Positive Quantification Is Not Elementary∗
(with Restricted Instantiation)

Aleksy Schubert, Paweł Urzyczyn, and Daria Walukiewicz-Chrząszcz

Institute of Informatics, University of Warsaw, Poland
{alx,urzy,daria}@mimuw.edu.pl

Abstract
We show that a restricted variant of constructive predicate logic with positive (covariant) quantification is of super-elementary complexity. The restriction is to limit the number of eigenvariables used in quantifier introductions rules to a reasonably usable level. This construction suggests that the known non-elementary decision algorithms for positive logic may actually be best possible.

1998 ACM Subject Classification F.4.1 Mathematical Logic, I.2.3 Deduction and Theorem Proving

Keywords and phrases constructive logic, complexity, automata theory

1 Introduction

Constructive logics are basis for many proof assistants [4, 12, 18, 3] as well as theorem provers [1, 13]. Since these tools are actively used for development of verified software [10, 9] and for formalisation of mathematics [7, 8] it is instructive to study computational complexity of various fragments of the logics.

One such fragment consists of positive formulas (understood here as formulas with positive quantification), shown decidable by Mints [11]. As defined there, a formula is positive when it is classically equivalent to one with a quantifier prefix of the form $\forall^*$. If we restrict attention to formulas built with $(\forall, \rightarrow)$ only, we can equivalently say that a formula $\varphi$ is positive if and only if all occurrences of $\forall$ are positive, where:

- The position of $\forall x$ in $\forall x \varphi$ is positive;
- Positive/negative positions in $\varphi$ are positive/negative in $\forall x \varphi$ and in $\psi \rightarrow \varphi$.
- Positive/negative positions in $\psi$ are respectively negative/positive in $\psi \rightarrow \varphi$.

It is not immediate to see that deciding provability for positive formulas is possible. Arbitrary proofs with positive quantification may introduce unbounded number of eigenvariables. However some of them may be regarded as equivalent — variables that ‘satisfy the same assumptions’ can be exchanged one with the other. Thus the identity of an eigenvariable is determined by the set to formulas it occurs in. In the simplest case, when the formula in question is $\forall x.\phi(x)$ and $\phi(x)$ has no quantifiers, we obtain exponential number of possibilities. In case a quantifier, say $\forall y$, occurs in the scope of another one, say our $\forall x$, the situation is more complicated as the variable $x$ may be involved in formulas with $y$ so an exponential number of different copies of $y$ gives rise to doubly exponential bound for the copies of $x$. This argument continues and we obtain a multiply exponential bound in general case.

Different decision algorithms for formulas of minimal positive logic that rigorously develop the idea sketched above were given by Dowek and Jiang [5, 6], Rummelhoff [14], and

∗ Project supported through NCN grant DEC-2012/07/B/ST6/01532.
Xue and Xuan [19]. Our sketch suggests non-elementary complexity and all these algo-

rithms are of non-elementary complexity (while the analogous problem of satisfiability for

$\exists^*$-sentences in classical first-order logic is only NP-complete [2, Thm. 6.4.3]). All of them

observe the pattern that each instantiation of an internal universal quantifier in a proof

creates a new “local environment” of assumptions, opaque, and separated from other such

environments. As a result unbounded depth of quantification makes it possible (in principle)
to represent hereditary finite sets.

As for the lower bound, the best result known up to date is only doubly exponential [16] hardness, and the attempts to prove non-elementary complexity failed; our own proof in [15] turned out incorrect.

While the question of an exact lower bound remains open, the contribution of the present paper makes the non-elementary conjecture quite plausible. As noted above the crucial arguments that make the known algorithms work rely on the fact that the number of eigen-

variables in a proof that occur in formulas serving as intermediate goals in the proof is restricted by appropriate multiply exponential number such that when the bound for the quantifier nesting level $n$ is $k$ then the bound for the level $n + 1$ is exponential in $k$. We show that in case this restriction becomes the part of the problem, i.e., in case we restrict the search space for proofs to ones where the number of eigenvariables occurring in targets of intermediate goals is restricted by an appropriate multiply exponential function, then the problem is non-elementary.

This does not mean that the original problem is non-elementary as it may be the case that there are proofs that violate the multiply exponential bound on eigenvariable occurrences, but are easy to find by some algorithm. However, this seems to be very difficult to imagine since then the algorithm would effectively represent a method to compress multiply exponential complicated structures.

Our upper bound is obtained by an analysis of an appropriate automata-theoretic model. Our Eden automata (or “expansible tree automata”) are alternating machines operating on data structured into trees of knowledge. The computation trees of Eden automata correspond directly to proofs (equivalently, $\lambda$-terms) and the tree structure of trees of knowledge makes it possible to manage the scopes of binders in proofs.

The bound on the number of eigenvariables in a proof corresponds to a bound on the trees of knowledge. Actually each node in the tree corresponds directly to an eigenvariable in a proof. Therefore our restriction on the proofs gives rise to trees of knowledge such that when $k$ is the number of children in a node that starts a tree of depth $n$ then $2^k$ is the number of children in a node that starts a tree of depth $n + 1$.

The version of Eden automata we use in this paper differs in one detail from the more general one present in our another work [16]. The difference is in that the more general version may create children of different kinds. To be more precise, the instruction to create a child may there result in different states whereas in the current paper we allow only one state after a child is created. This additional possibility is necessary to obtain a natural translation of full constructive logic to automata, and is exploited in case there are in a scope of a quantifier, multiple incomparable positions that contain nested quantifiers for child variables. It is natural then to translate them so that they correspond to different states obtained after a child node in the target Eden automaton is created. We could use them in the current paper, but it would force us to bring here notational burden that was not necessary. As current automata are special case and we prove lower bound, it also applies to the general case.

Structure of the paper Section 2 introduces some notation and states the principal def-
initions related to logic and lambda-terms used as proof notation. In Section 3 we give some insight into the intricacy of the problem. Then we introduce Eden automata and define the translation of automata into formulas. The main technical development to encode elementary Turing Machines as Eden automata is done in Section 4.

2 Preliminary definitions

We define \( \exp_0(n) = n \) and \( \exp_{k+1}(n) = 2^{\exp_k(n)} \). A tree is a finite partial order \( \langle T, \leq \rangle \) with a least element \( \varepsilon \in T \) (the root) and such that every non-root element \( w \in T \) has exactly one immediate predecessor (parent). The root of the tree has no proper ancestors. If an element \( w \) of the tree has a parent \( v \) then the set of proper ancestors of \( w \) consists of \( v \) together with all proper ancestors of \( v \). A node \( w \) with exactly \( h \) proper ancestors in \( T \) is said to be at depth \( h \), and then we may write \( |w| = h \). The depth of \( T \) is the maximal depth of a node in \( T \). A tree has uniform depth \( k \) when all its leaves are at depth \( k \).

It is sometimes convenient to refer to the level of a node \( w \) which is the depth of the subtree rooted at \( w \).

A labelled tree is a function \( T : \text{dom}(T) \to L \), where \( L \) is a set of labels. We sometimes identify \( T \) with \( \text{dom}(T) \). If \( L \) is a set of \( m \)-tuples we may say that the dimension of \( T \) is \( m \).

An immediate subtree of a node \( w \) in \( T \) is any of the trees that start in nodes \( v \) that have \( w \) as their parent.

If \( f \) is any function then \( f[x \mapsto a] \) stands for the function \( f' \) such that \( f'(x) = a \), and \( f'(y) = f(y) \), for \( y \neq x \). In particular, \( T[w \mapsto s] \) is a tree obtained from \( T \) by replacing the label at \( w \) by \( s \).

Formulas: We consider the monadic fragment (all predicates are unary) of first-order intuitionistic logic without function symbols and without equality. Therefore the only object terms are object variables, written \( x, y, z, \ldots \). For simplicity we only consider two logical connectives: the implication and the universal quantifier.

We are interested in positive formulas; those are defined in parallel with negative formulas:

- An atom \( P(x) \), where \( P \) is a unary predicate symbol and \( x \) is an object variable, is both a positive and a negative formula.
- If \( \varphi \) is positive and \( \psi \) is negative then \( (\varphi \rightarrow \psi) \) is a negative formula.
- If \( \varphi \) is negative and \( \psi \) is positive then \( (\varphi \rightarrow \psi) \) is a positive formula.
- If \( \varphi \) is positive and \( x \) is an object variable then \( (\forall x \varphi) \) is a positive formula.
- If \( \varphi \) is negative and \( x \) is an object variable then \( (\forall x \varphi) \) is a negative formula.

We use standard parentheses-avoiding conventions, in particular we take implication to be right-associative.

Lambda-terms: In addition to object variables, used in formulas, we also have proof variables occurring in proofs. We use capital letters, like \( X, Y, Z \), for proof variables and lower case letters, like \( x, y, z \), for object variables.

An environment is a set of declarations \( (X : \varphi) \), where \( X \) is a proof variable and \( \varphi \) is a formula. A proof term (or simply “term”) is one of the following:

- a proof variable,
- an abstraction \( \lambda X : \varphi. M \), where \( \varphi \) is a formula and \( M \) is a proof term,
- an abstraction \( \lambda x. M \), where \( M \) is a proof term,
- an application \( MN \), where \( M, N \) are proof terms,
an application $Mx$, where $M$ is a proof term and $x$ is an object variable.

The following type-assignment rules infer judgements of the form $\Gamma \vdash M : \varphi$, where $\Gamma$ is an environment, $M$ is a term, and $\varphi$ is a formula. In rule $$(\forall I)$$ we require $x \not\in \text{FV}(\Gamma)$ and $y$ in rule $$(\forall E)$$ is an arbitrary object variable.

$$
\Gamma, X : \varphi \vdash X : \varphi \quad (Ax)
$$

$$
\frac{\Gamma \vdash \lambda X : \varphi. M : \varphi \rightarrow \psi}{\Gamma \vdash \lambda X : \varphi. M : \varphi \rightarrow \psi} \quad (\rightarrow I)
$$

$$
\frac{\Gamma \vdash M : \varphi \rightarrow \psi \quad \Gamma \vdash N : \varphi}{\Gamma \vdash MN : \psi} \quad (\rightarrow E)
$$

$$
\frac{\Gamma \vdash \lambda x M : \forall x \varphi}{\Gamma \vdash \lambda x M : \forall x \varphi} \quad (\forall I)
$$

$$
\frac{\Gamma \vdash \forall x \varphi \quad \Gamma \vdash M y : \varphi[x := y]}{\Gamma \vdash My : \varphi} \quad (\forall E)
$$

We may write $\lambda X \varphi M$ for $\lambda X : \varphi. M$, and the upper index $\alpha$ in $M^\alpha$ means that term $M$ has type $\alpha$ in some (implicit) environment. Other notational conventions are as usual in lambda-calculus, in particular application is left-associative.

### 2.1 Long normal forms

A redex is a term of the form $(\lambda x M)y$ or of the form $(\lambda Y : \varphi. M)N$. A term which does not contain any redex is said to be in normal form. It is not difficult to see that normal forms are of the following shapes:

- $XN_1 \ldots N_k$, where all $N_i$ are normal forms;
- $\lambda X : \varphi. N$, where $N$ is a normal form;
- $\lambda x N$, where $N$ is a normal form.

It is known, see e.g.,[17, Ch.8], that every well-typed term reduces to one in normal form of the same type. In particular we know that:

*If $\Gamma \vdash M : \varphi$ then there exists a term $N$ in normal form with $\Gamma \vdash N : \varphi$*

In order to prove Lemma 7 we need to manipulate proofs in long normal form.

**Definition 1.** The notion of a long normal form (lnf) of a term is defined according to its type in a given environment.

- If $N$ is an lnf of type $\alpha$ then $\lambda x N$ is an lnf of type $\forall x \alpha$.
- If $N$ is an lnf of type $\beta$ then $\lambda X : \varphi. N$ is an lnf of type $\forall x \beta$.
- If $N_1, \ldots, N_n$ are lnf or object variables and $XN_1 \ldots N_n$ is of an atom type then the term $XN_1 \ldots N_n$ is an lnf.

**Lemma 2.** If $\Gamma \vdash M : \sigma$ and $M$ is in normal form then there exists a long normal form $N$ such that $\Gamma \vdash N : \sigma$.

**Proof.** First let us define a transformation $T$ which will be applied to eliminators in normal form. In $T^\alpha(M)$ we assume that $M$ is of type $\alpha$ in an appropriate environment; the definition is by induction with respect to $\alpha$:

- $T^{\forall x \alpha}(M) = \lambda x T^\alpha(M x)$;
- $T^{\alpha \rightarrow \beta}(M) = \lambda X : \alpha. T^\beta(M X)$;
- $T^\alpha(M) = M$ if $\alpha$ is an atom type.
Suppose that $M = XN_1 \ldots N_k$, where each $N_i$ is an lnf or an object variable. It is easy to see that if $M$ has type $\alpha$ then $T^\alpha(M)$ is an lnf of type $\alpha$.

Transformation $R$ takes an argument in normal form and returns its long normal form. In $R^\alpha(M)$ we assume that $M$ is of type $\alpha$ in some environment; the definition is by induction with respect to $M$:

- $R^{\alpha,\alpha} (\lambda x. P) = \lambda x R^\alpha(P)$
- $R^{\alpha \rightarrow \beta} (AX : \alpha . P) = \lambda X : \alpha . R^\beta(P)$
- $R^\alpha(XP_1 \ldots P_k) = T^\alpha(XP'_1 \ldots P'_k)$, where $P'_i$ is a result of applying $R$ to $P_i$ if $P_i$ is a term, and $P'_i = P_i$ otherwise.

The desired term $N$ equals $R^\alpha(M)$. Details are left to the reader. ▶

**Restricted positive logic**

The following lemma gives a direct characterisation of positive and negative formulas.

▶ **Lemma 3.**

1. Every positive formula is of the form $\forall \vec{x}_1 (\sigma_1 \rightarrow \forall \vec{x}_2 (\sigma_2 \rightarrow \cdots \forall \vec{x}_n (\sigma_n \rightarrow \forall \vec{x}_0 \ a) \ldots ))$, where $\sigma_i$ are negative, and $a$ is an atomic formula.

2. Every negative formula is of the form $\tau_1 \rightarrow \tau_2 \rightarrow \cdots \rightarrow \tau_n \rightarrow a$, where $\tau_i$ are positive, and $a$ is an atomic formula.

The rank of a formula $\varphi$, written $rk(\varphi)$, measures the nesting of occurrences of $\forall \vec{x}$ in $\varphi$. By induction we define:

- $rk(a) = 0$, when $a$ is an atomic formula;
- $rk(\psi \rightarrow \vartheta) = \max\{rk(\psi), rk(\vartheta))\}$;
- $rk(\forall x \psi) = rk(\psi)$, when $\psi$ begins with $\forall$;
- $rk(\forall x \psi) = 1 + rk(\psi)$, otherwise.

By Lemma 2, the type-assignment rules below make a complete proof system for positive formulas. Observe that if a judgement $\Gamma \vdash M : \varphi$ occurs in a long normal proof of a positive formula then all formulas declared in $\Gamma$ must be negative.

$$
\Gamma, X : \varphi \vdash X : \varphi \quad (Ax)
$$

$$
\Gamma, X : \varphi \vdash M : \psi \quad (\rightarrow I)
\Gamma, \lambda X : \varphi . M : \psi \rightarrow \psi
\Gamma \vdash \lambda X : \varphi . M : \varphi \rightarrow \psi

\Gamma \vdash M_i : \tau_i, \ i = 1, \ldots, n
\Gamma, X : \tau_1 \rightarrow \cdots \rightarrow \tau_n \rightarrow a \vdash X M_1 \ldots M_n : a
\Gamma \vdash M : \varphi
\Gamma \vdash \lambda x M : \forall x \varphi \quad (\forall I)
$$

In this paper we focus on the logic with unary predicates only and our hardness proof works already for such special case. It extends to languages that use predicates of greater arity since unary predicates can be simulated there.

Occurrences of a free variable $X$ in a term can be *nested*; this occurs when $X$ is free in some $N_i$ in the context $XN_1 \ldots N_k$ where $k \geq 0$. The maximal nesting $\nu(X, M)$ of a variable $X$ in a term $M$ is defined formally as:

- $\nu(X, X) = 1$, $\nu(X, Y) = 0$, when $X \not= Y$;
- $\nu(X, YN_1 \ldots N_k) = \nu(X, Y) + \max_i \nu(X, N_i)$;
- $\nu(X, \lambda Y N) = \nu(X, N)$, when $X \not= Y$;
\( \forall (X, \lambda y N) = \forall (X, N) \).

**Definition 4** (\(n\)-restricted proofs).

We say that a long normal proof \(M\) of a formula \(\varphi\) is \(n\)-restricted when it has the following property: in every subterm of the form \(\lambda X : \sigma. N\), where \(rk(\sigma) = k > 0\), the variable \(X\) has at most \(\exp_k(n)\) nested occurrences in \(N\), i.e., \(\forall (X, N) \leq \exp_k(n)\).

Observe that this definition restricts only the occurrences of bound variables. Free variables are not constricted here, but some control over their occurrences is also necessary if they are supposed to be bound later in the process. We introduce such restriction later. For the time being this definition is enough to formulate the following problem.

**Problem 5** (restricted decision problem for positive quantification).

**Input:** A positive formula \(\varphi\) and a number \(n\).

**Question:** Does \(\varphi\) have an \(n\)-restricted long normal proof?

### 3 Computational content of positive logic

As already mentioned the main complication of deciding provability in positive logic is that one quantifier can be introduced several times in a proof and may bring to the derivation several different “eigenvariables”. As a result we obtain a potential for unbounded storage. However, we can omit certain of the eigenvariables that cannot be distinguished one from the other. To see how this works and how the computations can be realised in positive logic proofs let us go through the following example.

**Example 6.** We show now that a run of a very simple procedure can be represented by a proof. The procedure has two main steps
1. in loop generate a number of bits,
2. check that there is a bit one and a bit zero among generated ones.

To implement it we need 5 nullary atoms\(^1\): \(Q_{\text{Loop}}, Q_{\text{Gen}}, Q_{\text{Check}}, Q_{\text{Is0}}, Q_{\text{Is1}}\) to represent the flow of the procedure; and two unary ones: \(1(x)\) and \(0(x)\) to represent bits. The intuitive meaning of the nullary atoms is as follows
- \(Q_{\text{Loop}}\) — is used to mark the entry point to the loop to generate bits,
- \(Q_{\text{Gen}}\) — is used to mark the operation to generate a bit,
- \(Q_{\text{Check}}\) — is used to mark the check the presence of bits 0 and 1,
- \(Q_{\text{Is0}}\) — is used to mark the check for presence of the bit 0,
- \(Q_{\text{Is1}}\) — is used to mark the check for presence of the bit 1.

The target formula we would like to prove is
\[
\phi = (\psi \rightarrow Q_{\text{Loop}}) \rightarrow Q_{\text{Loop}}
\]
where
\[
\psi = \forall x. \psi_{\text{gen0}}(x) \rightarrow \psi_{\text{gen1}}(x) \rightarrow \psi_{\text{check}}(x) \rightarrow \psi_{\text{is0}}(x) \rightarrow \psi_{\text{is1}}(x) \rightarrow Q_{\text{Gen}}
\]
with the following definitions (and their intuitive descriptions)
- \(\psi_{\text{gen0}}(x) = (0(x) \rightarrow Q_{\text{Loop}}) \rightarrow Q_{\text{Gen}}\) — generate bit 0 and go to the main loop,
- \(\psi_{\text{gen1}}(x) = (1(x) \rightarrow Q_{\text{Loop}}) \rightarrow Q_{\text{Gen}}\) — generate bit 1 and go to the main loop,
- \(\psi_{\text{check}}(x) = (1(x) \rightarrow Q_{\text{Check}}) \rightarrow Q_{\text{Gen}}\) — generate bit 1 and go to the check phase,
- \(\psi_{\text{check}}(x) = (1(x) \rightarrow Q_{\text{Check}}) \rightarrow Q_{\text{Gen}}\) — check if bit 0 and bit 1 are present,
- \(\psi_{\text{is0}}(x) = 0(x) \rightarrow Q_{\text{Is0}}\) — announce the presence of the bit zero,
- \(\psi_{\text{is1}}(x) = 1(x) \rightarrow Q_{\text{Is1}}\) — announce the presence of the bit one.

\(^1\) Nullary atoms, used for clarity, can be easily replaced by unary ones.
A long normal proof for the formula $\phi$ (a lambda term) must take the shape
\[
\lambda X \psi \rightarrow Q_{\text{Loop}}. X \psi \rightarrow Q_{\text{Gen}}. \lambda Y_1 Z_1 U_1 V_1 W_{0,1} W_{1,1} M_1
\]
where $Y_1 : \psi_{\text{gen}}(x_1)$, $Z_1 : \psi_{\text{gen}}(x_1)$, $U_1 : \psi_{\text{check}}(x_1)$, $V_1 : \psi_{\text{check}}(x_1)$, $W_{0,1} : \psi_{\text{0}}(x_1)$, $W_{1,1} : \psi_{\text{at1}}(x_1)$ and $M_1 : Q_{\text{Gen}}$. Observe that $x_1$ is an eigenvariable introduced to the proof through the only quantifier of the original formula $\phi$. We should also note here that with this partial proof we moved the state of the algorithm from the one represented by $Q_{\text{Gen}}$ to the one represented by $Q_{\text{Loop}}$. The term $M_1$ can now begin with any of the variables $Y_1, Z_1, U_1$, but the use of $U_1$ would force us to the phase to check that bits 0 and 1 are present, but none bits are currently present, so this option is impossible. We can take now either of $Y_1, Z_1$ and obtain a proper proof. Let us take $Y_1$ to make the bit 0 available and we are further forced to let $M_1 = Y_1 M_1^{0(x_1)}, N_1$ with

\[
N_1 : Q_{\text{Loop}} = X \psi \rightarrow Q_{\text{Gen}} \lambda x_2 \lambda Y_2 Z_2 U_2 V_2 W_{0,2} W_{1,2} M_2
\]
(1)

Here $x_2$ is a new eigenvariable introduced to the proof through the only quantifier of the original formula $\phi$. The term $M_2$ could now begin with any of the variables $Y_1, Z_1, U_1, Y_2, Z_2, U_2$. Since we already have the bit 0 (available through $T_1^{0(x_1)}$), we can proceed to $U_1$, which was not possible in the previous turn. However, we choose now to start the term with $U_2$. Before this, we would like to observe that taking $Y_2$ would result in $M_2 = Y_2 M_2^{0(x_2)}, N_2$ for some $N_2$ and this would make $x_1$ and $x_2$ equivalent in such a way that for the set $\Delta_{x_1}$ of formulas in which $x_1$ occurs and the set $\Delta_{x_2}$ of formulas in which $x_2$ occurs we would have $\Delta_{x_1} = \Delta_{x_2} [x_2 := x_1]$. As a result, taking this path leads to a situation that is practically the same as the current one.

Coming back to the main thread, let us take $M_2 = U_2 M_2^{1(x_2)}, N_2$. We can now define $N_2 : Q_{\text{Check}}$ in a different fashion than in case of $N_1$ in (1), namely

\[
N_2 = V_1(W_{0,1} T_1^{0(x_1)})(W_{1,2} T_2^{1(x_2)})
\]

Observe that the subproofs $W_{x_1(0,1)} T_1^{0(x_1)}$ and $W_{1,2} T_2^{1(x_2)}$ have types $Q_{\text{at1}}$ and $Q_{\text{at1}}$ respectively, and these correspond to the states of the original procedure that check for the presence of the bit 0 and the bit 1 respectively.

As a result we obtain the complete proof of $\phi$

\[
\lambda X \psi. X \psi \lambda x_1 \lambda Y_1 Z_1 U_1 V_1 W_{0,1} W_{1,1}.
Y_1 M_1^{0(x_1)} \lambda x_2 \lambda Y_2 Z_2 U_2 V_2 W_{0,2} W_{1,2}.
U_2 M_2^{1(x_2)}, V_1(W_{0,1} T_1^{0(x_1)})(W_{1,2} T_2^{1(x_2)}).
\]

As we mentioned before, a proof can involve an unbounded number of variables, actually this can be realised for the formula in the example above. In [5] it is shown rigorously how some eigenvariables may be eliminated, because “equivalent” variables can replace each other. The term “equivalent” is understood as “satisfying the same assumptions” and the basic instance of the equivalence is presented in Example 6.

The number of necessary non-equivalent eigenvariables is therefore essential to determine the complexity. A closer analysis of the algorithm in [5] reveals a super-elementary (tetration) upper bound, in other words the problem belongs to Grzegorczyk’s class E4.

Indeed, a formula of length $n$ has $O(n)$ different subformulas, so if it only has one quantifier $\forall x$ (like the one in our example) then the number of non-equivalent variables introduced for the quantifier is (in the worst case) exponential in $n$, as one has to account
for every selection from up to $O(n)$ subformulas including free occurrences of $x$. And here the quantifier depth comes into play. Consider a formula of the form $\forall x (\ldots \forall y \varphi(x,y) \ldots)$. For every eigenvariable $x'$ for $\forall x$ we now have $O(n)$ subformulas of $\varphi(x',y)$ and therefore up to exponentially many eigenvariables obtained from $\forall y$. Any set of such eigenvariables may potentially be created for a given eigenvariable for $\forall x$, and this gives a doubly exponential number of choices. Two eigenvariables coming from $\forall x$ may be assumed equivalent only when they induce the same choice, so we get a doubly exponential number of possible non-equivalent eigenvariables from $\forall x$. Every next quantifier nested in the scope of all previous ones increases the number of non-equivalent eigenvariables exponentially, and this yields the super-elementary upper bound.

In order to determine a lower bound we interpret proof-search in terms of an appropriate automaton. The idea is simple and, we believe, quite universal. When attempting to construct a proof of a formula $\varphi$, one encounters subproblems of the form $\Gamma \vdash \alpha$. We think of $\alpha$ as if it was a state of an automaton and of $\Gamma$ as of some kind of memory storage. If we restrict attention to long normal proofs then $\alpha$ is typically an atomic “subformula” of the initial proof goal $\varphi$. The finite number of atoms in $\varphi$ makes a finite set of states, if one can account for arbitrary instantiations of individual variables. In our construction this is handled by a pointer addressing an adequate position in memory, organised as a tree. The tree reflects the structure of instantiated quantifiers and stores “knowledge” about proof assumptions.

### 3.1 Eden automata

An Eden automaton (abbr. Ea) is an alternating computing device, organising its memory into a tree of knowledge of bounded depth but potentially unbounded width. The tree initially consists of a single root node and may grow during machine computation, not exceeding a fixed maximum depth. The machine can access memory registers at the presently visited node and its ancestor nodes. This access is limited to using the registers as guards: it can be verified that a flag is up, but checking that a flag is down is simply impossible. Every flag is initially down, but once raised, it so remains forever.

Formally, an Ea is a tuple $\mathcal{A} = (k, m, M, Q, q^0, \mathcal{I})$, where:

- $k \in \mathbb{N}$ is the depth of $\mathcal{A}$.
- $M$ is the finite set of registers; the number $m = |M|$ is the dimension of $\mathcal{A}$.
- $Q$ is the finite set of states, partitioned as $Q = \bigcup_{i \leq k} Q_i$. In addition, each $Q_i$ splits into disjoint sets $Q_i^\varnothing$ and $Q_i^\exists$ and we also define $Q^\varnothing = \bigcup_{i \leq k} Q_i^\varnothing$ and $Q^\exists = \bigcup_{i \leq k} Q_i^\exists$. States in $Q^\varnothing$, $Q^\exists$ are respectively universal and existential.
- $q^0 : k \rightarrow Q$ assigns the initial state $q^0_i \in Q_i$ to every $i \in k$.
- $\mathcal{I}$ is the set of instructions.

Instructions in $\mathcal{I}$ available in state $q \in Q_i$, may be of the following kinds:

1. “$q : \text{jmp } p$”, where $p \in Q_j$, and $|i - j| \leq 1$;
2. “$q : \text{check } R(h) \text{ jmp } p$”, where $p \in Q_i$ and $h \leq i$;
3. “$q : \text{set } R(h) \text{ jmp } p$”, where $p \in Q_i$, and $h \leq i$;
4. “$q : \text{new}$”, for $i < k$.

Instructions available in $q \in Q_i^\varnothing$, for any $i$, must be of kind 1, with $j = i$. If $q \in Q_h$ in 2 or 3 then we write $R$ instead of $R(h)$. An ID (instantaneous description) of $\mathcal{A}$ is a triple $\langle q, T, w \rangle$, where $q$ is a state and $T$ is a tree of depth at most $k$, labelled with elements of $\{0,1\}^M$ (i.e., functions from $M$ to $\{0,1\}$), called snakes. That is, if $v$ is a node of $T$ then
A. Schubert and P. Urzyczyn and D. Walukiewicz-Chrząszcz

\( T(v) \) is a snake, and \( T(v)(R) \in \{0, 1\} \) for any register \( R \). When \( T \) is known from the context, we write \( R(v) \) for \( T(v)(R) \). A snake can be identified with a binary string of length \( m \), for example \( \emptyset \) stands for a snake constantly equal to 0. Finally, the component \( w \) is a node of \( T \) called the current apple. We require that \( q \in Q_0 \). That is, the internal state always "knows" the depth of the current apple.

The IDs are classified as existential and universal, depending on their states. The initial ID is \( \langle q_0^0, T_0, \varepsilon \rangle \), where \( T_0 \) has only one node \( \varepsilon \), the root, labelled with \( \emptyset \) (all flags are down).

An ID \( C' = \langle p, T', w' \rangle \) is a successor of \( C = \langle q, T, w \rangle \), when \( C' \) is a result of execution of an instruction \( I \in \mathcal{J} \) at \( C \). We now define how this may happen. Assume that \( q \in Q_1 \), and first consider the case when \( I = "q : jmp \ p". \)

- If \( p \in Q_1 \), then \( C' = \langle p, T, w \rangle \) is the unique result of execution of \( I \) at \( C \). (The machine simply changes its internal state from \( q \) to \( p \).)
- If \( p \in Q_{i-1} \) then the only possible result is \( C' = \langle p, T, w' \rangle \), where \( w' \) is the parent node of \( w \). (The machine moves the apple upward and enters state \( p \).)
- If \( p \in Q_{i+1} \) then there may be many results of execution of \( I \), namely all IDs of the form \( C' = \langle p, T, w' \rangle \), where \( w' \) is any successor of \( w \) in \( T \). (The apple is passed downward to a non-deterministically chosen child \( w' \) of \( w \)). In case \( w \) is a leaf, there is no result (the instruction cannot be executed).

Let now \( I \) be of the form 2 and let \( v \in T \) be the (possibly improper) ancestor of \( w \) such that \( |v| = h \). If register \( R \) at \( v \) is 1 (i.e., \( T(v)(R) = 1 \)) then the only result of execution of \( I \) at \( C \) is \( \langle p, T, w \rangle \). Otherwise there is no result.

If \( I = "q : set \ R(h) \ jmp \ p" \) and \( v \) is the ancestor of \( w \) with \( |v| = h \), then the only result of execution of \( I \) at \( C \) is \( C' = \langle p, T', w \rangle \), where \( T' \) is like \( T \), except that in \( T' \) the register \( R \) at node \( v \) is set to 1. That is, \( T' = T[v \mapsto T(v)(R) \mapsto 1] \). Observe that it does not matter whether \( T(v)(R) = 1 \) or \( T(v)(R) = 0 \).

The last case is \( I = "q : new i" \) with \( i \neq k \). The result of execution of \( I \) at \( C \) is unique and has the form \( C' = \langle q_{i+1}^0, T', w' \rangle \), where \( T' \) is obtained from \( T \) by adding a new successor node \( w' \) of \( w \), with \( T'(w') = \emptyset \). (The apple goes to the new node and the machine enters the appropriate initial state.)

The semantics of Eas is defined in terms of eventually accepting IDs. We say that an existential ID is eventually accepting when at least one of its successors is eventually accepting. Dually, a universal ID is eventually accepting when all its successors are eventually accepting. Finally we say that an automaton is eventually accepting when its initial ID is eventually accepting.

Note that a universal ID with no successors is eventually accepting. By our definition this may only happen when no instruction is available in the appropriate universal state; such states may therefore be called accepting states.

A computation of an Ea, an alternating machine, should be imagined in the form of a tree of IDs. Every existential node represents a non-deterministic choice and has at most one child. Every universal node has as many children as there are successor IDs. (In other words, a computation represents a strategy in a game.) Such a computation is accepting if every branch ends in a universal leaf.

### 3.1.1 Restricted computation

We say that an ID \( \langle q, T, w \rangle \) of an Eden automaton is \( n \)-restricted when it satisfies the following condition:
Every node \( w \) of \( T \) which is at level \( k > 0 \) has at most \( \exp_k(n) \) children.

A computation of an Eden automaton is \( n \)-restricted when all its IDs are \( n \)-restricted.

### 3.2 The encoding

The goal of this section is to encode an Ea with a formula of positive first order predicate logic in such a way that the automaton has accepting \( n \)-restricted computation if and only if the formula is provable with an \( n \)-restricted long normal proof. Given an automaton \( A = (k, m, M, Q, q^0, \mathcal{J}) \), our formula uses unary predicate symbols \( q \) and \( R \), for all \( q \in Q \) and \( R \in M \). Each individual variable is of the form \( x_i \) or \( x_i^w \), where \( i \in k \) and \( w \) is a node in some tree of knowledge. For a root node \( \varepsilon \), we identify \( x_0^\varepsilon \) with \( x_0 \). Notation: If \( S \) is a set of formulas \( \{\alpha_1, \ldots, \alpha_k\} \) then \( S \rightarrow \beta \) abbreviates \( \alpha_1 \rightarrow \cdots \rightarrow \alpha_k \rightarrow \beta \). Similarly \( \lambda X^S.M \) abbreviates \( \lambda X_1^{\alpha_1} \cdots X_k^{\alpha_k}.M \).

Convention: Without loss of generality we can assume that for every \( i < k \) there is only one state \( q \in Q_i \) such that the instruction \( q : \text{new} \) belongs to \( \mathcal{J} \). Indeed, otherwise we can modify the automaton by adding designated “transfer states” \( q_i^* \) to \( Q_i \) and adding \( q : \text{jmp} q_i^* \) to \( \mathcal{J} \) when necessary.

#### 3.2.1 Encoding instructions

For every \( i \in k \), we define a set of formulas \( S_i \). With one exception (downward moves), formulas in \( S_i \) represent instructions available in states \( q \in Q_i \). The definition is by backward induction with respect to \( i \).

**Universal states:** Let \( q \in Q_i^p \), and let \( \text{“} q : \text{jmp} \ p_1 \text{”, \ldots, “} q : \text{jmp} \ p_r \text{”} \) be all the instructions available in \( q \). Then the following formula belongs to \( S_i \):

\[
p_1(x_i) \rightarrow \cdots \rightarrow p_r(x_i) \rightarrow q(x_i).
\]

**Existential states (downward moves):** For every instruction of the form \( \text{“} \text{jmp} \ p \text{”} \), where \( q \in Q_{i-1} \) and \( p \in Q_i \), the following formula belongs to \( S_i \):

\[
p(x_i) \rightarrow q(x_{i-1}).
\]

In this case the instruction is executed at depth \( i - 1 \), but the formula is in \( S_i \).

**Existential states (other moves):** Let now \( q \in Q_i^p \). For each of the following instructions available in \( q \), there is one formula in \( S_i \):

- For \( \text{“} q : \text{jmp} \ p \text{”} \), where \( p \in Q_j \) and \( j \in \{i, i - 1\} \), the formula is \( p(x_j) \rightarrow q(x_i) \).
- For \( \text{“} q : \text{check} \ R(h) \text{ jmp} \ p \text{”} \), the formula is \( p(x_i) \rightarrow R(x_h) \rightarrow q(x_i) \).
- For \( q : \text{set} \ R(h) \text{ jmp} \ p \), the formula is \( (R(x_h) \rightarrow p(x_i)) \rightarrow q(x_i) \).
- For \( q : \text{new} \text{”, the formula is } \forall x_{i+1}(S_{i+1} \rightarrow q_{i+1}^0(x_{i+1})) \rightarrow q(x_i) \).

The set of formulas \( S_i \) contains only one copy of \( S_{i+1} \) (state \( q_{i+1}^0 \) is fixed and by our convention so is \( q \)), whence the size of \( S_0 \) is polynomial in the size of \( A \). It is also worth pointing out that the rank of all the above formulas is zero, with the exception of the formula for \( \text{“} q : \text{new} \text{”, the rank of the latter is } k - i \) when \( q \in Q_i \) (note that \( i < k \)).

Note that the number of nested occurrences of a variable \( Z \) of type \( \forall x_{i+1}(S_{i+1} \rightarrow q_{i+1}^0(x_{i+1})) \rightarrow q(x_i) \) exactly corresponds to the number of different eigenvariables induced by the quantifier \( \forall x_{i+1} \). Indeed, the variable \( Z \) occurs in a long normal form in contexts of the form: \( \ldots Z(\lambda x_{i+1} \ldots Z(\lambda x_{i+1}^0 \ldots Z(\lambda x_{i+1}^p.M) \ldots) \ldots) \ldots \) and all the individual variables \( x_{i+1}, x_{i+1}^0, x_{i+1}^p, \ldots \) may be free inside \( M \).
3.2.2 Encoding IDs:

Let now $S$ be a set of formulas and let $w$ be a node in a tree of knowledge with $i$ as its depth. For every $j \leq i$, replace all occurrences of $x_j$ in $S$ by $x'_j$, where $v$ is an ancestor of $w$ of depth $j$. The result is denoted by $S[w]$, and is formally defined by induction with respect to $|w|:

$$S[w] = \begin{cases} S, & \text{if } w = \varepsilon; \\ S[v][x_{|w|}] := x'_w, & \text{if } w \text{ is a child of } v. \end{cases}$$

For a given tree of knowledge $T$, we define sets of formulas:

$$\Gamma_T^R = \{ R(x^w_i) \mid w \in T \land |w| = i \land T(w)(R) = 1 \};$$
$$\Gamma_T^F = \bigcup \{ S_i[w] \mid w \in T \land |w| = i \};$$
$$\Gamma_T = \Gamma_T^R \cup \Gamma_T^S$$

where $S_i$ is as defined above. Note that $FV(\Gamma_T) = \{ x^w_i \mid w \in T \land |w| = i \}$.

The following lemma reduces the halting problem for $n$-restricted computations of $\mathsf{Eden}$ to $n$-restricted provability of positive formulas. In order to state it in a form permitting a proof by induction we need to refine the definition of an $n$-restricted proof to take care of free assumptions. This is done with the following notion of a proof that respects a tree of knowledge.

An environment of the form $\Gamma_T$ contains for every $w \in T$ a declaration

$$Z_w : \forall x_{i+1}(S_{i+1}[w] \rightarrow q^0_{i+1}(x_{i+1})) \rightarrow q(x^w_i).$$

Now for every child $v$ of $w$ there is a variable $x^v_{i+1}$ in $FV(\Gamma_T)$. These eigenvariables should be thought of as reducing the limit of nested occurrences of $Z_w$ in proofs defined in $\Gamma_T$. Let $c_w$ be the number of children of $w$ in $T$. We say that a proof $\Gamma_T \vdash M : q(x_i)$ respects tree $T$ if $b(Z_w, M) \leq \exp_{k-i}(m) - c_w$, for every node $w$ at depth $i$ (that is, at level $k - i$).

**Lemma 7.** Let $\mathcal{A}$ be an Eden automaton. An ID of $\mathcal{A}$ of the form $(q, T, w)$ is eventually accepting by an $n$-restricted computation if and only if the positive judgement $\Gamma_T \vdash q(x^w_i)$ has an $n$-restricted long normal proof that respects $T$.

In particular, the automaton $\mathcal{A}$ is eventually accepting if and only if $\vdash \Gamma_T^F \rightarrow q^0_0(x_0)$, where $T_0$ is the initial tree of knowledge, has an $n$-restricted long normal proof.

**Proof.** ($\Rightarrow$) Let $\mathcal{A}$ be an Eden automaton and let $(q, T, w)$ be an ID of $\mathcal{A}$ that is eventually accepting using an $n$-restricted computation. We will show that $\Gamma_T \vdash q(x^w_i)$, where $i = |w|$, has an $n$-restricted proof that respects $T$. We proceed by induction with respect to the definition of eventually accepting IDs.

If $q$ is a universal state and $(q, T, w)$ is eventually accepting then all successors of $(q, T, w)$ are eventually accepting. Every successor ID corresponds to some instruction “$q : \text{ jmp } p_j$” for $j = 1, \ldots, k$. By the induction hypothesis we have $\Gamma_T \vdash p_j(x^w_i)$ for $j = 1, \ldots, k$.

By the definition of $\Gamma_T$, the formula $p_1(x_i) \rightarrow \cdots \rightarrow p_k(x_i) \rightarrow q(x_i)$ belongs to $S_i$ and $p_1(x^w_i) \rightarrow \cdots \rightarrow p_k(x^w_i) \rightarrow q(x^w_i)$ belongs to $S_i[w]$. Since $S_i[w] \subseteq \Gamma_T$ it follows that $\Gamma_T \vdash q(x^w_i)$.

If $q$ is an existential state and $(q, T, w)$ is eventually accepting then there exists a successor of $(q, T, w)$ which is eventually accepting. This successor $(p, T', w')$ is a result of execution of an instruction $I$ of $\mathcal{A}$, applicable in state $q$. We check the possible forms of $I$. 

A.Schubert and P.Urzyczyn and D.Walukiewicz-Chrząszcz 11
If $I$ is “$q : \text{jmp } p$”, where $q \in Q_i$, $p \in Q_j$, one has $T' = T$ and either $w = w'$ or $w'$ is an immediate predecessor or successor of $w$ in $T$. By the induction hypothesis we have $\Gamma_T \vdash p(x_i^{w'})$. Since $\Gamma_T$ contains the formula $p(x_i^{w'}) \rightarrow q(x_i^{w'})$, we conclude that $\Gamma_T \vdash q(x_i^{w'})$.

For “$q : \text{check } R(j) \text{ jmp } p$”, where $p, q \in Q_i$, let $v$ be the ancestor of $w$ in $T$ such that $|v| = j$. One has $T' = T$, $w' = w$ and the register $R$ at $v$ is set to 1 (since otherwise this instruction cannot be executed). By the induction hypothesis, $\Gamma_T \vdash p(x_i^{w'})$. Since $\Gamma_T$ contains the formula $p(x_i^{w'}) \rightarrow R(x_j) \rightarrow q(x_i^{w'})$ and the atom $R(x_j)$, we conclude that $\Gamma_T \vdash q(x_i^{w'})$.

For “$q : \text{set } R(j) \text{ jmp } p$”, where $p, q \in Q_i$, let $v$ be the ancestor of $w$ in $T$ such that $|v| = j$. One has $w' = w$ and $T' = T[v \mapsto T(v)[R \mapsto 1]]$. By the induction hypothesis we have $\Gamma_T \vdash p(x_i^{w'})$. Note that $\Gamma_T = \Gamma_T \cup R(x_j)$, and consequently $\Gamma_T \vdash R(x_j) \rightarrow p(x_i^{w'})$. Since $\Gamma_T$ contains the formula $p(x_i^{w'}) \rightarrow R(x_j) \rightarrow q(x_i^{w'})$ and the atom $R(x_j)$, we conclude that $\Gamma_T \vdash q(x_i^{w'})$.

In all the above cases, the assumptions used in the appropriate proof steps are formulas of rank $rk$ equal to zero. Therefore it follows immediately from the induction hypothesis that the obtained proofs are $n$-restricted and respect $T$. Only the last case involves quantification.

For “$q : \text{new } w$”, where $q \in Q_i$, $p = q_{i+1}^0$, the tree $T'$ is obtained from $T$ by adding a brand new child $w'$ of $w$ labelled $0^0$. From the induction hypothesis we know that $\Gamma_T \vdash M : q_{i+1}^0(x_i^{w'})$ where $M$ respects $T'$. Note that $\Gamma_T = \Gamma_T \cup S_{i+1}[w]$; consequently we may deduce that $\Gamma_T \vdash \lambda X S_{i+1}[w].M : S_{i+1}[w] \rightarrow q_{i+1}^0(x_i^{w'})$. The variable $x_i^{w'}$ does not appear in $\Gamma_T$, hence we also have

$$\Gamma_T \vdash \lambda x_{i+1}.\lambda X S_{i+1}[w].M : \forall x_{i+1}(S_{i+1}[w][x_i^{w'} := x_{i+1}] \rightarrow q_{i+1}^0(x_{i+1})).$$

Since $S_{i+1}[w][x_i^{w'} := x_{i+1}] = S_{i+1}[w]$ and $\Gamma_T$ contains the declaration

$$Z_w : \forall x_{i+1}(S_{i+1}[w] \rightarrow q_{i+1}^0(x_{i+1})) \rightarrow q_i(x_i^{w'})$$

we conclude that $\Gamma_T \vdash Z_w(\lambda x_{i+1}.\lambda X S_{i+1}[w].M) : q_i(x_i^{w'})$. This amounts to a single application of the proof variable $Z_w$. Since the number of children in $T'$ is smaller by one than the number in $T'$ and $M$ respects $T'$, the application respects the bound on the number $b(Z_w,Z_w(\lambda x_{i+1}.\lambda X S_{i+1}[w].M))$.

$(\Rightarrow)$ Suppose that $\langle q, T, w \rangle$ is an ID of an automaton $A$ such that $\Gamma_T \vdash N : q_i(x_i^{w'})$, where $i = |w|$ and where $N$ is an $n$-restricted long normal form that respects $T$. We show, by induction with respect to $N$, that $\langle q, T, w \rangle$ is eventually accepting and that the accepting computation is $n$-restricted. (In all cases but the last one, the second claim is obvious.) Since $q_i(x_i^{w'})$ is an atom, it must be the case that $N = XN_1 \ldots N_r$, for some $X$ and some long normal forms $N_1, \ldots, N_r$. In addition, we must have a declaration $(X : \varphi) \in \Gamma_T$, where $\varphi = \tau_1 \rightarrow \cdots \rightarrow \tau_r \rightarrow q_i(x_i^{w'})$, and $\Gamma_T \vdash N_1 : \tau_1$, for each $l$.

Let $q \in Q_{i+1}$. Let “$q : \text{jmp } p_j$”, for $j = 1, \ldots, s$, be all instructions available for $q$. By the definition of $\Gamma_T$, there is only one formula $\varphi$ that ends with the atom $q_i(x_i^{w'})$, namely $\varphi = p_{1}(x_i^{w'}) \rightarrow \cdots \rightarrow p_{s}(x_i^{w'}) \rightarrow q_i(x_i^{w'})$. Therefore, $r = s$ and for every $j = 1, \ldots, r$, we have $\Gamma_T \vdash N_j : p_j(x_i^{w'})$. By the induction hypothesis we know that the configurations $\langle p_j, T, w \rangle$ are eventually accepting. Since a universal ID is eventually accepting when all its successors are eventually accepting, we get the desired conclusion.

Let $q \in Q_i$. Since the formula $\varphi$ ends with $q_i(x_i^{w'})$, it must correspond to some instruction $I$ that is available in state $q$. We need to show that $I$ can be executed and that a result of execution of $I$ is eventually accepting. This will imply that also $\langle q, T, w \rangle$ is eventually accepting.

If $\varphi$ has the form $p(x_i^{w'}) \rightarrow q(x_i^{w'})$ for some variable $x_i^{w'}$ then $I$ is “$q : \text{jmp } p$”. Note that such a $\varphi$ may occur in $\Gamma_T$ only when $w'$ is a node of $T$, more precisely, node $w'$ is either $w$ or
it is an immediate predecessor or successor of \( w \) in \( T \). By the induction hypothesis applied to \( \Gamma_T \vdash N_1 : p(x^w_j) \), we conclude that \( \langle p,T,w \rangle \) is eventually accepting.

If \( \varphi \) is \( p(x^w_i) \rightarrow R(x^v_i) \rightarrow q(x^w_i) \) then \( I \) is "qu : check \( R(j) \) jmp \( p \)". We need to show that \( I \) can be executed, i.e., that \( T(v)(R) = 1 \) where \( v \) is the ancestor of \( w \) in \( T \) with \( |v| = j \). We know that \( \Gamma_T \vdash N_1 : p(x^w_i) \) and \( \Gamma_T \vdash N_2 : R(x^v_i) \). The only formula in \( \Gamma_T \) of the form \( \alpha_1 \rightarrow \cdots \rightarrow \alpha_k \rightarrow R(x^v_i) \) is \( R(x^v_i) \). By the definition of \( \Gamma_T \), if \( R(x^v_i) \in \Gamma_T \) then \( T(v)(R) = 1 \). Hence \( I \) can be executed. Since \( \Gamma_T \vdash N_1 : p(x^w_i) \), by the induction hypothesis, \( \langle p,T,w \rangle \) (the result of execution of \( I \) at \( \langle p,T,w \rangle \) is eventually accepting.

If \( \varphi \) is \( ( R(x^v_i) \rightarrow p(x^w_i) ) \rightarrow q(x^w_i) \) then \( I \) is "qu : set \( R(j) \) jmp \( p \)" and \( i < k \). The result of execution of \( I \) at \( \langle p,T,w \rangle \) is \( \langle p,T',w \rangle \), where \( T' = T[v \mapsto T[v][R \mapsto 1]] \). We know that \( \Gamma_T \vdash N_1 : R(x^v_i) \rightarrow p(x^w_i) \). Since \( N_1 \) is an inf, there exists \( N'_1 \) such that \( \Gamma_T,Y : R(x^v_i) \vdash N'_1 : p(x^w_i) \). By the induction hypothesis, \( \langle p,T',w \rangle \) is eventually accepting.

The last case is when \( \varphi = \forall \exists \bigwedge \Gamma(w) \Rightarrow q(x^w_i) \) is the type of \( \alpha \) and the instruction \( I \) is "qu : new". We have \( \Gamma_T \vdash Z_w \). For some inf \( N'_1 \), Substituting \( x^w_i \) for \( x^i \) we obtain the type assignment

\[
\Gamma_T,Y : S_i[w][x_i := x^w_i] \vdash N'_1[x_i := x^w_i] : q_0(T_i). \]

Note that \( S_i[w][x_i := x^w_i] \) equals \( S_i[w] \) and \( \Gamma_T,Y : S_i[w'] = \Gamma_T \), where \( T' \) is obtained from \( T \) by adding a new child \( w' \) of \( w \) labelled \( 0 \). The term \( N'_1[x_i := x^w_i] \) is \( n \)-restricted and respects \( T' \) because the top occurrence of \( Z_w \) was eliminated. Hence, by the induction hypothesis, the result \( \langle q_0(T_i),T',w' \rangle \) of execution of \( I \) is eventually accepting.

\section{ Eden programming}

We begin with a few examples demonstrating how Eden automata can be used to solve computational tasks. They present some techniques exploited in the hardness proof to follow and introduce the reader to the "pseudo-code" we use.

The access to knowledge in an Eden automaton is restricted in that it precludes the possibility to verify that a given bit is 0. This can be partly overcome by a simple trick: use two bits to encode one, 10 for 0 and 01 for 1. This works as long as one can ensure that the two bits are never set both to 1.

\begin{example}
To be more specific, if we fix 6 registers \( L_1, R_1, L_2, R_2, L_3, R_3 \) then any word of length 3 can be represented by a snake where exactly one register in each pair \( L_i, R_i \) is set to 1. For example, 101 is encoded by \( R_1 = L_2 = R_3 = 1 \) and \( L_1 = R_2 = L_3 = 0 \).

Consider an automaton \( A \) of depth 1, with \( q_0^0 = q_0, q_1^0 = q_1 \), and with the instructions (where \( q_0^0 \in Q_1^6 \), and other states are in \( Q^6_1 \)):

\[
q_0 : \text{new}; \quad q_1 : \text{jmp } q_1^1; \quad q_2 : \text{jmp } q_2^1; \quad q_3 : \text{jmp } q_3^1; \\
q_1^1 : \text{jmp } q_1^2; \quad q_2^1 : \text{jmp } q_2^2; \quad q_3^1 : \text{jmp } q_3^2; \\
q_1^2 : \text{set } L_1(1) \text{ jmp } q_2; \quad q_2^2 : \text{set } L_2(1) \text{ jmp } q_3; \quad q_3^2 : \text{set } L_3(1) \text{ jmp } q_4; \\
q_1^3 : \text{set } R_1(1) \text{ jmp } q_2; \quad q_2^3 : \text{set } R_2(1) \text{ jmp } q_3; \quad q_3^3 : \text{set } R_3(1) \text{ jmp } q_4.
\]

The automaton \( A \) starts in the initial ID in state \( q_0 \) with a root-only tree of knowledge. It creates an additional node \( d \), a successor of the root, and enters state \( q_1 \) at node \( d \). The procedure from state \( q_1 \) to state \( q_4 \) constitutes a for loop, informally written as follows:

\[
q_1 : \text{for } i = 1 \text{ to } 3 \text{ do } [\text{set } L_i \text{ OR set } R_i]; \text{ goto } q_4.
\]
The computation of our automaton has one branch, that ends in an ID where the only child of the root represents a non-deterministically generated word of length 3. The apple is at the child node and the machine is in state $q_4$.

We can now compose the automaton with another one $A'$ that runs after $A$. It has a number of states (see below) with $q_1', q_2', q_3', q_{acc} \in Q_1'$ and other states in $Q_1^3$:

- $q_1' : \text{jmp } q_1^1$;
- $q_1' : \text{jmp } q_1^\text{chk}$;
- $q_1' : \text{jmp } q_1^\text{cnt}$;
- $q_1^\text{chk} : \text{check } L_1(1) \text{ jmp } q_{acc}$;
- $q_1^\text{chk} : \text{check } L_2(1) \text{ jmp } q_{acc}$;
- $q_1^\text{chk} : \text{check } L_3(1) \text{ jmp } q_{acc}$;
- $q_1^\text{cnt} : \text{jmp } q_2'$;
- $q_1^\text{cnt} : \text{jmp } q_3'$;
- $q_1^\text{cnt} : \text{jmp } q_5$.

The automaton $A'$ is initiated in its starting ID in state $q_4$ with one node $d$, a successor of the root. The node has set one register in each of the pairs $L_1, R_1; L_2, R_2; L_3, R_3$. The automaton then enters state $q_1'$ at node $d$. The procedure from state $q_1'$ to state $q_4$ constitutes a universal for loop, informally written as follows:

$$q_1 : \text{for } i = 1 \text{ to } 3 \text{ do [check } L_i \text{ AND continue]; goto } q_5.$$  

In successful circumstances, the computation has 4 branches. Three of them end in an accepting ID in the state $q_{acc}$ and the fourth one ends in an ID where all bit markers are set as $L_1, L_2, L_3$ and encode the sequence of bits 000. The apple is at the child node and the machine is in state $q_0$.

These two automata may be viewed as procedures in a single program. The first procedure is realised by the automaton that generates non-deterministically a string of three bits and the second checks if all the bits are equal to 0.

The automaton $A'$ in Example 8 is purely universal; the “existential” states are actually deterministic because their instructions provide no choice.

When $q_5$ is reached the machine is in an ID that satisfies a specific property (in this case: all bits are 0). Actually, only a slight modification of the automaton makes it possible to check that the bits form a word from an arbitrary regular language.

### 4.1 Procedures

The main goal of this section is to show that in trees of knowledge we can faithfully represent numbers the binary representation of which can be realised in $\exp_k(n)$ bits for each natural $k > 0$ and $n > 2$. For this to work we need to limit computations to $n$-restricted ones. The trees and automata we consider here have dimension $2n + 9$. We think of the snakes as containing the following parts:

- **a base segment** consisting of $2n$ registers $L_0, R_0, \ldots, L_{n-1}, R_{n-1}$;
- **data registers**: $A_0, A_1$;
- **global registers**: $\text{Ready}, \text{Steady}$;
- **local registers**: $\text{New}, \text{Old}, \text{Done}_1, \text{Done}_2, \text{Done}_C$.

The base segment is capable to encode a binary word of length $n$, using the “two for one” trick, as demonstrated in Example 8. The data registers may contain a binary value of a node in a similar way: register $A_i$ set to 1 represents $i$ for $i = 0, 1$.

We identify binary words of length $\exp_k(n)$ with numbers from 0 to $\exp_{k+1}(n) - 1$, and we use trees of uniform depth $k$ to encode such words-numbers. (The encoding relation is not a function, i.e., one number is encoded by many trees.) Informally, the idea is as follows:

a word $a_0a_1 \ldots a_{r-1}$ of length $r$ can be represented as the set of pairs $\{(0, a_0), (1, a_1), \ldots, (r-$
induction with respect to the control should be passed to another subroutine. For every

4.1.1.1 Making a new word

4.1.1 Inductive hypotheses

and formulate the appropriate induction hypothesis. A successful computation of \( M_k \) ever uses (jumps, writes to or reads from registers at) any proper ancestor of \( d \).

A node \( d \) in a tree is said to encode a word when the subtree rooted at \( d \) encodes that word.

Remark. One can easily generalise the above definition to words over any \( l \)-element finite alphabet \( \Sigma = \{a_0, a_1, \ldots, a_{l-1}\} \) and trees of dimension \( 2n + l + 7 \), using data registers \( A_0, \ldots, A_{l-1} \) to represent symbols \( a_0, a_1, \ldots, a_{l-1} \).

We now show how Eden automata can manipulate binary words. The automata defined in this section should more adequately be called “procedures” as they are used as subroutines in our main construction. Each procedure is initiated at some specific start IDs which are expected to satisfy certain conditions.

We say that a computation initiated in a start ID is called a successful computation of a procedure if every its branch either ends in an accepting state, or in an end state where the control should be passed to another subroutine.

For every \( k \) and every \( l > k \) we define procedures \( M_k, E_k^l, S_k, C_k^0, \) and \( C_k^1 \), by simultaneous induction with respect to \( k \). For each of these procedures we first define start and end IDs and formulate the appropriate induction hypothesis.

4.1.1 Inductive hypotheses

4.1.1.1 Making a new word

For every \( k \geq 0 \) we define a procedure \( M_k \) to make new words.

Start ID: The current apple is a leaf \( d \) of the tree of knowledge, the snake at \( d \) is empty.

Induction hypothesis:

1. No computation of \( M_k \) ever uses (jumps, writes to or reads from registers at) any proper ancestor of \( d \).
2. A successful computation of \( M_k \) has only one end ID. At the end ID the apple is back at \( d \), but \( d \) is now a root of a subtree of uniform depth \( k \) and \( d \) encodes a non-deterministically chosen word \( w \) of length \( \exp_k(n) \).

The case 1 in the induction hypothesis is a separation condition which states that the procedure \( M_k \) does not have side effects. This is necessary since the procedure has end states, and computations continues after these are reached. In general there may be many occurrences of the end state in a computation tree for a particular run of a procedure. However, the computations of procedures we consider in this paper have this particular
property that once a procedure is initiated it reaches the end state at most one in its computation tree.

For the other subroutines we define no end states and no similar separation conditions; their only purpose is to accept.

4.1.1.2 Constant

Procedures $C^x_k$, where $x \in \{0, 1\}$, check that a given address is a constant.

Start ID: The apple is at node $d$ of level $k$, and $d$ encodes a binary word $w$ of multiexponential length $\exp_k(n)$. Local registers below node $d$ are empty.

Induction hypothesis: Let $C^0_k$ (resp. $C^1_k$) be initiated in a start ID. $C^0_k$ (resp. $C^1_k$) accepts iff the address of $d$ is $\vec{0}$ (resp. $\vec{1}$).

4.1.1.3 Equality

Procedure $E^l_k$, where $l > k$, verifies equality of two binary words.

Start ID: A start IDs of $E^l_k$ has the apple at node $d$, a root of a subtree of uniform depth $l$. At level $k$ there is exactly one descendant $e_O$ of $d$ satisfying $T(e_O)(\text{Old}) = 1$ and exactly one descendant $e_N$ satisfying $T(e_N)(\text{New}) = 1$. (There may be other nodes at level $k$ as well, and it may happen that $e_O = e_N$.) All local registers at level $k - 1$ and below are empty. Subtrees rooted at $e_O$ and $e_N$ encode binary words of length $\exp_k(n)$.

Induction hypothesis: Let $E^l_k$ be initiated in a start ID. $E^l_k$ accepts iff the addresses of $e_O$ and $e_N$ are the same.

4.1.1.4 Successor

Binary words are identified with numbers so that the successor relation holds between strings of the form $w011 \ldots 1$ and $w100 \ldots 0$. Procedure $S_k$ verifies this relation.

Start ID: The same as start IDs of $E^{k+1}_k$.

Induction hypothesis: Let $S_k$ be initiated in a start ID. $S_k$ accepts iff the address of $e_N$ is the successor of the address of $e_O$.

4.1.2 Realisation of the Procedures

4.1.2.1 Procedure $C^0_k$

We define our automata by mutual induction with respect to $k$. We begin with the relatively simple definition of $C^0_k$, written in informal pseudo-code. For $k = 0$, the definition of $C^0_k$ is a straightforward generalisation of the code of $A'$ in Example 8:

\[
\text{for } i = 1 \text{ to } n \text{ do } [\text{check } L_i \text{ AND continue}] \text{; jmp accept.}
\]

For $k > 0$, we assume that $C^0_{k-1}$, $C^1_{k-1}$, $S_{k-1}$ have already been defined, and we construct $C^0_k$, so that it executes the following algorithm. The almost identical definition of $C^1_k$ is omitted.

1. Descend to a child; goto 2 AND goto 3;
2. Run $C^0_{k-1}$ (accepting inside);
3. Check data register $A_0$; set register $Done_C$;
4. goto 5 OR goto 12;
5. Go up;
6. Descend to a child;
7. goto 8 AND goto 3;
8. Set register \textit{New}; go up to \textit{d};
9. Descend to a child;
10. Check register \textit{Done} \textsubscript{C}; set register \textit{Old}; go up;
11. Run \textit{S}_{k-1} (accepting inside);
12. Run \textit{C}_{k-1} \textsubscript{1} (accepting inside).

Before we describe the operation of the code let us remark that although the above description of the automaton is highly informal, it is implementable as an actual automaton, using a constant number of internal states.

First, let us make an informal account of the way the procedure operates. When \textit{C} \textsubscript{k} \textit{0} is initiated in a start ID at a node \textit{d} at level \( k \), it attempts to verify that data register \( A_0 \) is set to 1 at every address. It begins with a child with address 0, guessing it non-deterministically. At this point the computation splits into two branches. One branch verifies the correctness of the guess by running \textit{C} \textsubscript{k-1} \textsubscript{0} (and accepts if the verification is successful). Along the other branch we first check that \( A_0 \) is indeed set to 1, mark the present node as \textit{Done}, and then proceed to another child of \( d \) (step 6). The main loop in steps 3–7 should now be taken for every address in the increasing order. Each time the body of the loop is executed, the machine verifies that the address of the current apple is a successor of another address where register \textit{Done} is already set to 1 (i.e., the node was processed). This is done with help of another universal split in step 7. A separate branch of computation is activated. Within that branch, the present node \( e \) is marked as \textit{New}, then another child \( e' \) of \( d \) is selected and marked as \textit{Old}. But first we check register \textit{Done} at node \( e' \) to make sure that \( e' \) has already been processed.\footnote{It may happen that \( e' = e \) but in this case the successor test will fail.} It remains to run \textit{S}_{k-1} from node \( d \) to complete the verification branch (steps 8–11).

The main loop continues until we non-deterministically guess that we reached a node with address \( \vec{i} \). This is verified by initiating \textit{C} \textsubscript{k-1} \textsubscript{0}, and then the procedure accepts.

We can now turn to a proof of the induction hypothesis. \((\Rightarrow)\) Observe that in case the address of \( d \) encodes the word \( w = \vec{0} \) and \textit{C} \textsubscript{k} \textsubscript{0} is run from a correct start ID then the procedure may choose to take the child of \( d \) with address \( \vec{0} \) in step 1 and further all children in successor sequence in step 6. With this strategy of choice, the IDs in which the procedures \textit{C} \textsubscript{k-1} \textsubscript{0}, \textit{S}_{k-1}, and \textit{C}_{k-1} \textsubscript{1} are initiated are their correct start IDs. In particular, local registers at levels \( k - 2 \) and below are empty and can be safely used by each procedure. Note that every branch of computation uses its own private copy of these registers. This way alternation helps to avoid the limitations of our non-erasable memory. Assuming the induction hypothesis about \textit{C} \textsubscript{k-1} \textsubscript{0}, \textit{S}_{k-1}, \textit{C}_{k-1} \textsubscript{1}, we can prove by induction over the number \( i \) of bits in \( w \) represented by children of \( d \) marked as \textit{Done} in step 4 that all these children represent bit 0 (i.e., \( A_0 \) is set to 1). We also prove that for this strategy of choice each time the control of the automaton is in step 4 the number of such nodes increases. In conclusion we obtain that the procedure \textit{C} \textsubscript{k} \textsubscript{0} accepts.

\((\Rightarrow)\) Suppose now that \textit{C} \textsubscript{k} \textsubscript{0} initiated in a start ID accepts. Let \( l \) be the number of times the procedure enters step 4. Let \( D_i \) be the set of children of \( d \) marked as \textit{Done} at \( i \)-th entry to the step 4 of \textit{C} \textsubscript{k} \textsubscript{0}. Let \( a_i \) be the maximal address encoded by elements of \( D_i \). By simple downwards induction over \( l - i \) we show the following statement

\begin{quote}
For each accepting computation subtree of \textit{C} \textsubscript{k} \textsubscript{0} started at \( i \)-th entry to the step 4 and for each addresses \( a \) such that \( a_i < a < \exp_k(n) \), the node \( d \) has a child that encodes \( a \) and that has \( A_0 \) set to 1.
\end{quote}
Indeed for $i = l$ the set of addresses $a$ such that $a_i < a < \exp_k(n)$ is empty so the conclusion follows. In case $i < l$ the accepting computation must enter the loop and mark one child of $d$ with DoneC and then come back to the step 4. We have two subcases here depending on the relation between the elements $a_i$ and $a_{i+1}$. In case $a_i = a_{i+1}$ we observe that no node of the tree of knowledge could change in this turn of the loop (DoneC is only overwritten with the same value) so the conclusion follows by the induction hypothesis. In case $a_i \neq a_{i+1}$, there is $b_i \in D_{i+1} - D_i$. Let $b_i$ be the number encoded by the subtree with $b_i$ as root. In steps 8–11 it is verified that $b_i = a + 1$ for some $a$ encoded by $a \in D_i$, but actually $a$ must be $a_i$ as otherwise $a_i = a_{i+1}$. So we have $a_i < b_i < \exp_k(n)$. The node of $b_i$ has $A_0$ set to 1 as this is verified in step 3. As a result we obtain our conclusion for the address $b_i$. Since all other elements $a'$ such that $a_i < a' < \exp_k(n)$ must satisfy $a_{i+1} < a' < \exp_k(n)$, we obtain the conclusion by the induction hypothesis.

To conclude that acceptance of $C^0_k$ implies that all bits are 0 in the word encoded by $d$ we observe that a successful computation of the procedure requires that at the first entry to the step 4 only one child of $d$ is marked as DoneC, namely the one that encodes the address $\vec{0}$ (steps 1–2) with $A_0$ set (step 3). This is the first entry to the step 4 and further computation accepts so we can apply the proved above statement for $i = 1$ and obtain that $d$ has children that are root of trees that encode addresses $a$ such that $0 < a < \exp_k(n)$ and have all $A_0$ set to 1. As this applies also for the address 0 we immediately obtain that $d$ encodes $\vec{0}$.

Let us remark here that in step 6 the apple may be passed to a child already marked as DoneC, so that the main loop in steps 3–7 may be executed more times then needed and we effectively care about this case in the inductive step of the argument above.

There is one more subtlety here. Since there may be many nodes of the same address, a successful run of $C^0_k$ may verify (mark as DoneC) more nodes than needed. This is again not harmful, because nodes of the same address also have the same value.

### 4.1.2.2 Procedure $M_k$

We can now turn to the more complicated procedure $M_k$. In the induction step we assume that procedures $M_{k-1}$, $C^1_{k-1}$, $S_{k-1}$, $C^0_{k-1}$, and $C^1_{k-1}$ have already been defined and we describe $M_k$ as a pseudo-code “program” consisting of two phases. Recall that the computation begins at the root $d$ of the word to be constructed.

**Phase 1:** At first, procedure $M_k$ runs $M_{k-1}$ in a loop. The number of iterations is non-deterministic, but bounded due to the n-restrictedness condition, as each one creates a new child.

1. Create a new child and descend there;
2. Run $M_{k-1}$;
3. Set register $A_0$ OR set register $A_1$ (an existential choice);
4. go up; goto 1 OR goto 5 (an existential choice);
5. return (enter Phase 2);

Note the subtlety: once a new child is created the computation must commence from a fixed state (for the appropriate depth). Our construction respects this restriction: we perform exactly the same actions for every new child.

The end state of this phase is in step 5 above. An immediate inductive argument (for the loop in steps 1–4) shows that each time the computation reaches step 4:
1. The computation does not use (jumps, writes to or reads from registers at) any proper ancestor of $d$.
2. The apple is back at $d$, but $d$ has a non-empty set $C$ of children with $|C| \leq \exp_k(n)$. Each element of $C$ has either $A_0$ or $A_1$ set and starts a subtree that encodes a number in $\{0, \ldots, \exp_k(n) - 1\}$.

The restriction $|C| \leq \exp_k(n)$ is precisely the result of our $n$-restrictedness condition.

Phase 2: The second phase starts with the apple at node $d$ and goes as follows:

6. Descend to a child; goto 7 (verify) AND goto 8 (continue);
7. Run $C^0_{k-1}$ (accepting inside);
8. Set register Steady;
9. Go up to $d$;
10. Descend to a child; goto 11 (verify) AND goto 14 (continue);
11. Set register New; go up to $d$;
12. Descend to a child; check register Steady; set register Old; go up to $d$;
13. Run procedure $S_{k-1}$ (accepting inside);
14. Set register Steady;
15. goto 9 (continue) OR goto 16 (end of this phase);
16. Run $C^1_{k-1}$ (verify) AND goto 17 (continue);
17. Go up to $d$; return (end state).

In step 6 the computation splits into two branches. One proceeds (fingers crossed) along the main computation branch beginning at step 8. The other branch verifies that the present address is $\vec{0}$ and accepts. The whole computation can therefore accept only if the verification in step 7 was successful. In addition the auxiliary branch uses its own “private copy” of all resources, in particular it can set registers which remain empty for the main computation. Similar universal splits occur in steps 10 and 16. Note that registers Old and New remain intact outside of the subroutine 11–13. At the completion of the above we are again at node $d$.

Again an immediate inductive argument (for the loop in steps 9–15) shows that each time the computation reaches step 15:

1. The computation does not use (jumps, writes to or reads from registers at) any proper ancestor of $d$.
2. The apple is in a child of $d$, but $d$ has a non-empty set $C$ of children with $|C| \leq \exp_k(n)$ such that it contains an element that encodes $0$ and for each $e \in C$ marked as Steady either $e$ encodes $0$ or is a successor of some $e' \in C$.

Phase 2 reaches the end state only when it can verify that address $\vec{1}$ of length $\exp_k(n)$ is encoded by a child of $d$ that is marked with Steady. With $\vec{1}$ marked as Steady and the fact that children marked as Steady are closed on predecessor we obtain that all addresses of length $\exp_k(n)$ must be encoded by children of $d$. This is exactly the condition 2 for $M_k$. The condition 1 also holds by the conditions 1 in the properties of phase 1 and 2.

Remar: Observe that this procedure may be easily adapted to serve as a non-deterministic generator of words of length $\exp_k(n)$ over arbitrary alphabet $\Sigma$. It is enough to extend the registers with $\Sigma$ and to perform existential choice in step 3 of the automaton $M_k$ so that it chooses one of the elements in $\Sigma$ instead of $A_0$ and $A_1$; for inductive cases $k' < k$ the automata should remain the same.
4.1.2.3 Procedure $S_k$

Recall that we begin in a node $d$ which has (among others) exactly one child marked as $\text{Old}$ (i.e., satisfying $\text{Old} = 1$) and exactly one marked as $\text{New}$. Subtrees rooted there are assumed to encode binary words $w_{\text{old}}$ and $w_{\text{new}}$ of length $\exp_k(n)$. We want to verify that $w_{\text{old}} = w011\ldots 1$ and $w_{\text{new}} = w100\ldots 0$, for some $w$. For $k = 0$ this can be done with a simple for loop. For $k > 0$, we process children of $\text{Old}$ in order of increasing addresses. At each step we compare the data bit at the present node with the data bit at a child of $\text{New}$ with the same address. The compared bits should match in phase 1 (we begin with more significant ones) until we non-deterministically discover the point where they begin to differ (phase 2).

We now describe $S_k$ with a little more detail, but still informally, hoping that the colloquial expressions used below are easy to understand. For instance, “to descend to a child of $\text{Old}$” (step 1) means to descend to a child $e$ of $d$, check $e(\text{Old})$, and then go to a child of $e$. The phrase “Universally verify that...” is understood as “Verify that... AND continue”.

(A similar construction was already used in the definitions of $C_0^k$ and $M_k$.) In step 2 this is equivalent to the statement “Run $C_{k-1}^0$ AND goto 3”. Similarly, in steps 7 and 13 the verification branch calls procedure $S_{k-1}$, and steps 4, 9, 14 activate procedure $E_{k+1}$.

1. Descend to a child of $\text{Old}$;
2. Universally verify that the present address consists of only zeros;
3. goto 4 (phase 1) OR goto 9 (phase 2);
4. Universally verify that the data bit at the present node is the same as the data bit of a child of $\text{New}$ of the same address;
5. Mark the present node as $\text{Done}_1$; go up (to the node marked as $\text{Old}$);
6. Descend to a child;
7. Using $S_{k-1}$, universally verify that the present address is the successor of an address of a brother node already marked as $\text{Done}_1$;
8. goto 3;
9. Universally verify that the data bit at the present node is 0, while the data bit of a child of $\text{New}$ of the same address is 1;
10. Mark the present node as $\text{Done}_2$;
11. goto 12 (phase 2) OR goto 16 (end);
12. Go back to $d$; descend to a child of $\text{Old}$;
13. Universally verify that the present address is the successor of an address of a brother node already marked as $\text{Done}_2$;
14. Universally verify that the data bit at the present node is 1, while the data bit of a child of $\text{New}$ of the same address is 0;
15. Mark the present node as $\text{Done}_2$; goto 11;
16. Run $C_{k-1}^1$ (accepting inside).

Assuming that the start ID of $S_k$ is as expected, we can now refer to the induction hypothesis about $C_{k-1}^0$, $S_{k-1}$, and $E_{k-1}$. Indeed, all these procedures are run from their respective start IDs. In particular, local registers below level $k-1$ are available for use in the appropriate branches of computation. It follows that a successful computation of $S_k$ is only possible when the successor relation indeed holds as required.

Full proof of correctness requires now to formulate appropriate induction hypotheses for the two main loops inside this procedure. We encourage readers to formulate them and prove them themselves to better understand the mechanisms of the automata.
4.1.2.4 Procedure $E_k^l$

This procedure works in a similar way as $S_k$ except that only one phase is needed and the distance from $d$ to $Old$ and $New$ may be larger. We skip the details, but we want to remark on one difference between $E_k^l$ and $S_k$. It may happen that either of these procedures is run from an ID where the same node of the tree is marked $Old$ and $New$. This is not an obstacle: procedure $E_k^l$ will accept in this case while $S_k$ will not.

The formulation of the code for this procedure should now be within the readers’ reach so we encourage them to reconstruct it themselves for better understanding of the paper.

4.2 Simulation of a Turing Machine

The techniques introduced in Section 4.1 can be used to simulate a Turing Machine. Consider a deterministic Turing Machine $T$ working in $k$-EXPTIME, and fix an input word $x$ of length $n$. Without loss of generality\(^3\), we can assume that the machine works exactly in time $\sqrt{\exp_k(n)}$. Let $\Sigma = \Sigma_0 \cup (\Sigma_0 \times \Delta)$ where $\Sigma_0$ is the tape alphabet and $\Delta$ is the set of states of $T$. We already know (see the Remark on page 19) how to encode words over $\Sigma$ using trees of knowledge.

We use a triple $\langle t, a, s \rangle$ to express that the contents of the tape cell $a$ at time $t$ is $s$. Here, $s \in \Sigma$ is either a tape symbol of $T$ or a tape symbol plus an internal state (in case $T$ at time $t$ is at position $a$). A computation of $T$ is represented by a unique set of triples with only one $\langle t, a, s \rangle$ for every $t, a$. By our assumption, this set has exactly $exp_k(n)$ elements. In particular, the pair of numbers $t, a$ can be written in binary in space $\exp_{k-1}(n)$. Thus a triple $\langle t, a, s \rangle$ can be seen as a pair $\langle m, s \rangle$ where $m < exp_k(n)$ is a single number. The whole computation may therefore be identified with a word over $\Sigma$ of length $exp_k(n)$. This word may now be encoded, as in Section 4.1, by a tree of knowledge of depth $k$ and an appropriate width (extra data registers are needed to account for all elements of $\Sigma$). In this way we can represent a computation of $T$ in the memory of an Eden automaton.

A slight adjustment of the automaton $M_k$ of Section 4.1 (in step 3) yields a procedure to generate an arbitrary word over $\Sigma$ of length $\exp_k(n)$.

4.2.1 Procedure $N_k$

The definition of $N_k$ is similar to that of $M_k$, but now we have to only generate words representing accepting computations of $T$. Therefore, $N_k$ works in the following two phases:

1. It generates a sequence of triples.
2. It verifies that the set of triples represents a computation of $T$.

The phase 1 can be realised by a procedure similar to the phase 1 of $M_k$ (see the Remark on page 19). The phase 2 is more complicated, but it can similarly be related to the phase 2 of $M_k$. The steps 11–13 should be replaced with a longer verification routine. There are two subgoals of the routine:

1. To verify that the generated sequence of triples contains a representation of the input word $x$.
2. To verify that the generated sequence of triples follows the transition relation of $T$.

\(^3\) Using a routine padding technique one shows that every language in $k$-EXPTIME reduces in polynomial time to one of time complexity $\exp_k(n - 1)$, which is (for $k \geq 3$) less than the square root of $\exp_k(n)$. 
Positive Quantification Is Not Elementary (with Restricted Instantiation)

To verify 1 it has to be established that in every triple of the form \( \langle 0, a, s \rangle \), the value \( s \) is the symbol at position \( a \) in the initial configuration. To this end we use \( n+1 \) new procedures \( C^a \), defined for \( a \leq n \). Procedure \( C^a \) accepts from \( \langle t, a', s \rangle \) when \( t = 0 \), \( a = \min \{a', n\} \), and \( s = x_w \), where \( x_w \) is the appropriate symbol of the input (or blank for \( a = n \)). The definition of \( C^a \) is similar to that of \( C^a_k \). It is different in that it must verify that \( a' \geq n \). Observe that these procedures are initiated in separated branches of computation so they can use a single additional register to serve for \( Done_C \).

We verify 2 by an iteration over all triples \( \langle t, a, s \rangle \) for \( t > 0 \). The automaton expects that subtrees representing triples \( \langle t-1, a-1, s_1 \rangle \), \( \langle t-1, a, s_2 \rangle \), \( \langle t-1, a+1, s_3 \rangle \), are also present. This can be done by a non-deterministic guess in the course of descending to the trees. The roots of the trees can now be marked and we can run a subroutine \( E^a \) to confirm the guess. (The number of such subroutines is proportional to the size of the machine \( T \).) The definition of \( E^a \) combines the tricks used in the construction of \( S_{k-1} \) and \( E_k^k \). An additional complication is that it must compare halves of words rather than the whole words (recall that we merge \( t \) and \( a \) in \( \langle t, a, s \rangle \) into a single word). This is not a real problem, as the end of the first half is identified by an address of the form 011...1. The construction of \( E^a \), again, can be accomplished by a number of additional registers depending only on \( T \).

4.2.2 Automaton \( A_{T,x} \)

The automaton \( A_{T,x} \) first runs the procedure \( N_k \). Upon reaching the end state of \( N_k \) it checks that there is a triple \( \langle t, a, s \rangle \) where \( s = \langle a, f \rangle \) and \( f \) is an accepting state of \( T \). We have:

Lemma 9. Let \( T \) be a deterministic Turing Machine that works exactly in time \( \sqrt{\exp k (n)} \), and let \( x \) be a word of length \( n \). The automaton \( A_{T,x} \) has an \( n \)-restricted computation iff \( T \) accepts \( x \).

Proof. Suppose \( T \) accepts. Since the subgoals 1 and 2 of \( N_k \) are achieved, we have a computation that reaches an end state which properly encodes the computation of \( T \). All that remains is to verify that this computation contains an accepting state. This amounts to a single non-deterministic check.

Now suppose that \( A_{T,x} \) has an accepting computation tree. There is a computation leading to an end ID of \( N_k \), where the computation of \( T \) is properly encoded. Now there is no other way in which \( A_{T,x} \) can accept from such ID but to find an accepting state. So if \( A_{T,x} \) is accepting it must be the case that \( T \) accepts too.

The above combined with Lemma 7 allows us to conclude with the following theorem (because both reductions are polynomial):

Theorem 10. The restricted decision problem for positive quantification is not elementary.

We note that the above applies to monadic formulas (those involving only unary predicates). Indeed, the encoding in Section 3.2 did not require predicates of any higher arity.

Conclusions

The paper shows that only small restriction for the constructive predicate logic with positive quantification, that actually makes the proof of the decidability easier, gives rise to non-elementary complexity. The main difficulty of the proof is located in our procedures \( M_k \) that generate representation for long strings of bits. The notion of \( n \)-restriction is used
A. Schubert and P. Urzyczyn and D. Walukiewicz-Chrząszcz

precisely there. The rest of the construction is generic and could be applied also in a proof for the general case provided that a correct implementation of $M_k$ is given.

The restriction we propose is a bound on a particular kind of non-reusable resource within proofs. In the case of unrestricted positive logic this bound is waived. Therefore, the situation in which the complexity of this general case stabilised on a fixed exponential level would give a paradoxical conclusion that making more non-reusable resources available results in less complexity. This reinforces the claim that the non-elementary lower bound indeed holds for the fragment of first-order constructive logic that admits only positive quantification.

References

15 Aleksy Schubert, Paweł Urzyczyn, and Daria Walukiewicz-Chrząszcz. Positive logic is not elementary. Presentation at the Highlights conference, 2013.

19 Tao Xue and Qichao Xuan. Proof search and counter model of positive minimal predicate logic. *ENTCS*, 212(0):87–102, 2008.