *DA_init*	$I(a)$	$== a.2.1$ (the initial state, the second component)
A *DA_fin*	$F(a)$	$== a.2.2$ (the final states, the third component)
T *DA_act_wf*	$\forall Alph, States : U.\ \forall a :$ Automata($Alph$; $States$)$.\delta a \in States \to Alph \to States$	
T *DA_init_wf*	$\forall Alph, States : U.\ \forall a :$ Automata($Alph$; $States$)$.I(a) \in States$	
T *DA_fin_wf*	$\forall Alph, States : U.\ \forall a :$ Automata($Alph$; $States$)$.F(a) \in States \to \mathbb{B}$	

The leading symbol indicates either an abstraction, A, or a theorem, T.

4.2 Semantics of Automata

A finite automaton DA can be interpreted as a language recognizer by one of the methods discussed in section 2. That is, it defines a function from $Alph\ list$ to $\mathbb{B}$. The language accepted consists of those sentences on which DA computes true (tt).

This meaning of an automaton is given by providing a *meaning function* mapping an automaton DA in Automata($Alph$; $States$) to a formal language, i.e. to a map from $Alph\ list$ into propositions $\mathbb{P}$. We give this meaning by composing 3 simpler functions:

1. A function from an automaton and an input string to a state. This is called
   M *compute_list_ml*
   $DA(l) ==_r S$ *if* $null(l)$ then $I(DA)$ else $\delta DA\ DA(tl(l))hd(l)\ fi$

2. Associating with the resulting state of *compute_list_ml* a boolean value using the final state component, a function F: States $\to \mathbb{B}$.

3. Associating with the automaton the propositional function saying that the final state is tt. Let F be the final state function.

   A *auto_lang*
   $L(DA) == \lambda l.$True$(F(DA(l)))$ (We can also write this as $\lambda l. \uparrow (F(DA(l)))$.)

Hopcroft and Ullman follow a similar approach but they use only the first function and leave the other two implicit since they are so simple. They define the first function as

$$\hat{\delta}(q, \varepsilon) = q$$

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

for x a string and a a character of the alphabet. They say:

> a sentence x is said to be accepted by M if $\delta(q_0, x) = p$ for some p in F. The set of all x accepted by M is denoted $T(M)$. That is, $T(M) = \{x \mid \delta(q, x)$ is in $F\}$.

This is a very elegant definition, it can be even more compactly written without reference to q in $\hat{\delta}$, namely

$$\hat{\delta}(\varepsilon) = q_0$$

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

We adopt this definition. But notice that because Hopcroft and Ullman are not definite about the meaning of sentences, the symbol xa can be read either as $x@a$ or $x.a$. In the later case the normal convention is to write the cons operation either as $cons(a, x)$ or $a.x$. Using the cons operation keeps the reasoning simple and direct. So our definition of the computation of state of automaton DA on string x (thought of as $hd(x).tl(x)$) is :

$$\begin{aligned} DA(nil) &= I(DA) \\ DA(x) &= \delta(DA(tl(x)))hd(x). \end{aligned}$$

Recall that $I(DA)$ is the initial state.

This elegant definition has the consequence that we imagine processing a string from *tail to head*, and because we normally write lists from *head to tail* (left to right), this means that we should *think of processing the list from right to left rather than left to right as in Hopcroft and Ullman.*

There is potential for real confusion about this, and it is best to banish the notions of "left" and "right" as much as possible. It is especially confusing to think about "left and right invariant equivalence relations" in light of our reverse way of processing. (We need to use "left invariance" when Hopcroft and Ullman use right.)

4.3 Equivalence Relations and Quotient Types

Hopcroft and Ullman say: "A *binary relation* R on a set S is a set of pairs of elements in S. If (a, b) is in R, then we are accustomed to seeing this fact written aRb."

In set theory a set of pairs R can be defined in terms of its characteristic function $R : S \times S \to \mathbb{B}$. In type theory these sets are expressed as functions from $S \times S$ into $\mathbb{P}$ the propositions. In type theory all functions are computable, so we do not use maps into $\mathbb{B}$ unless the set or relation is decidable.

A relation R on S is said to be:

1. *reflexive* iff for each s in S, sRs,
2. *symmetric* iff for each $s, t \; in \; S, sRt$ implies tRs,
3. *transitive* iff for each $s, t, u \; in \; S, sRt \; and \; tRu$ imply sRu.

A reflexive, symmetric and transitive relation is called an *equivalence* relation. For an equivalence relation, we sometimes write $x = y \; mod \; R$ for xRy and say "x equals y modulo R." We sometimes also write xRy as Rxy when stressing that R is a function. The *equivalence class* of an element of S under R is the set $\{x \mid xRa\}$ denoted $[a]$ or $[a]_R$. The equivalence classes of S under R are clearly disjoint or equal since if $x \in [a] \cap [b]$ for $a \neq b$, then $aRx \; \& \; bRx$, hence xRb and by transitivity aRb so $[a] = [b]$. The set of equivalence classes is a *partition* of S. In set theory this structure is denoted S/R. The map $x \to [x]$ from S to S/R is called the *canonical mapping of S onto S/R.* It is common to think of the classes $[a]$ as new elements with equality between them defined by R, i.e. $[a] = [b]$ iff aRb.

If $f \in S \to T$ then we say that f is *functional on S/R* (or *compatible with R*) iff aRa' implies that $f(a) = f(a') \; in \; T$. Likewise for an (binary) operation on S, $g \in S \times S \to S$; we say g is *functional wrt R* iff aRa' and bRb' implies $f(a, b) = f(a', b') \; in \; S$.

Quotient sets and structures are central to mathematics, but their representation in set theory is not suitable for computation because the elements of a quotient set are equivalence classes which are infinite objects. To remedy this "computational defect" of set theory, type theory uses the notion of a *quotient type*. Given a type T and an equivalence relation E on T, there is a type called the quotient of T by E, written $T//E$ (or $x, y.T//xEy$ in fully expanded form). The elements of $T//E$ are the same as those of T, but the equality relation on $T//E$ is E.

In order to qualify as a function $f \in T//E \rightarrow S$, f must be a function $f \in T \rightarrow S$ which is *functional wrt* E. The canonical map $T \rightarrow T//E$ is just the identity function, so the functionality theorem becomes $f \in T \rightarrow S$ is functional wrt R iff $f \in T//R \rightarrow S$.

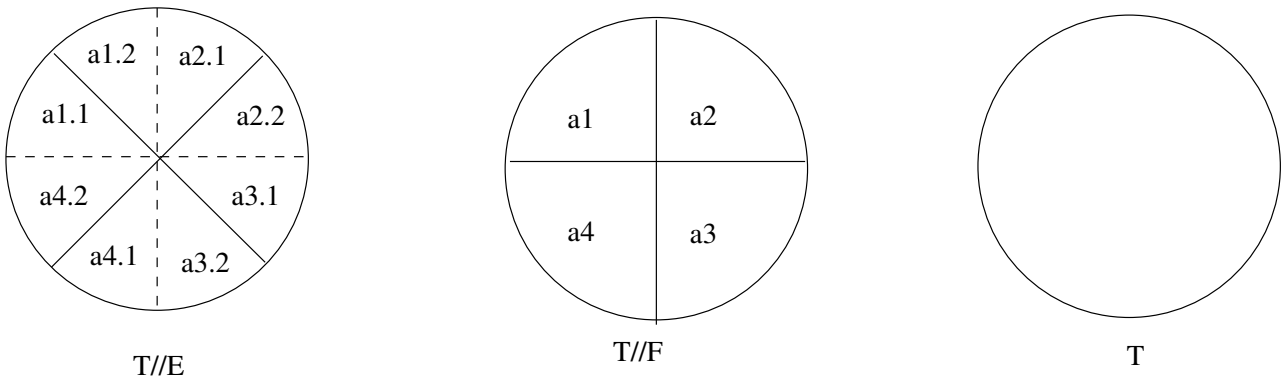
Here is another important fact about Nuprl's rules for the quotient type. The elements of T are elements of $T//E$ so T is a subtype of $T//E$. Also knowing xEy for x, y in T is sufficient to conclude $x = y$ in $T//E$, but not conversely. That is, if we know $x = y$ in $T//E$, we need not know constructively that xEy. (We can conclude this if E is decidable.)

To understand this feature of the quotient rules, we need to point out that according to Martin-Löf's semantics, the *computational content* of equality propositions, $x = y$ in A, is trivial. The theory only records that these propositions are proved, but ignores the details. Let us call this the "computational triviality of equality" principle. To preserve this semantic principle in the presence of quotient types requires that the rules "forget" the computational information in a proof of xEy when asserting $x = y$ in $T//E$.

4.4 Finite Index Equivalence Relations

An equivalence relation E on T is said to be of *finite index* iff $T//E$ is finite. E's *index* is the cardinality of $T//E$. A very important result we need is that if E is decidable and T is finite, then $T//E$ is finite, and its cardinaltiy is less or equal to that of T. Indeed if T is finite, say of size n, the index e of any finite index E satisfies $e \leq n$. See *quo_of_finite* in the relation library.

Given two equivalence relations E and F on T, we say that E *refines* F iff $xEy \Rightarrow xFy$. We write $E \sqsubseteq F$. This means that the equivalences classes of $T//F$ are possibly refined or decomposed into smaller classes. A suggestive picture is:



Although we will not discuss *subtyping* here, we note that in general $T_1 \sqsubseteq T_2$ iff $(x = y \text{ in } T_1)$ implies $(x = y \text{ in } T_2)$ (so $(x = x \text{ in } T_1) \Rightarrow (x = x \text{ in } T_2)$ which means T_1 is a "subtype" of T_2). We have for any equivalence relations, $E \sqsubseteq F$ implies $T//E \sqsubseteq T//F$ and $T \sqsubseteq T//E$. (We can think of T as $T//I$ where xIy iff $x = y$ in T; clearly $I \sqsubseteq E$ for *any* E on T, so $T//I \sqsubseteq T//E$.)

4.5 Equivalence Relations on Strings Induced by Finite Automata

Much of the theory of finite automata is concerned with a natural equivalence relation on strings induced by the automaton. Given DA in Automata($Alph$; $States$), we say that two strings x and y in $Alph\ list$ are equivalent $mod\ DA$, $x = y\ mod\ DA$, iff $DA(x) = DA(y)$ in $States$, that is, iff the strings are taken to the same state by the action of the automaton.

The remarkable fact is that a finite automaton is characterized by two properties: that the equivalence relation is of finite index and that it is invariant under the *extension* of the strings by the same characters. The last property is stated in terms of appending more characters, say a list z of them, to the *head of the input* (which means to the *end of the tape*).

Def: An equivalence relation E on $Alph\ list$ is called *extension invariant*[5]

$$iff\ for\ all\ x, y, z\ in\ Alph\ list\ xEy \Rightarrow (z@x)E(z@y).$$

Fact 1. The equivalence relation R induced by $DA \in$ Automata($Alph$; $States$) is of finite index and extension invariant.

It is easy to see that this is true. The largest number of equivalence class in $Alph\ list//R$ is the number of states of DA which is finite, and if $DA(x) = DA(y)$ then clearly

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

Fact 2. Any extension invariant equivalence relation R such that $Alph\ list//R$ is finite can be defined by a finite automaton.

We build the automaton by using the elements of $Alph\ list//R$ as states. Extension invariance allows us to define δ. These links between automata and finite index, extension invariant equivalence relations is independent of the final states. The link is defined in terms of *compute_list*.

When we add final state information, we can say more about the equivalence relation. Indeed another remarkable fact emerges, namely, if we designate those strings belonging to certain equivalence classes of R as "accepted", then we can find a minimal state automaton whose final states accept exactly the designated strings. Moreover, the automaton is essentially unique.

Fact 3. Given any language L, it induces an equivalence relation Rl_L defined by xRl_Ly iff for all z in $Alph\ list$, $z@x \in L \Leftrightarrow z@y \in L$. We call this the equivalence relation *induced by* L. If L is accepted by a finite automaton, then we can show that the equivalence relation induced by this automaton is a refinement of Rl_L. Moreover, we can build a finite automaton with $Alph\ list//Rl_L$ as states that will be the unique minimal automaton accepting L.

These remarkable facts are aggregated into the well-known Myhill-Nerode Theorem which we discuss and prove next. It is the centerpiece of Hopcroft and Ullman's section 3.2.

5 The Myhill-Nerode Theorem

The first section states and proves the HU version of the Myhill-Nerode theorem. We modified their account slightly to enable a constructive proof, namely, we require an *effective* union in statement 2

[5] Hopcroft and Ullman use $xEy \Rightarrow (x@z)E(y@z)$ and call these relations *right invariant* since the end of the tape is on the right. We could call them *left invariant* since the end of our tape is on the left, but we choose a geometrically neutral nomenclature.

and a *decidable* induced equivalence relation, Rl. These changes are highlighted by enclosing them in parentheses. We also use the terminology of the induced equivalence relation defined at the end of section 4 rather than defining that relation in the statement of the theorem as HU do.

After presenting the HU proof, we discuss its constructive formalization and then examine the details of the proofs of the three implications: (1) $\Rightarrow$ (2) called mn_12, (2) $\Rightarrow$ (3) called mn_23 and (3) $\Rightarrow$ (1) called mn_31. We include text from the on-line libraries.

5.1 Hopcroft and Ullman Version

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 (effective) 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 (and decidable).

Proof. (1). $\Rightarrow$ (2). Assume that L is accepted by $M = (K, Alph, \delta, q_0, F)$. Let R be the equivalence relation xRy 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 (2) is a refinement of Rl; that is, every equivalence class of R is entirely contained in some equivalence class of Rl. Thus the index of Rl cannot be greater than the index of R and so is finite. Assume that xRy. Then since R is extension invariant, for each z in $Alph\,list$, $z@xRz@y$, and thus $z@y$ is in L if and only if $z@x$ is in L. Thus $xRly$, and hence, the equivalence class of x in R is contained in the equivalence class of x in Rl. We conclude that each equivalence class of R is contained within some equivalence class of Rl.

(3) $\Rightarrow$ (1) Assume that $xRly$. 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 $w@xRlw@y$, and Rl is extension invariant. Now let K' be the finite set of equivalence classes of Rl and $[x]$ the element of K' containing x. Define $\delta([x], a) = [xa]$. The definition is consistent, since Rl is extension invariant. Let $q_0' = [\epsilon]$ 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 $T(M')$ if and only if $[x]$ is in F'. Note, F' is computable because we assume L is decidable.

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. 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 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 $\delta'(q'_0, x) = \delta'(q'_0, y) = q$.

5.2 Formalizing (1) ⇒ (2)

Formalizing the implication form (1) to (2) is quite direct and elegant in type theory. We go through it now step by step.

To say that a set $L \subseteq Alph\,list$ is accepted by some finite automaton means that there is an automaton, say $Auto$, accepting L. This, in turn, presupposes a set of states, say St such that $Auto \in \text{Automata}(Alph; St)$. So there is a mechanical translation of "accepted by some finite automaton" into

$$\exists St{:}\mathbb{U}.\exists Auto{:}\,Automata(Alph; St).Fin(St) \wedge L = L(Auto).$$

We are implicitly quantifying over L and $Alph$. This implicit translation is revealed in the first line of the proof, "let L be accepted by some $Auto = \langle K, Alph, \delta, q_0, F\rangle$."

Statement (2) is:

> L is the union of some of the equivalence classes of an extension invariant equivalence relation of finite index.

Translating this requires an equivalence relation, called R in the proof, so we call it R in the statement

$$\exists R{:}\{r{:}\,Alph\ list \to Alph\ list \to \mathbb{P} \mid EquivRel(Alph\ list; x, y.rxy)\}.$$

$EquivRel(Alph\ list\ x, y.Rxy)$ is defined as we would expect. It says that R is an equivalence relation over $Alph\,list$. It is a specialization of $EquivRel(T; x, y.Rxy)$.

We need to assert that R is of finite index which is just $Fin(x, y : Alph\ list//Rxy)$.[6] R must be extension invariant, i.e. $\forall x, y, z{:}\,Alph\,list.(Rxy \Rightarrow R(z@x)(z@y))$.

Next we consider how to express the idea of statement 2, that "L is the (effective) union of some of the equivalence classes of an extension invariant equivalence relation of finite index." The most direct translation of this would use some idea of union of equivalence classes, say $e_1, \ldots, e_m$ since there are finitely many. We could write $L = \bigcup_{i \in G} e_i$ where G is a subset of the indexes 1 to n. So, to say that the union is *effective* is to say that G is a decidable set.

We don't want to express the union idea this way (even though we could), because we are using the language of quotient types rather than that of equivalence classes. That is, we don't need to bring in the type of equivalence classes because we can use the type $Alph\,list//R$ instead.

We can transform $L = \bigcup_{i \in G} e_i$ into a statement about $Alph\ list//R$ as follows. Suppose that a function g picks out the classes in the union, so $g(e_i) = tt$ iff $i \in G$. Now notice that for $x \in Alph\,list$, the equivalence classes are $[x]_R$. So we have $x \in L$ iff $g([x]_R) = tt$. The map g must

[6] Recall that $x, y{:}\,A//Rxy$ is the fully expanded notation for $A//R$.

respect the equivalence relation R, so it can actually be defined as a function $g \in Alph\,list//R \to \mathbb{B}$. Each effective union determines such a map g and conversely.

To say that L is the effective union of some of the equivalence classes, we use a boolean valued function g to pick out which classes.

$$\exists g\!:\!(x, y\!:\!Alph\,list\,//Rx, y) \to \mathbb{B}.$$

$$\forall l\!:\!Alph\,list.\;\; L(l) \Longleftrightarrow \mathrm{True}(g(l)).$$

That is, l is in L iff $(g(l) = tt$ in $\mathbb{B})$. Note, $True(g(l))$ is also denoted $\uparrow(g(l))$.

Putting all this together we get the fully expanded formulation. It is named mn_12 for Myhill-Nerode $(1) \Rightarrow (2)$.

$*T\ mn\_12$
$\forall Alph\!:\!\mathbb{U}.\;\forall L\!:\!L(Alph).$
$\quad Fin(Alph)$
$\quad \Rightarrow (\exists St\!:\!\mathbb{U}.\;\exists Auto\!:\!DA\_(Alph; St).\; Fin(St) \wedge\; L = L(Auto))$
$\quad \Rightarrow (\exists R\!:\!\{r\!:\!Alph\,list\;\to Alph\,list \to \mathbb{P} \mid EquivRel(Alph\,list; x, y.rxy)\}$
$\qquad\quad \exists g\!:\!x, y\!:\!(Alph\,list)//(Rxy) \to \mathbb{B}$
$\qquad\quad\; Fin(x, y\!:\!(Alph\,list)//(Rxy))$
$\qquad\quad \wedge\; (\forall l\!:\!Alph\,list.\; L(l) \Leftrightarrow \uparrow(g(l)))$
$\qquad\quad \wedge\; (\forall x, y, z\!:\!Alph\,list.Rxy \Rightarrow\; R(z@x)(z@y)))$

The above version of the theorem is the one displayed on the Web, but we have worked to make both the theorem and the proof more readable. It is instructive to see how this can be done.

We first decided to suppress some of the detail in the statement that R is an equivalence relation by using a less detailed display form. The result is this display

$$\{r\!:\!Alph\,list \to Alph\,list \to \mathbb{P} \mid r \text{ is an Equivalence over } Alph\,list.\}.$$

Next we agreed to allow the assertion of a boolean without displaying the assert symbol, so $\mathrm{True}(g(l))$ becomes just $g(l)$. Next we display extension invariance as a simple phrase, "R is extension invariant." Finally we use a general abbreviation device of suppressing the leading universal quantifiers since this is a standard convention in mathematics, indeed used by Hopcroft and Ullman for this theorem. The result is:
$\vdash Fin(Alph) \Rightarrow$
$\quad (\exists St\!:\!\mathbb{U}, Auto\!:\!DA(Alph; St).\; Fin(St)\;\&\; L = L(Auto)) \Rightarrow$
$\quad \exists R\!:\!(\{r\!:\!(Alph\,list \to Alph\,list \to \mathbb{P} \mid r \text{ is an Equivalence over } Alph\,list\}),$
$\quad Alph\,list//R \to \mathbb{B}).$
$\quad Fin(Alph\,list//R) \wedge (\forall l\!:\!Alph\,list.\; L(l) \Leftrightarrow g(l))\;\&\; R$ is extension invariant.

The proof of this theorem is simple.

1. We define Rxy to mean $Auto(x) = Auto(y)$. It is immediate that R is extension invariant since $Auto(z@x) = \hat{\delta}(Auto(x), z) = \hat{\delta}(Auto(y), z) = Auto(z@y)$.

2. To show that R is of finite index is precisely to show $Fin(Alph\ list//R)$. We know that the number of states of $Auto$ is an upper bound to this cardinality. The exact size is in fact the number of *accessible states*. This fact comes out as we argue finiteness.

   Finiteness of $Alph\ list//R$ is proved by invoking the lemma in *Automata_3 inv_of_fin_is_fin*, $\forall T, S{:}\mathbb{U}.\ \forall f{:}T \to S.\ Fin(S)\ \wedge\ (\forall s{:}S.\ Dec(\exists t{:}T.\ f\,t = s)) \Rightarrow Fin(x, y{:}T//(f\,x = f\,y))$. We then prove the preconditions of the lemmas, mainly that

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

   The proof of $*$ requires showing that if there is a t such that $Auto(t) = s$, then there is a "short" t, namely of length less than n, the number of states. This is done by invoking the pumping lemma and its corollary from *Automata_1*.

   This in turn requires the pigeon hole lemma of *Automata_1, phole_lemma*. Finally, the proof of *inv_of_fin_is_fin* requires the key *Automata_3 lemma finite_decidable_subset*, $\forall T{:}\mathbb{U}.\ \forall B{:}T \to \mathbb{P}.\ Fin(T)\ \wedge\ (\forall t{:}T.\ Dec(B\,t)) \Rightarrow Fin(\{t{:}T \mid B\,t\})$.

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

The main steps of the on-line proof are displayed below using a presentation format that can be automatically generated from a mark-up of the original proof. The tools for creating these more readable proofs were created by Stuart Allen.

The key to this format is that parts of the proof are "put aside" to be read later, if at all. Allen calls these *side proofs*. They are indicated by the phrase SidePF followed by a name. In the on-line version it is possible to click on this proof to read it.

$\vdash\ Fin(Alph) \Rightarrow$
$(\exists\ St{:}\mathbb{U}, Auto{:}\text{Automata}(Alph;\ St).\ Fin(St)\ \&\ L = L(Auto)) \Rightarrow$
$\exists\ R{:}(\{r{:}(Alph\ list \to Alph\ list \to\ \mathbb{P} \mid r\ \text{is an Equivalence over}\ Alph\ list\}),$
$g{:}(Alph\ list//R\ \to \mathbb{B}).$
$Fin(Alph\ list//R) \wedge (\forall l{:}Alph\ list.\ L(l) \Leftrightarrow\ g(l))\ \&\ R$ is extension invariant

1. $Alph{:}\mathbb{U}$
2. $L{:}L(Alph)$
3. $Fin(Alph)$
4. $St{:}\mathbb{U}$
5. $Auto{:}DA(Alph; St)$
6. $Fin(St)$
7. $L = L(Auto)$

$\vdash\ \exists R{:}(\{r{:}(Alph\ list \to Alph\ list \to \mathbb{P} \mid r$ is an Equivalence over $Alph\ list\}),$
$g{:}(Alph\ list//R \to \mathbb{B}).$
$Fin(Alph\ list//R) \wedge (\forall l{:}Alph\ list.\ L(l) \Leftrightarrow g(l))\ \&\ R$ is extension invariant

8. $Auto(x) = Auto(y) \in St$ is an Equivalence in $x, y{:}Alph\ list$

$\vdash$ by SidePF $\to$ *mn_12_read_SidePf07*

$\vdots\ \ddots\quad \vdash Fin(x, y{:}Alph\ list//(Auto(x) = Auto(y) \in St))$

$\vdots\qquad\qquad (using\ THM{:}inv\_of\_fin\_is\_fin)$

$\vdots\qquad$ 6. $n{:}\mathbb{N}$

7. $\mathbb{N}n \sim St$
8. $L = L(Auto)$
9. $Auto(x) = Auto(y) \in St$ is an Equivalence in $x, y : Alph\ list$
10. $s : St$
11. $\#(St) = n$

$\vdash Dec(\exists t : Alph\ list.\ Auto(t) = s \in St)$

by SidePF $\rightarrow mn\_12\_read\_SidePf12$

$\vdash Dec(\exists k : \mathbb{N}(n+1), t : (\{l : Alph\ list \mid ||l|| = k \in \mathbb{N}\}).\ Auto(t) = s \in St)$

$(using\ THM : auto2\_lemma\_6\ by\ SidePF \rightarrow mn\_12\_read\_SidePf13)$

12. $t : \mathbb{N}(n+1)$
13. $t1 : \{l : Alph\ list \mid ||l|| = t \in \mathbb{N}\}$

$\vdash Dec(Auto(t1) = s \in St)\ \ (using\ THM : fin\_is\_decid)$

$\vdash Fin(St) \leftarrow 7$

$\vdash \forall l : Alph\ list.\ L(l) \Leftrightarrow (Auto\ accepts\ l) \leftarrow 7$

$\vdash (\lambda x, y.\ Auto(x) = Auto(y) \in St)$ is extension invariant
$(using\ THM : compute\_l\_inv)$

5.3 Formalizing (2) ⇒ (3)

We have seen how to formalize (1) and (2). To express (2) ⇒ (3) we need to formalize condition (3). First we define the relation Rl (this is how it appears in the libraries). This language is a function of a given language L, but that parameter is not always displayed although it is implicit. In Allen's display of the theorem, Rl is written as RlL.

$A\ lang\_rel\ \ Rl == \lambda x, y.\ \forall z : A\ list.\ L(z@x) \Leftrightarrow L(z@y).$
$T\ lang\_rel\_refl\ \ \forall A : \mathbb{U}.\ \forall L : L(A).\ \ Rl \in A\ list\ \rightarrow A\ list\ \rightarrow \mathbb{P}.$

We establish straight forwardly that Rl is an equivalence relation.

$T\ lang\_rel\_refl$
$\forall A : \mathbb{U}.\ \forall L : L(A).\ \ Refl(A\ list; x, y.x\ Rl\ y)$

$T\ lang\_rel\_symm$
$\forall A : \mathbb{U}.\ \forall L : L(A).\ \ Sym(A\ list; x, y.x\ Rl\ y)$

$lang\_rel\_tran$
$\forall A : \mathbb{U}.\ \forall L : L(A).\ \ Trans(A\ list; x, y.x\ Rl\ y)$

The formulation of (3) as in Hopcroft and Ullman would be that Rl is of finite index. But we will see that to prove (3) ⇒ (1) *constructively* we need to be explicit that L is a decidable language. So we take (3) to be: L is *decidable* and Rl is of finite index.

The proof of (3) from (2) appears to be the simplest of the implications. (From (2) we know immediately that L is decidable.) We show that if R refines Rl, then the index of Rl is no larger than that of R, that is,

$$\text{If}\ \ |\ Alph\ list//R\ |\ = k,\ \text{then}\ \ |\ Alph\ list//Rl\ |\ \leq k.$$

So the only nonroutine step is to show

$$* \;\; xRy \Rightarrow xRly.$$

This follows directly from the fact that R is extension invariant since $(z@x)R(z@y)$, but then

$$(z@x) \in L \;\text{ iff }\; (z@y) \in L \;\text{ (namely } g(z@x) = g(z@y)\text{), hence } (z@x)Rl(z@y).$$

This seems to be the whole story until we look at the details of the lemma

$$R \sqsubseteq Rl \Rightarrow index(Rl) \leq index(R).$$

It requires that we prove that the relation Rl is decidable (see *auto_2_lemma_8*). This complication suggests another more elegant proof which we outline after stating the theorem. This second proof is the one we formalize.

$T\ mn\_23$

$\forall n{:}\{1...\}.\ \forall A{:}\mathbb{U}.\ \forall L{:}L(A).\ \forall R{:}A\ list \to A\ list \to \mathbb{P}.$
$Fin(A)$
$\Rightarrow EquivRel(A\ list; x, y.xRy)$
$\Rightarrow 1-1{-}\text{Corresp}\ (\mathbb{N}n; x, y{:}(A\ list)//(xRy))$
$\Rightarrow (\forall x, y, z{:}A\ list.\ xRy \Rightarrow (z@x)R(z@y))$
$\Rightarrow (\exists g{:}x, y{:}(A\ list)//(xRy) \to \mathbb{B}.\ \forall l{:}A\ list.\ L(l) \Leftrightarrow\uparrow (g(l)))$
$\Rightarrow (\exists m{:}\mathbb{N}.\ 1-1{-}\text{Corresp}(\mathbb{N}m; x, y{:}(A\ list)//(x\ Rl\ y))) \wedge (\forall l{:}A\ list.\ Dec(L(l))$

We can use the same devices as before to render this theorem more readable. Here is Stuart Allen's version.

$\vdash Fin(A) \Rightarrow$
$(R$ is an Equivalence over $A\ list) \Rightarrow$
$\mathbb{N}n \sim A\ list//R \Rightarrow$
$(R$ is extension invariant $\Rightarrow$
$(\exists g{:}(A\ list//R \to \mathbb{B}).\forall l{:}A\ list.\ L(l) \Leftrightarrow g(l)) \Rightarrow$
$(\exists m{:}\mathbb{N}.\mathbb{N}m \sim A\ list//Rl)\ \&\ \forall l{:}A\ list.\ Dec(L(l))$

Proof

The key idea in this proof is to show that $Alph\ list//Rl\ =\ Alph\ list//Rg$ where Rg is like Rl but is defined on the quotient type using the boolean valued function g. This function g characterizes L in a simple way and is easier to work with than L itself. This leads us to work with an equivalence relation Rg instead of Rl. The proof is essentially establishing two isomorphisms,

$$Alph\ list//Rg \cong Alph\ list//R//Rg \cong \mathbb{N}m.$$

1. The first isomorphism follows from a lemma called *quo_of_quo*.
2. The second isomorphism follows from the lemma *quo_of_finite*. This is the heart of the proof. It requires that Rg is a decidable relation.

Qed

Here is the main line of the Web proof as it appears after applying Allen's technique to the full

Web proof. Recall that Rl will be displayed as RlL. Notice that there are two side proofs as well as a number of lemma references. These can be expanded in the on-line version just by clicking on the names.

1. $n : \{1...\}$
2. $A : \mathbb{U}$
3. $L : L(A)$
4. $L \in A\ list \to \mathbb{P}$
5. $R : A\ list \to A\ list \to \mathbb{P}$
6. $Fin(A)$
7. R is an Equivalence over $A\ list$
8. $\mathbb{N}n \sim A\ list//R$
9. R is extension invariant
10. $g : A\ list//R \to \mathbb{B}$
11. $\forall l : A\ list.\ L(l) \Leftrightarrow g(l)$

$\vdash (\exists m : \mathbb{N}.\mathbb{N}m \sim A\ list//RlL) \wedge \forall l : A\ list.\ Dec(L(l))$ by $D\ 0$

⋮ ⋱

⋮ ⋱ $\vdash \exists m : \mathbb{N}.\mathbb{N}m \sim A\ list//RlL$ by SidePF $\to mn\_23\_read\_SidePf01$

⋮ 12. $\forall x, y : A\ list//R.\ Dec(xRgy)$ $(using\ THM : mn\_23\_lem\_1\_EQUO2)$

⋮ 13. RlL is an Equivalence over $A\ list$ $(using\ THM : lang\_rel\_equi\_EQUI2)$

⋮ 14. Rg is an Equivalence over $A\ list//R$ $(using\ THM : lquo\_rel\_equi\_EQUO2)$

⋮ 15. $A\ list//RlL = A\ list//Rg \in \mathbb{U}$ $(using\ THM : mn\_23\_Rl\_equal\_Rg\_EQUO2)$

⋮ 16. $A\ list//Rg \sim A\ list//R//Rg$ $(using\ THM : quo\_of\_quo\_EQUO2)$

⋮ 17. $\exists m : \mathbb{N}(n+1).\mathbb{N}m \sim A\ list//R//Rg$ $(using\ THM : quotient\_of\_finite\_EQUO2)$

⋮ $\vdash$ by SidePF $\to mn\_23\_read\_SidePf02$

$\vdash \forall l : A\ list.\ Dec(L(l))$ by RWO"11"0....

The real work for us in proving this theorem was actually spent on building general facts about quotients and in defining Rg and showing that it is an equivalence relation on $Alph\ list//R$. This required a long sequence of lemmas. All of this is left *implicit* in Hopcroft and Ullman who need at least the properties of @ on quotient sets and facts about equivalence relations on quotient sets.

$A\ mn\_quo\_append$
$z@_q x == z@x$

$T\ mn\_quo\_append\_wf$
$\forall A : \mathbb{U}.\ \forall R : A\ list \to A\ list \to \mathbb{P}.$
$\mathrm{EquivRel}(A\ list; x, y.\ xRy)$
$\Rightarrow (\forall x, y, z : A\ list.\ xRy \Rightarrow (z@x)R(z@y))$
$\Rightarrow (\forall z : A\ list.\ \forall y : x, y : (A\ list)//(xRy).\ z@_q y \in x, y : (A\ list)//(xRy))$

$T\ mn\_quo\_append\_assoc$
$\forall Alph : \mathbb{U}.\forall R : Alph\ list \to Alph\ list \to \mathbb{P}.$
$EquivRel\ Alph\ list; x, y.\ xRy)$
$\Rightarrow (\forall x, y, z : Alph\ list.\ xRy \Rightarrow (z@x)R(z@y))$
$\Rightarrow (\forall z1, z2 : Alph\ list.\ \forall y : x, y : Alph\ list)//(xRy).\ z1@z2@_q y = z1@_q z2@_q y$

$A\ lquo\_rel$
$Rg == \lambda x, y.\ \forall z{:}A\ list.\ \uparrow(gz@_q x) \Leftrightarrow \uparrow(gz@_q y)$

$T\ lquo\_rel\_wf$
$\forall A{:}\mathbb{U}.\ \forall R{:}A\ list \to A\ list \to \mathbb{P}.$
$EquivRel(A\ list; x, y.\ xRy)$
$\Rightarrow (\forall x, y, z{:}A\ list.\ xRy \Rightarrow (z@x)R(z@y))$
$\Rightarrow (\forall g{:}x, y{:}(A\ list)//(xRy) \to \mathbb{B}$
$Rg{\in}x, y{:}(A\ list)//(xRy) \to x, y{:}(A\ list)//(xRy) \to \mathbb{P})$

$T\ lquo\_rel\_equi$
$\forall A{:}\mathbb{U}.\ \forall R{:}A\ list \to A\ list \to \mathbb{P}.$
$EquivRel(A\ list; x, y.\ xRy)$
$\Rightarrow (\forall x, y, z{:}A\ list.\ xRy \Rightarrow (z@x)R(z@y))$
$\Rightarrow (\forall g{:}x, y{:}(A\ list)//(xRy) \to \mathbb{B}.\ EquivRel(x, y{:}(A\ list)//(xRy); u, v.uRgv))$

TRl_iff_Rg
$\forall A{:}\mathbb{U}.\ \forall R{:}A\ list \to A\ list \to \mathbb{P}.$
$EquivRel(A\ list; x, y.\ xRy)$
$\Rightarrow (\forall x, y, z{:}A\ list.\ xRy \Rightarrow (z@x)R(z@y))$
$\Rightarrow (\forall g{:}x, y{:}(A\ list)//(xRy) \to \mathbb{B}.\ \forall L{:}L(A).$
$(\forall l{:}A\ list.\ L(l) \Leftrightarrow \uparrow(g(l))) \Rightarrow (\forall x, y{:}A\ list.\ xRly \Leftrightarrow xRgy))$

5.4 Formalizing (3) ⇒ (1)

Our goal is to build a finite automaton called M'. We follow Hopcroft and Ullman exactly, taking the set of states to be $Alph\ list//Rl$, defining $\delta([x], a) = [ax]$, taking [nil] as the start state and defining $F([x]) = tt$ exactly when $x \in L$. In the next section we refer to this automaton as $A(g)$.

To show that M' as defined is a finite automaton accepting L, we need to show that δ is well defined on the equivalence classes, i.e. if $[x] = [y]$ then $\delta([x], a) = \delta([y], a)$. Since $\delta([x], a) = [ax]$ and $\delta([y], a) = [ay]$, we need to know that $[ax] = [ay]$. This is true iff $ax \in L$ iff $ay \in L$. But this is an instance of the definition of $x = y$ iff $xRly$ since $xRly$ iff $\forall z{:}\ Alph\ list.\ z@x \in L \Leftrightarrow z@y \in L$. Here is the formal statement followed by a compressed proof. In the compressed proof we use ...assertion... to indicate that an assertion was cut into the proof; that assertion is the goal of the following line. Direct Computation is key to the proof, and we display its main step by writing $\lceil formula \rceil * \Rightarrow formula'$

$\vdash Fin(Alph) \Rightarrow$
$\quad (Fin(Alph\ list//RlL) \wedge \forall l{:}Alph\ list.\ Dec(L(l))) \Rightarrow$
$\quad \exists St{:}\mathbb{U}, Auto{:}DA(Alph; St).Fin(St) \wedge L = L(Auto)$
1. $Alph{:}\mathbb{U}$
2. $L{:}L(Alph)$
3. RlŁ is an Equivalence over $Alph\ list$ $(using\ THM{:}lang\_rel\_equi\_EQUI2)$
4. $Fin(Alph)$
5. $Fin(Alph\ list//RlL)$
6. $\forall l{:}Alph\ list.\ Dec(L(l))$
$\vdash \exists St{:}\mathbb{U}, Auto{:}DA(Alph; St).Fin(St) \wedge L = L(Auto)$
by SidePF $\to mn\_31\_read\_SidePf01$
7. g$: Alph\ list \to \mathbb{B}$

8. $\forall t : Alph\ list. L[t] \Leftrightarrow g[t]$
$\vdash \exists Auto : DA(Alph; Alph\ list//RlL).\ Fin(Alph\ list//RlL) \wedge L = L(Auto)$
.....assertion .....
$\vdash < (\lambda s, a.a.s), nil, g > \in DA(Alph;\ Alph\ list//RlL)$
by SidePF $\rightarrow mn\_31\_read\_SidePf02$
9. $< (\lambda s, a.a.s), nil, g > \in DA(Alph;\ Alph\ list//RlL)$
$\vdash L = L(< (\lambda s, a.a.s), nil, g >)$
10. $l : Alph\ list$
$\vdash L(l) \Leftrightarrow (< (\lambda s, a.a.s), nil, g >$ accepts $l)$
$\vdash g(l) \Leftrightarrow (< (\lambda s, a.a.s), nil, g >$ accepts $l)$
.....assertion .....
$\vdash g(l) =< (\lambda s, a.a.s), nil, g >$ accepts l
by DirComp $g(l) = \lceil < (\lambda s, a.a.s), nil, g >$ accepts $l \rceil * \Rightarrow g[< (\lambda s, a.a.s), nil, g > (l)]$
$\vdash l =< (\lambda s, a.a.s), nil, g > (l) \in Alph\ list$ *by ListInd* 10
$\vdash nil =< (\lambda s, a.a.s), nil, g > (nil) \in Alph\ list$
by DirComp $nil = (\lceil < (\lambda s, a.a.s), nil, g > (nil) \rceil * \Rightarrow nil) \in Alph\ list$ ....
11. $u : Alph$
12. $v : Alph\ list$
13. $v =< (\lambda s, a.a.s), nil, g > (v) \in Alph\ list$
$\vdash (u.v) =< (\lambda s, a.a.s), nil, g > (u.v) \in Alph\ list$
by DirComp

6 State Minimization

6.1 Textbook Proof

Recall Theorem 3.2 reproduced in section 4. We restate it here as:

> **Theorem 3.2** The automaton M' of Theorem 3.1 has the least number of states of any automaton accepting L, and any automaton accepting L with this minimum number of states is isomorphic to M'.

There are several notable points about this theorem and its proof that bear on their formalization. First, notice that the statement of the theorem refers to M' which is defined in the proof of Theorem 3.1. This is a very economical device, but it is more common to make such definitions explicit as we do with *lang_auto*. This defines an automaton $A(g)$ given a function g to define the final states. The definitions are

A lang_auto
$A(g) == \lambda s, a.(a :: s), [\,], g$

T lang_auto_wf 3
$\forall Alph : \mathbb{U}.\ \forall L :$ Language$(Alph).\ \forall g : x, y : (Alph\ list)//(x Rly) \rightarrow \mathbb{B}.$
$A(g) \in$ Automata$(Alph; x, y : (Alph\ list)//(x Rly))$

T lang_auto_compute 4
$\forall Alph : \mathbb{U}.\ \forall L :$ Language$(Alph).\ \forall g : x, y : (Alph\ list)//(x Rly) \rightarrow \mathbb{B}.\ \forall l : Alph\ List.$

$A(g)(l) = l$
Let us review the proof exactly as written in [11].

> **Proof.** In the proof of Theorem 3.1 we saw that any finite automaton $M = (K, Alph, \delta, q_0, F)$ accepting L defines an equivalence relation which is a refinement of R. 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 $\delta'(q_0', x) = \delta'(q_0', y) = q$.

Notice that the properties of M' are proved *in the context of this specific theorem.* There is no effort to abstract them as general principles. So, for example, the notion of isomorphism is only mentioned in the proof, but never defined. We make this explicit in *Automata_4* discussed below. In addition the key argument that any automaton M accepting L defines a refinement of Rl is a observation from the proof of Theorem 3.1 that is not stated as a separate fact. And the consequence that the number of states of M is greater than the number of M' is an important general fact that is not abstracted from the theorem. We state these as separate theorems *card_le* and *card_ge*.

$*A\ card\_le$
$|S| \leq |T| == \exists f : S \to T.\mathrm{Inj}(S; T; f)$
$A\ card\_ge$
$|S| \geq |T| == \exists f : S \to T.\mathrm{Surj}(S; T; f)$

A notable point about this Hopcroft and Ullman proof is that while it is based on a neat idea, it is flawed because key details are omitted. The correspondence between states is not shown to be an isomorphism. (Hopcroft and Ullman don't hint at the proof they have in mind.) Failing to prove this led them to insert a *derivative fact*, namely that the automaton M is connected. This is not necessary. Let us outline their argument again.

6.2 Filling in Gaps in Textbook Proof

the proof
We are given L and a specific machine they call M' which accepts it. We call this machine $A(g)$. They let M be any other machine accepting L; by Theorem 3.1 we know $M \sqsubseteq M'$ hence $|M| \geq |M'|$. If M is also minimal, then $|M| = |M'|$. Using this equality they define a map from M to M'; let's call it f. They show f is well defined and claim without proof that it is an isomorphism.

The definition of f is on the connected set of states, say $K = \{q : St \mid \exists x : Alph\ list\ \delta(q_0, x) = q\}$. Given such a q let x be any string such that $\delta(q_0, x) = q$, then define $f(q) = [x]$. This is well-defined because if we pick a different string taking us from q_0 to q, say y with $\delta(q_0, y) = q$, then $x = y \bmod M$ so $x = y \bmod Rl$ by Theorem 3.1. Thus $[x] = [y]$ in $Alph\ list//Rl$.

It is not hard to show that f is an isomorphism between K and the states of M'. This implies that $K = St$. *But Hopcroft and Ullman do not carry out this argument.* (Instead, they prove separately that $K = St$.) Let us see what the right argument is.

First we show that f is onto. Given $[x]$ a state of $A(g)$, notice that $\delta(q_0, x)$ is in K for any x and that $f(\delta(q_0, x)) = [x]$. This means that f is onto which means $|K| \geq |A(g)|$.

If f is not 1-1, then $|K| > |A(g)|$. But $|K| \leq |M| = |A(g)|$. Since $K \subseteq M$, then $|K| \leq |M|$, and since we are assuming $|M| = |M'|$, then $|K| \leq |M'|$. So $|K| = |A(g)|$. Thus it is contradictory to assert that f is not 1-1. (By classical logic this means that f is 1-1. Constructively, this is true as well since the property of being 1-1 in these types is decidable.)

Notice that these final steps are subtle in terms of constructive reasoning. They also use basic facts about finite sets that are habitually considered "immediately" or "obviously" true. But they are in fact not "obvious" to Nuprl until we prove them.

a lacuna

There is another gap in the proof that is glossed over even in the above more detailed account. That account assumes that we can compute with equivalence classes as if they were concrete objects. As sets they are "infinite objects," so we have adopted the approach of *quotient types* discussed in section 4.3.

In order to precisely define the isomorphism f discussed above, we need to assign an element of $Alph\ list//Rl$ to q in K. We said that $f(q) = [x]$ for some x such that $\delta(q_0, x) = q$. But how do we find this x? The definition of K assures that it exists, but the semantics of the set type does not allow us to use the witness in a proof of this fact.

We could use a much stronger definition of the connected set of states, requiring the string x be kept with the state. That is, we could take $\hat{K} = q : St \times \{y : Alph\ list \mid \delta(q_0, y) = q\}$. Then the function f has access to x; it is the witness in the second component of the pair.

An approach that is more similar to the Hopcroft and Ullman proof is to notice that we can actually compute the string x given $q \in K$. We could for example pick the *least* string x with respect to the lexicographical ordering of $Alph\ list$. Suppose $x \prec y$ is this ordering. It is a well-ordering, and there is a least x such that $\delta(q_0, x) = q$. So we can define $f(q) = [\mu x.\delta(q_0, x) = q]$ where μx computes the least x. We will not actually formalize either of these approaches. It turns out that once we define the lexicographical ordering, then there is a more direct argument than the one in Hopcroft and Ullman. None of the facts about lexicographical ordering is mentioned in Hopcroft and Ullman.

We can avoid entirely the argument by contradiction (to f being 1-1) whose computational version is complex. We briefly discuss our approach next. It is presented in $Automata\_7$ on the Web.

6.3 Minimization Theorem

If we want to compute on an automaton like $A(g)$, then we probably want to use a more convenient representation where the states are natural numbers. We can define this directly in terms of the finiteness theorem for $A(g)$

Suppose $A(g)$ has k states, then there are maps

$$rep : Alph\ list//Rl \rightarrow [l...k]$$
$$unrep : [l...k] \rightarrow Alph\ list//Rl.$$

We could define the canonical minimal automaton, $M(g)$, in Automata($Alph$; $[l...k]$) by

$$\begin{aligned}\delta_M(i,x) &= rep(\delta_{A(g)}(unrep(i),x))\\ I(M) &= rep([nil])\\ F_M(i) &= F_{A(g)}(unrep(i)).\end{aligned}$$

It is now straight forward to build an isomorphism between $A(g)$ and $M(g)$ and between $M(g)$ and M of the theorem. (In this case the onto property is proved as $\forall i{:}[l...k].\ \exists q{:}K.(f(q)=i).$)

We can summarize the minimization work in the following way. First, we take the disjoint union of Automata($Alph$; St) over all finite types for states. In type theory this is

$$\text{Automata}(Alph) == St{:}\mathbb{U} \times Fin(St) \times \text{Automata}(Alph; St)).$$

The minimization result can be stated as: For every automaton A over $Alph$, there is an equivalent one with the minimum number of states. To formalize this, we write A *is_equivalent_to* M to mean that they accept the same languages, i.e. $\forall x : Alph\ list.\ L(A)(x) \Leftrightarrow L(M)(x)$. Then we say M is minimal iff for any other A equivalent to M, A has at least as many states.

$$\begin{aligned}A\ is\_equivalent\_to\ M\ &== \forall x{:}Alph\ list.\ (L(A)(x) \Leftrightarrow L(M)(x))\\ Minimal(M)\ &== \forall A{:}\text{Automata}(Alph).\ (A \equiv M \Rightarrow\ |\ States(A)| \geq |\ States(M)|).\end{aligned}$$

Now we can easily prove

Minimization Theorem:
$\forall Alph :\ \mathbb{U}.\ \ Fin(Alph) \Rightarrow$
$\forall A : \text{Automata}(Alph).\ \exists M : \text{Automata}(Alph).\ A\ is\_equivalent\_to\ M\ \&\ \text{Minimal}(M).$

From this theorem we can extract a function $Reduce\ \epsilon\ \text{Automata}(Alph) \rightarrow \text{Automata}(Alph)$ which produces the minimal machine.

In Automata_5 we show also that the minimal automaton is connected, where connected is defined in Automata_4 as $Con(A) == \forall s{:}St.\exists l{:}Alph\ list.A(l) = s$.

Define $MinAuto(Auto) == A(\lambda l.\ Auto(l)\downarrow)$ where $A(g)$ was defined before as $\langle \lambda s, a.\ (a.s), [], g\rangle$.

Theorem (min_auto_con)**:**
$\forall Alph,\ St{:}\mathbb{U}.\ \forall Auto : Automata(Alph,\ St).\ Fin(Alph)\ \&\ Fin(St) \Rightarrow Con(MinAuto(Auto))$.

We also show that the minimal automaton is unique up to isomorphism among all connected automata. Isomorphism is defined in Automata_4 as:
$A_1 \equiv A_2 == \exists f : S_1 \rightarrow S_2.\ By(S_1; S_2; f)\ \&$
$(\forall s{:}S_1.\forall a{:}Alph.\ f(\delta A_1\ is\ a) = \delta A_2(fs)a)\ \&$
$(f(I(A_1)) = I(A_2))\ \&$
$\forall S{:}s_1.\ F(A_1)s = F(A_2)(fs)$.

Theorem ($any\_iso\_min\_auto$)**:**
$\forall Alph, St{:}\mathbb{U}.\ \forall Auto{:}Automata(Alph; St).\ \forall S{:}\mathbb{U}.\ \forall A{:}Automata(Alph; S).$
$Fin(Alph) \Rightarrow Fin(S) \Rightarrow Con(A) \Rightarrow 1-1-Correspondence(S; x, y{:}Alph\ list//xRly)$
$\Rightarrow L(Auto) = L(A) \Rightarrow A \equiv MinAuto(A)$.

6.4 Computational Behavior

The Nuprl system is designed to extract and execute the computational content of constructive theorems even when it is only implicitly mentioned. So it is possible to actually perform state minimization on an automaton without the need to write a separate minimization algorithm. Instead, just as HU mention, we can extract the algorithm from the proof of the Myhill/Nerode theorem.

To illustrate this point concretely, note that given an automaton, *Auto*, Theorem 3.1 tells us that *Alph list//Rl* will be the set of states of the minimal machine and that this set is finite. Indeed, we should be able to compute the size, $|$ *Alph list//Rl* $|$. We have carried out this computation for some automata in *Automata_6*.

The algorithms implicit in our proofs of Theorem 3.1 are quite inefficient because we did not attend to the efficiency issues in the proofs. We saw this as an experiment in using quotient types to capture the HU results as exactly as we could. It is possible to redo some of the basic lemmas to achieve acceptable performance for a minimization procedure.

7 Conclusion

It is our view that the Nuprl formalization adds value to the Myhill/Nerode theorem and the state minimization corollary. With more work the proofs can be rendered as clearly as the informal ones—this is a challenge for us. We believe it is possible to formalize Hopcroft and Ullman's chapters 1-11 at the level exhibited here for Myhill/Nerode and to do it with our four-person team in less than 18 months. The collaboration methods we have learned would extend to larger teams. We examine these points in more detail.

1. Added value of Formalization

   Of course the formal account is more complete and each step has been checked by a machine, hence it is more *reliable*. There are no holes as in the proof Theorem 3.2. We are considerably more certain of the inferences because each one that we display has been checked by a *person and a machine*.

   Since the account is complete, we can study it *quantitatively*. We can talk about shorter proofs or more abstract ones. We can answer questions of dependence, e.g. does the theory use the axiom of choice for example?

   The formal account is *interactive*. We can change a formula or justification and replay the proof. We can mechanically reuse the theorems, tactics, and even *individual inferences*.

   Since the theory is itself a *formal mathematical object*, we can manipulate it. We could develop it first classically and then *transform* it automatically to the explicitly constructive version seen here. We could uniformly translate it to a constructive set theory (like IZF).

   All of the algorithms of the theory can be evaluated, e.g. we can experiment with state minimization, we can "run" the automata. Since the algorithms are terms of the theory, we can transform them to internal models of other languages or transfer them to algorithms in other programming languages as Nuprl automatically does (into Lisp, SML or Caml Light).

2. Proofs can be variously presented.

A "raw" Nuprl proof is not guaranteed to be easy to grasp—some are, some aren't. But the proof editor allows us to create *readings* or *presentations* of the proof by a variety of devices. We can *suppress* boring inferences; we can collect inferences into a single node, suppress them and replace the node with a comment that expands to the full node. We can create a term that displays the inferences in a more compact form using what Stuart Allen calls *side proofs*.

We need to systematically explore this aspect of Nuprl. Many ideas are being pursued. The Mizar work is encouraging that good displays are possible with simple general devices.

3. Going further in Hopcroft and Ullman

   We have already formalized other parts of Hopcroft and Ullman, for example an account of grammars from Chapter 2 and nondeterministic automata. We think we could finish Chapter 3 in another month of work by this team. This involves the challenging theorem that two-way automata are equivalent to one-way.

   A more appealing way to accomplish this formalization is with the help of other users world-wide who collaborate with us on definitions and connections. We judge that Chapters 6, 7, 8, and Chapter 9 would be especially easy to formalize, but Chapter 4 would be challenging. Chapter 12 on deterministic pushdown automata is not a sufficiently elegant chapter to formalize, Chapter 13 on stack automata is not of general interest, and the undecidability results of Chapter 14 could be obtained better in other settings.

4. Collaboration methods and related work

   Our method of working on Myhill/Nerode was to discuss the formalization at group meetings weighing the advantages and disadvantages of every definition and theorem statement. Once the definitions and theorems were fixed and the proof methods discussed, we would continue on independently with proof details.

   There were unexpected difficulties with quotients because the tactics were not strong enough to make the reasoning natural, but for the automata theory, the proofs could proceed as we have shown.

   This method of work would allow the theorem proving task to be distributed further. The key point is to agree on definitions and statements of the main theorems, e.g. those in the text being formalized. These constitute *specifications* for the task of *programming a proof* using tactics.

   We believe that with two other teams of provers like ours, Chapters 1 -11 of Hopcroft and Ullman could be formalized in six months. One way to expedite this endeavor would be for experienced users to formulate all of the definitions and theorems and invite a wider community to help with the proofs. We welcome and invite participation in this task.

It would be especially interesting to collaborate with other theorem proving systems as Howe and his colleagues are doing with HOL and Nuprl. Much of a classical treatment of languages can easily be re-interpreted constructively (see [12]). It would be especially fruitful to collaborate with other constructive provers such as Alf, Coq and Lego or with Isabelle which has formalized Martin-Löf type theory. Although these provers are based on different formalizations of constructive mathematics, they all share the critical properties that the computational notion essential to Hopcroft and Ullman can be expressed, and they all allow extraction of code form proofs.

Grant Support/Acknowledgments
We acknowledge the support granted by the National Science Foundation. We also thank Stuart

Allen and Karl Crary for the discussions and input concerning this topic and Karla Consroe for help in preparing the document.

References

[1] Stuart F. Allen. *A non-type-theoretic semantics for type-theoretic language.* PhD thesis, Cornell University, 1987.

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

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

[4] 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.

[5] 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.

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

[7] N. G. deBruijn. Set theory with type restrictions. In V.T. Sos A. Jahnal, R. Rado, editor, *Infinite and Finite Sets*, pages 205–314. *vol. I, Coll. Math. Soc. J. Bolyai 10*, 1975.

[8] Michael Gordon and T. Melham. *Introduction to HOL.* University Press, Cambridge, 1993.

[9] 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.

[10] Jason J. Hickey. Objects and theories as very dependent types. In *Proceedings of FOOL 3*, July 1996.

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

[12] Douglas J. Howe. Importing mathematics from HOL into Nuprl. In *Proceedings of The 1996 International Conference on Theorem Proving in Higher Order Logics*, 1996. To appear.

[13] Douglas J. Howe. Semantic foundations for embedding HOL in Nuprl. In *Proceedings of AMAST'96*, 1996. To appear.

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

[15] Miroslava Kaloper and Piotr Rudnicki. Minimization of finite state machines. Mizar User's Association, 1996.

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

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

[18] 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.

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

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

[21] L. C. Paulson. *Isabelle: A Generic Theorem Prover, Lecture Notes in Computer Science, Vol. 78.* Springer-Verlag, 1994.

[22] 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.