Improving the Efficiency of Nuprl Proofs

Aleksey Nogin

August 11, 1997

Abstract

In order to use Nuprl system [1] as a programming language with built-in verification one has to improve the efficiency of the programs extracted from the Nuprl proofs.

In the current paper we consider proofs from the Nuprl automata library [2]. In some of these proofs (pigeon-hole principle, decidability of the state reachability, decidability of the equivalence relation on words induced by the automata language) sources of exponential-time complexity have been detected and replaced by new complexity cautious proofs. The new proofs now lead to polynomial-time algorithms extracted by the same Nuprl extractor, thus eliminating all known unnecessary exponentials from the Nuprl automata library.

General principles of efficient programming on Nuprl are also discussed.

Key Words and Phrases: automata, constructivity, Myhill-Nerode theorem, Nuprl, program extraction, program verification, state minimization.

1 Introduction

The Nuprl system (cf.[1]) is designed to extract and execute 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* $f(i) = f(j)$, we can extract a program which takes n, f and computes these i, j. In other words, Nuprl can be regarded as a programming language with build-in verification: a proof is an algorithm and its verification at the same time.

However, the computational efficiency of some existing proofs in Nuprl is very poor, since the authors of these proofs did not attend to the efficiency issues in their first efforts.

In the current paper we demonstrate that the computational performance of Nuprl can be much better if we write efficiency cautious proofs. In this respect, Nuprl is similar to other programming languages, where there are slow programs and faster programs, computing the same function.

In particular, we give an exposition of the results of revising the Nuprl proof [2] of Myhill-Nerode automata minimization theorem. In the existed proof [2] three sources of exponential-time complexity have been detected:

1. pigeon-hole principle,
2. decidability of the state reachability,
3. decidability of the equivalence relation on words induced by the automata language.

The convenient modular structure of Nuprl theories allows us to just write the proofs of several corresponding lemmas in order to fix the entire proof. Now, after the proofs of these lemmas have been analyzed and rewritten the resulting extracted programs (extracts) became polynomial. Although it took about 24 hours to evaluate the extract from the old version of minimization theorem applied to some small automaton, the new extract applied to the same automaton was evaluated during only about 40 seconds on the same computer.

The current proof of the minimization theorem illustrates that programming by extract can really work. Apart from the refining of the proof of the Myhill–Nerode theorem, we also discuss some general principles of effective programming by extract.

Compete proofs can be found:
New ones —
at `http://www.cs.cornell.edu/Info/People/nogin/automata/`
Old ones — at `http://www.cs.cornell.edu/Info/Projects/NuPrl/Nuprl4.2/Libraries/Automata/`.

2 Pigeon-Hole Principle

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's why, to compare the performance we took the function

$$F \ = \ \lambda x. \text{ if } (x = 0) \text{ then } 0 \text{ else } x - 1 \text{ fi}$$

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	7,610 ms	1,790 ms
12	29,140 ms	2,340 ms
20	??	5,160ms

2.1 Old Proof

Lemma PHOLE_AUX, AUTOMATA_1 theory[1].

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

A Nuprl proof was done by induction.
Base. Obviously, $f(0) = f(1)$ $(= 0)$.
Induction step. If there exist such $0 <= k < n$ that $f(n) = f(k)$ then we can take $i = k,\ j = n$

If not then the function $g\ =\ \lambda x\,.\,\text{if}\ (f(x) = n-1)\ \text{then}\ f(n)\ \text{else}\ f(x)\ \text{fi}$ 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 $f(i) = f(j)$.

Here is the Nuprl-proof of the induction step. (Proofs of all wellfoundness 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
|
```

[1]In the Nuprl system each file with several definitions (abstractions) and lemmas (theorems) is called a theory.

```
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'
```

The extracted algorithm was:

1. Take $n_0 = n$, $f_0 = f$.
2. 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 $n_{k+1} = n_k - 1$,
   $f_{k+1} = \lambda x : \mathbb{N}n.$ if $(f_k\, x = n_k - 1)$ then $f_k\, n_k$ else $f_k\, x$ fi
3. 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 must be calculated 2^k times.

This proof can be fixed by using

$$f_{k+1} = \lambda x : \mathbb{N}n.\ ((\lambda f x.\ \text{if } (f x = n_k - 1) \text{ then } f_k\, n_k \text{ else } f x \text{ fi})\ (f_k\ x)).$$

The refined proof will work in polynomial time but it will be much slower then the proof described in 2.2.

2.2 New Proof

Lemma PHOLE_AUX, FINITE SETS theory.

The idea of new algorithm is to check whether $f(i) = f(j)$ for all pairs $0 \le j < i \le n$. We check i's from n to 1 and for each i, the j's from $i - 1$ down to 0.

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

Proof

Nuprl proof is done by (2-level) induction.

Level 1 - Base. Obviously, f(1)=f(0) (=0)

Level 1 - Induction step. This part of the proof "programs" the main part of the algorithm. It is done by proving some sort of invariant - if at some point we haven't found the necessary pair i, j then it exists among the pairs that we haven't checked yet:

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

"We checked all ii's from n down to $iii+1$ and haven't found a necessary pair. So there is a pair $0 \leq j < i \leq iii$ 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. The only difference is that here the induction hypothesis is used to prove that our algorithm will never come to some point, so in fact it will never be evaluated.
Level 2 — Induction Step. Check whether there is a jj in $\{0..iii^-\}$ such that $(f\,ii) = (f\,jjj)$ (Nuprl is capable of automatically proving that properties like $\exists jj : \mathbb{N}iii.\ (f jj = f iii)$ are decidable.) If such jj is found then we are done. If not then we can

- (in terms of the proofs) use the level 2 induction hypothesis to prove the main goal.
- (in terms of the algorithms) take $iii := iii - 1$ and go back to the beginning of the main cycle.

3 State Reachability

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

"For all finite automata on finite alphabet and for all states of that automaton the property this state is reachable *is decidable."*

3.1 Old Proof

In the old version of the library the proof of the decidability of the state reachability is in MN_12 theorem (AUTOMATA_3 theory) itself.

First, the pumping lemma (PUMP_THM_COR, AUTOMATA_1 theory) was used to prove

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

then the proof of the decidability of

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

used AUTO2_LEMMA_6 (AUTOMATA_2 theory) twice. AUTO2_LEMMA_6 states that for every finite set T

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

The proofs of the finiteness of $\mathbb{N}n$ and of $\{l : \mathit{Alph\,List} | \, ||l|| = k\}$ (for finite Alph) are called NSUB_IS_FINITE and AUTO2_LEMMA_5 (both were in AUTOMATA_2 theory) respectively.

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.

3.2 New Proof

The idea of the new algorithm is to compute the list of all reachable states and then to check whether some state is reachable each time when needed. The extracted algorithm will just check whether the state appears in that list (if some set S is finite then the property $x = y \in S$ is decidable).

The existence of the list of all reachable states is proved in more general terms using the notion of action sets [2]. An *action set* S over an alphabet *Alph* ($S : \mathit{ActionSet}(\mathit{Alph})$ is a pair of a *carrier* ($S.car \in \mathbb{U}$) and an *action* ($S.act \in \mathit{Alph} \to S.car \to S.car$). Automata consist of examples of action sets where the carrier is the set of states and the action is the automata function.

The Nuprl theorem REACH_AUX (DETERMINISTIC AUTOMATA theory):

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

where $\mathit{mem\_f}(T, a, L)$ — a *of type* T *is an element of* T *List* L; $S : w \leftarrow si$ is a display form for the *maction* — function that "takes" an action set S, a word w in the alphabet of this set and an "initial" element si in $S.car$ and "computes" the result of transforming the initial value with $S.act$ using letters of w as the second argument of $S.act$. The formal recursive definition is

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

[2]In the previous version of library action sets were used to prove the pumping lemma

In the new algorithm we keep two lists of elements of $S.car$ — RL is the main list of reachable elements (each element should appear in RL not more then once) and RLa — list of the elements "pending addition to RL".

The main part of the algorithm is "programmed" in REACH_LEMMA:

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

Let us try to decipher this rather scary formula. We have an alphabet $Alph$, an action set S over this alphabet, the initial element si in the $S.car$, nn — the size of $Alph$; and functions f and g give us a one-to-one correspondence between $Alph$ and $\mathbb{N}nn$. REACH_LEMMA says that given all these objects, we can for every natural n find a list RL (with the first element equal to si) satisfying one of the following conditions.

1. RL consists exactly of all reachable (from si) elements of $S.car$
2. RL consists of $n+1$ different reachable elements of $S.car$ and for each $k \leq nn$ we can construct RLa with the following properties:
    (a) all elements of $S.car$ immediately reachable from the elements of RL (but its head) are in RL or RLa
    (b) for all $a : Alph$ such that their numbers $g\,a < k$ the element $(S.act\,a\,hd(RL))$ should appear in RL or in RLa
    (c) all elements of RLa are reachable (from si)

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 on 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 are finished and at this point we prove (list induction on w) that 1) is actually holds. If we've found that s then we add it to the top of RL and then we take the rest of RLa for $k = 0$ and 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 we use the pigeon–hole principle to prove that 2) can not be true — the elements of RL can not be all distinct if RL has more elements than $S.car$.

4 Decidability of Language Equivalence Relation

In both versions of the library this fact was proven in MN_23_LEM_1:

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

where (by definition and ASSERT_IFF_EQ lemma)

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

The language L in alphabet $Alph$ such that $\forall w : Alph\,\mathrm{List}.\ L(w)\ \Leftrightarrow\uparrow(g\,w)$ mentioned in the old version of the library was clearly redundant there, because g itself already defines this language.

4.1 Old Proof

The main scheme of the old proof resembles the one of the old proof of decidability of state reachability. First, AUTO2_LEMMA_0

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

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

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

Then some sort of pumping has been used to prove that

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

where n is the size of $x, y : (Alph\,\mathrm{List})//(x\,R\,y)$. Actually the PUMP_THM_CORR itself applied to something like the action set Sp defined in the new proof could be also used, but here the pumping was proved directly. 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.

4.2 New Proof

First we introduce a new action set. Its carrier is the set of pairs $< x, y >$ defined by:

$$(x, y : (Alph\,\mathrm{List})//(x\,R\,y)) \times (x, y : (Alph\,\mathrm{List})//(x\,R\,y))$$

and its action is

$$\lambda a : Alph.\ \lambda xy.\ \mathrm{let}\ < x, y > = xy\ \mathrm{in}\ < a :: x, a :: y >$$

This definition is correct because $x\,R\,y \Rightarrow (a :: x)\,R\,(a :: y)$. Let us denote this action set as Sp[3]. We can prove that $Sp : z \leftarrow < x, y > = < z@x, z@y >$ (as pairs of equivalence classes).

Then we use REACH_LEMMA to get the list of all pairs "reachable" from the pair $< x, y >$ in this action set. Then we compute the function g on both elements of each pair in that list and check whether in the list there exists such a pair $< x_i, y_i >$ that $g\,x_i \neq g\,y_i$.

4.3 Most Recent Proof

This version of the proof of MN_23_LEM_1 is called MN_23_LEM in MYHILL–NERODE THEOREM theory.

The difference between this proof and the previous one is that instead of computing a list of "reachable" elements for each pair $< x, y >$ we compute the list of all pairs $< x.y >$ such that $\neg(x\,\mathrm{R}g\,y)$ and then just check whether our particular pair is in that list. Unfortunately the current evaluator evaluates this list anew each time we need it, so this version works slower then the previous one under the current evaluator.

First, for each element of *Sp.car* we compute the list of all elements immediately reachable from that element (actually we compute a list of all these lists because we want these lists to be computed once and "memorized"). Then, using these lists we compute for each element of *Sp.car* the list of all elements from which it can be immediately reached (BACK_LISTIFY).

We also take the list of all elements of *Sp.car* (FIN_LISTIFY) and leave only such pairs $< x, y >$ in it that $g\,x \neq g\,y$ (BOOL_LISTIFY). Then we take it

[3]This notation does not appear in the actual proof

as initial list and proceed mostly as in REACH_AUX but going backward (with BACK_LISTIFY) instead of going forward getting the necessary list (list of all pairs $< x.y >$ such that $\neg(x \, \mathrm{R}g \, y)$) at the end.

The speed of [the extract from] this proof may be significantly (and easily) improved if we use the particular structure of our Sp — actually it can be considered as some sort of product of two equal smaller action sets. But this work has been postponed until a better evaluator appears.

5 Conclusion

5.1 General Principles

Here we would like to present a short review of general principles of the computationally efficient programming in Nuprl, introduced in this work.

The first principle due to Robert Constable is *reverse engineering.* The idea behind this is that one starts to write a proof already having some concrete algorithm in mind. Then some invariant of this algorithm should be found and proved in Nuprl in such a way that the extract from this proof actually works as the desired algorithm. In this way the cycles of the algorithm usually become the inductions in the proof, the "if" operator becomes something like DECIDE tactics and so on.

The second method is called the *lists as memory* principle. It is illustrated by our "Most recent proof" of the decidability of the language relation. Here the idea is in evaluating a sufficient amount of data in advance, the evaluator gets to that by reuse it instead of evaluating it each time it is needed. Under this approach one has to put all the necessary data in several lists and look through these lists when necessary. A small problem with this approach is that it does not seem to work with the current Nuprl evaluator.

5.2 Suggestions 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 should be done in both the automata library and the Nuprl system itself to make the proofs shorter, faster and more readable. Here are some suggestions

1. The Nuprl evaluator should be essentially rewritten. The current one very often unnecessarily evaluates the same things several times. Probably a new evaluator should use dag-structures for representing terms.

2. If MN_23_LEM will work faster than MN_23_LEM_1 with the new evaluator then it should be improved in section 4.3.

3. The definition of *finite* turned out to be very inconvenient. We propose alternative definitions:

$$Fin(T) \;==\; \exists FL : T\,\text{List}.\ \forall t : T.\ mem\_f(T, t, FL)$$

$$FinDec(T) \;==\; Fin(T) \;\wedge\; \forall t1, t2 : T.\ Dec(t1 = t2 \in T)$$

   (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).

4. 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 wellfoundness lemma. It really creates a lot of technical difficulties in some lemmas. A better way is to prove an extra wellfoundness lemma for *append* itself.

6 Acknowledgments

I am greatly indebted to R.Constable for his generous sharing with me his observations concerning complexity of the programs extracted from the Nuprl proofs. I am grateful to A.Razborov and L.Beklemishev for educating me about the complexity theory and the proof theory, and P.Naumov for teaching me Nuprl, and R.Eaton for his patient and friendly cooperation.

References

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

[2] R. L. Constable, P. B. Jackson, P. Naumov, and J. Uribe. Constructively formalizing automata. In *Proof Language and Interaction: Essays in Honour of Robin Milner.* MIT Press, Cambridge, 1997.

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

[4] J. T. Sasaki. *The Extraction and Optimization of Programs from Constructive Proofs.* PhD thesis, Cornell University, 1985.