TTL-PSS00 Preliminary Version

Aleksey Nogin

Computer Science Department,
Cornell University
Ithaca, NY 14853, USA.
E-mail: NOGIN@CS.CORNELL.EDU

Writing Constructive Proofs Yielding Efficient Extracted Programs

Abstract

The NuPRL system [3] was designed for interactive writing of machine–checked constructive proofs and for extracting algorithms from the proofs. The extracted algorithms are guaranteed to be correct ⋆ which makes it possible to use NuPRL as a programming language with built-in verification[1,5,7,8,9,10]. However it turned out that proofs written without algorithmic efficiency in mind often produce very inefficient algorithms — exponential and double-exponential ones for problems that can be solved in polynomial time.

In this paper we present some general principles of efficient programming in constructive type theory as well as describe a case study that shows how these principles apply to particular problems. We consider the proof of the Myhill–Nerode automata minimization theorem from the NuPRL automata library [4] which leaded to a double–exponential (in time) extracted program. Systematic use of the presented principles allowed us to build a new complexity cautious proof leading to polynomial-time algorithm extracted by the same NuPRL extractor.

We believe that the principles presented in this paper in combination with other methods may lead to an efficient technique of programming-by-proofs.

Key Words and Phrases: automata, constructivity, Myhill-Nerode theorem, NuPRL, program extraction, program verification, programming by extracts, state minimization.

1 Introduction

The NuPRL system [3] is capable of extracting and executing the computational content of constructive theorems even when it is only implicitly mentioned. For example, given a NuPRL proof of the pigeon-hole principle in the form *for any natural number* n *and for any function* f *from* $\{0, 1, \ldots, n\}$ *to* $\{0, 1, \ldots, n-1\}$ *there exists a pair of numbers* $0 \leq i < j \leq n$ *such that*

⋆ Provided that the trusted parts of the NuPRL system work correctly.

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

$f(i) = f(j)$, we can extract a program which takes n, f and computes such i, j. In other words, NuPRL can be regarded as a programming language with build-in verification: a proof is both a program and its verification at the same time.

However the computational efficiency of some NuPRL proofs turned out to be very poor, since they were written without paying attention to efficiency issues.

In the current paper we are presenting some methods that can be used to write efficient proof–programs (section 2). Then we are going to demonstrate how these methods allowed us to write efficiency cautious proofs in NuPRL. In this respect NuPRL is similar to other programming languages, where there often exist slow programs and faster programs, computing the same function.

In particular, we give an exposition of the results of revising the NuPRL proof [4] of Myhill-Nerode automata minimization theorem which is a formalization of [6]. The convenient modular structure of NuPRL theories allowed us to only rewrite the proofs of several inefficient lemmas in order to fix the entire proof. This eliminated all known unnecessary exponential-time proofs from the NuPRL Automata Library and, in particular, the extract from the minimization theorem became polynomial.

We will start by giving a brief overview of the NuPRL Automata Library (section 3). Then, in sections 5,6 and 7 we will describe three most inefficient proofs from the library and show how the principles described in section 2 were applied to turn them from exponential and double-exponential (in time) to polynomial ones.

2 General Principles of Efficient Proof–Programming

Here is a short review of general principles of the computationally efficient programming in the NuPRL type theory, introduced in this work.

One of the basic observations concerning programming by extract is that quite often very elegant proofs yield surprisingly inefficient extracts — such as an exponential–time program extracted from the early proof of the pigeonhole principle. Therefore one can not just write some proof and hope that the extract would do something efficient. We believe that one should start writing a proof while already having some understanding of how the extracted algorithm should work. Some of the principles presented below correspond to well-known principles of efficient programming in traditional imperative languages. But what is new here, is the "translation" of those principles to the "language" of proof–programming as well as the observation that these principles work quite well even for proof–programming.

Use "expensive" statements non-computationally. When it is known or suspected that a certain statement would yield a computationally expensive extract, we should try to avoid using that statement in computational context. We can still use it in non-computational context such as proving that a certain "bad" state is impossible (and, as a result, our algorithm would not get stuck). See section 5.3 for an example of an application of this principle.

Induction steps are the main source of computational complexity. In traditional imperative programming languages most time is usually spent inside various loops and programmers often have to concentrate their efforts on making loops more efficient. In programming by extract, loop code is usually extracted from a proof of some induction step. Consequently, proof writers have to concentrate on writing efficient proofs for induction steps.

Usually the first step towards making an induction step efficient is coming up with a good induction statement. Here are some methods of doing it:

Turn loop invariants into induction statements. In situations where the proof writer has some idea how the loop is going to work, it is often beneficial to try to find some invariant of that loop and *reverse engineer* it into such a proof that its extract actually works as the desired algorithm. In this way the loops of the algorithm are usually programmed by using inductions in the proof, the "if" operator is usually programmed using something like Decide tactics and so on. All the examples in this paper make use of this method.

Use existential quantifiers as memory. When the induction statement has a form $\forall x\, \exists u, v, w, \ldots$ (where x is the object we are doing induction on), $u, v, w, \ldots$ represent the objects that are being computed and saved at each loop iteration. Using, for example, $\forall x : T_1\, \exists y : T_2\, \exists z : T_3.\ A(x, y) \wedge B(x, y, z)$ instead of just $\forall x : T_1\, \exists y : T_2.\, A(x, y)$ prevents us from producing an algorithm that goes back and recomputes z every time it is needed.

Use lists as memory. Evaluate a sufficient amount of data in advance so that the extracted algorithm gets to reuse it instead of recomputing it each time it is needed. Under this approach one has to put all the necessary data in several lists by asserting and proving that a list with certain properties exists and look through these lists when necessary (see sections 6.2, 7.2 and 7.3 for examples of an application of this idea.).

3 NuPRL Automata Library

3.1 Introduction to Automata Library

In this section we are going to give a short overview of the NuPRL Automata Library. This library is based on the Hopcroft – Ullman book [6]. A detailed description of the NuPRL Automata Library can be found in [4].

Two versions of the library are available — the older one, which is described in [4] and which contains several inefficient proofs, and the new one, in which the objects are organized into theories [2] differently and in which the inefficient proofs have been replaced by more efficient ones. All the theories constituting

[2] In NuPRL a theory is a collection of abstraction definitions, theorems (along with proofs) and, possibly, some tactics code and/or comments.

each version of the library are available on the Web. The original theories can be found at `http://nuprlauto-b.nogin.org/` and the updated ones — at `http://nuprlauto.nogin.org/`.

Below you can find a theory-by-theory description of those parts of the updated library that are needed for better understanding of this paper. We provide many definitions in detail because slight difference in details can make writing proofs and especially writing efficient constructive proofs much easier or much more difficult. Some common NuPRL notations are also explained.

3.2 Fnite Sets *theory*[3]

In this theory it is defined what it means for a set s to be finite[4]:

$$Fin(s) \;==\; \exists n : \mathbb{N}.\ \exists f : \mathbb{N}n \to s.\ Bij(\mathbb{N}n; s; f)$$

where $\mathbb{N}n$ is the NuPRL notation for the type $\{0, \ldots, n-1\}$ and $Bij(\mathbb{N}n; s; f)$ says that f is a bijection between $\mathbb{N}n$ and s. From this definition it follows that the equality between elements of a finite set is decidable. This theory also proves several properties of finite sets and of lists of elements of a finite set. In particular, it proves that $\mathbb{N}n$ itself is finite and that the set of fixed length lists of elements of a finite set is finite.

Finally, the pigeon-hole principle is proved (see also section 5).

3.3 Language *theory*[5]

This theory gives a definition of a language. A language over some alphabet $Alph$ is a predicate over $Alph$ List (finite lists of elements of $Alph$). The theory also gives definitions of language operations: intersection, union, product and complement.

3.4 Action Sets *theory*[6]

An action set over some type T is a pair consisting of a carrier type car and an action function that takes a $t \in T$ and $c \in car$ and produces $c' \in car$: $ActionSet(T) \;==\; car : \mathbb{U} \times (T \to car \to car)$. Another way to think about it is that an action set assigns an action $car \to car$ to each element of T. The theory also gives a definition of the multi-action function that naturally extends the definition of the action function from the single elements of T to lists of elements of T:

$$(S : L \leftarrow s) \;==_r\; \texttt{if}\ null(L)\ \texttt{then}\ s\ \texttt{else}\ (S.act\, hd(L)\, (S : tl(L) \leftarrow s))\ \texttt{fi}$$

[3] `http://nuprlauto.nogin.org/finite_sets/`
[4] See section 8 for a discussion of this definition and possible alternatives.
[5] `http://nuprlauto.nogin.org/language/`
[6] `http://nuprlauto.nogin.org/action_sets/`

where $S.act$ is defined to be the second element of S - i.e. S's action, hd and tl are list head and tail operations, $null(L)$ is true *iff* L is an empty list and $==_r$ means that this definition is recursive [7]. Informally, a list $l \in T\,\text{List}$ corresponds to a multi-action that is equal to a composition of actions corresponding to elements of l.

Finally, the pumping lemma is proved. This lemma states that for any action set S with a finite carrier of size n if some multi-action l goes from A to B ($A, B \in S.car$), then there exists a multi-action l' of length $\leq n$ that also goes from A to B. Indeed, by pigeon-hole principle, if l has over n elements, multi-action l has to visit some element $C \in S.car$ at least twice. That means that we can remove the section of l that corresponds to a loop from C to C and obtain a shorter multi-action that still takes A to B. We can repeat this operation until we get a multi-action that is short enough.

3.5 Deterministic Automata *theory* [8]

This theory gives a definition of deterministic automata over some alphabet and a set of states. The automata are defined as triples of a transition function, an initial state, and a function that tells whether a state is a final state:

$$\begin{aligned} &Automata(Alph; States) \;== \\ &\qquad act : (States \to Alph \to States) \times init : States \times (States \to \mathbb{B}) \end{aligned}$$

where $\mathbb{B}$ is a boolean type [9]. The theory also defines the operations δa, $I(a)$ and $F(a)$ that return the three components of an automaton a. Then the theory gives definitions of what state the automaton DA is in after processing an input string l and whether the input string l is accepted:

$$\begin{aligned} DA(l) \quad &==_r \texttt{ if } null(l) \texttt{ then } I(DA) \texttt{ else } ((\delta DA)\ DA(tl(l))\ hd(l)) \texttt{ fi} \\ DA(l)\downarrow &== \ F(DA)\ DA(l) \end{aligned}$$

Finally, the REACH_DEC theorem proves that it is decidable [10] whether some state of an automaton DA is reachable from $I(DA)$ (see also section 6.2).

3.6 Myhill–Nerode Theorem *theory* [11]

[7] NuPRL systems implements recursive definitions using the Y-combinator.

[8] `http://nuprlauto.nogin.org/det_automata/`

[9] NuPRL uses *propositions as types* approach, so it may be undecidable whether a proposition is true or not. On the other hand, boolean type contains only two elements — `tt` and `ff` and it is decidable whether a boolean is true or not.

[10] If NuPRL proves that t is a function, then t must represent a total computable function. Because of that, we can define decidability of a proposition $P(x)$ over some type T as $\forall x : T.\ P(x) \vee \neg P(x)$, which is the same as a dependent function $x : T \to P(x) \vee \neg P(x)$.

[11] `http://nuprlauto.nogin.org/myhill_nerode/`

First a relation Rl_L: $x\,Rl_L\,y$ *iff* $\forall z : A\,\mathrm{List}.\ L(z@x) \Leftrightarrow L(z@y)$ (where L is some language and @ is the list append operator) is defined. Then it is proved that for any language L, Rl_L is an equivalence relation. The relation Rg_g is defined similarly for the case when a language is defined using a function $Alph\,\mathrm{List} \to \mathbb{B}$ instead of a predicate [12].

Also in this theory, the MN_23_LEM_1 lemma is proved. MN_23_LEM_1 states that for any equivalence relation R on $Alph\,\mathrm{List}$ such that for any $x, y, z \in Alph\,\mathrm{List}$, $x\,R\,y$ implies $(z@x)\,R\,(z@y)$, if the number of equivalence classes of R is finite, then for any $g \in Alph\,\mathrm{List} \to \mathbb{B}$ that respects R, the relation Rg_g is decidable (see also section 7).

Finally, the Myhill–Nerode automata minimization theorem is proved. For information on the proof see [4]. Here we are only going to outline the minimization procedure that gets extracted from the proof.

Given an automaton DA, first the reachable states are taken using the decision procedure extracted from REACH_DEC. Then we take $x\,R\,y$ to be the relation $DA(x) = DA(y)$ (automaton goes to the same state after seeing either x or y), $g(x)$ to be $DA(x)\downarrow$ and then use equivalence classes of Rg_g as the states of the minimal automaton. Finally, the decision procedure extracted from MN_23_LEM_1 lemma is used to enumerate the states.

3.7 The rest of the library

The rest of the library includes a definition and properties of non-deterministic automata, proofs of the existence of a deterministic automaton equivalent to a given nondeterministic automaton and other theorems. For information on these parts of the library see [4].

4 Sources of Exponential Complexity

In the existed proof [4] three sources of exponential-time complexity have been detected [13]:

(i) pigeon-hole principle (see sections 4 and 5)

(ii) decidability of the state reachability (see sections 10 and 6)

(iii) decidability of the equivalence relation on words induced by the automata language (the MN_23_LEM_1 lemma — see sections 12 and 7)

Now, after the proofs of these lemmas have been analyzed and rewritten, the resulting extracted programs became polynomial. Whereas it took about 24 hours to evaluate the extract from the old version of the minimization theorem applied to a certain small automaton, the new extract applied to the

[12] A function from $Alph\,\mathrm{List}$ to type of propositions $\mathbb{P}$.

[13] We had to search for sources of exponential complexity manually. Hopefully, Ralph Benzinger's work [2] would lead to a tool capable of doing that automatically.

same automaton was evaluated during only about 40 seconds on the same computer[14].

The current proof of the minimization theorem illustrates that programming by extract can really work.

5 Pigeon-Hole Principle

5.1 Performance

For algorithms extracted from both old and new proofs of this principle the worst case is the case when the only pair of $i > j$ such that $f(i) = f(j)$ is $i = 1$, $j = 0$. That is why for performance comparison we took the function $F = \lambda x.\ \texttt{if}\ (x = 0)\ \texttt{then}\ 0\ \texttt{else}\ x - 1\ \texttt{fi}$ and evaluated the extract from the proof applied to this F and different n. The following table shows how long it took for the evaluator to get the answer:

n	old proof	new proof
10	7.6 sec	1.8 sec
12	29.1 sec	2.3 sec
20	> 20 min	5.2 sec

5.2 Original Exponential Proof

The main part of the pigeon-hole principle is proved in the PHOLE_AUX lemma[15]:

$$\forall n : \{1...\}.\ \forall f : \mathbb{N}(n+1) \to \mathbb{N}n.$$
$$\exists i : \mathbb{N}(n+1).\ \exists j : \{(i+1)..(n+1)^{-}\}.\ f\,i = f\,j$$

where $\{1...\}$ is the NuPRL notation for the set of positive integers and $\{m..n^{-}\}$ is a notation for $\{i \mid m \le i < n\}$.

A NuPRL proof was done by induction.

Base. ($n = 1$) Obviously, $f(0) = f(1)$ $(= 0)$.

Induction step. If there exist such $0 \le k < n$ that $f(n) = f(k)$, then we can take $i = k$, $j = n$. Otherwise, the function $g = \lambda x.\texttt{if}\ (f(x) = n - 1)\ \texttt{then}\ f(n)\ \texttt{else}\ f(x)\ \texttt{fi}$ is a function from $\mathbb{N}n$ to $\mathbb{N}(n-1)$ and we can use the induction hypothesis. Then we can easily prove that if $g(i) = g(j)$, then $f(i) = f(j)$.

[14] All performance numbers in this paper were produced on relatively old hardware. If tested on modern hardware, they should become significantly smaller. Also currently NuPRL interprets the extracted terms. If compiled, the performance of extracted programs should be much better.

[15] http://nuprlauto-b.nogin.org/automata_1/phole_aux..html

For the printout of the NuPRL proof of the induction step, see the Appendix (section 9).

The extracted algorithm was:

(i) Take $n_0 = n$, $f_0 = f$.

(ii) At the kth step:

  (a) Compare $(f_k\, n_k)$ with $(f_k\, i)$ for all $0 \le i < n_k$.

  (b) If for some i $(f_k\, n) = (f_k\, i)$, then $i = i$ and $j = n_k$ is an answer.

  (c) Else take $n_{k+1} = n_k - 1$,
  $f_{k+1} = \lambda x : \mathbb{N}n.\ \texttt{if}\ \ (f_k\, x = n_k - 1)\ \ \texttt{then}\ \ f_k\, n_k\ \ \texttt{else}\ \ f_k\, x\ \ \texttt{fi}$

(iii) On $n-1$th step $(n_{n-1} = 1)$ $i = 0$, $j = 1$ is an answer.

The problem with this algorithm is that in order to calculate $(f_k\, i)$ for some i, the evaluator needs to calculate f_{k-1} twice and calculate f_{k-2} four times and so on up to the f_0, which gets calculated 2^k times.

This proof can be fixed by using

$$g = \lambda x : \mathbb{N}n.\ ((\lambda y.\ \texttt{if}\ \ (y = n-1)\ \ \texttt{then}\ \ f\ \ n\ \ \texttt{else}\ \ y\ \ \texttt{fi})\ (f\ \ x)).$$

The refined proof will work in polynomial time [16] but it will be much slower than the proof described in 5.3.

5.3 Polynomial Proof

The new version of the proof is also called the PHOLE_AUX lemma [17]:

$$\forall n : \{1...\}.\ \forall f : \mathbb{N}(n+1) \to \mathbb{N}n.\ \exists i : \mathbb{N}(n+1).\ \exists j : \mathbb{N}i.\ f i = f j$$

Proof The proof is done by induction over n.
Level 1 - Base. Obviously, f(1)=f(0) (=0)
Level 1 - Induction step. Clearly, the proof of this induction step is the main source of computational complexity. We decided that proof–programming an algorithm that would make a recursive call with $n = n - 1$ (as in old proof) would be inefficient, so we need to prove the induction step without using the induction hypothesis computationally.

To find i, j we decided to go over all pairs $0 \le j < i \le n$ and to check whether $f(i) = f(j)$ [18]. We check i's from n down to 1 and for each i, the j's from $i-1$ down to 0. To program this algorithm we used our *loop invariants into induction statements* principle. Obviously, the invariant of the loop over i's is that we have not found the correct i, j yet and that such a pair still exists

[16] This is because NuPRL evaluator is essentially *call-by-need*.
[17] `http://nuprlauto.nogin.org/finite_sets/phole_aux..html`
[18] The goal of this case study was to get the time complexity down to some reasonable polynomial, but not necessarily to the smallest possible one, so we do not consider this to be too inefficient.

ahead of us, so we asserted that

$$\forall iii : \mathbb{N}(n+1).\ \forall ii : \{(iii+1)..(n+1)^{-}\}.\ \forall jj : \mathbb{N}ii.\ \neg(f\,ii = f\,jj) \ \Rightarrow \\ (\exists i : \mathbb{N}(iii+1).\ \exists j : \mathbb{N}i.\ f\,i = f\,j)$$

"We checked all ii's from n down to iii+1 and haven't found a necessary pair. So there is a pair $0 \le j < i \le iii$ such that $f(i) = f(j)$".

This statement is proved by induction:
Level 2 — Base. $iii = 0$ and we want to prove that

$$\forall ii : \{1..(n+1)^{-}\}.\ \forall jj : \mathbb{N}ii.\ \neg(f\,ii = f\,jj)) \ \Rightarrow \ldots$$

By the level 1 induction hypothesis we prove that the premise of this implication is false. This argument is similar to the old proof, but from the algorithmic standpoint here we are saying that we are going to find our i, j before hitting $i = 0$
Level 2 — Induction Step. Check whether there is a jj in $\{0..iii^{-}\}$ such that $(f\,ii) = (f\,jjj)$ [19]. If such jj is found, then we are done. Otherwise we can use the level 2 induction hypothesis to prove the main goal which corresponds to making a recursive call with $iii := iii - 1$ in our algorithm.

6 State Reachability

$$\forall Alph, St : \mathbb{U}.\ \forall Auto : Automata(Alph; St). \\ Fin(Alph) \Rightarrow Fin(St) \Rightarrow \forall s : St.\ Dec(\exists w : Alph\,\mathrm{List}.\ Auto(w) = s)$$

"For any finite automaton over a finite alphabet and for any state of that automaton the property this state is reachable *is decidable."*

6.1 Original Exponential Proof

In the old version of the library the proof of the decidability of the state reachability is inside the proof of the MN_12 theorem [20].

First, the pumping lemma [21] was used to prove that

$$(\exists t : Alph\,\mathrm{List}\ (Auto(t) = s)) \Leftrightarrow \\ (\exists k : \mathbb{N}(n+1).\ \exists t : \{l : Alph\,\mathrm{List} \mid ||l|| = k\}.\ Auto(t) = s)$$

[19] NuPRL is capable of automatically proving that properties like $\exists jj : \mathbb{N}iii.\ (f\,jj = f\,iii)$ are decidable.
[20] `http://nuprlauto-b.nogin.org/automata_3/mn_12..html`
[21] see section 7 or `http://nuprlauto-b.nogin.org/automata_1/pump_thm_cor..html`

where $||l||$ is the length of list l. Then the proof of the decidability of

$$\exists k : \mathbb{N}(n+1).\ \exists t : \{l : Alph\,\mathrm{List} \mid ||l|| = k\}.\ Auto(t) = s$$

used twice the AUTO2_LEMMA_6 [22] which states that for every finite set T

$$\forall P : T \to \mathbb{P}.\ (\forall t : T.\ Dec(P(t))) \Rightarrow Dec(\exists t : T.\ P(t))$$

The algorithm extracted from the proof of AUTO2_LEMMA_6 simply checks $P(t)$ for all t in T from $f(n-1)$ down to $f(0)$ or to the first t such that $P(t)$ holds (where n is the cardinality of T and f is the "enumerating" function that comes from definition of *"finite"*). So the algorithm extracted from the proof of the decidability of state reachability just checked all words in the alphabet $Alph$ whose length is less or equal to the number of states.

6.2 *New Polynomial Proof*

As per *lists as memory* principle, we are going to compute the list of all reachable states and then to check whether some state is in the list each time we need to know whether some state is reachable. According to the *existential quantifiers as memory* principle, we need to prove the existence of the list of all the reachable states.

In order to prove it, we use a more general notion of action sets (see section 3.4) which will allow us to reuse this theorem later (see section 7.2). Obviously, automata can be regarded as action sets where the carrier is the set of states and the action is the automata transition function. Here is what we prove (REACH_AUX lemma [23]):

$$\begin{aligned}&\forall Alph : \mathbb{U}.\ \forall S : ActionSet(Alph).\ \forall si : S.car.\\ &\qquad Fin(S.car) \Rightarrow Fin(Alph) \Rightarrow (\exists RL : S.car\,\mathrm{List}\ \forall s : S.car.\\ &\qquad\qquad (\exists w : Alph\,\mathrm{List}.\ (S : w \leftarrow si) = s) \Leftrightarrow mem\_f(S.car; s; RL)),\end{aligned}$$

where $mem\_f(T, a, L)$ stands for "a of type T is an element of $T\,\mathrm{List}\ L$".

The idea of our algorithm is to keep all the states we already know to be reachable in a list and for each state s from that list to go over all the letters of the alphabet to get all the states immediately reachable from s and to repeat this procedure until no new states can be added to our list. For efficiency, we want to make sure that we only compute the transition function once for any pair s, α of a reachable state and an alphabet letter. In order to do that, we are going to keep a list of all reachable states for which we have already computed the transition function in RL, and a list of all states immediately

[22] `http://nuprlauto-b.nogin.org/automata_2/auto2_lemma_6..html`
[23] `http://nuprlauto.nogin.org/det_automata/reach_aux..html`

reachable from RL — in RLa. This means that we want to have the following invariant:

After adding n states to RL (putting the initial state si into RL does not count as a step) at least one of the following two statements is true:

(i) RL consists exactly of all reachable (from si) elements of $S.car$

(ii) RL consists of $n+1$ distinct reachable elements of $S.car$ and if we go over the first k letters of the alphabet, we can have the RLa with the following three properties:

  (a) all elements of $S.car$ immediately reachable from the elements of RL (other than its head, which was just added) are either in RL or in RLa (possibly both)

  (b) for any letter a out of the first k letters of the alphabet, the element $(S.act\, a\, hd(RL))$ should appear in either RL or RLa (or both)

  (c) all elements of RLa are reachable (from si)

We turned this invariant into a statement of the REACH_LEMMA[24]:

$$\begin{aligned}
&\forall Alph : \mathbb{U}.\ \forall S : ActionSet(Alph).\ \forall si : S.car.\\
&\forall nn : \mathbb{N}.\ \forall f : \mathbb{N}nn \to Alph.\ \forall g : Alph \to \mathbb{N}nn.\\
&hspace2mmFin(S.car) \Rightarrow InvFuns(\mathbb{N}nn; Alph; f; g) \Rightarrow (\forall n : \mathbb{N}\\
&\quad \exists RL : \{y : \{x : S.car\ \text{List} \mid 0 < ||x|| \wedge ||x|| \le n+1\} \mid y[(||y||-1)] = si\}\\
&\qquad (\forall s : S.car.\ (\exists w : Alph\ \text{List}.\ (S : w \leftarrow si) = s) \Leftrightarrow mem_f(S.car; s; RL))\\
&\quad \vee (||RL|| = n+1\ \wedge\ (\forall i : \mathbb{N}||RL||.\ \forall j : \mathbb{N}i.\ \neg(RL[i] = RL[j]))\\
&\qquad \wedge (\forall s : S.car.\ mem_f(S.car; s; RL) \Rightarrow (\exists w : Alph\ \text{List}.\ (S : w \leftarrow si) = s))\\
&\qquad \wedge (\forall k : \mathbb{N}.\ k \le nn \Rightarrow (\exists RLa : S.car\ \text{List}\\
&\qquad\quad (\forall i : \{1..||RL||^{-}\}.\ \forall a : Alph.\\
&\qquad\qquad mem_f(S.car; S.act\, a\, RL[i]; RL) \vee mem_f(S.car; S.act\, a\, RL[i]; RLa))\\
&\qquad\qquad \wedge (\forall a : Alph.\ ga < k \Rightarrow mem_f(S.car; S.act\, a\, hd(RL); RL)\\
&\qquad\qquad\quad \vee mem_f(S.car; S.act\, a\, hd(RL); RLa))\\
&\qquad\qquad \wedge (\forall s : S.car.\ mem_f(S.car; s; RLa) \Rightarrow\\
&\qquad\qquad\quad (\exists w : Alph\ \text{List}.\ (S : w \leftarrow si) = s))))))
\end{aligned}$$

where $RL[i]$ is the i-th element of RL. REACH_LEMMA states that given an alphabet $Alph$, an action set S over this alphabet, an initial element si in the $S.car$, nn — the size of $Alph$; and functions f and g that give us a one-to-one correspondence between $Alph$ and $\mathbb{N}nn$, we can satisfy our invariant for every

[24] `http://nuprlauto.nogin.org/det_automata/reach_lemma..html`

natural number n.

Now we need to *reverse engineer* the algorithm into a proof of this statement. We start (base case, $n = 0$) with si as the only element of RL. Then we take empty RLa (for $k = 0$) and we go from $k = 1$ up to $k = nn$ (proof by induction) adding $S.act\,(f\ (k-1))\ si$ to RLa at each step.

In the main cycle (induction step of the main induction) we take elements from RLa (list induction) and check whether it is already in RL until either we've found some element s in RLa but not in RL or RLa becomes empty. If RLa becomes empty, then we can prove (by list induction on w) that statement (1) holds, so we are done. And if we've found such an s, then we add it to the top of RL and then we take the rest of RLa as a new RLa for $k = 0$ and then start a cycle (induction) for k from 1 up to nn adding $S.act\,(f\ (k-1))\ s$ to RLa on each step.

To prove REACH_AUX we take n equal to the size of $S.car$, get the correspondent RL from REACH_LEMMA and then use the pigeon–hole principle to prove that (2) can not hold — the number of distinct elements in RL can not be larger than the total number of elements in $S.car$.

7 Decidability of Language Equivalence Relation

In both versions of the library this fact was proved in MN_23_LEM_1 [25]:

$$\begin{aligned}
&\forall Alph : \mathbb{U}.\ \forall R : Alph\,\mathrm{List} \to Alph\,\mathrm{List} \to \mathbb{P}\\
&\quad Fin(Alph) \Rightarrow EquivRel(Alph\,\mathrm{List}; x, y.x\,R\,y)\\
&\quad \Rightarrow\ Fin(x, y : (Alph\,\mathrm{List})//(x\,R\,y))\\
&\quad \Rightarrow\ (\forall x, y, z : Alph\,\mathrm{List}.\ x\,R\,y\ \Rightarrow\ (z\,@\,x)\,R\,(z\,@\,y))\\
&\quad \Rightarrow\ (\forall g : x, y : (Alph\,\mathrm{List})//(x\,R\,y) \to \mathbb{B}.\ \forall x, y : x, y : (Alph\,\mathrm{List})//(x\,R\,y)\\
&\qquad Dec(x\,\mathrm{R}g\,y))
\end{aligned}$$

where $x, y : T//(x\,R\,y)$ is a quotient type [26] and Rg is (by definition and ASSERT_IFF_EQ lemma [27])

$$x\,\mathrm{R}g\,y\ \Leftrightarrow\ \forall z : Alph\,\mathrm{List}.\ g\,(z@x)\ =\ g\,(z@y)$$

[25] http://nuprlauto-b.nogin.org/automata/mn_23_lem_1..html,
http://nuprlauto.nogin.org/myhill_nerode/mn_23_lem_1..html

[26] The quotient type $x,y : T//(x\,R\,y)$ has the same members as the original type T, but with R as its equality relation.

[27] http://nuprlauto-b.nogin.org/automata/assert_iff_eq..html,
http://nuprlauto.nogin.org/myhill_nerode/assert_iff_eq..html

7.1 Original Exponential Proof

The main scheme of the old proof resembles the one of the old proof of decidability of state reachability. First, auto2_lemma_0 [28]

$$\forall T : \mathbb{U}.\ \forall P : T \to \mathbb{P}.$$
$$(\forall x : T.\ Dec(P\,x)) \wedge Dec(\exists x : T\,\neg(P\,x)) \Rightarrow Dec(\forall x : T.\ (P\,x))$$

is used. The proof of $Dec(g\,(z@x) \ = \ g\,(z@y))$ is trivial, so the only fact left to prove is

$$Dec(\exists w : Alph\,\text{List}.\ \neg(g\,(z@x) \ = \ g\,(z@y)))$$

Then some sort of pumping has been used to prove that

$$\exists w : Alph\,\text{List}.\ \neg(g\,(z@x) \ = \ g\,(z@y)) \ \Leftrightarrow$$
$$\exists k : \mathbb{N}(n * n + 1).\ \exists z : \{l : Alph\,\text{List} \,|\, ||l|| = k\}.\ \neg((g\,(z@x) \ = \ g\,(z@y))$$

where n is the size of $x, y : (Alph\,\text{List})//(x\,R\,y)$. The pumping here was proved directly, although the pump_thm_corr applied to something like the action set Sp defined for the new proof could have been used. Then auto2_lemma_6 has been used twice to establish the decidability.

So, the extracted algorithm had to check all words in the alphabet $Alph$ with the length up to $n * n$ to get an answer.

7.2 Polynomial Proof

First we introduce a new action set Sp [29]. Its carrier is the set of pairs $< u, v >$ of equivalence classes defined as $(x, y : (Alph\,\text{List})//(x\,R\,y)) \times (x, y : (Alph\,\text{List})//(x\,R\,y))$ and its action is $\lambda a : Alph.\ \lambda uv.\ \text{let } < u, v >= uv \text{ in } < a :: u, a :: v >$. This definition is valid because $u\,R\,v \ \Rightarrow \ (a :: u)\,R\,(a :: v)$. We can prove that $Sp : w \leftarrow < u, v > = < w@u, w@v >$ (as pairs of equivalence classes).

Then we use reach_lemma to get the list of all pairs "reachable" from the pair $< u, v >$ in this action set. Then, using a trivial list induction, we compute the function g on both elements of each pair in that list and check whether there exists a pair $< u_i, v_i >$ in the list such that $g\,u_i \neq g\,v_i$.

7.3 Another Polynomial Proof

This version of the proof of mn_23_lem_1 is called mn_23_lem [30].

The difference between this proof and the previous one is that instead of computing a list of "reachable" elements for each pair $< u, v >$ we compute

[28] `http://nuprlauto-b.nogin.org/automata_2/auto2_lemma_0..html`
[29] This notation does not appear in the actual proof.
[30] `http://nuprlauto.nogin.org/myhill_nerode/mn_23_lem..html`

the list of all pairs $< u, v >$ such that $\neg(u\, Rg\, v)$ and then just check whether our particular pair is in that list.

First, for each element s of *Sp.car* we compute the list of all elements immediately reachable from s and we put all those lists into a big list of lists. Then, using these lists we compute for each element s of *Sp.car* the list of all elements from which s can be immediately reached (BACK_LISTIFY[31]). Then we take the list of all elements of *Sp.car* and filter such pairs $< u, v >$ in it that $g\,u = g\,v$ (BOOL_LISTIFY[32]). Then we take it as initial list and proceed mostly as in REACH_AUX but going backward (with BACK_LISTIFY) instead of going forward getting the list of all pairs $< u, v >$ such that $\neg(u\,\mathrm{R}g\,v)$ in the end.

8 Possibilities for Further Improvement

Although the algorithms extracted from the new proofs in the NuPRL automata library work fast on small automata, a lot of further improvements may be done in both the automata library and the NuPRL system itself to make the proofs shorter, faster and more readable. Here are some of them.

(i) The NuPRL evaluator should be substantially rewritten. The current one very often unnecessarily evaluates the same terms several times. Ideally, the evaluator should be turned into a compiler.

(ii) If MN_23_LEM would work faster than MN_23_LEM_1 with the new evaluator (it works slower with the current one since it tries to recompute each list anew when it is needed), then it should be used instead of MN_23_LEM_1. And the speed of the extract from MN_23_LEM proof can easily be further significantly improved if we take advantage of the particular structure of our *Sp* — it can be regarded as some sort of product of two equal smaller action sets.

(iii) New tactics should be written to make writing efficient proofs more automatic. In particular, a tactic that adds a new existential quantifier to the induction statement without destroying the existing (possibly unfinished) proof needs to be written. Such tactic would correspond to declaring a new variable.

(iv) More induction principles should be added to the system. For example, an induction principle that allows us to refer to the induction hypothesis for $n = m$ for any $m < n$, not just $n - 1$.

(v) The definition of *finite* turned out to be very inconvenient. It would be better to separate the “finiteness” from the decidability of the equality

[31] `http://nuprlauto.nogin.org/myhill_nerode/back_listify..html`

[32] `http://nuprlauto.nogin.org/myhill_nerode/bool_listify..html`

by using, for example, the following definitions:

$$Fin(T) == \exists FL : T\,\mathrm{List}.\ \forall t : T.\ mem\_f(T, t, FL)$$
$$FinDec(T) == Fin(T) \wedge \forall t1, t2 : T.\ Dec(t1 = t2 \in T)$$

(It can be easily proven in NuPRL that *FinDec* is equivalent to the current definition of *finite*). If the automata library were rewritten with these definitions, then many lemmas would have much shorter proofs (especially INV_OF_FIN_IS_FIN) and minimization would work faster, at least with a new evaluator (above).

(vi) In the current version of the library (as well as in the previous ones) the new abstraction *mn_quo_append* has been introduced, which is equal to *append* but has special wellformedness lemma. It creates technical difficulties in many lemmas. A better way is to prove an extra wellformedness lemma for *append* itself.

Acknowledgement

I am grateful to Robert Constable for generously sharing his observations and for guiding me through all the stages of this work.

9 Appendix

Here is the printout of the NuPRL proof of the induction step from the original proof of the pigeon–hole principle (see section 15). (Proofs of all wellformedness subgoals are omitted).

```
1.  n:  {2...}
2.  ∀f:ℕn → ℕ(-1 + n).  ∃i:ℕn.  ∃j:{(1 + i)..n⁻}.  f i = f j
3.  f:  ℕ(n + 1) → ℕn
⊢ ∃i:ℕ(1 + n).  ∃j:{(1 + i)..(1 + n)⁻}.  f i = f j
|
BY (Decide ⌈∃k:ℕn.  f n = f k⌉ ...a)
|\
| 4.  ∃k:ℕn.  f n = f k
| |
1 BY (D 4 THENM InstConcl [⌈k⌉;⌈n⌉] ...)
\
4.  ¬(∃k:ℕn.  f n = f k)
|
BY (RWW "not_over_exists" 4 ...a)
|
4.  ∀k:ℕn.  ¬(f n = f k)
|
```

```
BY With ⌈λx:ℕn.  if (f x =z n - 1) then f n else f x fi ⌉ (D 2)
THENM Reduce (-1)
|
2.  f:  ℕ(n + 1) → ℕn
3.  ∀k:ℕn.  ¬(f n = f k)
4.  ∃i:ℕn.  ∃j:{(1 + i)..n⁻}
if (f i =z n - 1) then f n else f i fi  =
if (f j =z n - 1) then f n else f j fi
|
BY (ExRepD THENM InstConcl [⌈i⌉;⌈j⌉] ...a)
|
4.  i:  ℕn
5.  j:  {(1 + i)..n⁻}
6.  if (f i =z n - 1) then f n else f i fi  =
if (f j =z n - 1) then f n else f j fi
⊢ f i = f j
|
BY MoveToConcl 6 THENM SplitOnConclITEs THENA Auto'
```

References

[1] Bates, J. L. and R. L. Constable, *Proofs as programs*, ACM Transactions on Programming Languages and Systems **7** (1985), pp. 53–71.

[2] Benzinger, R., *Automated complexity analysis of Nuprl extracted programs* (2000), to appear in Journal of Functional Programming.

[3] Constable, R. L., S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki and S. F. Smith, "Implementing Mathematics with the Nuprl Development System," Prentice-Hall, NJ, 1986.

[4] Constable, R. L., P. Jackson, P. Naumov and J. Uribe, *Constructively formalizing automata theory*, Proof, Language and Interaction: Essays in Honour of Robert Milner (1998).

[5] Constable, R. L., T. Knoblock and J. Bates, *Writing programs that construct proofs*, J. Automated Reasoning **1** (1984), pp. 285–326.

[6] Hopcroft, J. E. and J. D. Ullman, "Formal Languages and Their Relation to Automata," Addison-Wesley, Reading, Massachusetts, 1969.

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

[8] Paulin, C. and B. Werner, *Extracting and executing programs developed in the inductive construction systems*, in: *Proceedings of First Annual Workshop of Logical Frameworks*, Sophia-Antipolis, France, 1990, pp. 349–361.

[9] Paulin-Mohring, C. and B. Werner, *Synthesis of ML programs in the system Coq*, Journal of Symbolic Computations **15** (1993), pp. 607–640.

[10] Sasaki, J. T., "The Extraction and Optimization of Programs from Constructive Proofs," Ph.D. thesis, Cornell University (1985).