

Formalizing Automata II: Decidable Properties*

Robert L. Constable
Cornell University

Abstract

Is it possible to create formal proofs of interesting mathematical theorems which are mechanically checked in every detail and yet are readable and even faithful to the best expositions of those results in the literature? This paper answers that question positively for theorems about decidable properties of finite automata. The exposition is from Hopcroft and Ullman's classic 1969 textbook *Formal Languages and Their Relation to Automata*. This paper describes a successful formalization which is faithful to that book.

The requirement of being faithful to the book has unexpected consequences, namely that the underlying formal theory must include primitive notions of computability. This requirement makes a constructive formalization especially suitable. It also opens the possibility of using the formal proofs to decide properties of automata. The paper shows how to do this.

1 Introduction

1.1 context

This is a self contained second paper in a series about the formalization of automata theory. As in the first paper [25], the object of formalization is Hopcroft and Ullman's 1969 classic book *Formal Languages and Their Relation to Automata* (called Hopcroft and Ullman) [40]. In the first paper we formally proved the Myhill/Nerode Theorem and its corollary that there is a unique (up to isomorphism) minimal finite automaton accepting any regular language. A constructive proof was created using the Nuprl proof development system [22, 24], and from it we obtained an algorithm for correctly minimizing automata.¹

In this paper we are concerned with the decidability results in Chapter 3. For instance, Theorem 3.11 says “there is an algorithm to determine if a finite automaton accepts zero, a finite number, or an infinite number of sentences.” Theorem 3.12 says “there is an algorithm to determine if two finite automata are equivalent (i.e., if they accept the same language).” Throughout Chapter 3 there is mention of the fact that a finite automaton operates “effectively;” that is, we can effectively tell whether or not it accepts any string x .

*Supported in part by NSF grants CCR-9423687, DUE-9555162.

¹The Nuprl (“new pearl”) proof development system automatically *extracts* an efficient algorithm for minimization from the proof.

1.2 foundational challenge

This work uses automata theory to explore the issues that arise in expressing the intuitions underlying *computational discrete mathematics*. The work is part of a research program to explore the foundations of *computational mathematics*. An attempt to formalize the undecidability results of Hopcroft and Ullman’s Chapter 3 in classical set theory (say as in Mizar [46]) or in classical type theory (say HOL [34, 43, 42] or PVS [60]) would encounter severe difficulties. *It could not remain close to the style of the book*. It could not capture the computational intuitions that motivate and permeate the whole account. But in *constructive set theory* the Hopcroft and Ullman account can be followed essentially *verbatim*.²

The major difficulty in formalizing Chapter 3 in classical set theory lies in defining the notion of an algorithm and the concept of decidability. We could formalize the concept of an algorithm in ZF set theory, say using Turing machines. But the formalization of finite automata in Hopcroft and Ullman is a step on the way to defining (in Chapter 6) Turing machines! *So trying to follow their account in set theory, these decidability results would be inexpressible*. This situation might lead one to believe that Hopcroft and Ullman’s account is just “classically incoherent” and perhaps altogether incoherent. But that is not so since the ideas can be faithfully expressed in constructive set theory. This is a situation that is representative for all aspects of computational mathematics we have examined: namely, key results cannot be naturally expressed in classical set theory but can be in effective set theory.

Ultimately the problem of expressing computability in classical set theory is deeper than the issue of tracking Hopcroft and Ullman. For example, even if we agreed to define Turing machines and accepted Church’s thesis in order to talk about computability and decidability, there is first the *practical question* of whether this Turing machinery, or any equivalent machinery, is a tolerable way to faithfully carry out computational mathematics. The preponderance of evidence from articles and texts that present such mathematics is a resounding “no!”³ Hopcroft and Ullman’s book is representative.

Hopcroft and Ullman did not intend that their book be interpreted in *ordinary classical set theory* because they discuss the use of *effective procedures* in the introduction. This is not a basic notion of classical set theory, and attempts to define it using Turing machines presuppose the very subject they are introducing. So they seem to have in mind a foundation that includes Set Theory + Effective Procedures. Alas, there was no careful account of such a theory for them to rely on in 1969 as there are now: Alf, Coq [4, 28, 51], IZF [6, 32, 33, 54, 67], Nuprl [30, 3, 45, 43]. Hopcroft and Ullman also *intended* that there be a way to provide data to algorithms explicitly.⁴

²This theory is also called *type theory* to stress its difference from *classical set theory* and its similarity to the notion of type in programming languages. Martin-Löf also refers to it as a *set theory* [59], a term that might be natural when equality between types is extensional. However, the fact that set membership, $a \in A$, is not expressed by a predicate but by a judgment makes it less appealing to call this “set theory” instead of type theory.

³Among some mathematicians there is a *philosophical* question of whether classical set theory can adequately model effective computability. Intuitionists, for example, maintain that effectiveness is a primitive concept that is not reducible to the primitives of set theory since set theory lacks both the concept of *explicit* data and *reducibility*. Although we will not deal with this philosophical question directly, some of what we say about formalizing Hopcroft and Ullman will bear on it.

⁴In person, Hopcroft has confirmed this. Those of us at Cornell who have taught from the book (Constable, Hartmanis, Hopcroft, and Kozen) all found some way to introduce explicit data.

1.3 role of Nuprl

It turns out that the Nuprl logical theory, call it *formal effective set theory*, can express computability in a way that allows formalizing Hopcroft and Ullman *nearly verbatim*. Moreover, we directly formalize the intuitive ideas motivating the work. Making that point is a major theme of this article. Another important theme is that our presentation is also *completely formal*: the proofs were all done in the Nuprl proof development system by extending results from [25] with help of Pavel Naumov, who posted his work on the Nuprl web page.⁵

1.4 outline

Section 2 reviews the basic concepts of automata theory and casts them in a rigorous language. Section 3 reviews the decidability results from Hopcroft and Ullman and raises the foundational questions. Section 4 works out the consequences of needing explicit data in order to compute, and Section 5 works out the logical concepts needed to express decidability theorems. Section 6 looks at the details of the decidability proofs and examines a complete formal proof for the algebraic generalization of the pumping lemma used in the Nuprl decidability proofs. The conclusion mentions related work and future plans.

2 Set Theory Preliminaries

2.1 symbols and sentences

Hopcroft and Ullman write in a *naive set theory* (see Halmos [36]) with effective procedures. Let us call it *naive effective set theory*. Their first definition says that an *alphabet* is any finite set of symbols, Σ .⁶ They say that these finite sets come from some *countable set*, say *Symbols*. They say that a set is countable if and only if it can be put in a one-to-one correspondence with the integers. Let us note that the natural numbers $\mathbb{N} = \{0, 1, 2, \dots\}$ are a better choice than integers \mathbb{Z} , and so we assume

$$j: \mathbb{N} \xrightarrow[\text{onto}]{1-1} \text{Symbols} \text{ and } \Sigma \subseteq \text{Symbols}.$$

It is not clear whether *Symbols* is to be taken as a primitive set. Typically in set theory we deal with pure sets or possibly with \mathbb{N} as a primitive set whose elements are not sets. Since *Symbols* is in one-to-one correspondence with \mathbb{N} , we can simplify matters by taking it to be \mathbb{N} .⁷

The notion of a string is taken to be an element of Σ *list*. We denote this as Σ^* . In general if A is a type, then A^* denotes A *list*. In Hopcroft and Ullman the notion of Σ^* is more abstract but this abstractiveness is not used, and in fact defining Σ^* as Σ *list* adds clarity.

The concept of a *list type*, A^* , is basic to computer science. In classical set theory we need to provide an encoding of *lists* into *sets*. Typically, this is done by an inductive definition. For a modern account see Moschovakis [56]. We define it as a *recursive set*:

$$A^* = 1 \cup A \times A^*$$

⁵The URL is www.cs.cornell.edu/Info/Projects/NuPrl/html/Interactive_Formal_Courseware.html.

⁶They actually make the claim that a *noncountably infinite* number of symbols exists (on page 1), but say they will draw all their symbols from a countable subset of them. For us the claim amounts to the assertion that there is a *discrete uncountable set*. It is interesting to prove *constructively* that such sets exist.

⁷In Nuprl we could also take *Symbols* to be *Atom*.

where $\mathbf{1}$ is a canonical set with one element and \cup is the *union* symbol.

This definition depends on the concept of a *Cartesian product* of two sets, $S \times T$; this is the set of ordered pairs. In classical set theory, the Cartesian product is also used to define the *graphs* (of functions); namely, using the *power set* of all subsets of $A \times B$, denoted $\mathcal{P}(A \times B)$,⁸

$$Fun(A, B) = \{f : \mathcal{P}(A \times B) \mid \forall x : A. \exists! y : B. \langle x, y \rangle \in f\}.$$

But in Hopcroft and Ullman we directly define the effective procedures from A to B . We denote these as $A \rightarrow B$.

2.2 automata

According to Hopcroft and Ullman, the definition of a deterministic finite automaton is a “quintuple” $(K, \Sigma, \delta, q_0, F)$ where K is a finite nonempty set of states; Σ is a finite set (the alphabet); δ is a function from $K \times \Sigma$ to K called the *transition function*; q_0 is an element of K called the *start state*; and F is a subset of K , the *final states*. This quintuple is clearly a set; one way to express the “typing” of automata as follows. Let $Finite$ be the class of all finite sets, and let $Finite_+ = \{x : Finite \mid x \neq \emptyset\}$, let $\mathcal{P}_f(Symbols)$ be the set of all finite subsets of $Symbols$.

A finite automaton is an element of the class

$$\{\langle K, \Sigma, \delta, q_0, F \rangle \mid K \in Finite_+ \ \& \ \Sigma \in \mathcal{P}_f(Symbols) \ \& \ \delta \in (K \times \Sigma \rightarrow K) \ \& \ q_0 \in K \ \& \ F \subseteq K\}$$

The “official” definition we adopt for automata is parametrized by Σ and K . This allows us to easily state properties of these sets and define an automaton as an element of a Cartesian product of three sets. We use a “curried typing” for δ , treat F as a propositional function, and we will impose finiteness conditions on Σ and K in the theorems (see section 6).

Definition 1 *Given types Σ and K , let*

$$Automata(\Sigma, K) = (K \rightarrow \Sigma \rightarrow K) \times K \times (K \rightarrow \mathbb{B}).$$

explicit form

When we imagine quintuples, $\langle K, \Sigma, \delta, q_0, F \rangle$, we think of the elements given *explicitly*. Suppose as in Hopcroft and Ullman functions are represented as graphs, $Fun(K \times \Sigma, K)$, then an explicit automaton is

$$\begin{aligned} \{ \{0, 1, 2\}, \{0, 1\}, \{ \langle \langle 0, 0 \rangle, 1 \rangle \ \langle \langle 0, 1 \rangle, 0 \rangle, 0, \{2\} \} \\ \langle \langle 1, 0 \rangle, 2 \rangle \ \langle \langle 1, 1 \rangle, 0 \rangle \\ \langle \langle 2, 0 \rangle, 0 \rangle \ \langle \langle 2, 1 \rangle, 0 \rangle \} \end{aligned}$$

But there is nothing in the axiomatization of *classical* set theory that distinguishes explicit data from inexplicit. We could take the size of the alphabet, Σ , to be a number n_0 defined as the largest

⁸In general the *power set* of a set T , written $\mathcal{P}(T)$, is the set of all subsets of T . This is one of the most basic concepts of classical set theory, and it gives it an impredicative character since in defining a subset S of T we can quantify over *any set in $\mathcal{P}(T)$* , including S in $\mathcal{P}(T)$ which we are defining (see [8]).

perfect number if there is one, otherwise 17. Knowing n_0 depends on solving the open problem of determining the largest perfect number, if one exists. The states, K , could be the two real numbers $e \cdot \pi$ and $e + \pi$. One of these must be transcendental (if not, then the coefficients of $x^2 - x \cdot (e + \pi) + e \cdot \pi$ are algebraic, so its roots must be as well; but they are the roots of $((x - e) \cdot (x - \pi))$). We take the least transcendental as the start state—again, this is unknown. The transition function could be any finite function from $K \times \Sigma$ to K . We might define $f(x, i)$ to be $e + \pi$ if i is odd and Goldback’s conjecture is true; otherwise, $e \cdot \pi$. We could take $e + \pi$ to be the final state.

Classically, one might argue that the above description determines a well-defined finite automaton, but we don’t know much about it explicitly. For instance, we could say that any finite function must be effective, so this transition function is. It does not matter that we don’t know which function it is. But the problem lies deeper than this. We don’t have a way to say that a number is explicitly given, say 17 *vs* the n_0 above; and we don’t have a way of saying that a finite function is computable. This lack of control over the form of an automaton means that, in a classical set theory formalization, we cannot effectively solve any problems about automata as defined; we must try to encode the automata first. There is actually no explicit data type in which to encode them in classical set theory. *But Hopcroft and Ullman do not even try that!*

One might reply that we can say these things in the *metalanguage*, by defining decimal numerals and algorithmic notations. But if the metalanguage is itself classical set theory, then explicitness cannot be achieved there either. At some point we require a language with links to computation. To formalize the Hopcroft and Ullman account faithfully requires that these concepts are directly expressible in the theory (rather than in the metatheory). This is what Nuprl and other effective set theories allow.

2.3 semantics of automata

A finite automaton da can be interpreted as a language recognizer; that is, it defines a function from Σ^* to \mathbb{B} . The language accepted consists of those sentences on which da computes true (\mathbf{tt}).

This meaning of an automaton is given by providing a *meaning function* which maps an automaton da in $Automata(\Sigma; K)$ to a formal language, i.e., to a map from Σ *list* into propositions \mathbb{P} .⁹

Hopcroft and Ullman give the meaning by extending δ to strings. They define $\hat{\delta}$ as:

$$\begin{aligned}\hat{\delta}(q, \varepsilon) &= q \\ \hat{\delta}(q, xa) &= \delta(\hat{\delta}(q, x), a)\end{aligned}$$

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

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

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

$$\begin{aligned}\hat{\delta}(\varepsilon) &= q_0 \\ \hat{\delta}(xa) &= \delta(\hat{\delta}(x), a),\end{aligned}$$

⁹We discuss propositions in section 5.

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 appending x to a , $x@a$, or as constructing a list from x and a (called “consing” a to x). In the second 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 the resulting state of automaton da on string x (thought of as $hd(x).tl(x)$) is:

$$da(nil) = I(da)$$

$$da(x) = \delta_{da}(da(tl(x)), hd(x)),$$

where $I(da)$ is the initial state of da , and δ_{da} is the transition function.

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.¹⁰

In the Nuprl libraries this definition is broken up into three simpler functions (listed below), and we “curry” the arguments to δ , i.e., instead of $\delta_{da}(q, x)$ we write $\delta_{da}(q)(x)$ or just $\delta_{da}q x$.

1. Define a function from an automaton and an input string to a state. This is called

M compute_list_ml

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

2. Associate with the resulting state of *compute_list_ml* a Boolean value using the final state component, a function $F : States \rightarrow \mathbb{B}$.
3. Associate with the automaton the propositional function saying that the final state is \mathbf{tt} .

A auto_lang

$L(da) == \lambda l.(F(da(l)) = \mathbf{tt})$.

3 Solvable Problems Concerning Finite Automata

In this section we examine what Hopcroft and Ullman say about the existence of algorithms to answer questions about finite automata. Then in section 6 we formalize these results.

3.1 results from Hopcroft and Ullman

The most basic solvability result in Hopcroft and Ullman is that given an automaton M and a string x in Σ^* we can decide whether M accepts x ; that is, whether $x \in L(M)$.¹¹ Put another way, we can *compute* the function $\lambda(x. F(\hat{\delta}(q_0)(x)))$ from Σ^* to \mathbb{B} . Let us call this function $fun(M)$. It belongs to $\Sigma^* \rightarrow B$. Let us call this the basic result. It is used in all the other results.

Basic Fact: $fun(M) \in \Sigma^* \rightarrow B$, is effectively computable.

¹⁰There 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.

¹¹Hopcroft and Ullman use M for a finite automaton and $T(M)$ for the set of all accepted strings. When we quote them directly we use their notations.

This result is not stated explicitly but is used to prove two theorems.

Theorem 3.11: *The set of sentences accepted by a finite automaton with n states is: (1) nonempty iff the automaton accepts a sentence of length less than n ; (2) omitted here. Thus, there is an algorithm to determine if a finite automaton accepts zero...number of sentences.*

Here is their proof of part 1.

Proof. The “if” portion is obvious. Suppose that a finite automaton $M = (K, \Sigma, \delta, q_0, F)$, with n states accepts some word. Let w be a word as short as any other word accepted. We might as well assume that $|w| \geq n$, else the result is proven for M . Since there are but n states, M must pass through the same state twice accepting w . Formally, we can find q in K such that we can write $w = w_1w_2w_3$, with $w_2 \neq \epsilon$, $\hat{\delta}(q_0, w_1) = q$, $\hat{\delta}(q, w_2) = q$, and $\hat{\delta}(q, w_3) \in F$. Then w_1w_3 is in $T(M)$, since

$$\hat{\delta}(q_0, w_1w_3) = \hat{\delta}(q_0, w_1w_2w_3).$$

But $|w_1w_3| < |w|$, contradicting the assumption that w is as short as any word in $T(M)$.

In part (1), the algorithm to decide if $T(M)$ is empty is: “See if any word of length up to n is in $T(M)$.” Clearly, there is such a procedure which is guaranteed to halt.

Qed

Theorem 3.12: *There is an algorithm to determine if two finite automata are equivalent (i.e., if they accept the same language).*

Proof. Let M_1 and M_2 be finite automata, accepting L_1 and L_2 , respectively. By Theorem 3.6, $(L_1 \cap \overline{L_2}) \cup (\overline{L_1} \cap L_2)$ is accepted by some finite automaton, M_3 . It is easy to see that M_3 accepts a word if and only if $L_1 \neq L_2$. Hence, by Theorem 3.11, there is an algorithm to determine if $L_1 = L_2$.

Qed

3.2 critique of solvability results

The intuition behind the Basic Fact and Theorems 3.11 and 3.12 is that given an automaton da and a string x , we can “run da on x .” This means we can compute $\hat{\delta}x$ and decide whether the resulting state is in F . In the formal notation it means we can reduce $F(da(x))$ to either **tt** or **ff**.

The intuition depends on the *unstated assumption* that when we are given da , we have explicit access to the next state function δ and the final state function F *as algorithms*. It also assumes that when we are given x , we can reduce it to an explicit sequence of symbols. We have seen in section 2 under *explicit form* that if the definitions are read in *classical* set theory, then these assumptions are invalid.

So can these ideas be expressed in classical set theory? Hopcroft and Ullman recognize that they need effective set theory to state their results, but they suggest that once they have defined Turing machines in Chapter 6, then by citing *Church’s thesis* they can connect Turing machines to these procedures. To this end they define the notion of an algorithm.

A *procedure* is a finite sequence of instruction that can be mechanically carried out. We are somewhat vague in our definition of a procedure. ... if we cannot determine whether or not a step can be mechanically carried out, then we reduce the step to a sequence of simpler ones which we can determine can be carried out. ...a procedure which always terminates is called an *algorithm*.

Readers familiar with computability theory might think that by using the notion of a Turing machine, the decidability results (like Theorems 3.11, and 3.12) could be easily formalized in classical set theory and that Hopcroft and Ullman have this in mind when they say on page 80 “It has been hypothesized by Church that any process which could naturally be called a procedure can be realized by a Turing machine.... We shall accept Church’s hypothesis....”

We want to explore in section 4 the consequences of working out this analysis of the algorithms proposed above. What we will see is that it is necessary to present an account of data types as part of the notion of effective procedure. Indeed, it turns out that *this is by far the most demanding part!* It is not the concept of *step* that needs to appear in our mathematics and be elaborated even more carefully but the concept of *explicit data*. This is entirely missing in the Hopcroft and Ullman analysis, yet it leads to a profound reconstruction of the classical set theory itself. Once this is done, as it must be, then *there is no longer any need for the classical set theory in these results; effective set theory is enough*.

Even more profoundly, when we take seriously the task of keeping track of effectiveness, we will rediscover Brouwer’s results that the logical operations must also be refined. Brouwer [15] made his discoveries for analysis and set theory. It is interesting that the same observations apply in automata theory, and with special force in the case of the automata called Turing machines.

4 Explicit Data and Effective Computability

We will analyze the purported “algorithms” from section 3 and see what must be done to make them clearly effective. This will entail that we analyze natural numbers, \mathbb{N} , since they appear in the definition of finite given below. We also want the Booleans \mathbb{B} , although they can be reduced to \mathbb{N} . Next we must examine pairs in Cartesian products $A \times B$, since they are used to group states and symbols as input to the transition function. Then we must be clear about lists, especially Σ^* , since it defines the input data to automata. Then we need to be clear about computable functions $S \times \Sigma^*$ to S and Σ^* to \mathbb{B} . This takes care of the explicit data. In section 5 we will see that these decisions also affect our use of elementary logic.

4.1 natural numbers

In order to carry out the most basic computation in automata theory, or in almost any other branch of mathematics, it must be possible to compute with natural numbers. To compute with them requires that we can reduce an expression for a natural number to decimal form. Indeed, we could follow Bishop [11] in defining a natural number to be an expression which reduces to a decimal numeral or to another canonical form like binary or tally notation.

The canonical natural numbers are 0 and $suc(n)$ where n is a natural number. Two canonical numbers are equal *iff* they are identical, so $0 = 0$ and if $n = m$ then $suc(n) = suc(m)$. The basic

operations of addition (+) and multiplication (*) are given by primitive recursive definitions of *noncanonical expressions* in the standard way:

$$\begin{array}{ll} 0 + y \rightarrow y & 0 * y \rightarrow 0 \\ \text{suc}(n) + y \rightarrow \text{suc}(y + n), & \text{suc}(n) * y \rightarrow y * n + y, \end{array}$$

where \rightarrow denotes a one step reduction. We can argue inductively that its transitive closure must terminate when n and y are natural numbers. Whenever $t \rightarrow t'$, the equality relation cannot distinguish t and t' , so we also know

$$\begin{array}{ll} 0 + y = y & 0 * y = 0 \\ \text{suc}(n) + y = \text{suc}(y + n), & \text{suc}(n) * y = y * n + y. \end{array}$$

In general we can allow any form of recursive definition of new operators that is guaranteed to terminate. Nuprl allows any recursive definition $f(n, y) = E$ as long as it is possible to show that $f(n, y)$ has a value in a type.

The operations + and * are *functional* in each argument in the sense that if $y = y'$ and $n = n'$, then $n + y = n' + y'$ and $n * y = n' * y'$. We can see this by induction on the recursive argument.

4.2 mathematical objects and their canonical forms

It is an important insight of Per Martin-Löf that *all* mathematical objects in a *type* or *constructive set* S can be defined following the pattern for numbers. First, *canonical notations* are defined for elements of S . Then equality is specified on these to determine their abstract referential nature. There are also noncanonical expressions for elements of a set S . These are built from operators which have a computational meaning in the sense that there are mechanical rules for *reducing* them to canonical expressions of the set S . To say that a noncanonical expression e belongs to S is to say that under the rules for the operators in e , e reduces to a canonical member of S . Martin-Löf stipulates that the reduction rules are given independently of type information. This stipulates a strong sense of mechanical computability.

Martin-Löf's approach supports a computational account of all mathematical objects by relying on canonical expressions to achieve *explicitness* of the data and by relying on equality relations to achieve *mathematical abstractness*. In order to achieve the explicitness needed for computation, he relies on specific notation. This is true for any account of computation whether by Turing machines, Markov algorithms, programming languages or systems of recursive equations. At first glance, this approach might seem too "linguistic" or too concrete or too formal, *but computation requires it*. Also, it might seem to belong to the metamathematics rather than abstract mathematics. However, by treating the equality relations explicitly, he achieves an abstract account that is independent of specific choices of notation and which is not necessarily a formalization of a mathematical language, i.e., not a metalanguage but instead a mathematically rigorous use *of language*.

In the next sections we show how to use these ideas to explain other constructive sets needed to define automata.

Cartesian products and dependent products

An automaton is a triple $\langle \delta, q_0, F \rangle$ whose type is parameterized by states K and alphabet Σ . Triples can be built from pairs, namely $\langle a, b, c \rangle$ is $\langle a, \langle b, c \rangle \rangle$. So to define an automaton explicitly we need to have *canonical forms for pairs*. We take these to be $\langle a, b \rangle$ where a belongs to a type A and b belongs to a type B . Two pairs $\langle a, b \rangle$, and $\langle a', b' \rangle$ are equal precisely when $a = a'$ in A and $b = b'$ in B . Pairs $\langle a, b \rangle$ with this equality belong to the type $A \times B$ called the *Cartesian product* of A and B .

There are two commonly needed operations on pairs—selecting the first component and selecting the second. Here are mnemonic notations and reduction rules:

$$\begin{aligned} 1\text{of}(\langle a, b \rangle) &\rightarrow a \\ 2\text{of}(\langle a, b \rangle) &\rightarrow b. \end{aligned}$$

It is immediately clear that these operations respect equality, so we have $\langle a, b \rangle = \langle a', b' \rangle$ in $A \times B$ which entails

$$\begin{aligned} 1\text{of}(\langle a, b \rangle) = 1\text{of}(\langle a', b' \rangle) &\text{ in } A, & \text{and} & & 1\text{of}(\langle a, b \rangle) = a &\text{ in } A, \\ 2\text{of}(\langle a, b \rangle) = 2\text{of}(\langle a', b' \rangle) &\text{ in } B, & & & 2\text{of}(\langle a, b \rangle) = b &\text{ in } B. \end{aligned}$$

In the presence of these operators, we have more noncanonical notations for numbers. For instance, $1\text{of}(\langle 3, x \rangle)$ is a number.

function spaces

There are two common ways to specify functions: one is based on using variables and *abstraction*, and the other uses constants and *combinators*. These lead to different canonical notations for functions. In the case of variables, canonical expressions are those like “ $x + 2$ is a function of x ” or $x \mapsto x + 2$ or $\lambda(x . x + 2)$. In the case of combinators, there are names for constant functions like *identity*, I , where $Ix = x$, or *projections* like $P_1^2 x y = x$, $P_2^2 x y = y$ and constants C_2 , where $C_2 x = 2$ and *Add* where $Add x y = x + y$. Martin-Löf develops a theory based on abstractions as does Nuprl.

The basic idea behind a function f from A to B is that it expresses the notion of a rule for producing an element of B given an element of A . Given f and a in A , the corresponding element of B is usually denoted by an expression combining f and a such as $ap(f; a)$ or $f(a)$ called an *application* of f to a . This is a noncanonical form because there will be rules for reducing $f(a)$ to some canonical b in B . The form of these rules depends on the notation for canonical functions. If we build these notations on the idea of abstraction, then we start with expressions (canonical or noncanonical) for elements of B that are built with variables x ranging over A , say e_x . If this expression has the property that for any a in A , e_a reduces to a canonical element of B , and if $a = a'$ in A implies that $e_a = e_{a'}$ in B , then e_x is said to be *functional in x* . We say that e_x is a function in x . Church’s lambda notation $\lambda(x . e_x)$ is a conventional way to symbolize this.

When λ terms are used as the canonical names for functions, then the reduction rule for $ap(f; a)$ is to reduce f to canonical form, say $\lambda(x . e_x)$, then reduce $ap(\lambda(x . e_x); a)$ by a beta reduction to e_a , then reduce e_a to a canonical b in B .¹²

¹²This is call-by-name reduction. If we first reduce a to canonical form, a' , and then reduce $e_{a'}$, this is call-by-value reduction.

A term $\lambda(x . e(x))$ will be in type $A \rightarrow B$ provided that for any a in A , $e(a)$ is an element of B . That means that $e(a)$ will reduce to a canonical element of B .

lists

The set of Σ lists is critical to automata theory for $\Sigma \subseteq Symbols$. We define the more general concept of A lists for any set A . The canonical elements are inductively defined. There is the atomic element nil . Given any a in A and e in A list, then $cons(a; e)$ is an A list. We abbreviate $cons(a; e)$ by $(a . e)$. If a_1, a_2, \dots are elements of A , then $nil, cons(a_1; nil), cons(a_2; cons(a_1; nil)), \dots$ are in A list. We also write $nil, (a_1 . nil), (a_2 . (a_1 . nil)), \dots$. Another common abbreviation is $(a_n a_{n-1} \dots a_2 a_1)$ for $(a_n . (a_{n-1} \dots (a_2 . (a_1 . nil)) \dots))$.

The noncanonical forms for decomposing lists are built recursively following the inductive structure of the lists. An expression in a set B can be defined from a list and an element y from a set A recursively as follows:

$$\begin{aligned} f(nil, y) &= g(y) \\ f(cons(a; e), y) &= h(a, e, y, f(e, y)). \end{aligned}$$

This expression is in the set B provided that $g(y)$ is and $h(a, e, y, f(e, y))$ is.

The expression is reduced to canonical form by recursive computation. To reduce $f(e_1, e_2)$, reduce e_1 to a canonical list. If the list is nil , then reduce $g(e_2)$. If the list is $cons(a; e)$, then reduce $h(a, e, e_2, f(e, e_2))$. This might require reducing $f(e, e_2)$ but we can argue inductively, based on the structure of $cons(a; e)$, that the reduction procedure will terminate provided that g and h are computable functions.

subset types and finiteness

Finiteness of the set of states, K , is critical to the decidability results of Chapter 3. We say that a set S is *finite* if and only if there is a number k and a bijection f from $0, \dots, k - 1$ to S . In section 6 we will discuss the way bijections are handled. Here we look at the definition of subsets of \mathbb{N} and subsets in general.

We want to define subsets $0, \dots, k - 1$, which are abbreviated as $\mathbb{N}k$. The definition is $\{i : \mathbb{N} \mid 0 \leq i < k\}$ understood in the naive way that elements of $\mathbb{N}k$ are natural numbers i for which we can prove $0 \leq i < k$. To know that i is in $\mathbb{N}k$ we must know that i is a number and that there is a proof of $0 \leq i < k$.

In general we can form subsets of the form $\{i : \mathbb{N} \mid P(i)\}$ for any predicate P on \mathbb{N} ; and in general to know that i is in the set we need to know that i is in \mathbb{N} and that there is a proof of $P(i)$. This construction of subset types is general. Given any type A , and predicate P on A , we can form the subset type $\{x : A \mid P(x)\}$.

In classical set theory, if we know that $a \in \{x : A \mid P(x)\}$, then we know that $P(a)$ is true. Consider a predicate such as $\exists y : B . R(x, y)$. Classically, if we know $a \in \{x : A \mid \exists y : B . R(x, y)\}$, we presume that there is a witness, say b , such that $R(a, b)$.

In effective set theory, we require that there is a proof of $P(a)$, but we do not assume that we have access to information contained in the proof of $P(a)$. If this information is needed, it must be supplied explicitly.

For simple predicates such as $0 \leq i < k$, if we know that i belongs to \mathbb{N} , we can decide whether the predicate is true, and we can reconstruct the information from the proof. But for predicates like $\exists y : B . R(x, y)$, which might not be decidable, it might not be possible to reconstruct information contained in a proof of $\exists y : B . R(x, y)$. If this information is needed in computations, it must be supplied. We will see in the next section how this is done.

4.3 operations on families of sets

The first definition of automata in section 2 used a product operation on a family of sets. We use the notation $x : A \times B_x$ for this product. That is, given sets B_a indexed by a set A , we form $\Sigma x : A . B_x$, the dependent product set. The elements of this set are pairs, $\langle a, b \rangle$, with $a \in A$, $b \in B_a$. The canonical forms of elements are pairs, and projection operators $1\text{of}(p)$, $2\text{of}(p)$ are used to decompose them.

The function space constructor can be extended to families of sets as well. We take $\Pi x : A . B_x$ to be the set of all functions f such that $f(a) \in B_a$ for all $a \in A$. We also use the notation $x : A \rightarrow B_x$.

We take these two operations on families of sets to be *primitive* in effective set theory. In classical set theory they are built from union, subset and comprehension.

5 Logic and Decidability

5.1 computational problems

Decidability or undecidability of a problem is a concept of basic interest in computer science—as the Hopcroft and Ullman text amply illustrates. Throughout this textbook there is a discussion of various *problems* such as:

emptiness problem	does automaton A accept any string?
halting problem on x	does automaton A halt on input x ?
halting problem on blank tape	does Turing machine M halt on blank tape?
equivalence problem for finite automata	do automata A_1, A_2 accept the same language?

Problems are also described as *relations* or *predicates*. So for instance the emptiness problem can be stated as the predicate

$$\exists x : \Sigma^* . da \text{ accepts } x .$$

The “problem” of deciding whether a function f from $\mathbb{N} \rightarrow \mathbb{N}$ has a zero can be stated as

$$\exists x : \mathbb{N} . f(x) = 0 .$$

Hopcroft and Ullman stress that a *problem* usually covers many *instances*. So if we ask about emptiness of automata, there is an instance of the problem for each given automaton da . In the formulation as a predicate, $\exists x : \Sigma^* . da \text{ accepts } x$, the free variable of the problem is da .

Almost any topic in computer science could be chosen to illustrate this important concept of a problem and its decidability. For example, in graph theory we are interested in deciding whether

two graphs are isomorphic or not, or whether a graph is planar or not. In geometry we want to know whether a point is inside or outside of a polygon or whether a polygon is convex or not.

How do we express that a problem is decidable? If the problem $P(x)$ is a predicate on a set A , then Hopcroft and Ullman use the idea that a function f from A to \mathbb{B} decides P when for all a in A

$$f(a) = \mathbf{tt} \text{ in } \mathbb{B} \text{ iff } P(a) \text{ is true.}$$

$$f(a) = \mathbf{ff} \text{ in } \mathbb{B} \text{ iff } P(a) \text{ is false.}$$

We could define the above relation as f decides P or $f(x)$ decides $P(x)$ in x .

In the case of a particular proposition $P(a)$, what would it mean to say that it is decidable? Hopcroft and Ullman do not address this issue, so they do not obtain a general theory of problems. But the above approach gives an answer; we say that $f(a)$ decides $P(a)$ under the above conditions.

We would like to ask whether $P(x)$ or an instance, $P(a)$, is decidable without having to explicitly mention f . We would like to pose the question as a problem:

(i) *Decidable* ($P(x)$) iff there is an effective function f in $A \rightarrow \mathbb{B}$. f decides $P(x)$ in x ,

(ii) *Decidable* ($P(a)$) iff there is a Boolean b in \mathbb{B} . b decides $P(a)$.

The relationship can be naturally generalized as follows. We say that b in \mathbb{B} decides P or Q .

if b reduces to \mathbf{tt} then P is true.

if b reduces to \mathbf{ff} then Q is true.

The previous relation is then

$$\textit{Decidable} (P) \text{ iff } \exists b : \mathbb{B}. b \text{ decides } P \text{ or } \neg P.$$

Paraphrasing we get the English, “ P is decidable if and only if there is a Boolean expression that decides whether P or not.” We also have

$$\textit{Decidable} (P, Q) \text{ iff there is a } b \text{ in } \mathbb{B} \text{ which decides } P \text{ or } Q.$$

There is a problem with using these logical definitions of *Decidable*(P) and *Decidable*(P, Q). We need to interpret the statement that “there is an effective function” and the statement “there is a Boolean b ” in such a way that we can effectively find an explicit algorithm f and an expression b that effectively reduces to \mathbf{tt} or \mathbf{ff} . *This is not what the ordinary classical quantifiers say.* If we write

$$\textit{Decidable} (P(x)) \text{ iff there is an } f \text{ in } A \rightarrow B \text{ such that } f \text{ decides } P(x) \text{ in } x$$

this might not say what we want. The so-called *classical* existential quantifier does *not* have this meaning. Hopcroft and Ullman do not use the quantifier but instead always give the algorithm explicitly when discussing decidability. So they are stipulating a constructive requirement when proving an existential statement.

We can capture their way of speaking if we introduce the *constructive existential quantifier*. This will entail other changes to the logic as well, namely introducing a constructive universal quantifier. It is an interesting design fact about logics that we can derive the constructive quantifiers and the constructive connectives for “or” and “implies” from a careful treatment of problems and the relations on them.

Another key computational idea that we use to reason about decidability is the notion of *reducibility* among problems. We say that problem Q is reducible to problem P , written $Q \leq P$, if solving P enables solving Q . To say that the decidability of Q is reducible to that of P is expressed as $Decidable(Q) \leq Decidable(P)$. For example, in the study of Turing machine halting problems, we have

$$\begin{aligned} Halt_on_Index(M_i) & \text{ iff } M_i(i) \text{ halts and} \\ Halt_on_Blanks(M_i) & \text{ iff } M_i(nil) \text{ halts.} \end{aligned}$$

We can easily prove that there is a function f in $\mathbb{N} \rightarrow \mathbb{N}$ such that

$$Decidable(Halt_on_Blanks(M_i)) \leq Decidable(Halt_on_Index(M_{f(i)}).$$

We also use the notation $P \Rightarrow Q$ for $Q \leq P$.

The key distinction that Hopcroft and Ullman need when they say that a deciding algorithm f “exists” is that they can actually construct f explicitly. They also use this idea in the state minimization algorithm for finite automata. In Chapter 3 they say that a minimum state machine can be found. We need a construction to capture this idea. Here is what we do. We extend the effective set operators to combine sets and propositions. We take $x : A \times P(x)$ to be the set of pairs $\langle a, p \rangle$ where a is an element of A in the constructive sense, and p is a proof of $P(a)$. So we are treating the propositional function $P(x)$ in x as a family of sets. We think of $P(a)$ as the set of all proofs of $P(a)$. We will speak later about what proofs are.

The set $x : A \times P(x)$ will serve perfectly as the constructive existential quantifier which we also write as $\exists x : A . P(x)$. The classical existential quantifier will be denoted as $Exists\ x : A . P(x)$. We do not need it here and, more significantly, it can be defined as we note below.

Once we have combined propositions $P(x)$ with the set constructors, an idea also used in Morse’s set theory [55], we can also form $P \times Q$, $P \rightarrow Q$ and $x : A \rightarrow P(x)$. It is remarkable that these operations make sense in terms of a “logic of problems.” [47]. We have already seen $P \rightarrow Q$ in the definition of $Q \leq P$. To say $P \rightarrow Q$ is to say that there is an effective function that converts a proof of P to a proof of Q . This function f reduces Q to P . The operation $P \times Q$ corresponds to the combined problem; to solve $P \times Q$ we must solve P and solve Q .

The function space $x : A \rightarrow P(x)$ contains effective functions f which, given an element a in A , will produce a proof $f(a)$ of $P(a)$. This notion corresponds exactly to the constructive universal quantifier, so we write it also as $\forall x : A . P(x)$. With this operator, we can define

$$\begin{aligned} Exists\ x : A . P(x) & \text{ iff } \neg \forall x : A . \neg P(x) \\ All\ x : A . P(x) & \text{ iff } \neg \exists x : A . \neg P(x). \end{aligned}$$

We can also define the classical notion of disjunction, P *classical-or* Q , symbolized as $P \odot Q$. It is defined as

$$P \odot Q \text{ iff } \neg(\neg P \& \neg Q).$$

The concept of negation used here is that $\neg P$ holds when $P \Rightarrow False$, i.e., $False \leq P$.

5.2 axiomatizing a logic of problems

It is a remarkable fact discovered by Kolmogorov [47] and Heyting [37] that the operators $Decidable(P, Q)$, $Q \leq P$, \exists , \forall used to make distinctions about decidability, reducibility, construction and uniform solution obey the ordinary laws of logic of the corresponding classical operators with only one exception. Namely, to show $Decidable(P, Q)$ we must either show P or show Q , so the inference rule is

$$\frac{P}{Decidable(P, Q)} \quad \frac{Q}{Decidable(P, Q)}.$$

We lose the classical axiom $Decidable(P, \neg P)$, although we have $P \otimes \neg P$ trivially.

So we adopt the Heyting axioms for these operators along with the usual axioms for P & Q . The result is a *computational predicate logic (of problems)*. From these we can define the classical predicate logic (of truth values) using the definitions given above.

function comprehension

One of the basic connections between relations and functions is called *function comprehension*. It is used to define functions from proofs and can be stated effectively using the “exists a unique” operator:

$$\exists! y : B.P(y) \text{ iff } \exists y : B.(P(y) \ \& \ \forall z : B.P(z) \Rightarrow z = y \text{ in } B).$$

The principle is

$$\forall x : A. \exists! y : B. R(x, y) \Rightarrow \exists f : A \rightarrow B. \forall x : A. R(x, f(x)).$$

This principle is clearly true in effective set theory. But even more is true for the constructive quantifiers, namely a *principle of choice*:

$$\forall x : A. \exists y : B. R(x, y) \Rightarrow \exists f : A \rightarrow B. \forall x : A. R(x, f(x))$$

5.3 expressing finiteness computationally

When Hopcroft and Ullman say that the state set of a finite automaton da is finite, they mean that we know explicitly how big it is. That is, given da , we can compute a decimal representation of the size of the state set. We can see that they mean this from their proof of Theorem 3.11.

This notion of a *computationally finite set* is needed throughout automata and language theory and throughout computational discrete mathematics. It is quite interesting that we can specify this computational notion of finiteness by simply using the computable logical operators instead of the classical ones in the usual definition. Let’s see how this works.

We can say that S is finite if and only if there is a k and a bijection f such that f maps $\{0, \dots, k-1\}$ onto S . Symbolically, S is finite iff $\exists k : \mathbb{N}. \exists f : \mathbb{N}_k \rightarrow S. Bij(\mathbb{N}_k; S; f)$

$Bij(\mathbb{N}_k; S; f)$ means that f is injective and surjective.

$Inj(A; B; f)$ iff $\forall a_1, a_2 : A. (f(a_1) = f(a_2) \text{ in } B \Rightarrow a_1 = a_2 \text{ in } A)$.

$Surj(A; B; f)$ iff $\forall y : B. \exists x : A. (f(x) = y \text{ in } B)$.

Let us now decode this definition of finite using the computational operators. To say that S is finite means that we can find a decimal numeral k which is the size of S . But moreover, we can explicitly count out the elements of S by a prescribed method, namely the function f . The function maps $\mathbb{N}k$ onto S in such a way that each element i in $\mathbb{N}k$ corresponds uniquely under f to exactly one element of S ; i.e., the element $f(i)$ associated with i is not associated to any other number $j \in \mathbb{N}k$. That is, if $f(i) = f(j)$, then $i = j$. This is the injective property of f . Furthermore, every element of S is associated to some element of $\mathbb{N}k$; that is, given s in S there is a number i in $\mathbb{N}k$ such that $f(i) = s$. This is exactly the *surjective* property of f . Since f is injective, the number i corresponding to s is unique. Indeed, we can define an inverse function g from S to $\mathbb{N}k$ such that $f(g(s)) = s$ and $g(f(i)) = i$. This fact is formally stated in the Nuprl `fun_1` library. Notice, the theorem says that we are explicitly building g which is computable.

Theorem 5.1 (*bij_imp_exists_inv*):

$$\forall A, B : Type. \quad \forall f : A \rightarrow B. \quad \text{Bij}(A; B; f) \Rightarrow \exists g : B \rightarrow A. \quad \text{InvsFuns}(A; B; f; g)$$

We say that a set S is *discrete* if and only if its equality relation, $s = t$ in S , is decidable. Clearly \mathbb{N} and all $\mathbb{N}k$ are discrete. It is easy to see from the definition of finiteness that any finite set is discrete because $s = t$ in S iff $g(s) = g(t)$ in $\mathbb{N}k$ for g , the inverse function just discussed.

6 Decidability results for automata

6.1 emptiness problem

We now examine the formal treatment of Theorems 3.11 and 3.12 from Hopcroft and Ullman. To say that we can decide the *emptiness problem* for a finite automaton, da , is to say that we can decide whether there is a string x in the language defined by da . The language is $L(da)$, and x is accepted if and only if the propositional function $L(da)$ is true at x ; i.e., $L(da)(x)$ holds. So the problem is

$$\text{NonEmpty}(da) == \exists x : \Sigma^*. L(da)(x).$$

We want to show that for all finite automata da , $\text{Decidable}(\text{NonEmpty}(da))$.

Theorem 6.1

$$\forall \Sigma, K : Type. \forall da : \text{Automata}(\Sigma, K). \quad \text{Fin}(\Sigma) \ \& \ \text{Fin}(K) \Rightarrow \text{Decidable}(\text{NonEmpty}(da)).$$

The key to the proof is the following lemma which uses the concept of the length of a string x , denoted $\|x\|$.

Lemma 6.1

$$\begin{aligned} \forall \Sigma, K : Type. \quad \forall da : \text{Automata}(\Sigma, K). \text{Fin}(\Sigma) \Rightarrow \\ \forall n : \mathbb{N}. \quad \exists f : \mathbb{N} \rightarrow K. \text{Bij}(\mathbb{N}n; K; f) \Rightarrow \\ (\exists x : \Sigma^*. L(da)(x) \Leftrightarrow \exists k : \mathbb{N}(n + 1). \exists y : \Sigma^*. \|y\| \leq k \ \& \ L(da)(y)). \end{aligned}$$

This lemma tells us that to decide whether $L(da)$ is empty, we only need to check strings whose length is bounded by the number of states of da .

The second lemma we need is

Lemma 6.2

$$\forall \Sigma, K : Type. \forall da : Automata(\Sigma, K). \forall n : \mathbb{N}. Fin(\Sigma) \ \& \ Fin(K) \Rightarrow \\ Decidable(\exists x : \Sigma^*. ||x|| = n \ \& \ L(da)(x)).$$

The lemma can be proved by invoking general facts about decidability. The idea is that if $P(k)$ is a decidable problem and A is a finite set, then $\exists x : A.P(x)$ is a decidable problem because we can check $P(x_i)$ for each element x_i of A for $i = 1, \dots, |A|$, where $|A|$ is the size of A .

The proof of Lemma 6.1 uses results from the proof of the Myhill/Nerode theorem in Hopcroft and Ullman, the most key being the “pumping lemma” to pump down. The lemma says that if $L(da)(w)$ for $n < ||w||$, then the shortest such string w has the form $ab^k c$. So if a string longer than n is accepted, then we can keep removing copies of b until there is an accepted string whose length is n or less.

6.2 abstract pumping lemma

Automata theory can be seen as a branch of algebra [57, 66, 29, 48]. This is well illustrated in Eilenberg’s books, *Automata, Languages and Machines* [29].¹³ Recently Kozen has discovered a beautiful generalization of regular expressions in the notion of a *Kleene algebra* [48]. We can see the flavor and power of these algebraic results by looking at an algebraic form of the pumping lemma used above in the decidability results.

The algebraic notion of a set with an action, an action set, is a good generalization of an automaton.

Definition 2 *An action set over A , $ActionSet(A)$, is a set S and a mapping $act: A \rightarrow (S \rightarrow S)$. Each element of a of A induces an action $act(a)$ on S .*

When A is a monoid with unit 1 and operator written multiplicatively, say ab for combining actions a and b , then a left-action by A on S is defined by requiring

$$act(1)(s) = s \\ act(ab)(s) = act(a)(act(b)(s))$$

If we write the action also as concatenation, then the requirements on a left action are:

$$1s = s \\ (ab)s = a(bs)$$

¹³Notice that Eilenberg has presented his account constructively. He says in the 1974 preface,

One very characteristic feature of the subject should be mentioned here. All the arguments and proofs are constructive. A statement asserting that something exists is of no interest unless it is accompanied by an algorithm (i.e., an explicit or effective procedure) for producing this “something.” Thus, each proof is also an algorithm. Usually but not always, the simplest proof also gives the most efficient algorithm.

A finite automaton on Σ with states K also defines an $ActionSet(\Sigma), \langle K, \hat{\delta} \rangle$ where $\hat{\delta}$ is the transition function. (Eilenberg calls an $ActionSet$ over Σ with a left-action a *left Σ -module*.)

In the automata library we define a left action by Σ^* on S (called there a “multi-action”) with this notation for $s \in S$.

$$\begin{aligned} nil \leftarrow s &= s \\ (h.t) \leftarrow s &= act(h)(t \leftarrow s). \end{aligned}$$

If we also write $act(a)(s)$ as $a \leftarrow s$ then this definition can be written

$$\begin{aligned} nil \leftarrow s &= s \\ (h.t) \leftarrow s &= h \leftarrow (t \leftarrow s). \end{aligned}$$

The notation is suggestive of an automaton starting in state s acting from right to left on the list $(h.t)$. Notice that Σ^* is an example of a monoid. The definition could be generalized to any monoid M , but we did not do that. The pumping lemma we need can be stated for action sets as follows.

Theorem 6.2 *For Σ^* acting on a finite set S of size n , we know that for any s in S and any x in Σ^* whose length is greater than n , there are elements a, b, c , of Σ^* such that $x = (a@b)@c$ and for all k in \mathbb{N}*

$$x \leftarrow s = ((a@b^k)@c) \leftarrow s.$$

Corollary 1 *If there is a string x such that $(x \leftarrow s) = f$ then there is a string y , such that*

$$(y \leftarrow s) = f \text{ and } ||y|| \leq n.$$

Here is a formal statement of Theorem 6.2.

Theorem 6.3

$$\begin{aligned} \forall \Sigma : Type. \forall \langle K, \delta \rangle : ActionSet(\Sigma). \forall n : \mathbb{N}^+. \forall s : K \\ card(K) = n \Rightarrow \forall x : \Sigma^*. (n < ||x|| \Rightarrow \exists a, b, c, : \Sigma^*. \\ 0 < ||b|| \ \& \ x = (a@b)@c \ \& \\ \forall k : \mathbb{N}. ((a@b^k)@c \leftarrow s) = (x \leftarrow s)). \end{aligned}$$

Proof (gloss)

Given alphabet Σ , and $\langle K, \hat{\delta} \rangle$ an $ActionSet$ Σ with the cardinality of K equal to n , let s be any element of K and x any element of Σ^* whose length exceeds n .

We show that there are three elements of Σ^* , say a, b, c which when appended give x as $(a@b)@c$ and such that for any number k

$$(a@b^k)@c \leftarrow s = x \leftarrow s.$$

To find a, b, c , first index the elements of K produced in the action of s on x . Say these are s_1, \dots, s_m for m the length of x . Notice that $n < m$, so by the pigeon-hole lemma, two of these are equal, say $s_i = s_j$ for $i < j$.

Let c be the elements of x that lead to s_i ; that is, $(c \leftarrow s) = s_i$. We can write c as $x[m - i \dots m^-]$ where m^- is $m - 1$. Let b be the elements of x such that $(b \leftarrow s_i) = s_j$.

We can write b as $x[m - j \dots m - (i + 1)]$. Let a be $x[0 \dots m - (j + i)]$. Since $(a @ b) @ c = a @ (b @ c)$,

$$a @ (b @ c) \leftarrow s = a \leftarrow (b @ c \leftarrow s) = a \leftarrow s_j.$$

Thus $a \leftarrow s_j = x \leftarrow s$. We now show by induction on k that

$$(a @ b^k) @ c \leftarrow s = x \leftarrow s.$$

In the base case we must show that

$$(a @ c) \leftarrow s = x \leftarrow s.$$

By definition $(c \leftarrow s) = s_i$ and $(a \leftarrow s_i) = x \leftarrow s$. Since $(a @ c) \leftarrow s = a \leftarrow (c \leftarrow s)$ and $(c \leftarrow s) = s_i$, the result follows. For induction, assume that

$$(a @ b^{k-1}) @ c \leftarrow s = x \leftarrow s.$$

We must show that $(a @ b^k) @ c \leftarrow s = x \leftarrow s$. Notice that $a @ b^k @ c = a @ (b^{k-1} @ b) @ c = a @ b^{k-1} @ (b @ c)$ and $b @ c \leftarrow s = c \leftarrow s$. Thus $a @ b^{k-1} @ (b @ c) \leftarrow s = a @ b^{k-1} @ c \leftarrow s$. So by the induction hypothesis we are done.

Qed.

Here is how Nuprl presents this argument.

Proof (formal)

Decompose the conclusion (UnivCD), THEN Unfold the definition of cardinality and the definition of 1–1 Correspondence, splitting apart the two conjuncts to get declarations of f and g . The following sequent is the result. (Where we take K to be the carrier of the action set, i.e., $K == S.car$.)

1. $\Sigma : Type$
2. $S : ActionSet(\Sigma)$
3. $n : \mathbb{N}^+$
4. $s : K$
5. $f : K \rightarrow \mathbb{N}n$
6. $\exists g : \mathbb{N}n \rightarrow K.InvFuns(S.car; \mathbb{N}n; f; g)$
7. $x : \Sigma^*$

8. $n < \|x\|$

$\vdash \exists a, b, c, : \Sigma^*. 0 < \|b\| \ \& \ x = (a@b)@c \ \& \ \forall k : \mathbb{N}((a@b^k)@c \leftarrow s) = x \leftarrow s$

Now Instantiate Lemma ‘phole_lemma’ with $n, m, \lambda(i.f(x[m - i..m^-] \leftarrow s))$ where $m = \|x\|$ and m^- is $m - 1$.

The ‘phole_lemma’ lemma is the following:

$\forall n : \{1..\}. \forall m : \{n + 1..\}. \forall f : \mathbb{N}m \rightarrow \mathbb{N}m. \exists i, j : \mathbb{N}m. (i < j \ \& \ fi = fj)$

The sequent that results adds these assumptions 9 to 12.

9. $i : \mathbb{N}m$

10. $j : \mathbb{N}m$

11. $i < j$

12. $\lambda(i.f x[m - i..m^-])i = \lambda(i.f x[m - i..m^-])j$

Next, take a to be $x[0..m - (j + 1)]$ ¹⁴

b to be $x[m - j..m - (i + 1)]$, and

c to be $x[m - i..m^-]$.

This decomposes x as follows

$$\begin{array}{ccc} x[0..m - (j + 1)] & x[m - j..m - (i + 1)] & x[m - i..m^-] \\ a & b & c \end{array}$$

The goal is now

$\vdash 0 < \|b\| \ \& \ x = (a@b)@c \ \& \ \forall k : \mathbb{N}((a@b^k)@c \leftarrow s) = x \leftarrow s$

Next decompose the goal; this results in two subgoals.

simple goal: $\vdash 0 < \|b\| \ \& \ x = (a@b)@c$

action goal: $\vdash \forall k : \mathbb{N}((a@b^k)@c \leftarrow s = x \leftarrow s)$

The simple goal is further decomposed into two subgoals. The first one, $0 < \|b\|$, is immediately proved by a lemma called ‘length_segment,’ knowing that $m - j < m - i$ since $i < j$. The second is immediately proved by two lemmas on appending segments (append_segment and whole_segment.)

The interesting step is the action goal. We want to find the segments that produce the same state. So we apply g , the inverse function to f , to conclude that $x[m - i..m^-] \leftarrow s = x[m - j..m^-] \leftarrow s$. This is done by decomposing assumption 6. The result is a renumbered list of hypotheses as follows:

¹⁴The rule is With $x[0..m - (j + 1)]$ (D0) where (D0) means that the conclusion (clause 0) is decomposed.

1. $\Sigma : Type$
2. $S : ActionSet(\Sigma)$
3. $n : \mathbb{N}^+$
4. $s : K$
5. $f : K \rightarrow \mathbb{N}n$
6. $g : \mathbb{N}n \rightarrow K$
7. $g \circ f = Id$
8. $f \circ g = Id$
9. $x : \Sigma^*$
10. $n < ||x||$
11. $i : \mathbb{N}m$
12. $j : \mathbb{N}m$
13. $i < j$
14. $f(x[m - i \dots m^-] \leftarrow s) = f(x[m - j \dots m^-] \leftarrow s)$
15. $x[m - i \dots m^-] \leftarrow s = x[m - j \dots m^-] \leftarrow s$

Next we do induction on \mathbb{N} in the goal, generating two subgoals.

base case $\vdash (a@b^0)@c \leftarrow s = x \leftarrow s$

ind case $\vdash (a@b^k)@c \leftarrow s = x \leftarrow s$

In the base case we use the definition of *lpower* to see that b^0 is *nil*, which we also write as $[]$. This gives the goal

$\vdash (a@[])@c \leftarrow s = x \leftarrow s.$

Using the lemma ‘append_back_nil’ the goal becomes

$\vdash a@c \leftarrow s = x \leftarrow s.$

To prove this we use the ‘maction_lemma’.

$\forall \Sigma : Type. \forall s : ActionSet(\Sigma). \forall s : S.car.$

$\forall L_1, L_2, L : \Sigma^*. (L_1 \leftarrow s = L_2 \leftarrow s \Rightarrow L@L_1 \leftarrow s = L@L_2 \leftarrow s).$

We take L_1 to be c , L_2 to be $b@c$ and L to be a . The goal then follows from 15.

To prove the induction case, notice that we have this sequent

1 - 15 as before.

16. $k : \mathbb{Z}$
17. $0 < k$
18. $(a@b^{k-1})@c \leftarrow s = x \leftarrow s$

$\vdash (a@b^k)@c \leftarrow s = x \leftarrow s$

Now unfold the definition of b^k , *lpower*.

The definition is $L \uparrow n == \text{if } n = 0 \text{ then } [] \text{ else } L \uparrow (n - 1) @ L \text{ fi}$
(that is $b^k = \text{if } k = 0 \text{ then } [] \text{ else } b^{k-1} @ b \text{ fi}$).

In the case $k = 0$ the result is immediate which leaves the new goal:

$\vdash (a@b^{k-1}@b)@c \leftarrow s = x \leftarrow s$

Next we use associativity of append to rewrite the goal; the lemma is ‘append_assoc’, and we use it twice

$\vdash (a@(b^{k-1}@b)@c) \leftarrow s = x \leftarrow s.$

Now we use the ‘maction_lemma’ with L_1 as c , L_2 as $b@c$ and L as $a@b^{k-1}$ to conclude $(a@b^{k-1}@c) \leftarrow s = (a@b^{k-1}@(b@c)) \leftarrow s$. The conclusion follows immediately from 18 (the induction hypothesis).

Qed.

The corollary needed for the decidability results is stated below using $\#(S.car)$ to denote the size of the set of states. The result is that if any element of Σ^* acts on s to produce s' , then a short string, y , acts on s to produce s' .

Corollary

$\vdash \forall n : \mathbb{N}^+. \forall \Sigma : Type. \forall S : ActionSet(\Sigma). \forall s, s' : S.car. [\#(S.car) = n \Rightarrow (\exists x : \Sigma^*. (x \leftarrow s) = s' \Rightarrow \exists y : \Sigma^*. [|y|] \leq n \ \& \ (y \leftarrow s) = s')]$

Proof (formal)

Decompose the goal (and let $K == S.car$)

1. $n : \mathbb{N}^+$
2. $\Sigma : Type$
3. $S : ActionSet(\Sigma)$
4. $s : K$
5. $s' : K$
6. $\#(K) = n$

$$7. \exists x : \Sigma^*(x \leftarrow s) = s'$$

$$\vdash \exists y : \Sigma^*. (||y|| \leq n \ \& \ (y \leftarrow s) = s')$$

Next assert the following conjecture

$$\forall k : \mathbb{N}. \exists x : \Sigma^*. (||x|| < k \ \& \ (x \leftarrow s) = s') \Rightarrow \exists y : \Sigma^*. ||y|| \leq n \ \& \ (y \leftarrow s) = s'.$$

This creates two subgoals; one to prove the asserted formula and the other to prove the main goal given the asserted formula as a new assumption. The second subgoal is trivially proved by decomposing assumption 7 and taking $||x|| + 1$ for k in the assertion. Hence we focus on the first subgoal. We prove it by *induction on k*. This produces the following sequent

1. $n : \mathbb{N}^+$
2. $\Sigma : Type$
3. $S : ActionSet(\Sigma)$ (K is the carrier of S)
4. $s : K$
5. $s' : K$
6. $\#(K) = n$
7. $\exists x : \Sigma^*. (x \leftarrow s) = s'$
8. $k : \mathbb{Z}$
9. $0 < k$
10. $\exists x : \Sigma^*. ||x|| < k - 1 \ \& \ (x \leftarrow s) = s' \Rightarrow \exists y : \Sigma^*. (||y|| \leq n \ \& \ (y \leftarrow s) = s')$ (induction hypothesis)
11. $x : \Sigma^*$
12. $||x|| < k$
13. $(x \leftarrow s) = s'$

$$\vdash \exists y : \Sigma^*. ||y|| \leq n \ \& \ (y \leftarrow s) = s'$$

Next Instantiate the ‘pump_theorem’ with Σ, K, n, s (the theorem is stated above). Then see if $||x|| \leq n$. If so we are done; otherwise we have the assumption $\neg(||x|| \leq n)$. This adds assumptions 14, 15 and 16:

14. $\forall z : \Sigma^*. n < ||z|| \Rightarrow \exists a, b, c, : \Sigma^*. 0 < ||b|| \ \& \ z = (a @ b) @ c \ \& \ \forall k : \mathbb{N}. ((a @ b^k) @ c \leftarrow s) = (z \leftarrow s)$
15. $\neg(||x|| \leq n)$

Now take x for z in 14 and use the fact that $\neg(||x|| \leq n)$ to decompose x into a, b, c . The new assumptions from 13 on are:

16. $\neg(\|x\| \leq n)$
17. $a : \Sigma^*$
18. $b : \Sigma^*$
19. $c : \Sigma^*$
20. $0 < \|b\| \ \& \ x = (a@b)@c$
21. $\forall k : \mathbb{N}.((a@b^k)@c \leftarrow s = x \leftarrow s)$
 $\vdash \exists y : \Sigma^* \|y\| \leq n \ \& \ (y \leftarrow s) = s'$

Next take 0 for k in 19 to delete a piece of x . Use the recursive definition of *lpower* (the definition of b^k) to replace b^0 by *nil*. Then compute the result of appending *nil*.

22. $0 < \|b\| \ \& \ x = (a@b)@c$
23. $(a@[])@c \leftarrow s = x \leftarrow s$

Next invoke the induction hypothesis, 10. This changes to the goal to

$$\vdash \exists x : \Sigma^*. \|x\| < k - 1 \ \& \ (x \leftarrow s) = s'.$$

Now take $a@c$ for x in the conclusion.

$$\vdash \|a@c\| < k - 1 \ \& \ (a@c) \leftarrow s = s'$$

.

Decompose the goals

$$\vdash \|a@c\| < k - 1$$

$$\vdash (a@c) \leftarrow s = s'.$$

The second subgoal is immediate from 13 and 19. In order to prove the first we note that in 12 $\|(a@b)@c\| < k$, and we have removed a non-zero segment b . The exact lemma is ‘length_append’ which says that $\|a@c\| = \|a\| + \|c\|$ and $\|(a@b)@c\| = \|a\| + \|b\| + \|c\|$.

Qed.

We can see in the proof of the corollary the computational procedure used to find y from x . It proceeds to reduce x by applying the pumping lemma over and over again until the accepted string has length less than or equal to n . This is not a particularly efficient procedure. It would be better to compress x to a string y in which no state repeats. But our proof generalizes the Hopcroft and Ullman one to *ActionSets*.

7 Conclusion

7.1 assessment of results

The book *Formal Languages and Their Relation to Automata* has remained useful and relevant for 30 years because it is such a clear and succinct exposition of concepts and theorems of enduring value to computer science, linguistics and mathematics. The Nuprl formalization adds value to Chapter 3 of this book. The formal proofs achieve the highest level of rigor and assurance known. The on-line hypertext and its automatic links into a formal data base of *all* the basic mathematics needed for Chapter 3 increase the utility of the results significantly. The Nuprl system also allows readers to execute the algorithms implicit in the definitions and theorems, thus further adding to the utility of the results.

What is remarkable to me in looking back on the formal proofs two years after doing the work is how readable and clarifying they are. The Hopcroft and Ullman proofs are very good, but the formal proofs add sharpness and logical clarity to an already exceptionally clear presentation. I hope that the reader can sense this from the small examples in section 6.

The hypertext proofs will continue to exist even if Nuprl ceases to run on any commercial computer hardware (a very unlikely event for the next decade at least). The extracted algorithms will run as long as Lisp and ML are supported, and their interpreter can easily be written in any higher type language.

7.2 related work

My colleagues and I at Cornell have carried out other faithful formalizations of results in both discrete and continuous computational mathematics. Several people have formalized the fundamental theorem of arithmetic (FTA) constructively [41].¹⁵

Paul Jackson has shown how to formalize a large amount of computational algebra of the kind that underlies symbolic algebra systems [45]. These results are accessible on the Web from Nuprl home page. James Caldwell [18] is finishing a PhD thesis that presents decidability theorems for classical and intuitionistic propositional calculus, and Pavel Naumov is finishing a PhD thesis that presents a formal semantics for a fragment of Java.

In the realm of continuous mathematics, there is a constructive proof of the Intermediate Value Theorem from Bishop's book [11]. The most up-to-date library is by Forester [31], building on work of Chirimar and Howe [20].

The Coq and Alf research groups have also formalized a number of interesting results in constructive mathematics [44]. Pollock [64] proved a strong normalization theorem for Lego. Werner and Paulin-Mohring [61] have proved a decidability theorem for propositional logic. A topological completeness proof for predicate logic was proved in Alf by Persson [63] and the Hahn-Banach theorem in Coq [19]. Anthony Bailey is finishing a PhD thesis on Galois' theory in Lego with Peter Aczel [5].

¹⁵Boyer and Moore's account in primitive recursive arithmetic (PRA) [14] is also constructive, but PRA does not correspond to ordinary mathematical practice. Chet Murthy produced a constructive proof of Higman's Lemma, but the proof was obtained by automatically translating a classical one and is not readable [65]. Also see Berger and Schwichtenberg [7].

7.3 future directions

The Nuprl mathematics is noteworthy in large measure because it expresses in a unified way both the algorithmic and descriptive aspects of mathematics. The Nuprl theory can capture the basic intuitions behind the Hopcroft and Ullman book, but it cannot directly express notions of computational complexity that would be part of a more modern account. Indeed, we might want to state the Hopcroft result [39] that a finite automaton with n states can be minimized in $n \log n$ time. Work is being done to extend Nuprl's theory in just this manner [23].

The Nuprl *system* takes the formalization beyond what can be achieved for classical logics. The Evaluator is able to mechanically reduce expressions to canonical form; this means it can compute with functions and evaluate the extracts of proofs. We say that the system *animates* the mathematics.

It would be possible to animate parts of classical mathematics as well, but I know of no way to do it which is as conceptually clear as that provided by constructive formalizations such as Alf, Coq and Nuprl. The conceptual basis should be exceptionally clear because it is the foundation for reasoning about computation. As the importance of computation increases in all science, the need for a clear and solid foundation will become paramount.

Finally let me mention the Nuprl Prover. This system of tactics and decision procedures is among the most extensive and expressive among all modern theorem provers, but as with most active theorem provers it is constantly being extended. Tactics are being grouped into larger collections which act as small theorem provers on their own (like Auto tactic). We sometimes call these *super tactics* [2, 1]. They automate more of the proof building. Eventually we hope to support *proof plans* in the sense of Bundy [16, 17].

As systems like Nuprl are rebuilt and overhauled (Nuprl is now on version 5), their basic algorithms usually become faster. So overall the provers are becoming “more powerful;” that is, their algorithms are faster, there are more decision procedures, tactics, supertactics, and plans, and all of them run on faster hardware. As the power of provers increases, they can become easier for the nonexpert to use because more work is done automatically. The Nuprl group is also committed to making the Prover easier to use by improving its user interface – a goal we share with the CTCoq project using the Centaur system [68, 10, 9].

As provers become more powerful and user-friendly, we expect that they will be more widely used. Eventually nonexperts will use these provers to create a world wide digital library of formalized mathematics. Chapter 3 of *Formal Languages and Their Relation to Automata* will be part of it.

Acknowledgements

I want to thank Joan Lockwood and Karla Consroe for helping prepare the manuscript, and Pavel Naumov for carrying out the formal proof of Theorem 3.11, and Jim Caldwell for improving earlier drafts.

References

- [1] Mark Aagaard and Miriam Leeser. Verifying a logic synthesis tool in Nuprl. In Gregor Bochmann and David Probst, editors, *Proceedings of Workshop on Computer-Aided Verification*, pages 72–83. Springer-Verlag, June 1992.
- [2] Mark D. Aagaard, Miriam E. Leeser, and Phillip J. Windley. Towards a super duper hardware tactic. In Jeffrey J. Joyce and Carl-Johan H. Seger, editors, *HOL Theorem Proving System and its Applications*, pages 400–413. Springer-Verlag, 1993.
- [3] Stuart F. Allen. *A Non-Type-Theoretic Semantics for Type-Theoretic Language*. PhD thesis, Cornell University, 1987.
- [4] L. Augustsson, T. Coquand, and B. Nordstrom. A short description of another logical framework. In *Proceedings of the First Annual Workshop on Logical Frameworks*, pages 39–42, Sophia-Antipolis, France, 1990.
- [5] Antony J. Bailey. *The Machine-Checked Literate Formalization of Algebra in Type Theory*. PhD thesis, University of Manchester, 1998.
- [6] M. J. Beeson. *Foundations of Constructive Mathematics*. Springer-Verlag, 1985.
- [7] U. Berger and H. Schwichtenberg. The greatest common divisor: a case study for program extraction from classical proofs. In *Logic Notes in Computer Science*, June 1995. Workshop on Proofs and Types.
- [8] Paul Bernays and A. A. Fraenkel. *Axiomatic Set Theory*. North-Holland Publishing Company, Amsterdam, 1958.
- [9] J. Bertot, Y. Bertot, Y. Coscoy, H. Goguen, and F. Montagnac. *User Guide to the CTCOQ Proof Environment*. INRIA, Sophia Antipolis, February 1997. System Revision 1.22; Documentation Revision 1.31.
- [10] Y. Bertot, G. Kahn, and L. Théry. Proof by pointing. In *Theoretical Aspects of Computer Software*, Lecture Notes in Computer Science, volume 789, pages 141–160, 1994.
- [11] E. Bishop. *Foundations of Constructive Analysis*. McGraw Hill, NY, 1967.
- [12] 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.
- [13] N. Bourbaki. *Elements of Mathematics, Theory of Sets*. Addison-Wesley, Reading, MA, 1968.
- [14] R. S. Boyer and J. S. Moore. *A Computational Logic*. Academic Press, New York, 1979.
- [15] L. E. J. Brouwer. *Collected Works* A. Heyting, ed., volume 1. North-Holland, Amsterdam, 1975. (see On the foundations of mathematics 11-98.).
- [16] A. Bundy. The use of explicit plans to guide inductive proofs. In *Ninth Conference on Automated Deduction*, Lecture Notes in Computer Science, Vol. 203, pages 111–120. Springer-Verlag, 1988.

- [17] A. Bundy. The use of proof plans for normalization. In R.S. Boyer, editor, *Essays in Honor of Woody Bledsoe*, pages 149–166. Kluwer, 1991.
- [18] James Caldwell. Classical propositional decidability via Nuprl proof extraction. Submitted to TPHOLs'98: The 11th International Conference on Theorem Proving in Higher Order Logics, March 1998.
- [19] Jan Cederquist, Thierry Coquand, and Sara Negri. The Hahn-Banach theorem in type theory. In G. Sambin and J. Smith, editors, *Proceedings of Twenty-Five Years of Constructive Type Theory*. Oxford University Press, 1997. To appear.
- [20] J. Chirimar and Douglas J. Howe. Implementing constructive real analysis: a preliminary report. In *Symposium on Constructivity in Computer Science*, pages 165–178. Springer-Verlag, 1991.
- [21] 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.
- [22] Robert L. Constable. The structure of Nuprl's type theory. In Helmut Schwichtenberg, editor, *Logic of Computation*, pages 123–156. Springer, 1997.
- [23] Robert L. Constable. A note on complexity measures for inductive classes in constructive type theory. *Information and Computation*, To appear 1998.
- [24] 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.
- [25] Robert L. Constable, Paul B. Jackson, Pavel Naumov, and Juan Uribe. Constructively formalizing automata. In *Proof, Language and Interaction: Essays in Honour of Robin Milner*. MIT Press, Cambridge, 1998.
- [26] Thierry Coquand and G. Huet. The Calculus of Constructions. *Information and Computation*, 76:95–120, 1988.
- [27] N. G. deBruijn. Set theory with type restrictions. In A. Jahnal, R. Rado, and V. T. Sos, editors, *Infinite and Finite Sets*, volume I of *Collections of the Mathematics Society*, pages 205–314. J. Bolyai 10, 1975.
- [28] G. Dowek, A. Felty, H. Herbelin, G. Huet, C. Murthy, C. Parent, C. Paulin-Mohring, and B. Werner. *The Coq Proof Assistant User's Guide*. INRIA, Version 5.8, 1993.
- [29] Samuel Eilenberg. *Automata, Languages and Machines*, volume A. Academic Press, New York, 1974.
- [30] Solomon Feferman. A language and axioms for explicit mathematics. In J. N. Crossley, editor, *Algebra and Logic*, Lecture Notes in Mathematics, Vol. 480, pages 87–139. Springer, Berlin, 1975.
- [31] Max B. Forester. Formalizing constructive real analysis. Technical Report TR93-1382, Computer Science Department, Cornell University, Ithaca, NY, 1993.

- [32] H. Friedman. Set theoretic foundations for constructive analysis. *Annals of Math*, 105:1–28, 1977.
- [33] H. Friedman and A. Sčedrov. Set existence property for intuitionistic theories with countable choice. *Annals of Pure and Applied Logic*, 25:129–140, 1983.
- [34] Michael Gordon and T. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher-Order Logic*. University Press, Cambridge, 1993.
- [35] 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.
- [36] Paul R. Halmos. *Naive Set Theory*. Springer-Verlag, New York, 1974.
- [37] Arend Heyting. *Mathematische Grundlagenforschung. Intuitionismus. Beweistheorie*. Springer, Berlin, 1934.
- [38] Jason J. Hickey. Nuprl-light: An implementation framework for higher-order logics. In William McCune, editor, *Proceedings of the 14th International Conference on Automated Deduction*, volume 1249 of Lecture Notes on Artificial Intelligence, pages 395–399, Berlin, July 13–17 1997. Springer. CADE '97.
- [39] J. E. Hopcroft. An $n \log n$ algorithm for minimizing states in a finite automaton. In *Theory of Machines and Computations*, pages 189–196. Academic Press, New York, 1971.
- [40] John E. Hopcroft and Jeffrey D. Ullman. *Formal Languages and Their Relation to Automata*. Addison-Wesley, Reading, Massachusetts, 1969.
- [41] Douglas J. Howe. Implementing number theory: An experiment with Nuprl. *Eighth International Conference on Automated Deduction*, Lecture Notes in Computer Science, Vol. 230, pages 404–415, July 1987.
- [42] 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 Lecture Notes in Computer Science, pages 267–282. Springer-Verlag, Berlin, 1996.
- [43] 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 Lecture Notes in Computer Science, pages 85–101. Springer-Verlag, Berlin, 1996.
- [44] G. Huet and A. Saïbi. Constructive category theory. In Gordon Plotkin, Colin Stirling and Mads Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner*. MIT Press, 1998. Presented at CLICS-TYPES BRA '95.
- [45] Paul B. Jackson. *Enhancing the Nuprl Proof Development System and Applying it to Computational Abstract Algebra*. PhD thesis, Cornell University, Ithaca, NY, January 1995.
- [46] Miroslava Kaloper and Piotr Rudnicki. Minimization of finite state machines. Mizar User's Association, 1996.
- [47] A. N. Kolmogorov. Zur deutung der intuitionistischen logik. *Mathematische Zeitschrift*, 35:58–65, 1932.

- [48] Dexter Kozen. A completeness theorem for Kleene algebras and the algebra of regular events. *Information and Computation*, 110:366–390, 1994.
- [49] Dexter Kozen. *Automata and Computability*. Springer, New York, 1997.
- [50] C. Kreitz. Constructive automata theory implemented with the Nuprl proof development system. Technical Report 86–779, Cornell University, Ithaca, New York, September 1986.
- [51] L. Magnusson and B. Nordström. The ALF proof editor and its proof engine. In Springer-Verlag, editor, *Types for Proofs and Programs*, volume 806 of Lecture Notes in Computer Science, pages 213–237, 1994.
- [52] Per Martin-Löf. Constructive mathematics and computer programming. In *Proceedings of the Sixth International Congress for Logic, Methodology, and Philosophy of Science*, pages 153–175, Amsterdam, 1982. North Holland.
- [53] Per Martin-Löf. On the meaning of the logical constants and the justification of the logical laws. Lectures in Siena, 1983.
- [54] David McCarty. Realizability and recursive set theory. *Journal of Pure and Applied Logic*, 32:153–183, 1986.
- [55] A. Morse. *A Theory of Sets*. Academic Press, 1965.
- [56] Y. Moschovakis. *Notes on Set Theory*. Springer-Verlag, New York, 1994.
- [57] A. Nerode. Linear automaton transformations. In *Proceedings of the American Mathematical Society*, volume 9, pages 541–544, 1958.
- [58] Alexei Nogin. Improving the efficiency of Nuprl proofs. Moscow State University, unpublished, 1997.
- [59] B. Nordstrom, K. Petersson, and J. Smith. *Programming in Martin-Löf’s Type Theory*. Oxford Sciences Publication, Oxford, 1990.
- [60] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE, Saratoga NY, 1992)*, volume 607 of Lecture Notes in Computer Science, pages 748–752. Springer-Verlag, 1992.
- [61] Christine Paulin-Mohring and Benjamin Werner. Synthesis of ML programs in the system Coq. *Journal of Symbolic Computations*, 15:607–640, 1993.
- [62] L. C. Paulson. *Isabelle: A Generic Theorem Prover*. Lecture Notes in Computer Science, No. 78. Springer-Verlag, 1994.
- [63] Henrik Persson. A formalization of a constructive completeness proof for intuitionistic predicate logic. Chalmers University of Technology, Goteborg, September 1995. Draft.
- [64] Robert Pollack. *The Theory of LEGO: A Proof Checker for the Extended Calculus of Constructions*. PhD thesis, Department of Computer Science, University of Edinburgh, April 1995.
- [65] J. Russell and Chetan Murthy. A direct constructive proof of Higman’s Lemma. In *Proceedings of the Fifth Symposium on Logic in Computer Science*, pages 257–269. IEEE, 1990.

- [66] Arto Salomaa. Two complete axiom systems for the algebras of regular events. *Journal of the ACM*, 13:158–169, 1966.
- [67] Andrej Šcedrov. Intuitionistic set theory. In Morley, Šcedrov, Harrington and Simpson, editors, *Harvey Friedman’s Research on the Foundations of Mathematics*, pages 257–284. North-Holland, 1985.
- [68] L. Théry, Y. Bertot, and G. Kahn. Real theorem provers deserve real user-interfaces. In *Software Engineering Notes*, volume 17(5), pages 120–129. Fifth Symposium on Software Development Environments, 1992.