Restricted Positive Quantification Is Not Elementary*

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 [3, 4, 13, 19] as well as theorem provers [1, 14]. Since these tools are actively used for development of verified software [9, 11] and for formalization 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 [12]. 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 $(\forall, \to)$ only, we can equivalently say that a formula φ is positive if and only if all occurrences of $\forall$ in φ are positive, where:

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

It is not immediate to see that deciding provability for positive formulas is possible. The same positive quantifier may be introduced several times in a proof, and this requires a fresh eigenvariable each time. The number of eigenvariables occurring in a proof is in general unbounded, so the search space for proofs is potentially infinite. However some of the eigenvariables may be regarded as equivalent — variables that "satisfy the same assumptions" can be exchanged with each other. Thus the identity of an eigenvariable x is determined by the set of assumptions made about it. With n assumptions there is 2^n such sets, so we need 2^n eigenvariables. The number of variables to consider grows exponentially at each level of nested quantification (see the discussion following Example 6), but altogether it remains finite.

Decision algorithms for formulas of minimal positive logic that rigorously develop the idea sketched above were given by Dowek and Jiang [5, 6], Rummelhoff [15], and Xue and

* Project supported through NCN grant DEC-2012/07/B/ST6/01532.

Xuan [21]. It should come as no surprise that these algorithms 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]).

As for the lower bound, the best result known up to date is only doubly exponential hardness [17], and our own attempt to prove non-elementary complexity failed; the proof in [16] 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, raising the quantifier nesting by one yields an at most exponential increase of the number of eigenvariables. This is a crucial argument in the known decidability proofs. We show that if this restriction becomes a part of the problem, i.e., if we require that the number of eigenvariables occurring in proofs is bounded by an appropriate multiply exponential function, then the problem is non-elementary.

This does not necessarily mean that the original problem is non-elementary, as there may be 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 hardness proof is inspired by an automata-theoretic interpretation of proof-search. The idea is simple and, we believe, quite universal. When attempting to construct a proof of a formula φ, one encounters subproblems of the form $\Gamma \vdash \alpha$. We think of α as if it was a state of an automaton and of Γ as of some kind of memory storage. Applying a proof tactic to $\Gamma \vdash \alpha$, which yields a new proof obligation $\Gamma' \vdash \alpha'$, can be seen as changing the state from α to α' and updating the memory Γ to Γ'. This way, proof construction can simulate a computation of an automaton.

Our *Eden automata* (or "expansible tree automata") are alternating machines operating on data that is structured into *trees of knowledge*. The computation trees of Eden automata correspond directly to proofs (equivalently, λ-terms) and the trees of knowledge represent the structure of binders in proofs. In fact, a slightly more general definition of Eden automata in [17] yields an exact equivalence between proofs and computations. Here, we stick to the weaker version, as we are only interested with a lower bound for the restricted case.

A specific feature of Eden automata is their monotone (non-erasing) access to data, very much as in the works by Leivant or even earlier by Wang [10, 20]. This is so because in a fully-structural logic assumptions are never deleted.

Structure of the paper Section 2 introduces some notation and states the principal definitions 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 $\langle \mathrm{T}, \leq \rangle$ with a least element $\varepsilon_{\mathrm{T}} \in \mathrm{T}$ (the root) and such that every non-root element $w \in \mathrm{T}$ has exactly one immediate predecessor (parent) v, in which case we say that w is a *child* of v. A *labelled tree* is a function $T : \mathrm{T} \to L$, where T is a tree, and L is a set of labels. We often confuse T with its domain T. If L is a set of m-tuples we may say that the *dimension* of T is m. A *proper ancestor* of a node w in a tree T is either a parent v of w or a proper ancestor of v. (The root of the tree has no proper ancestors.) 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 (maximal elements) are at depth k.

It is sometimes convenient to refer to the *level* of a node w which is the depth of the subtree $T_w = \{v \in T \mid w \leq v\}$, rooted at w. An *immediate subtree* of a node w in T is any tree T_v, where v is a child of w.

If $k \in \mathbb{N}$ then $\mathbf{k} = \{0, \ldots, k\}$. 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 use standard parentheses-avoiding conventions, in particular we take implication to be right-associative, e.g., $\varphi \to \psi \to \vartheta$ stands for $\varphi \to (\psi \to \vartheta)$.

We deal with *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 φ is positive and ψ is negative then $(\varphi \to \psi)$ is a negative formula.
- If φ is negative and ψ is positive then $(\varphi \to \psi)$ is a positive formula.
- If φ is positive and x is an object variable then $(\forall x\, \varphi)$ is a positive formula.
- If φ is negative and x is an object variable then $(\forall x\, \varphi)$ is a negative formula.

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

▶ **Lemma 1.**

1. *Every positive formula is of the form* $\forall\vec{x}_1(\sigma_1 \to \forall\vec{x}_2(\sigma_2 \to \cdots \to \forall\vec{x}_n(\sigma_n \to \forall\vec{x}_0\, \mathbf{a}) \ldots))$, *where* σ_i *are negative, and* $\mathbf{a}$ *is an atomic formula.*
2. *Every negative formula is of the form* $\tau_1 \to \tau_2 \to \cdots \to \tau_n \to \mathbf{a}$, *where* τ_i *are positive, and* $\mathbf{a}$ *is an atomic formula.*

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

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

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 Γ of declarations $(X : \varphi)$, where X is a proof variable and φ is a formula. We often identify Γ with the set of formulas $\{\varphi \mid (X : \varphi) \in \Gamma,\ \text{for some } X\}$. A *proof term* (or simply "term") is one of the following:

- a proof variable,
- an abstraction $\lambda X\!:\!\varphi.\, M$, where φ 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 Γ is an environment, M is a term, and φ is a formula. In rule $(\forall\mathcal{I})$ we require $x \notin FV(\Gamma)$ and y in rule $(\forall\mathcal{E})$ is an arbitrary object variable.

$$\Gamma, X : \varphi \vdash X : \varphi \quad (\mathcal{A}x)$$

$$\frac{\Gamma, X : \varphi \vdash M : \psi}{\Gamma \vdash \lambda X\!:\!\varphi.\, M : \varphi \to \psi}\,(\to\mathcal{I}) \qquad \frac{\Gamma \vdash M : \varphi \to \psi \quad \Gamma \vdash N : \varphi}{\Gamma \vdash MN : \psi}\,(\to\mathcal{E})$$

$$\frac{\Gamma \vdash M : \varphi}{\Gamma \vdash \lambda x\, M : \forall x \varphi}\,(\forall\mathcal{I}) \qquad \frac{\Gamma \vdash M : \forall x \varphi}{\Gamma \vdash My : \varphi[x := y]}\,(\forall\mathcal{E})$$

We may write $\lambda X^{\varphi} M$ for $\lambda X\!:\!\varphi.\, M$, and the upper index α in M^{α} means that term M has type α in some (implicit) environment. Other notational conventions are as usual in lambda-calculus, in particular application is left-associative, i.e., MNP stand for $((MN)P)$.

2.1 Restricted proofs and 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:

- $XN_1 \dots N_k$, where all N_i are normal forms or object variables;
- $\lambda X\!:\!\varphi.\, N$, where N is a normal form;
- $\lambda x\, N$, where N is a normal form.

Normal forms correspond to normal proofs in natural deduction (or to cut-free proofs in sequent calculus). It is known, see e.g., [18, 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 $\Gamma \vdash N : \varphi$.

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 $XN_1 \dots N_k$ where $k \geq 0$. The maximal nesting $\flat(X, M)$ of a variable X in a normal term M is defined formally as:

- $\flat(X, X) = 1$, $\flat(X, Y) = 0$, when $X \neq Y$;
- $\flat(X, YN_1 \dots N_k) = \flat(X, Y) + \max_i \flat(X, N_i)$;
- $\flat(X, \lambda Y\, N) = \flat(X, N)$, when $X \neq Y$, and $\flat(X, \lambda X\, N) = 0$;
- $\flat(X, \lambda y\, N) = \flat(X, N)$.

▶ **Definition 2** (n-restricted proofs).
We say that a normal proof M 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., $\flat(X, N) \leq \exp_k(n)$. A judgement is *$n$-provable* when it has an n-restricted normal proof.

▶ **Problem 3.** [restricted decision problem for positive quantification]
Given a positive formula φ and a number n, decide if φ is n-provable.

The process of proof search is easier to control if we restrict our attention to proofs in *long normal form.*

▶ **Definition 4.** The notion of a term in long normal form (lnf) is defined according to its type in a given environment.

- If N is an lnf of type α then $\lambda x\, N$ is an lnf of type $\forall x\, \alpha$.
- If N is an lnf of type β then $\lambda X : \alpha.\, N$ is an lnf of type $\alpha \to \beta$.
- 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 5.** *If $\Gamma \vdash M : \sigma$ and M is in normal form then there exists a long normal form N such that $\Gamma \vdash N : \sigma$. In addition, if M is n-restricted then so is N.*

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

- $T^{\forall x.\alpha}(M) = \lambda x\, T^\alpha(Mx)$;
- $T^{\alpha\to\beta}(M) = \lambda X : \alpha.\, T^\beta(MX)$;
- $T^\alpha(M) = M$ if α 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 α then $T^\alpha(M)$ is an lnf of type α.

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 α in some environment; the definition is by induction with respect to M:

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

Observe that transformations T and R have the following property:

- They do not change the number and relative position of existing occurrences of free nor bound proof variables in a term;
- Whenever a new variable is added, it only occurs once in the result.

The desired term N equals $R^\sigma(M)$. Details are left to the reader. The two properties above ensure that N remains n-restricted when so is M. ◀

The logic of long normal proofs We say that a judgement $\Gamma \vdash \varphi$ is *positive* when φ is positive, and all formulas in Γ are negative. The type-assignment rules below preserve positivity, and by Lemma 5 they make a complete proof system for positive judgments.

$$\frac{\Gamma, X : \varphi \vdash M : \psi}{\Gamma \vdash \lambda X : \varphi.\, M : \varphi \to \psi}\ (\to\mathcal{I}) \qquad \frac{\Gamma \vdash M_i : \tau_i, \quad i = 1, \ldots, n}{\Gamma, X : \tau_1 \to \cdots \to \tau_n \to \mathbf{a} \vdash XM_1 \ldots M_n : \mathbf{a}}\ (\to\mathcal{E})$$

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

3 Computational content of positive logic

As already mentioned, the main complication of deciding provability of a positive formula 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. To see how this works let us go through the following example.

▶ **Example 6.** Let $\mathbf{1}$ and $\mathbf{0}$ be unary predicate symbols, and let $\mathsf{G}, \mathsf{L}, \mathsf{U}, \mathsf{Z}$, be nullary atoms.[1] Consider the formulas:

$$\varphi = (\psi \to \mathsf{L}) \to (\mathsf{Z} \to \mathsf{U} \to \mathsf{L}) \to \mathsf{L};$$
$$\psi = \forall x.\, \mathit{Gen}_0(x) \to \mathit{Gen}_1(x) \to \mathit{Zero}(x) \to \mathit{One}(x) \to \mathsf{G},$$

with the following components, whose intended meaning will soon become clear.

$$\begin{array}{rcl} \mathit{Gen}_0(x) & = & (\mathbf{0}(x) \to \mathsf{L}) \to \mathsf{G}; \\ \mathit{Gen}_1(x) & = & (\mathbf{1}(x) \to \mathsf{L}) \to \mathsf{G}; \\ \mathit{Zero}(x) & = & \mathbf{0}(x) \to \mathsf{Z}; \\ \mathit{One}(x) & = & \mathbf{1}(x) \to \mathsf{U}. \end{array}$$

We show how the process of finding a long normal proof of the formula φ represents a computation of a simple "procedure" consisting of two phases:

1. Nondeterministically generate a number of bits in a loop;
2. Check that there is at least one 0 and at least one 1 among the generated bits.

The atoms correspond to states of the procedure:

- L – the entry point to the main Loop;
- G – Generate a bit;
- Z – test for the presence of Zero;
- U – test for the presence of 1 (a "Unit").

A long normal proof of the formula φ (a lambda term of type φ) must take the shape $\lambda X^{\psi\to\mathsf{L}} \lambda Y^{\mathsf{Z}\to\mathsf{U}\to\mathsf{L}}.\, M$, where the term M of type L should begin either with X or with Y. Let us consider the first possibility, i.e., let $M = XM_1$. The long normal term M_1 has type ψ, so we must have $M_1 = \lambda x_1 \lambda Z_1 U_1 V_1 W_1.\, N_1$, where $Z_1 : \mathit{Gen}_0(x_1)$, $U_1 : \mathit{Gen}_1(x_1)$, $V_1 : \mathit{Zero}(x_1)$, $W_1 : \mathit{One}(x_1)$, and $N_1 : \mathsf{G}$. This way we have replaced the proof goal L by a new proof goal G. We interpret it as passing from state L to state G in a computation.

The term N_1 can now begin with any of the variables Z_1, U_1, V_1, W_1, so let us try Z_1, i.e., take $N_1 = Z_1(\lambda T_1^{\mathbf{0}(x_1)}.\, M_2)$, with M_2 of type L. The variable T_1, which can occur in M_2, is a new assumption of the form $\mathbf{0}(x_1)$ added to the proof environment. Our computational interpretation of this phase is that the bit zero has been just written to the memory cell represented by the eigenvariable x_1 and the control went back to state L.

Asking about M_2 we note that one possibility is $M_2 = X(\lambda x_2 \lambda Z_2 U_2 V_2 W_2.\, N_2)$, with $Z_2 : \mathit{Gen}_0(x_2)$, $U_2 : \mathit{Gen}_1(x_2)$, $V_2 : \mathit{Zero}(x_2)$, $W_2 : \mathit{One}(x_2)$, and $N_2 : \mathsf{G}$. This step introduces to the proof a new eigenvariable x_2 (or allocates a new memory cell x_2). We may now construct $N_2 = Z_2(\lambda T_2^{\mathbf{0}(x_2)}.\, M_3)$, and repeat the loop once more in a slightly

[1] Nullary atoms, used for clarity, can be easily replaced by unary ones.

different way, by taking $M_3 = X(\lambda x_3\lambda Z_3U_3V_3W_3.\,U_3(\lambda T_3^{\mathbf{1}(x_3)}.\,M_4))$. Now we have three memory locations x_1, x_2, x_3, containing respectively the values $0, 0, 1$. We could continue in this fashion by introducing more locations and more bits, but now we can also complete the proof construction by choosing, for example, $M_4 = Y(V_1(T_1))(W_3(T_3))$. This step represents entering states Z and U, to check the presence of memory locations holding zero and one. Note that this two actions happen independently in parallel (it is a universal computation step). As a result we obtain a complete proof of φ:

$$
\begin{aligned}
&\lambda X^{\psi\to\mathsf{L}}\lambda Y^{\mathsf{Z}\to\mathsf{U}\to\mathsf{L}}.\,X(\lambda x_1\lambda Z_1U_1V_1W_1.\,Z_1(\lambda T_1^{\mathbf{0}(x_1)}.\\
&\qquad\qquad X(\lambda x_2\lambda Z_2U_2V_2W_2.\,Z_2(\lambda T_2^{\mathbf{0}(x_2)}.\\
&\qquad\qquad\qquad X(\lambda x_3\lambda Z_3U_3V_3W_3.\,Z_3(\lambda T_3^{\mathbf{1}(x_3)}.\,Y(V_1(T_1))(W_3(T_3)))))))).
\end{aligned}
$$

In the above proof, the subterm $V_1(T_1)$ using the variable x_1, can be replaced by $V_2(T_2)$, because assumptions made about x_2 and x_1 are exactly the same. We may say that variables x_1 and x_2 are "equivalent", and from this point of view, introducing x_2 was not necessary. Indeed, the middle line of the above term could simply be deleted without any harm.

As we mentioned before, a proof (for instance a proof of the formula in Example 6) can involve an unbounded number of variables. 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 a basic instance of such 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 $\mathcal{O}(n)$ different subformulas, so if it only has one quantifier $\forall x$ (like the one in our example) then the number of non-equivalent eigenvariables introduced for the quantifier is (in the worst case) exponential in n, as one has to account for every selection from up to $\mathcal{O}(n)$ 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 $\mathcal{O}(n)$ 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 for $\forall x$. Any additional nested quantifier increases the number of non-equivalent variables exponentially, and this yields the super-elementary upper bound.

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 $\mathcal{A} = \langle\, k, m, \mathcal{R}, Q, q^0, \mathfrak{I}\,\rangle$, where:

- $k \in \mathbb{N}$ is the *depth* of $\mathcal{A}$ (recall the notation $\mathbf{k} = \{0, \ldots, k\}$);
- $\mathcal{R}$ is the finite set of *registers*; the number $m = |\mathcal{R}|$ is the *dimension* of $\mathcal{A}$.

- Q is the finite set of *states*, partitioned as $Q = \bigcup_{i\in\mathbf{k}} Q_i$. In addition, each Q_i splits into disjoint sets $Q_i^\forall$ and $Q_i^\exists$ and we also define $Q^\forall = \bigcup_{i\in\mathbf{k}} Q_i^\forall$ and $Q^\exists = \bigcup_{i\in\mathbf{k}} Q_i^\exists$. States in $Q^\forall$, $Q^\exists$ are respectively *universal* and *existential*.
- $q^0 : \mathbf{k} \to Q$ assigns the *initial state* $q_i^0 \in Q_i$ to every $i \in \mathbf{k}$.
- $\mathfrak{I}$ is the set of *instructions*.

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

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

Instructions available in $q \in Q_i^\forall$, 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 $\{0,1\}^{\mathcal{R}}$ (i.e., functions from $\mathcal{R}$ to $\{0,1\}$), called *snakes*. That is, if v is a node of T then $T(v)$ 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 $\vec{0}$ stands for a snake constantly equal to 0. Finally, the component w is a node of T called the *current apple*. We require that $q \in Q_{|w|}$. 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 ε, the root, labelled with $\vec{0}$ (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 $I \in \mathfrak{I}$ at C. We now define how this may happen. Assume that $q \in Q_i$, and first consider case (1) where $I =$ "q : $\texttt{jmp}$ p".

- If $p \in Q_i$ 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), i.e., $I =$"q : $\texttt{check}$ $R(h)$ $\texttt{jmp}$ p", 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 is of the form (3), i.e., $I =$ "q : $\texttt{set}$ $R(h)$ $\texttt{jmp}$ 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 (4), i.e., $I =$ "q : $\texttt{new}$" 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 $T'(w') = \vec{0}$. (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

The idea of a tree of knowledge is that each node in the tree corresponds directly to an eigenvariable in a proof. Therefore our restriction on proofs gives rise to a restriction for trees: if every child of a node w has at most n children, then the number of children of w should not exceed 2^n. This motivates the following definition. 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 $i > 0$ has at most $\exp_i(n)$ children.

We are interested in *n-restricted computations*, where all IDs are n-restricted. More formally, we say that an ID is *eventually n-accepting* if it is n-restricted, and

– either it is existential and it has an eventually n-accepting successor,
– or it is universal and all its successors are eventually n-accepting.

3.2 The encoding

Throughout this section we assume that the parameter n is fixed. Our goal is to encode an Ea with a positive first-order formula in such a way that the automaton has an accepting n-restricted computation if and only if the formula has an n-restricted normal proof. Given an automaton $\mathcal{A} = \langle k, m, \mathcal{R}, Q, q^0, \mathfrak{I} \rangle$, our formula uses unary predicate symbols q, for all $q \in Q$, and R, for all $R \in \mathcal{R}$. Each individual variable is of the form x_i or x_i^w, where $i \in \mathbf{k}$ and w is a node in some tree of knowledge. For a root node ε, we identify x_0^ε with x_0.

Notation: If S is a set of formulas $\{\alpha_1, \ldots, \alpha_k\}$ then $S \to \beta$ abbreviates the formula $\alpha_1 \to \cdots \to \alpha_k \to \beta$. Similarly $\lambda X^S.\, M$ and $\lambda \vec{X} : S.\, M$ abbreviate $\lambda X_1^{\alpha_1} \ldots X_k^{\alpha_k}.M$.

Convention: Without loss of generality we can assume that for every $i < k$ there is exactly one state $q \in Q_i$ such that the instruction "$q : \texttt{new}$" belongs to $\mathfrak{I}$. Indeed, otherwise we can modify the automaton by adding designated "transfer states" $q_i^\star$ to Q_i and replacing each "$q : \texttt{new}$" by "$q : \texttt{jmp}\ q_i^\star$" and "$q_i^\star : \texttt{new}$" when necessary.

3.2.1 Encoding instructions

For every $i \in \mathbf{k}$, 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 $q \in Q_i^\forall$, and let "q : $\texttt{jmp}$ p_1", …, "q : $\texttt{jmp}$ p_r" be all the instructions available in q. Then the following formula belongs to S_i :

$$p_1(x_i) \to \cdots \to p_r(x_i) \to q(x_i).$$

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

$$p(x_i) \to q(x_{i-1}).$$

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

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

- For "q : $\texttt{jmp}$ p", where $p \in Q_j$ and $j \in \{i,\, i-1\}$, the formula is $p(x_j) \to q(x_i)$.
- For "q : $\texttt{check}$ $R(h)$ $\texttt{jmp}$ p", the formula is $p(x_i) \to R(x_h) \to q(x_i)$.
- For "q : $\texttt{set}$ $R(h)$ $\texttt{jmp}$ p", the formula is $(R(x_h) \to p(x_i)) \to q(x_i)$.
- For "q : $\texttt{new}$", the formula is $\forall x_{i+1}(S_{i+1} \to q^0_{i+1}(x_{i+1})) \to q(x_i)$.

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

The number of nested occurrences of a variable $Z : \forall x_{i+1}(S_{i+1} \to q^0_{i+1}(x_{i+1})) \to q(x_i)$ exactly corresponds to the number of different eigenvariables induced by the quantifier $\forall x_{i+1}$. Indeed, Z occurs in contexts of the form "$Z(\lambda x_{i+1} \ldots Z(\lambda x'_{i+1} \ldots Z(\lambda x''_{i+1}.\, M) \ldots) \ldots)$", and all the individual variables x_{i+1}, x'_{i+1}, x''_{i+1}, … may be free inside M.

3.2.2 Encoding IDs:

Let now S be a set of formulas and let w be a node of depth i in a tree of knowledge. For every $j \le i$, replace all occurrences of x_j in S by x_j^v, 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 $|w|$:

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

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

$$\Gamma^R_T = \{R(x^w_i) \mid w \in T \wedge |w| = i \wedge T(w)(R) = 1\};$$
$$\Gamma^S_T = \bigcup\{S_i[w] \mid w \in T \wedge |w| = i\};$$
$$\Gamma_T = \Gamma^R_T \cup \Gamma^S_T$$

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

The following lemma reduces the halting problem for n-restricted computations of Eas 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 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 Γ_T contains, for every non-leaf node $w \in T$, a declaration

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

where i is the depth of w. Now for every child v of w there is a variable x^v_{i+1} in $\mathrm{FV}(\Gamma_T)$. These eigenvariables should be thought of as reducing the limit of nested occurrences of Z_w in proofs defined in Γ_T. Let ch^T_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 $\flat(Z_w, M) \leq \exp_{k-i}(n) - ch^T_w$, for every $i \leq k$ and every node w at depth i.

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

In particular, the initial ID is eventually n-accepting if and only if $\vdash \Gamma^S_{T_0} \to q^0_0(x_0)$, 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 $\langle\, q, T, w\,\rangle$ be an eventually n-accepting ID of $\mathcal{A}$. 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 n-accepting IDs.

If q is a universal state and $\langle\, q, T, w\,\rangle$ is eventually n-accepting then all successors of $\langle\, q, T, w\,\rangle$ are eventually n-accepting. Every successor ID corresponds to some instruction "q : `jmp` p_j" for $j = 1, \ldots, s$. By the induction hypothesis we have $\Gamma_T \vdash p_j(x^w_i)$ for $j = 1, \ldots, s$.

By the definition of Γ_T, the formula $p_1(x_i) \to \cdots \to p_s(x_i) \to q(x_i)$ belongs to S_i and $p_1(x^w_i) \to \cdots \to p_s(x^w_i) \to q(x^w_i)$ 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 $\langle\, q, T, w\,\rangle$ is eventually n-accepting then there exists a successor of $\langle\, q, T, w\,\rangle$ which is eventually n-accepting. This successor $\langle\, p, T', w'\,\rangle$ is a result of execution of an instruction I of $\mathcal{A}$, applicable in state q. We check the possible forms of I.

If I is "q : `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 $\Gamma_T \vdash p(x^{w'}_j)$. Since Γ_T contains the formula $p(x^{w'}_j) \to q(x^w_i)$, we conclude that $\Gamma_T \vdash q(x^w_i)$.

For "q : `check` $R(j)$ `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, $\Gamma_T \vdash p(x^w_i)$. Since Γ_T contains the formula $p(x^w_i) \to R(x^v_j) \to q(x^w_i)$ and the atom $R(x^v_j)$, we conclude that $\Gamma_T \vdash q(x^w_i)$.

For "q : `set` $R(j)$ `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 $\Gamma_{T'} \vdash p(x^w_i)$. Note that $\Gamma_{T'} = \Gamma_T \cup R(x^v_j)$, and consequently $\Gamma_T \vdash R(x^v_j) \to p(x^w_i)$. Since Γ_T contains the formula $(R(x^v_j) \to p(x^w_i)) \to q(x^w_i)$, we conclude that $\Gamma_T \vdash q(x^w_i)$.

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. These proofs are also long normal, as all are of the form $X\vec{N}$, where $\vec{N}$ are long normal by induction.

Only the last case involves quantification. For "q : `new`", where $q \in Q_i$, $p = q^0_{i+1}$, the tree T' is obtained from T by adding a brand new child w' of w labelled $\vec{0}$ (empty registers). From the induction hypothesis we know that $\Gamma_{T'} \vdash M : q^0_{i+1}(x^{w'}_{i+1})$ where M respects T'. Note that $\Gamma_{T'} = \Gamma_T \cup S_{i+1}[w']$, so we may deduce that $\Gamma_T \vdash \lambda\vec{X}^{S_{i+1}[w']}.\, M : S_{i+1}[w'] \to q^0_{i+1}(x^{w'}_{i+1})$. The variable $x^{w'}_{i+1}$ does not appear in Γ_T, hence we also have

$$\Gamma_T \vdash \lambda x_{i+1}.\lambda\vec{X} : S_{i+1}[w'][x^{w'}_{i+1} := x_{i+1}].\, M : \forall x_{i+1}(S_{i+1}[w'][x^{w'}_{i+1} := x_{i+1}] \to q^0_{i+1}(x_{i+1})).$$

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

$$Z_w : \forall x_{i+1}(S_{i+1}[w] \to q^0_{i+1}(x_{i+1})) \to q_i(x_i^w),$$

we conclude that $\Gamma_T \vdash Z_w(\lambda x_{i+1}.\lambda \vec{X}^{S_{i+1}[w']}.M) : q_i(x_i^w)$. This is a long normal proof introducing a single application of the proof variable Z_w. It respects T because M respects T' and the number of children of w in T is smaller by one than the number in T'. Also the obtained proof is n-restricted, because so is M and because M respects T', in particular the number of nested occurrences of $Z_{w'}$ in M is at most $exp_{k-i-1}(n)$ (node w' has no children).

($\Leftarrow$) Suppose that $\langle q, T, w \rangle$ is an ID of an automaton $\mathcal{A}$ such that $\Gamma_T \vdash N : q(x_i^w)$, 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 n-accepting. Since $q(x_i^w)$ is an atom, we must have $N = XN_1 \dots N_r$, for some X and some long normal forms $N_1, \dots, N_r$. In addition, there must be a declaration $(X : \varphi) \in \Gamma_T$, where $\varphi = \tau_1 \to \dots \to \tau_r \to q(x_i^w)$, and $\Gamma_T \vdash N_l : \tau_l$, for each l.

Let $q \in Q^i_\forall$ and let "q : `jmp` p_j", for $j = 1, \dots, s$, be all instructions available in state q. By the definition of Γ_T, there is only one formula φ that ends with the atom $q(x_i^w)$, namely $\varphi = p_1(x_i^w) \to \dots \to p_s(x_i^w) \to q(x_i^w)$. Therefore, $r = s$ and for every $j = 1, \dots, r$, we have $\Gamma_T \vdash N_j : p_j(x_i^w)$. By the induction hypothesis we know that $\langle p_j, T, w \rangle$ are eventually n-accepting. Since a universal ID is eventually n-accepting when all its successors are eventually n-accepting, we get the desired conclusion.

Let $q \in Q^i_\exists$. Since the formula φ ends with $q(x_i^w)$, 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 n-accepting. This will imply that also $\langle q, T, w \rangle$ is eventually n-accepting.

If φ has the form $p(x_j^{w'}) \to q(x_i^w)$, for some variable $x_j^{w'}$, then I is "q : `jmp` p". Note that such a φ may occur in Γ_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 $\Gamma_T \vdash N_1 : p(x_j^{w'})$, we conclude that $\langle p, T, w' \rangle$ is eventually n-accepting.

If φ is $p(x_i^w) \to R(x_j^v) \to q(x_i^w)$ then I is "q : `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 $\Gamma_T \vdash N_1 : p(x_i^w)$ and $\Gamma_T \vdash N_2 : R(x_j^v)$. The only formula in Γ_T of the form $\alpha_1 \to \dots \to \alpha_k \to R(x_j^v)$ is $R(x_j^v)$. By the definition of Γ_T, if $R(x_j^v) \in \Gamma_T$ then $T(v)(R) = 1$. Hence I can be executed. Since $\Gamma_T \vdash N_1 : p(x_i^w)$, 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 φ is $(R(x_j^v) \to p(x_i^w)) \to q(x_i^w)$ then I is "q : `set` $R(j)$ `jmp` p" and $j \leq i$. The result of execution of I at $\langle q, T, w \rangle$ is $\langle p, T', w \rangle$, where $T' = T[v \mapsto T(v)[R \mapsto 1]]$. We know that $\Gamma_T \vdash N_1 : R(x_j^v) \to p(x_i^w)$. Since N_1 is an lnf, there exists N_1' such that $\Gamma_T, Y : R(x_j^v) \vdash N_1' : p(x_i^w)$. By the induction hypothesis, $\langle p, T', w \rangle$ is eventually accepting.

The last case is when $\varphi = \forall x_{i+1}(S_{i+1}[w] \to q^0_{i+1}(x_{i+1})) \to q(x_i^w)$ is the type of Z_w and the instruction I is "q : `new`". We have $\Gamma_T \vdash Z_w N_1 : q(x_i^w)$ and we also know that $\Gamma_T \vdash N_1 : \forall x_{i+1}(S_{i+1}[w] \to q^0_{i+1}(x_{i+1}))$. Since N_1 is an lnf, it must have the form $N_1 = \lambda x_{i+1} \lambda \vec{Y}{:}S_{i+1}[w].N_1'$, for some lnf N_1'. Substituting $x_{i+1}^{w'}$ for x_{i+1} we obtain the type assignment

$$\Gamma_T, \vec{Y} : S_{i+1}[w][x_{i+1} := x_{i+1}^{w'}] \vdash N_1'[x_{i+1} := x_{i+1}^{w'}] : q^0_{i+1}(x_{i+1}^{w'}).$$

Note that $S_{i+1}[w][x_{i+1} := x_{i+1}^{w'}]$ equals $S_{i+1}[w']$ and $\Gamma_T, \vec{Y} : S_{i+1}[w'] = \Gamma_{T'}$, where T' is obtained from T by adding a new child w' of w labelled $\vec{0}$. The term $N_1'[x_{i+1} := x_{i+1}^{w'}]$ is n-restricted and respects T' because the top occurrence of Z_w was eliminated, and because w'

has no children in T'. Hence, by the induction hypothesis, the result $\langle q^0_{i+1}, T', w' \rangle$ of execution of I is eventually n-accepting. ◀

4 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 flags are never raised together.

▶ **Example 8.** 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 $\mathcal{A}$ of depth 1, with $q^0_0 = q_0$, $q^0_1 = q_1$, and with the instructions (where $q_0 \in Q^\exists_0$, and other states are in $Q^\exists_1$):

q_0 : `new`;

q_1 : `set` $L_1(1)$ `jmp` q_2;	q_2 : `set` $L_2(1)$ `jmp` q_3;	q_3 : `set` $L_3(1)$ `jmp` q_4;
q_1 : `set` $R_1(1)$ `jmp` q_2;	q_2 : `set` $R_2(1)$ `jmp` q_3;	q_3 : `set` $R_3(1)$ `jmp` q_4.

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

$$q_1 : \text{for } i = 1 \text{ to } 3 \text{ do } [\texttt{set } L_i \text{ OR } \texttt{set } R_i]; \text{ goto } q_4.$$

The computation of our automaton has one branch, which 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, $\mathcal{A}'$, which runs after $\mathcal{A}$, i.e., it commences in state q_4. Among its states, $q'_1, q'_2, q'_3, q_{\mathrm{acc}}$ are in $Q^\forall_1$ and other states are in $Q^\exists_1$.

q_4 : `jmp` q'_1;

q'_1 : `jmp` q^{chk}_1;	q'_2 : `jmp` q^{chk}_2;	q'_3 : `jmp` q^{chk}_3;
q'_1 : `jmp` q'_2;	q'_2 : `jmp` q'_3;	q'_3 : `jmp` q_5;
q^{chk}_1 : `check` $L_1(1)$ `jmp` q_{acc};	q^{chk}_2 : `check` $L_2(1)$ `jmp` q_{acc};	q^{chk}_3 : `check` $L_3(1)$ `jmp` q_{acc};

The automaton $\mathcal{A}'$ is initiated in state q_4 in node $\mathbf{d}$, a successor of the root. At node $\mathbf{d}$, one register in each of the pairs L_1, R_1; L_2, R_2; L_3, R_3 is set to 1. The automaton then enters state q'_1 at node $\mathbf{d}$. The procedure from state q'_1 to state q_5 constitutes a universal for loop, informally written as follows:

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

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 L_1, L_2, L_3 are set to encode the sequence of bits 000. The apple is at the child node and the machine is in state q_5.

These two automata may be viewed as procedures in a single program. The first procedure generates non-deterministically a string of three bits and the second works like a finite automaton that checks if all the bits are equal to 0. (Note that any loop-free finite automaton can be simulated this way.)

4.1 Procedures

Throughout this section we assume that the parameter n is fixed, and we only consider n-restricted computations (a computation which is not n-restricted is illegal).We show how to deal with numbers up to $\exp_k(n)$ using trees of knowledge of depth k.

The trees and automata we consider here have dimension $2n+7$. 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;
- a *global* register *Steady*;
- *local* registers: *New, Old, Done, Gone.*

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: for $i = 0, 1$, register A_i set to 1 represents the bit i.

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. 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 $\{(0, a_0), (1, a_1), \ldots, (r-1, a_{r-1})\}$. If a tree T encodes a number i and has value a_i at the root then T represents a pair (i, a_i). A word $a_0a_1 \ldots a_{r-1}$ of length r can thus be encoded by a tree consisting of a root node and a number of immediate subtrees representing the pairs (i, a_i). (Observe that i is then encoded by a string of length of order $\log r$.) Once we know how to encode binary words of length d we can interpret them as numbers from 0 to $2^d - 1$, and use the above method to give an encoding for words of length 2^d.

More precisely, we define what it means that a tree T of uniform depth k *encodes* a word w of length $\exp_k(n)$. To begin with $k = 0$, a tree T consisting of a single node $\mathbf{d}$ *encodes* a binary word $x_0x_1 \ldots x_{n-1}$ of length n when, for each number $i = 1, \ldots, n-1$, we have $T(\mathbf{d})(L_i) = 1$ iff $x_i = 0$, and $T(\mathbf{d})(R_i) = 1$ iff $x_i = 1$. (Note that there are other registers as well, so many trees encode the same number.) A tree T of uniform depth k *encodes* a word $w = x_0x_1 \ldots x_{r-1}$ of length $r = \exp_{k+1}(n)$ when

- T has exactly r immediate subtrees, each encoding a different number $i \in \{0, \ldots, r-1\}$;
- If $\mathbf{d}$ is the root of an immediate subtree encoding i then $T(\mathbf{d})(A_j) = 1$ iff $x_i = j$. (We say that i is the *address* of $\mathbf{d}$ and j is called the *value* of $\mathbf{d}$.)

A node $\mathbf{d}$ in a tree is said to *encode* a word when the subtree rooted at $\mathbf{d}$ encodes that word.

▶ **Remark 9.** One can easily generalize the above definition to words over any l-element finite alphabet $\Sigma = \{a_0, a_1, \ldots, a_{l-1}\}$ with trees of dimension $ln + l + 5$, and 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 branch either ends in an accepting ID, or in an *end ID* (an ID with a specified *end state*), where the control should be passed to another subroutine. In general there may be many occurrences of end IDs in a computation. However, the procedures we consider in this paper have this particular property that every computation contains at most one end ID.

For every k and every $l > k$, we define procedures $\mathcal{M}_k$, $\mathcal{E}_k^l$, S_k, $\mathcal{C}_k^0$, and $\mathcal{C}_k^1$, 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 in the form of a input-output condition. PU, 18 IV

4.1.1 Induction hypotheses

Making a new word

For every $k \geq 0$ we define a procedure $\mathcal{M}_k$ to make new words.

Start ID: The current apple is a leaf $\mathbf{d}$ of the tree of knowledge, the snake at $\mathbf{d}$ is empty.

Claim: PU, 18 IV

1. No computation of $\mathcal{M}_k$ ever uses (jumps, writes to or reads from registers at) any proper ancestor of $\mathbf{d}$.
2. A successful computation of $\mathcal{M}_k$ has only one end ID. At the end ID the apple is back at $\mathbf{d}$, but $\mathbf{d}$ is now a root of a subtree of uniform depth k and $\mathbf{d}$ encodes a non-deterministically chosen word w of length $\exp_k(n)$. All local registers are empty in the subtree rooted at $\mathbf{d}$.

Part 1 of the induction hypothesis is a separation condition which states that the procedure $\mathcal{M}_k$ does not have side effects. This is necessary since the procedure has end states, and computations continues after these are reached.

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

Constant

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

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

Claim: Procedure $\mathcal{C}_k^0$ (resp. $\mathcal{C}_k^1$) accepts iff the address of $\mathbf{d}$ is $\vec{0}$ (resp. $\vec{1}$).

Equality

Procedure $\mathcal{E}_k^l$, where $l > k$, verifies equality of two binary words.

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

Claim: Procedure $\mathcal{E}_k^l$, initiated in a start ID, accepts iff the addresses of $\mathbf{e}_O$ and $\mathbf{e}_N$ are the same.

Successor

Binary words are identified with numbers so that the successor relation holds between strings of the form $w011\ldots1$ and $w100\ldots0$. Procedure $\mathcal{S}_k$ verifies this relation.

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

Claim: Procedure $\mathcal{S}_k$, initiated in a start ID, accepts iff the address of $\mathbf{e}_N$ is the successor of the address of $\mathbf{e}_O$.

4.1.2 Procedures

To provide a gentle introduction we begin our presentation with the relatively simple procedure $\mathcal{C}^0_k$; after that we proceed in the order of the previous subsection.

Procedure $\mathcal{C}^0_k$

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

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

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

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

First, let us make an informal account of the way the procedure operates. When $\mathcal{C}^0_k$ is initiated in a start ID at a node $\mathbf{d}$ 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 $\vec{0}$, guessing it non-deterministically. At this point the computation splits into two branches. One branch verifies the correctness of the guess by running $\mathcal{C}^0_{k-1}$ (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 *Done*, and then proceed to another child of $\mathbf{d}$ (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 which has already been 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 $\mathbf{e}$ is marked as *New*, then another child $\mathbf{e}'$ of $\mathbf{d}$ is selected and marked as *Old*. But first we check register *Done* at node $\mathbf{e}'$ to make sure that $\mathbf{e}'$ has been processed.[2] It remains to run $\mathcal{S}_{k-1}$ from node $\mathbf{d}$ to complete the verification branch (steps 8–11).

The main loop continues until we non-deterministically guess that we reached a node with address $\vec{1}$. This is verified by initiating $\mathcal{C}^1_{k-1}$, and then the procedure accepts.

Let us remark here that, although the above description of the algorithm is informal, it is precise enough to be implemented as an actual automaton, using a number of internal PU, 18 IV states proportional to n. Now we can show that $\mathcal{C}^1_k$ satisfies the specification.

[2] It may happen that $\mathbf{e}' = \mathbf{e}$ but in this case the successor test will fail.

($\Leftarrow$) Observe that in case the address of $\mathbf{d}$ encodes the word $w = \vec{0}$ and $\mathcal{C}^0_k$ is run from a correct start ID then the procedure may choose to take the child of $\mathbf{d}$ with address $\vec{0}$ in step 1 so that $\mathcal{C}^0_{k-1}$ accepts in step 2. Then all other children are chosen in step 6 in order of increasing addresses, so that it is always possible in step 9 to choose an appropriate predecessor address, guaranteeing termination in step 11. A more formal proof should go by induction with respect to the number of children of $\mathbf{d}$ marked as *Done*. Note that local registers at levels $k-2$ and below are empty and can be safely used by each procedure. 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.

($\Rightarrow$) Suppose now that $\mathcal{C}^0_k$ accepts. Let l be the number of times the procedure enters step 4 in the accepting computation. Let D_i be the set of children of $\mathbf{d}$ marked as *Done* at the i-th entry to step 4. Let a_i be the maximal address encoded by an element of D_i. By induction with respect to $l - i$ we show the following statement

> *For each accepting computation subtree of $\mathcal{C}^0_k$ started at the i-th entry to step 4 and for each address b such that $a_i < b < \exp_k(n)$, the node $\mathbf{d}$ has a child that encodes the number b and has A_0 set to 1.*

Indeed, for $i = l$, the set of addresses a such that $a_i < b < \exp_k(n)$ is empty, so the conclusion follows. If $i < l$ then an accepting computation must enter the loop and mark one child of $\mathbf{d}$ with *Done* 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 (*Done* 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 $\mathbf{b}_i \in D_{i+1} - D_i$. Let b_i be the number encoded by $\mathbf{b}_i$. In steps 8–11 it is verified that $b_i = a + 1$, for some a encoded by $\mathbf{a} \in D_i$, but actually b must be a_i as otherwise $a_i = a_{i+1}$. This also means that $b_i = a_{i+1}$. Node $\mathbf{b}_i$ has A_0 set to 1, as this is verified in step 3. 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.

Now observe that at the first entry to step 4 only one child of $\mathbf{d}$ is marked as *Done*, and it must encode the address $\vec{0}$ (steps 1–2) with A_0 set (step 3). As the further computation accepts, we can apply the statement proven above for $i = 1$ and obtain that $\mathbf{d}$ has children that encode addresses b such that $0 < b < \exp_k(n)$ and all have A_0 set to 1. This applies also for the address 0. Since by assumption $\mathbf{d}$ encodes a a word of length $\exp_k(n)$, this must be the number of children of $\mathbf{d}$. Therefore $\mathbf{d}$ encodes $\vec{0}$.

Let us remark here that in step 6 the apple may be passed to a child already marked as *Done*, 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.

A diggression before we proceed to the next procedure: The above algorithm can easily be adapted to verify if the binary string encoded by $\mathbf{d}$ belongs to any fixed regular language.

Procedure $\mathcal{M}_k$

We can now turn to the more complicated procedure $\mathcal{M}_k$. For the base case $k = 0$ we generalize the automaton $\mathcal{A}$ of Example 8:

$$\texttt{for } i = 1 \texttt{ to } n \texttt{ do } [\texttt{set } L_i \texttt{ OR set } R_i].$$

In the induction step we assume that procedures $\mathcal{M}_{k-1}$, $\mathcal{E}^l_{k-1}$, $\mathcal{S}_{k-1}$, $\mathcal{C}^0_{k-1}$, and $\mathcal{C}^1_{k-1}$, have already been defined, and we describe $\mathcal{M}_k$ as a pseudo-code "program" consisting of two phases. Recall that the computation begins at the root $\mathbf{d}$ of the word to be constructed.

Phase 1: At first, procedure $\mathcal{M}_k$ runs $\mathcal{M}_{k-1}$ in a loop. The number of iterations is chosen non-deterministically, but it is bounded due to the n-restrictedness condition, as each iteration creates a new child.

1. Create a new child and descend there;
2. Run $\mathcal{M}_{k-1}$;
3. Set register A_0 OR set register A_1;
4. go up; goto 1 (continue) OR goto 5 (enter Phase 2);

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

An immediate inductive argument (for the loop in steps 1–4) shows that

1. The computation does not use (jumps, writes to or reads from registers at) any proper ancestor of $\mathbf{d}$.
2. At the entry to step 4 the apple is back at $\mathbf{d}$, and $\mathbf{d}$ 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 to 1 and starts a subtree that encodes a number in $\{0, \ldots, \exp_k(n) - 1\}$.

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

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

5. Descend to a child; goto 6 (verify) AND goto 7 (continue);
6. Run $\mathcal{C}^0_{k-1}$ (accepting inside).
7. Set register *Steady*;
8. goto 9 OR goto 16;
9. Go up to $\mathbf{d}$;
10. Descend to a child;
11. goto 12 (verify) AND goto 7 (continue);
12. Set register *New*; go up to $\mathbf{d}$;
13. Descend to a child;
14. Check register *Steady*; set register *Old*; go up to $\mathbf{d}$;
15. Run $\mathcal{S}_{k-1}$ (accepting inside);
16. Run $\mathcal{C}^1_{k-1}$ (verify) AND goto 17 (continue);
17. Go up to $\mathbf{d}$ (end state).

The second phase works very much like the procedure $\mathcal{C}^0_k$. In step 5 the computation splits into two branches. One proceeds (fingers crossed) along the main computation branch beginning at step 7. 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 6 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 11 and 16. Note that registers *Old* and *New* remain intact outside of the subroutine 12–15. At the completion of the above we are back at node $\mathbf{d}$. Again an immediate inductive argument (for the loop in steps 7–11) shows that:

1. The computation does not use (jumps to, writes to or reads from registers at) any proper ancestor of $\mathbf{d}$.

2. Each time the computation reaches step 8, the apple is in a child of **d**, and **d** has a non-empty set C of children with $|C| \leq \exp_k(n)$. The set of numbers encoded by nodes in C is closed with respect to predecessor (in particular it contains zero).

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 closure with respect to predecessor we obtain that all addresses of length $\exp_k(n)$ must be encoded by children of **d**. And each of them only once, because the computation is n-restricted. This is exactly part 2 of the induction hypothesis for $\mathcal{M}_k$. Part 1 follows from (1) above.

▶ **Remark 10.** 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 Σ. It is enough to use more registers and to adjust step 3 of the automaton $\mathcal{M}_k$ so that it chooses one of the registers corresponding to elements of Σ instead of A_0 or A_1.

Procedure $\mathcal{S}_k$

Recall that we begin in a node **d** which has (among others) exactly one child marked as *Old* (i.e., satisfying $Old = 1$) and exactly one marked as *New*.

Subtrees rooted at these nodes are assumed to encode binary words w_{old} and w_{new} of length $\exp_k(n)$. We want to verify that $w_{old} = w011\ldots1$ and $w_{new} = w100\ldots0$, for some w. For $k = 0$ this can be done with a simple for loop. For $k > 0$, we process children of *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 *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 $\mathcal{S}_k$ with a little more detail, but on a higher level of abstraction than the previous procedures. We believe that this account is still precise enough, and at the same time easier to understand. To make it even more comprehensive, let us first explain some of the phrases used below For instance, "to descend to a child of *Old*" (step 1) means to descend to a child **e** of **d**, check $\mathbf{e}(Old)$, 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 $\mathcal{C}^0_k$ and $\mathcal{M}_k$.) In step 2 this is equivalent to the statement "Run $\mathcal{C}^0_{k-1}$ AND goto 3". Similarly, in steps 7 and 13 the verification branch calls procedure $\mathcal{S}_{k-1}$, and steps 4, 9, 14 activate procedure $\mathcal{E}^{k+1}_{k-1}$.

1. Descend to a child of *Old*;
2. Universally verify that the present address consists of only zeros;
3. goto 4 (phase 1) OR goto 9 (end of phase 1);
4. Universally verify that the data bit at the present node is the same as the data bit of a child of *New* of the same address;
5. Mark the present node as *Done*; go up (to the node marked as *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 *Done*;
8. goto 3;
9. Universally verify that the data bit at the present node is 0, while the data bit of a child of *New* of the same address is 1;
10. Mark the present node as *Gone*;
11. goto 12 (phase 2) OR goto 16 (end);

12. Go back to **d**; descend to a child of *Old*;
13. Universally verify that the present address is the successor of an address of a brother node already marked as *Gone*;
14. Universally verify that the data bit at the present node is 1, while the data bit of a child of *New* of the same address is 0;
15. Mark the present node as *Gone*; goto 11;
16. Run $\mathcal{C}^1_{k-1}$ (accepting inside).

Assuming that the start ID of $\mathcal{S}_k$ is as expected, we can now refer to the induction hypothesis about $\mathcal{C}^0_{k-1}$, $\mathcal{S}_{k-1}$, and $\mathcal{E}^l_{k-1}$. 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.

Procedure $\mathcal{E}^l_k$

This procedure works in a similar way as $\mathcal{S}_k$ 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 $\mathcal{E}^l_k$ and $\mathcal{S}_k$. 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 $\mathcal{E}^l_k$ will accept in this case while $\mathcal{S}_k$ will not.

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 $\mathcal{T}$ working in time $\exp_k(n^{\mathcal{O}(1)})$ and fix an input word $\mathbf{x}$ of length n. Without loss of generality[3] we can assume that the machine works exactly in time $\sqrt{\exp_k(n)} - 1$. Let $\Sigma = \Sigma_0 \cup (\Sigma_0 \times \Delta)$ where Σ_0 is the tape alphabet and Δ is the set of states of $\mathcal{T}$. We already know (see Remark 9) how to encode words over Σ using trees of knowledge.

We use a triple $\langle\, t, a, \mathtt{s}\,\rangle$ to express that the contents of the tape cell a at time t is $\mathtt{s}$. Here, $\mathtt{s} \in \Sigma$ is either a tape symbol of $\mathcal{T}$ or a tape symbol plus an internal state (in case $\mathcal{T}$ at time t is at position a). A computation of $\mathcal{T}$ is represented by a unique set of triples with only one $\langle\, t, a, \mathtt{s}\,\rangle$ for every t, a. Note that $a, t \in \{0, \ldots, \sqrt{exp_k(n)} - 1\}$. Consequently there are exactly $exp_k(n)$ pairs $\langle\, t, a\,\rangle$ and they can be identified with numbers less than $exp_k(n)$. The whole computation of machine $\mathcal{T}$ may therefore be seen as a word over Σ 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 dimension (extra data registers are needed to account for all elements of Σ). In this way we can represent a computation of $\mathcal{T}$ in the memory of an Eden automaton.

A slight adjustment of the automaton $\mathcal{M}_k$ of Section 4.1 (in step 3) yields a procedure to generate an arbitrary word over Σ of length $\exp_k(n)$.

---

[3] Using a routine padding technique one shows that every language in $\mathrm{D{\small TIME}}(\exp_k(n^{\mathcal{O}(1)})$ 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 $\exp_k(n)$.

4.2.1 Procedure $\mathcal{N}_k$

The definition of $\mathcal{N}_k$ is similar to that of $\mathcal{M}_k$, but now we have to only generate words representing accepting computations of $\mathcal{T}$. Therefore, $\mathcal{N}_k$ 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 $\mathcal{T}$.

Phase 1 is similar to phase 1 of $\mathcal{M}_k$ (see Remark 10). Phase 2 is more complicated, but it can similarly be related to phase 2 of $\mathcal{M}_k$. Steps 12–15 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 an encoding of the input word $\mathbf{x}$.
2. To verify that the sequence obeys the transition relation of $\mathcal{T}$.

For part (1) it has to be established that in every triple of the form $\langle\, 0, a, \mathtt{s}\,\rangle$, the value $\mathtt{s}$ is the symbol at position a in the initial configuration. To this end we use $n+1$ new procedures $\mathcal{C}^a$, defined for $a \le n$. Procedure $\mathcal{C}^a$ accepts from $\langle\, t, a', \mathtt{s}\,\rangle$ if $t = 0$, and $\mathtt{s} = x_a$, where $a = \min\{a', n\}$, and x_a is the appropriate symbol of the input (or blank for $a = n$). The definition of $\mathcal{C}^a$ is similar to that of $\mathcal{C}^x_k$. Observe that procedures $\mathcal{C}^a$ are initiated in separated branches of computation so they can use the same registers.

We handle (2) by an iteration over triples $\langle\, t, a, \mathtt{s}\,\rangle$ for $t = 0, \ldots, \sqrt{\exp_k(n)} - 1$. The automaton expects that subtrees encoding triples $\langle\, t{-}1, a{-}1, \mathtt{s}_1\,\rangle$, $\langle\, t{-}1, a, \mathtt{s}_2\,\rangle$, $\langle\, t{-}1, a{+}1, \mathtt{s}_3\,\rangle$, are also present. Those can be nondeterministically guessed and their roots appropriately marked (using four special registers for this purpose). Then we can run a subroutine $\mathcal{E}^{\mathtt{q}}$ (where $\mathtt{q}$ is a transition of $\mathcal{T}$) to confirm the guess. (The number of such subroutines is proportional to the size of the machine $\mathcal{T}$.) The definition of $\mathcal{E}^{\mathtt{q}}$ combines the tricks used in the construction of $\mathcal{S}_{k-1}$ and $\mathcal{E}^k_{k-1}$. An additional complication is that it must compare halves of words rather than the whole words (recall that we merge t and a in $\langle\, t, a, \mathtt{s}\,\rangle$ 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 $011\ldots1$. The construction of $\mathcal{E}^{\mathtt{q}}$, again, can be accomplished by a number of additional registers depending only on $\mathcal{T}$.

4.2.2 Automaton $\mathcal{A}_{\mathcal{T},\mathbf{x}}$

The automaton $\mathcal{A}_{\mathcal{T},\mathbf{x}}$ first runs the procedure $\mathcal{N}_k$. Upon reaching the end state of $\mathcal{N}_k$ it checks that there is a triple $\langle\, t, a, \mathtt{s}\,\rangle$ where $\mathtt{s} = \langle\, \mathbf{a}, f\,\rangle$ and f is an accepting state of $\mathcal{T}$.

▶ **Lemma 11.** *Let $\mathcal{T}$ be a deterministic Turing Machine that works in time $\sqrt{\exp_k(n)} - 1$, and let $\mathbf{x}$ be a word of length n. The automaton $\mathcal{A}_{\mathcal{T},\mathbf{x}}$ has an n-restricted accepting computation iff the machine $\mathcal{T}$ accepts $\mathbf{x}$.*

Proof. Suppose $\mathcal{T}$ accepts. By construction, the automaton $\mathcal{N}_k$ has a computation that reaches an end ID which properly encodes the computation of $\mathcal{T}$. All that remains is to verify that this computation contains an accepting ID. This amounts to a single non-deterministic `check`.

Now suppose that $\mathcal{A}_{\mathcal{T},\mathbf{x}}$ has an accepting computation tree. This computation contains an end ID of $\mathcal{N}_k$, where the computation of $\mathcal{T}$ is properly encoded. Now there is no other way in which $\mathcal{A}_{\mathcal{T},\mathbf{x}}$ can accept from such ID but to find an accepting state. So if $\mathcal{A}_{\mathcal{T},\mathbf{x}}$ is accepting, it must be the case that $\mathcal{T}$ accepts the word $\mathbf{x}$. ◀

The above combined with Lemma 7 yields a polynomial-time reduction of any language in $\mathrm{DTIME}(\exp_k(n^{\mathcal{O}(1)})$ to Problem 3. We can thus conclude with the following theorem:

▶ **Theorem 12.** *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.

Conclusion

We have demonstrated that the provability problem for intuitionistic logic with positive quantification becomes non-elementary under an apparently small restriction on proofs (computations). Technically, the only use of this restriction is in the definition of procedures $\mathcal{M}_k$ that generate representation for long strings of bits. Therefore, if an unrestricted implementation of $\mathcal{M}_k$ is possible then the original (unrestricted) problem is also not elementary.

The restriction we propose is a bound on a particular kind of a certain non-reusable resource. Under this restriction, the decidability of our formulas becomes immediate, as it reduces the search space to a finite size. In fact, the argument in e.g. the work by Dowek and Jiang [5] or in the work by Minc [12] shows that the actual use of this resource in a proof of a formula φ is essentially equivalent to that in an $\mathcal{O}(n)$-restricted proof, where n is the size of φ. Still, the proof itself does not have to be n-restricted. Shall we prove it has, the general result will follow from our consideration. Although the opposite seems unlikely, the conjecture remains an open question.

References

1 Ulrich Berger, Kenji Miyamoto, Helmut Schwichtenberg, and Monika Seisenberger. Minlog: a tool for program extraction supporting algebras and coalgebras. In *Proc. of CALCO'11*, volume 6859 of *LNCS*, pages 393–399. Springer, 2011.

2 Egon Börger, Erich Grädel, and Yuri Gurevich. *The Classical Decision Problem.* Perspectives in Mathematical Logic. Springer, 1997.

3 Ana Bove, Peter Dybjer, and Ulf Norell. A brief overview of Agda – a functional language with dependent types. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics*, volume 5674 of *LNCS*, pages 73–78. Springer, 2009.

4 Robert L. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1986.

5 Gilles Dowek and Ying Jiang. Eigenvariables, bracketing and the decidability of positive minimal predicate logic. *Theoretical Computer Science*, 360(1–3):193–208, 2006.

6 Gilles Dowek and Ying Jiang. Enumerating proofs of positive formulae. *Computer Journal*, 52(7):799–807, 2009.

7 Georges Gonthier. The four colour theorem: Engineering of a formal proof. In D. Kapur, editor, *Computer Mathematics*, volume 5081 of *LNCS*. Springer, 2008.

8 Georges Gonthier. Advances in the formalization of the odd order theorem. In M. van Eekelen, H. Geuvers, J. Schmaltz, and F. Wiedijk, editors, *Interactive Theorem Proving*, volume 6898 of *LNCS*, page 2. Springer, 2011.

9 Gerwin Klein et al. seL4: Formal verification of an OS kernel. *Communications of the ACM*, 53(6):107–115, 2010.

10 Daniel Leivant. Monotonic use of space and computational complexity over abstract structures. Technical Report CMU-CS-89-21, Carnegie Mellon University, 1989.

11 Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

12 Grigori E. Mints. Solvability of the problem of deducibility in LJ for a class of formulas not containing negative occurrences of quantifiers. *Steklov Inst.*, 98:135–145, 1968.

13 Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

14 Jens Otten. ileanTAP: An intuitionistic theorem prover. In D. Galmiche, editor, *Proc. of TABLEAUX'97*, volume 1227 of *LNCS*, pages 307–312. Springer, 1997.

15 Ivar Rummelhoff. *Polymorphic* Π1 *Types and a Simple Approach to Propositions, Types and Sets*. PhD thesis, University of Oslo, 2007.

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

17 Aleksy Schubert, Paweł Urzyczyn, and Daria Walukiewicz-Chrząszcz. How hard is positive quantification? In preparation, 2014.

18 Morten H. Sørensen and P. Urzyczyn. *Lectures on the Curry-Howard Isomorphism*, volume 149 of *Studies in Logic and the Foundations of Mathematics*. Elsevier, 2006.

19 The Coq Development Team. *The Coq Proof Assistant. Reference Manual*. INRIA, December 2011.

20 Hao Wang. A variant to Turing's theory of computing machines. *Journal of the ACM*, 4(1):63–92, 1957.

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