How Hard Is Positive Quantification?*

Aleksy Schubert, Paweł Urzyczyn, and Daria Walukiewicz-Chrząszcz

University of Warsaw, Institute of Informatics
[alx,urzy,daria]@mimuw.edu.pl

Abstract. We show that the constructive predicate logic with positive (covariant) quantification is hard for doubly exponential universal time, i.e., for the class *co*-2-NEXPTIME. Our approach is to represent proof-search as computation of an alternating automaton. The memory of the automaton is structured in a way that strictly corresponds to scopes of the binders used in the constructed proof. This provides an application of automata-theoretic techniques in proof theory.

1 Introduction

Constructive logics are basis for many proof assistants [4, 15, 19] and theorem provers [2, 16]. Since these tools are actively used for development of verified software [13, 10] 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 [14]. 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 positive/negative in $\forall x\, \varphi$ and in $\psi \to \varphi$.
- Positive/negative positions in ψ are respectively negative/positive in $\psi \to \varphi$.

Decision algorithms for formulas of minimal positive logic (and positive formulas of System F) were given by Dowek and Jiang [5, 6], Rummelhoff [17], and Xue and Xuan [21].

Despite the modesty of the positive minimal logic, the decision algorithms given by these proofs are not easy, and the obtained upper bound is not elementary. This is because each instantiation of an internal universal quantifier creates a new "local environment" of assumptions, opaque, and separated from other such environments. With the unbounded depth of quantification, such local environments may, in principle, represent arbitrary hereditarily finite sets. We conjecture that this actually can be done. Unfortunately, our attempt to

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

prove the non-elementary lower bound turned out to be incorrect [18] and the question remains open.

However, as we show below, the inherent complexity (and thus also the expressive power) of positive quantification is enormous anyway. We prove that the problem is hard for doubly exponential universal time, i.e., hard for the class *co*-2-NEXPTIME. This holds even when only the connectives $(\forall, \to)$ and unary predicates are used. The result stays in contrast with the situation in classical logic where the analogous problem of satisfiability for $\exists^*$-sentences is NP-complete [3, Thm. 6.4.3].

The lower bound is obtained by interpreting 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 φ, one encounters subproblems of the form $\Gamma \vdash \alpha$. We think of α as if it were a state of an automaton and of Γ as of some kind of memory storage. If we restrict attention to long normal proofs then α is typically an atomic "subformula" of the initial proof goal φ. The finite number of atoms in φ makes a finite set of states, if one can account for arbitrary instantiations of individual variables. In our *Eden automata* (or "expansible tree automata") this is handled by a pointer addressing an adequate position in memory, organized as a *tree of knowledge*. The tree reflects the structure of instantiated quantifiers and stores "knowledge" about proof assumptions. Computations of Eden automata correspond directly to proofs (equivalently, λ-terms) and the tree structure of memory makes it possible to manage the scopes of binders.

Related work Eden automata work on structured data. Simple data structures in form of flat registers were used in tree automata by Kaminski and Tan [9]. However, there are significant differences between our approach and theirs. Our automata store binary information in registers while the automata of Kaminski and Tan store elements of an infinite set. Moreover, our registers are structured in trees and the access to data is not random, but depends on a location of a pointer in the tree. As to navigation of the pointer, it makes our model similar to the tree-walking automata [1]. However, when passing from a father to a child node, an Eden automaton cannot control which child is taken. What is most prominent, our automata modify the tree data structure they operate on while tree walking ones only examine it.

Eden automata store information in monotonic (non-erasing) fashion similar to that in [20, 12]; in addition the access to information resembles the use of positive queries in [12] (the computation diverges on a negative response). But in [12] every bit is identified by a fully addressable "relational pointer" while in our case the access to data is mostly nondeterministic.

Structure of the paper In Section 2 we give some insight into the intricacy of the problem. Then we introduce Eden automata and define the translation of automata into formulas. In Section 3 we use examples to explain tools used in programming Eden automata. The main technical development to encode Turing Machines as Eden automata is done in Section 4.

Preliminary definitions We take $\mathbf{n} = \{0, \ldots, n\}$ and 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

$\varepsilon_T \in T$ (the root) and such that every non-root element $x \in T$ has exactly one immediate predecessor (parent). A node x with exactly h proper ancestors in T is said to be *at depth* h, and then we may write $|x| = h$. The *depth* of T is the maximal depth of a node in T. A *labeled tree* is a function $T : \mathrm{dom}(T) \to L$, where L is a set of labels. We sometimes identify T with $\mathrm{dom}(T)$. If L is a set of m-tuples we may say that the *dimension* of T is m.

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.

2 Computational content of positive logic

A naive idea of a decision algorithm is as follows: the only quantifier rule needed to prove a positive formula is $\forall$-introduction. Therefore one essentially deals only with open proof goals and routine methods are applicable. The complication here is that one quantifier can be introduced several times in a proof and may require several different "eigenvariables". This is demonstrated by the following example.

Example 1. Let G_1, G_2, Q_{Loop} be nullary[1] and $Q_{Body}(x)$ a unary predicate. We show a formula that represents a simple iteration over its subformulas. Let

$$\alpha = \beta \to Q_{Loop}, \qquad \beta = \forall x(\gamma_{Body}(x)) \to Q_{Loop}$$

where we use the abbreviations $\gamma_{Body}(x) = \varphi(x) \to \psi(x) \to \vartheta(x) \to Q_{Body}(x)$, $\varphi(x) = (G_1 \to Q_{Loop}) \to Q_{Body}(x)$, $\psi(x) = G_1 \to (G_2 \to Q_{Loop}) \to Q_{Body}(x)$, and $\vartheta(x) = G_2 \to Q_{Body}(x)$. A long normal proof of the formula α (seen as a lambda-term) must take the shape $\lambda X^\beta.\, X^\beta(\lambda x_1 \lambda U_1 V_1 Y_1.\, M_1)$, with $X^\beta : \beta$, $U_1 : \varphi(x_1)$, $V_1 : \psi(x_1)$, $Y_1 : \vartheta(x_1)$, and $M_1 : Q_{Body}(x_1)$. The term M_1 could begin with either of U_1, V_1, Y_1 but the use of V_1 and Y_1 is impossible due to the presence of blocking guards G_1, G_2 in their types. So the only workable solution must be $M_1 = U_1(\lambda Z_1 : G_1.\, N)$ with $N = X^\beta(\lambda x_2 \lambda U_2 V_2 Y_2.\, M_2) : Q_{Loop}$. Here, x_2 is a new eigenvariable, and $U_2 : \varphi(x_2)$, $V_2 : \psi(x_2)$, $Y_2 : \vartheta(x_2)$. If we look at the possible shapes of M_2 we find out that it may begin either with U_2 or with V_2, as G_1 is now available, but cannot begin with Y_2, as G_2 is not. Observe that the shortest inhabitant of α must involve yet another iteration using ϑ and introducing a variable x_3. As a result, it has the form

$$\begin{aligned}\lambda X^\beta.\, X^\beta(\lambda x_1 \lambda U_1 V_1 Y_1.\, U_1(\lambda Z_1 : G_1.\\ X^\beta(\lambda x_2 \lambda U_2 V_2 Y_2.\, V_2 Z_1(\lambda Z_2 : G_2.\\ X^\beta(\lambda x_3 \lambda U_3 V_3 Y_3.\, Y_3 Z_2))))).\end{aligned}$$

This term represents a simple looping over the subformulas $\varphi(x), \psi(x)$, and $\vartheta(x)$. With a little more effort one can give examples where a similar loop is executed four or more times. In general, a proof can involve an unbounded number of variables. In [5] it is shown how some eigenvariables may be eliminated, because

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

"equivalent" variables can replace each other. The term "equivalent" is understood as "satisfying the same assumptions".

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 copies of x 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 $\forall x\,(\dots\forall y\,\varphi(x,y)\dots)$. For every copy x' of x we now have $\mathcal{O}(n)$ subformulas of $\varphi(x',y)$ and therefore up to exponentially many copies of y introduced as eigenvariables for $\forall y\,\varphi(x',y)$. Any set of such eigenvariables may potentially be created for a given copy of x, and this gives a doubly exponential number of choices. Two copies of x may be assumed equivalent only when they induce the same choice, so we get a doubly exponential number of possible non-equivalent copies of x. Every next quantifier now increases the number of non-equivalent eigenvariables exponentially, and this yields the super-elementary upper bound.

2.1 Eden automata

In order to prove the lower bound we introduce an appropriate automata-theoretic model. An *Eden automaton* (abbr. Ea) is an alternating computing device, organizing 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 positive guards: it can be verified that a flag is up, but checking a flag which is down results in a failure. Every flag is initially down, but once raised, it so remains forever.

Formally, an Ea is a tuple $\mathcal{A} = \langle\, k, m, M, Q, q^0, \mathfrak{I}\,\rangle$, 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 $Q = \bigcup_{i\in\mathbf{k}} Q_i$. 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 : `jmp` p", where $p \in Q_j$, and $|i-j| \le 1$;
2. "q : `check` $R(h)$ `jmp` p", where $p \in Q_i$ and $h \le i$;
3. "q : `set` $R(h)$ `jmp` p", where $p \in Q_i$ and $h \le i$;
4. "q : `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, labeled 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 $T(v)$ 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, labeled 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 the simplest case when $I =$ "q : `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 nondeterministically 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`" 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 nondeterministic 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.

2.2 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 is eventually accepting if and only if the formula is provable. Given an automaton $\mathcal{A} = \langle\, k, m, M, Q, q^0, \mathfrak{I}\,\rangle$, 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 $i \in \mathbf{k}$ 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 $\{\alpha_1, \ldots, \alpha_r\}$ then $S \to \beta$ abbreviates the formula $\alpha_1 \to \cdots \to \alpha_r \to \beta$.

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 : `new` belongs to $\mathfrak{I}$. Indeed, otherwise we can modify the automaton by adding designated "transfer states" $q_i^\star$ to Q_i and adding q : `jmp` $q_i^\star$ to $\mathfrak{I}$ when necessary.

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 : `jmp` p_1", ..., "q : `jmp` p_k" be all the instructions available in q. Then the following formula belongs to S_i :

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

Existential states (downward moves): For each instruction of the form "q : `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 : `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 : `check` $R(h)$ `jmp` p", the formula is $p(x_i) \to R(x_h) \to q(x_i)$.
- for "q : `set` $R(h)$ `jmp` p", the formula is $(R(x_h) \to p(x_i)) \to q(x_i)$.
- for "q : `new` ", the formula is $\forall x_{i+1}(S_{i+1} \to q_{i+1}^0(x_{i+1})) \to q(x_i)$.

Note that 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 $\mathcal{A}$. Also note that the quantifier $\forall x_{i+1}$ occurs above at a negative position.

Encoding IDs: Let now S be a set of formulas, and let w be a node in a tree of knowledge and i be its depth. For every $j \leq i$, 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:

$$\Gamma_T^R = \{R(x_i^w) \mid w \in T \wedge |w| = i \wedge T(w)(R) = 1\};$$
$$\Gamma_T^S = \bigcup\{S_i[w] \mid w \in T \wedge |w| = i\};$$
$$\Gamma_T = \Gamma_T^R \cup \Gamma_T^S\,.$$

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

The purpose of this encoding is the following lemma, which states that the halting problem for Ea is reducible to provability of positive formulas.

Lemma 2. *Let $\mathcal{A}$ be an Eden automaton. An ID of $\mathcal{A}$ of the form $\langle q, T, w\rangle$ is eventually accepting if and only if the positive formula $\Gamma_T \to q(x_i^w)$, where $i = |w|$, has a proof. In particular, the automaton $\mathcal{A}$ is eventually 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.*

3 Eden programming

In this section we show a number of examples demonstrating the computational power of Eden automata. Each of these examples contributes a different technique to be later used in our hardness proof. We explain the behavior of our automata using quite informal pseudocode "programs" involving for- and while-loops, auxiliary variables etc. All these constructs can be implemented by actual automata using e.g. the internal states.

The „positive only" access to knowledge in an Eden automaton makes it impossible 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.

Example 3. 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 $q_0^0 = q_0$, $q_1^0 = q_1$, and with the instructions (where $q_0 \in Q_0^\exists$, $q_1, q_2, q_3 \in Q_1^\forall$, and other states are in $Q_1^\exists$):

q_0 : `new`;

q_1 : `jmp` q_1^L;	q_2 : `jmp` q_2^L;	q_3 : `jmp` q_3^L;
q_1 : `jmp` q_1^R;	q_2 : `jmp` q_2^R;	q_3 : `jmp` q_3^R;
q_1^L : `set` $L_1(1)$ `jmp` q_2;	q_2^L : `set` $L_2(1)$ `jmp` q_3;	q_3^L : `set` $L_3(1)$ `jmp` q_4;
q_1^R : `set` $R_1(1)$ `jmp` q_2;	q_2^R : `set` $R_2(1)$ `jmp` q_3;	q_3^R : `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 w, a successor of the root, and enters state q_1 at node w. The procedure from state q_1 to state q_4 constitutes a universal for loop, informally written as follows:

$$q_1 : \mathsf{for}\ i = 1\ \mathsf{to}\ 3\ \mathsf{do}\ [\mathtt{set}\ L_i\ \mathsf{AND}\ \mathtt{set}\ R_i];\ \mathsf{goto}\ q_4.$$

The computation of our automaton has 8 branches, each ending in an ID where the only child of the root represents a word of length 3, different word at every branch. The apple is at the child node and the machine is in state q_4. □

From now on we assume that our automata have two special registers L and R. The idea is that, at any node of the tree of knowledge, register L set to 1 identifies the node as the "left" child of its parent, while register R is 1 in the "right" child. It must be ensured by construction that exactly one of those bits is set to 1 at every node but the root. The sequence of R's and L's read downward from the root to w makes a *location* of node w. More precisely: the root has the empty location, and a child w of node v has location πL (resp. πR) when v has location π and $L(w) = 1$ (resp. $R(w) = 1$).

Our second example demonstrates how an Eden automaton of size $\mathcal{O}(n)$ can construct a tree of knowledge with at least 2^n leaves in which every location of length n is present.

Example 4. This automaton has depth n and uses three registers R, L, and OK. It builds a full binary tree of knowledge of depth n, and stops when this task is completed. The automaton is purely nondeterministic: it has no universal states (except one final state q_0^{up}).

The "program" of the automaton consists essentially of two "subroutines". One is executed when the machine is in state q_i^0 (the initial state for depth i), for $0 < i < n$, the other when in state $q_i^{up} \in Q_i$, for $0 < i \leq n$. The computation begins with the instruction:

$$q_0^0 : \mathtt{new}.$$

The machine creates a new node at level 1 and enters state q_1^0. Then, for all $i = 1, \ldots, n-1$, we run the first subroutine, informally described as follows:

$$\begin{aligned} q_i^0 :\ & \text{Set register } L(i)\ \ \mathsf{OR}\ \ \text{if } OK(i-1) \text{ then set register } R(i); \\ & \mathtt{new}\ ; \end{aligned}$$

The notation OR represents nondeterministic choice. The instruction `new` creates a new node at level $i+1$ and the machine enters state q_{i+1}^0. The bottom level n is an exception, because it is no longer possible to go down. Thus, after choosing between $L(n)$ and $R(n)$, the automaton enters state q_n^{up} and this activates the iteration of the second subroutine. Below we take $0 < i \leq n$:

$$\begin{aligned} q_n^0 :\ & \text{Set register } L(n)\ \ \mathsf{OR}\ \ \text{if } OK(n-1) \text{ then set register } R(n); \\ & \mathtt{jmp}\ q_n^{up}\ ; \\ q_i^{up} :\ & \mathtt{check}\ R(i)\ \mathtt{jmp}\ q_{i-1}^{up}\ \ \mathsf{OR}\ \ \mathtt{check}\ L(i)\ \mathtt{jmp}\ q_{i-1}^{stop}\ ; \\ q_i^{stop} :\ & \mathtt{set}\ OK(i);\ \mathtt{new}. \end{aligned}$$

Recall that the `new` instruction moves the control of the automaton to the state q^{up}_{i+1}, i.e., calls the first subroutine. State q^{up}_0 is an accepting state (a universal state with no applicable instructions).

How does it work? The first phase of computation builds the "leftmost path" of the tree of knowledge (since registers OK are initially zero, there is no other choice but to set $L(i)$). At the end of this phase, the current apple is at the only leaf of the single "leftmost" branch.

Then we begin iterating the main loop, which starts in a node at level n in state q^{up}_n. The induction hypothesis (loop invariant) is as follows. The ID of the automaton is of the form $\langle\, q^{up}_n, T, w \,\rangle$ where $|w| = n$. All nodes above w, except the root, have exactly one bit R or L set to 1.

Let $v \leq w$ be the last node with $T(v)(L) = 1$. Under these circumstances the only possible behavior of our machine is to go up to node v and then to its parent u, and enter state q^{stop}_j, where $j = |u|$. Indeed, the behavior of the machine in state q^{up}_i is essentially this:

$$q^{up}_i : \text{If } R(i) \text{ then go to } q^{up}_{i-1} \text{ else go to } q^{stop}_{i-1}.$$

At node u (in state q^{stop}_{i-1}) register OK is set to 1, and a new child x of u is created. Then the first procedure is started at node x with $i = j + 1$. The difference with the initial use of this procedure is that now $OK(i-1)$ holds, and the new node may (but does not have to) be marked as a right node by raising the flag R. In this case we go down to level n again and create a new leaf, the location of which is the lexicographic successor of the location of w. Indeed, a suffix of the form $LR \ldots R$ in the address of w is replaced by $RL \ldots L$ for the new leaf.

It does not have to be the case, and we may obtain a node with a location ending with the suffix $LL \ldots L$. However, in order to successfully terminate, the computation must eventually enter q^{up}_0 at the root. For this, it must necessarily create a node with location R^n, and this is only possible after building a tree with *at least* one leaf for every location. Indeed, by inspecting the graph of transitions between states, one can show by induction that:

- The only way to reach state q^{up}_0 is from location R^n visited in state q^{up}_n;
- State q^{up}_n at location $\pi RL...L$ is only reachable from state q^{stop}_i at location π;
- The only way to be in state q^{stop}_i at location π is to get there from location $\pi LR...R$ visited in state q^{up}_n.

That is, to visit any location of length n one must first see the lexicographic predecessor of that location. Hence all of them must occur. □

A drawback of the above construction is that we cannot guarantee that the tree T has exactly one leaf for every location. But it is still possible to ensure that only one leaf per location is actually used. This is demonstrated by the next example (also purely nondeterministic). It shows how a tree of knowledge of depth n can be used to store a word of length 2^n over the alphabet $\{0, 1\}$. This requires two additional registers, Z for 0 and J for 1. Locations in $\{L, R\}^n$ can be read as binary numbers from 0 to $2^n - 1$, let us write $\sharp\pi$ for the number given by π.

We say that a tree of knowledge T of depth n *encodes* a word $x \in \{0,1\}^{2^n}$ when:

- For every location $\pi \in \{L,R\}^n$ there is exactly one leaf w in T with location π such that either $J(w) = 1$ or $Z(w) = 1$;
- If w has location π and $\sharp\pi = \ell$ then $J(w) = 1$ if the ℓ-th symbol in x is 1, otherwise $Z(w) = 1$;
- It is never the case that $J(w) = 1$ and $Z(w) = 1$.

Example 5. This is a modification of the machine in Example 4. The machine behaves exactly as before, until it reaches state q_0^{up} at the root. The tree of knowledge is now fully constructed, and we write data into the registers Z and J at the leaf nodes, from "left" to "right". We use new internal states $p_i^{up}, p_i^{stop}, p_i^{go}, p_i^{down} \in Q_i$. First we go down to a leaf with location L^n, using the following instructions (where $0 < i < n$):

$$q_0^{up} \ : \ \texttt{jmp}\ p_1^{down}.$$
$$p_i^{down} \ : \ \texttt{check}\ L(i)\ \texttt{jmp}\ p_{i+1}^{down};$$

In state p_n^{down} we enter the main loop, first choosing between 0 and 1:

$$p_n^{down} \ : \ \texttt{check}\ L(n); [\,\texttt{set}\ Z(n)\ \ \texttt{OR}\ \texttt{set}\ J(n)];\ \texttt{jmp}\ p_{n-1}^{stop};$$

The main loop is a search for a lexicographically next leaf:

$$p_i^{up} \ : \ \texttt{check}\ R(i)\ \texttt{jmp}\ p_{i-1}^{up}\ \texttt{OR}\ \texttt{check}\ L(i)\ \texttt{jmp}\ p_{i-1}^{stop} \qquad (0 < i < n);$$
$$p_i^{stop} \ : \ \texttt{jmp}\ p_{i+1}^{go} \qquad (0 \le i < n);$$
$$p_i^{go} \ : \ \texttt{check}\ R(i)\ \texttt{jmp}\ p_{i+1}^{down} \qquad (0 < i < n);$$
$$p_n^{go} \ : \ \texttt{check}\ R(i); [\,\texttt{set}\ Z(i)\ \ \texttt{OR}\ \texttt{set}\ J(i)];\ \texttt{jmp}\ p_{n-1}^{up}.$$

Observe that the behavior of our automaton is essentially deterministic: making an incorrect choice leads to a failure. The available transitions between states enforce the following routine: the apple goes up from a location $\pi LR...R$ to π and then down to $\pi RL...R$. It follows that we visit exactly one leaf node per location, ending up in state p_0^{up} at the root. At this point, the tree of knowledge encodes a unique word of length 2^n. □

The next example puts together the tricks introduced in Examples 3 and 5.

Example 6. Consider an automaton which initially works as in Example 5, but it creates a tree of knowledge of depth $n+1$, rather than n. The set of registers is now larger too: let $M = \{L, R, L_1, R_1, L_2, R_2, \ldots, L_n, R_n, J, Z\}$. At the end of the initial phase we have two main subtrees encoding two binary words x and y of length 2^n (and the apple is at the root).

The automaton can now verify that these two words are identical, using universality. It starts 2^n parallel processes by a universal "for" loop:

$$\text{for } i = 1 \text{ to } n \text{ do } \texttt{set}\ L_i(0)\ \texttt{AND}\ \texttt{set}\ R_i(0). \qquad (**)$$

Executing the loop creates 2^n independent computation branches, each one with a different choice of n bits in the root snake. This choice indicates a certain

location $\pi \in \{L, R\}^n$; let us suppose that $\sharp\pi = \ell$. The goal to be achieved now is to verify that the ℓ-th symbol of x and y is the same. (Remember that for every ℓ we have a dedicated branch of computation.)

The automaton first guesses the bit in question and stores it using one of the registers J, Z, at the root snake. Then it goes down into the left subtree and reaches a leaf addressed by $L\pi$. This requires a sequence of nondeterministic steps but the result of such nondeterminism is fully *determined*. Upon getting there, the machine verifies that the same register, say J, is true at the present leaf and at the root. Then it goes back to the root and repeats the same effort to get down to $R\pi$. It remains to check the appropriate bit there.

The above procedure terminates successfully if and only if all the 2^n processes correctly verify the information at the corresponding leaves of the two subtrees.

Observe that the universal verification routine cannot be replaced by a sequential one (comparing the corresponding nodes pair after pair), because the location u has to be remembered and the space it occupies at the root cannot be re-used.

A binary word of length 2^n can be seen as a number between 0 and $2^{2^n} - 1$. Our next exercise is to verify the successor relation on such numbers (the lexicographic successor on words).

Example 7. We modify the automaton of Example 6 so that it can verify that the word y encoded by the right subtree is the lexicographic successor of the word x encoded by the left subtree. It happens when $y = v10^k$ and $x = v01^k$, for some v and k. We adjust the for loop (**) using the virtual variable *position* (implementable using internal states).

position := *Middle*;
for $i = 1$ to n do if *position* = *Middle* then
((set $L_i(0)$; *position* := *Left*) AND set $R_i(0)$)
OR
((set $L_i(0)$ AND (set $R_i(0)$; *position* := *Right*)))
else (set $L_i(0)$ AND set $R_i(0)$).

As in Example 6, the loop initiates 2^n independent processes, each corresponding to a different location of a leaf in a binary tree of depth n. In exactly one of these processes, the value of *position* is *Middle*. This process identifies a certain location π; processes representing leaves to the left of π have *position* equal to *Left*, to the right of π we have *position* = *Right*. Each of the processes now checks the corresponding bits in x and y at π. Depending on *position*, we expect:

- equal bits in case of *Left*;
- 0 in x, and 1 in y, in case of *Middle*;
- 1 in x, and 0 in y, in case of *Right*. □

At this point the reader should find it obvious that similar procedures can be defined to verify various relations between words encoded as immediate subtrees of the root of a given tree of knowledge. An easy exercise is to check that a given

word consist of only zeros or only ones. A more involved one can be as follows: given three subtrees encoding numbers m_1, m_2, m_3, respectively, assume that the root of each of those subtrees has exactly one among registers $C_0, \ldots, C_r$ set to 1, say $C_{v_i} = 1$ at the root of m_i. For a given function $F : \mathbf{r} \to \mathbf{r}$, verify the equations $m_2 = m_1 + 1$, $m_3 = m_2 + 1$ and that $v_3 = F(v_1, v_2)$.

The following is our most complicated example, revealing the main idea of the proof of Theorem 10. It shows how to create and verify a tree of knowledge with (at least) doubly exponential branching at the root.

Example 8. We define an automaton of depth $n+1$ and dimension $2n+7$. The intended meaning of the snake components is as follows:

- The first two bits encode (as usual) the relative position of every node with respect to its predecessor.
- The next two bits encode a 0 or 1 so that subtrees of depth n may represent words of length 2^n (as in Example 6).
- Three bits represent three "markers": *Ready*, *New*, and *Old*.
- The remaining $2n$ bits are used for comparing two subtrees of depth n, as in Example 7.

The following "program" informally defines the behavior of the automaton. The instruction "construct a word x" abbreviates a procedure as in Example 5: a subtree of depth n, representing a word x, is constructed. Observe that at the end of this procedure the apple is at the root of the newly defined word.

1. Create a new child of the root and descend there;
2. Nondeterministically construct a word x of length 2^n;
3. Go to 4 OR go to 5;
4. Verify that $x = \vec{0}$; go to 12;
5. Go to 6 AND go to 12;
6. Mark top of x as *New*;
7. Go up; descend to the root of some other word x';
8. Check that the top of x' is marked as *Ready*;
9. Mark the top of x' as *Old*;
10. Verify that x is the successor of x';
11. Accept.
12. Mark top of x as *Ready*;
13. Go to 14 OR go to 15;
14. Go up; go to 1;
15. Verify that $x = \vec{1}$;
16. Accept.

Our automaton runs in a loop and creates a certain number of immediate subtrees of the root, each representing a word of length 2^n. This happens in step 1. The purpose of the remaining steps is to make sure that all words of length 2^n will eventually be created (possibly with repetitions). Execution of the loop body is therefore successful only if:

– either the new word x consists of only zeros, or:
– the lexicographic predecessor of x is already present.

We use alternation to verify that a newly generated word satisfies one of the two conditions. In step 3 the machine guesses which of the two cases holds. In the first case, the equality $x = \vec{0}$ is immediately verified in step 4, like in Example 6. This phase begins and ends at the root of x, and when it is successfully completed, the top of x is marked as *Ready*.

In the second case the situation is a little more complex. In step 5 the computation splits into two branches. The main branch proceeds (fingers crossed) directly to step 12, the other verifies if the guess was correct. The whole computation is therefore accepting only if the auxiliary branch can successfully reach step 11.

The verification makes an essential use of the register *Ready* which is now set to 1 at the top of every "old" word but not at the root of x. Thus, in step 8, we can make sure that the root of x' is not the same as the root of x. (That is why this step must be done right after x is created.) In step 10 the automaton executes a routine similar to that in Example 7: the markers *Old* and *New* allow the automaton to identify trees x and x' without confusion. This works because a separate branch of the universal procedure is devoted to any pair of trees being compared, and in every such branch exactly one tree is marked *Old* and exactly one is marked as *New*.

The loop may be exited in step 13, when $x = \vec{1}$. This must be verified in step 15. Note that (as in case of step 4) there is no need for an auxiliary branch of computation, because the apple is at top of x.

Upon a successful completion of the above procedure, the root of the tree has at least 2^{2^n} children, each being a root of a full binary tree of depth n. This is because we must go from 00...0 to 11...1 and verify each successor step.

The bad news is that there may be more than 2^{2^n} children, and some binary words are encoded several times (by unnecessarily repeating the same sequences of guesses in Step 1). It seems that we have no way to prevent such repetitions.

4 Hardness for *co*-2-NEXPTIME

The class *co*-2-NEXPTIME consists of languages recognized by alternating Turing machines, as defined e.g. in [11], but exhibiting purely universal behavior (all states are universal "and"-states)[2] and working in doubly exponential time.

A configuration of such a machine is considered *eventually accepting* when it is either a final configuration (with an accepting state) or all its successor configurations are eventually accepting. A computation can be seen as a tree of configurations, and it is accepting iff all branches end with final configurations. Note that, for a given input word, there is *exactly one* computation, although it may consist of many branches.

[2] We avoid introducing a name for such machines: the most adequate one, "universal TM", has already a well-established, and quite different, meaning.

Our hardness proof is by a direct coding of such a machine, let us therefore fix a machine M and an input word $a_0...a_{n-1}$. For convenience we assume that M admits exactly two transitions for every state and every scanned symbol, i.e., that the transitions of M are defined by a function

$$\delta : (Q \times \Sigma) \to (Q \times \Sigma \times \{-1, 0, +1\})^2,$$

where Q is the the set of states and Σ is the machine alphabet, including the input symbols and a blank. For each q, x, and each "direction" $d \in \{1, 2\}$, we refer to $\pi_d(\delta(q, x))$ as to the *d-th choice for* q, x. That is, computations of our machine are binary trees.

We assume that the machine works on a single tape, infinite to the right, and that it never attempts to move left when scanning the leftmost tape cell. We also assume that the number of configurations on every computation path is exactly 2^{2^n}. It is a routine exercise in the padding technique to see that the halting problem for such machines is *co*-2-NEXPTIME-complete.

We construct an Eden automaton $\mathcal{A}$ of depth $n+2$ to simulate M. The width of $\mathcal{A}$ is $n + c$, where c is a constant large enough to provide registers to encode tape symbols, states, and a few "markers" like *Ready*, *New*, etc.; the meaning of those will be explained later. Then $n + 1$ registers are used to identify locations of leaves as in Examples 6 and 7.

Let $\max = 2^n - 1$. A node w in a tree of knowledge has *address* (t, p) when:

- $0 \leq t, p \leq \max$;
- Node w is a root of a subtree T of depth $n + 1$;
- The subtree T encodes the binary word $t \cdot p$ of length $2^n + 2^n$.

The snake at any node w can represent an element of the set $\Sigma \cup (Q \times \Sigma)$, called a *value* of w. Yet another pair of bits D_1, D_2 is used to store a number 1 or 2, called *direction at* w.

The intended meaning of an address (t, p) and a value a is to represent the contents of tape cell p after t steps of computation of M. Of course we mean t steps along a certain branch of a computation, because the machine is alternating. The direction $d = 1, 2$, indicates that the t-th step was executed according to the d-th choice.

The idea is to write a history of a (single branch of) computation as a tree of knowledge of depth $n + 2$, where each immediate subtree of the root encodes an address, has a value, and a direction. This must be done in a coherent way: for any fixed t, the direction at every address (t, p) must be the same, and if any (t, p) occurs twice, the associated values must be equal. And of course, the collected data should correspond to an actual computation branch. This can be verified locally: indeed, the contents of tape cell p in time $t+1$ is fully determined by the direction used and the contents of at most three tape cells at time t.

It is useful to make this precise. We define a relation $\sqsubset^0$ between addresses so that $(t-1, p-1), (t-1, p), (t-1, p+1) \sqsubset^0 (t, p)$, and we take $\sqsubseteq$ to be the reflexive and transitive closure of $\sqsubset^0$. Clearly, $\sqsubseteq$ is a well-founded partial order.

The algorithm of $\mathcal{A}$, written in pseudocode, is given below. Recall that the machine starts in a root-only tree.

We explain the abbreviations used in the pseudocode. The main loop begins in step 1 when the apple is at the root. The machine creates a new child w of the root and executes a procedure similar to Example 5 to create a subtree of depth $n+1$ rooted at w. Leaves of the subtree are used to encode the word $t \cdot p$, i.e. the address (t, p). The instruction "Check that...", occurring e.g. in step 3, abbreviates a procedure verifying the address associated to the current apple w. This phase begins and ends at w. The phrase "set value" means "set the appropriate register at w, and similarly for the direction. If not stated otherwise, the value and direction are set nondeterministically (but in steps 3–5 the nondeterministic guess must be verified).

The universal step 7 divides the computation into three branches. One proceeds directly to step 22 (end of the loop body), the two others verify the correctness of the guess. In step 8 a node with address $(t, 0)$ is identified and the direction d is checked. (It is expected that all nodes referring to time t should have the same direction.)

The meaning of "Compare...", as e.g. in step 9, is as follows: the value at position (t, p) should be determined by the direction at (t, p) and the values at the positions which are immediate predecessors of (t, p) with respect to relation $\sqsubseteq$. This can be verified using the technique developed in Example 7. Each of the steps 9, 13, and 21 requires a separate computation branch.

1. Create a new subtree with address (t, p);
2. Go to 3 OR 4 OR 5 OR 6 OR 10 OR 14;
3. Check that $t = p = 0$; set value (q_0, a_0); go to 22.
4. Check that $t = 0$ and $0 < p < n$; set value a_p; go to 22.
5. Check that $t = 0$ and $n \leq p \leq \max$; set value blank; go to 22.
6. Check that $t > 0$ and $0 < p < \max$; set value a and direction d;
7. Go to 8 AND goto 9 AND goto 22;
8. Verify that $(t, 0)$ has direction d; accept.
9. Compare with $(t-1, p-1)$, $(t-1, p)$, $(t-1, p+1)$ for the d-th choice; accept.
10. Check that $t > 0$ and $p = \max$; set value a and direction d;
11. Go to 12 AND goto 13 AND goto 22;
12. Verify that $(t, 0)$ has direction d; accept.
13. Compare with $(t-1, \max-1)$, $(t-1, \max)$ for the d-th choice; accept.
14. Check that $t > 0$ and $p = 0$;
15. Go to 16 AND goto 19;
16. Set value a and direction 1;
17. Go to 18 AND goto 22;
18. Compare with $(t-1, 0), (t-1, 1)$ for the 1-st choice; accept.
19. Set value a and direction 2;
20. Go to 21 AND goto 22;
21. Compare with $(t-1, 0), (t-1, 1)$ for the 2-nd choice; accept.
22. Go to 23 OR goto 24;
23. Set *Ready*; go up; go to 1.
24. Check that the current value is (q_f, a); accept.

The main loop works as follows: Starting from a fresh child w of the root, a new tree T of depth $n+1$, rooted at w, is created in step 1, together with a nondeterministically assigned address (t,p). If $t=0$ then the value at w is set according to the initial configuration (steps 3–5). For $t,p\neq 0$, a value and direction is set nondeterministically in step 6 or 10; the latter must be the same as the direction associated with $(t,0)$, cf. step 8. Then the correctness of the value is established by verifying the values at two or three other nodes (steps 9, 13). When $t>0$ and $p=0$ the computation is split into two in step 15: one branch uses direction 1, the other uses direction 2. The iteration terminates in step 24 when a "final" value is discovered.

To make the above construction a little more precise let us say that a tree of knowledge T is *adequate* when:

- It consists of the root and a non-zero number of subtrees of depth $n+1$, each with an address (t,p) and a value;
- Every immediate subtree of the root with $t>0$ has also a direction;
- All but one are marked as *Ready* and their values are not of the form (q_f,a);
- For every t, p, p', if the addresses (t,p) and (t,p') occur in T then they have the same direction;
- Every pair of nodes of the same address have the same value;
- The set of addresses occurring in the tree is downward closed with respect to the relation $\sqsubseteq$.

An adequate tree defines a partial function F_T from addresses to values, and a partial function G_T from addresses to directions. Let $\max_T$ stands for the largest t such that an address of the form (t,p) is in T.

Obviously, every computation branch B of TM defines a partial function F_B from addresses to values, and a partial function G_B from addresses to directions. Let $\max_B\geq 0$ be the number of machine steps along B.

We say that T is *adequate for* B, when $\max_T=\max_B$, $F_T\subseteq F_B$, and $G_T\subseteq G_B$. Note that we do not require equality of the functions; the tree does not have to contain all relevant addresses.

An ID of $\mathcal{A}$ is *adequate* (for B) when the tree of knowledge is adequate (for B), the apple is at the only child of the root not marked as *Ready*, and the machine enters step 22.

Lemma 9. *Assume that $\mathcal{C}$ is an ID of $\mathcal{A}$ adequate for a branch B ending in a configuration Δ of TM. Then Δ is eventually accepting if and only if so is $\mathcal{C}$.*

Proof. The proof from left to right is an easy induction with respect to two parameters: (1) the size of an accepting TM computation starting at Δ, and (2) the cardinality of the difference $\mathrm{Dom}(F_B)-\mathrm{Dom}(F_T)$.

If the value at the current apple is (q_f,a) then $\mathcal{A}$ may accept immediately. Otherwise, if the second parameter is not zero, then there are addresses not yet present in $\mathcal{C}$. Let (t,p) be a minimal such address with respect to $\sqsubseteq$; then we have $t\leq\max_B$. By a single iteration of the main loop (making only correct

choices) the automaton can add a new subtree with address (t, p) and the appropriate value and direction. This reduces the second parameter by one, while the inclusions $F_{T'} \subseteq F_B$ and $G_{T'} \subseteq G_B$ still hold for the expanded tree T'. Otherwise all addresses $(\max_B, p)$ are already present in T. The automaton $\mathcal{A}$ can now correctly guess the address $(\max_B +1, 0)$ and divide its computation into two branches. The IDs commencing the two branches are adequate for the two successor configurations of Δ.

From right to left we go by induction with respect to the size of the accepting computation of $\mathcal{A}$. The base case is when $\mathcal{A}$ goes right away to step 24 and accepts. This means that Δ is an accepting configuration. Otherwise every iteration of the main loop introduces a new address. There is no problem with new addresses (t, p) when $t = 0$ or $p \neq 0$; it is clear that an accepting computation of $\mathcal{A}$ must correctly guess the value and direction, so that the next time step 22 is executed in an ID which is adequate for the same branch B.

If $p = 0$ then the computation splits into two, and if the address $(t, 0)$ is generated for the first time then $t = \max_T +1$. Indeed, the address $(t, 0)$ must be created before any other (t, p) is created, due to step 8. This step exactly corresponds to the branching of the TM. That is, we obtain two IDs adequate for the two extensions of branch B, and we can apply the induction hypothesis to the two new ID's.

Otherwise, the address $(t, 0)$ is already present at the moment of the split. Suppose the direction associated to $(t, 0)$ (and all other addresses (t, p)) is 1. Now a subtree with address $(t, 0)$ is added to both the new IDs. In one of them the direction is set to 1, in the other it is set to 2. The latter is not adequate, because the directions at $(t, 0)$ are inconsistent. But the former is adequate for B, as all the nodes with address $(t, 0)$ have the same direction 1. We apply induction to this ID. □

Theorem 10. *The problem to decide if a given positive first-order formula is constructively provable is hard for universal doubly-exponential time.*

Proof. To show correctness of the simulation one uses Lemma 9 for the initial ID with the empty set of addresses.

Conclusion

We have shown that the "positive" fragment of first-order intuitionistic logic is *co*-2-NEXPTIME-hard. The question of a matching upper bound remains an obvious target for the future work.

A related issue is the complexity of other classes of the Mints hierarchy, where we classify a formula according to the quantifier prefix it would obtain if classically normalized. This is a subject of a forthcoming paper.

Our approach has yet another aspect not developed in this paper. Consider an accepting computation tree of an Ea. This tree morally reflects the shape of a long normal proof of the formula corresponding to the automaton. In this respect we can see our automata as tree acceptors and this suggests a natural method to develop proof theory.

References

1. A.V. Aho and J.D. Ullman. Translations on a context-free grammar. *Information and Control*, 19(5):439–475, 1971.
2. 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.
3. Egon Börger, Erich Grädel, and Yuri Gurevich. *The Classical Decision Problem*. Perspectives in Mathematical Logic. Springer, 1997.
4. R. 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. Michael Kaminski and Tony Tan. Tree automata over infinite alphabets. In A. Avron, N. Dershowitz, and A. Rabinovich, editors, *Pillars of Computer Science, Essays Dedicated to Boris (Boaz) Trakhtenbrot on the Occasion of His 85th Birthday*, volume 4800 of *LNCS*, pages 386–423. Springer, 2008.
10. Gerwin Klein et al. seL4: Formal verification of an OS kernel. *Communications of the ACM*, 53(6):107–115, 2010.
11. Dexter Kozen. *Theory of Computation*. Springer-Verlag, New York, 2006.
12. Daniel Leivant. Monotonic use of space and computational complexity over abstract structures. Technical Report CMU-CS-89-21, Carnegie Mellon University, 1989.
13. Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.
14. G.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.
15. Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.
16. Jens Otten. ileanTAP: An intuitionistic theorem prover. In D. Galmiche, editor, *Proc. of TABLEAUX'97*, volume 1227 of *LNCS*, pages 307–312. Springer, 1997.
17. Ivar Rummelhoff. *Polymorphic Π1 Types and a Simple Approach to Propositions, Types and Sets*. PhD thesis, University of Oslo, 2007.
18. Aleksy Schubert, Paweł Urzyczyn, and Daria Walukiewicz-Chrząszcz. Positive logic is not elementary. Presentation at the Highlights conference, 2013.
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.