RAIRO-Theor. Inf. Appl. 47 (2013) 97–109
DOI: 10.1051/ita/2012030

Available online at:
www.rairo-ita.org

SOME RESULTS ON COMPLEXITY OF μ-CALCULUS EVALUATION IN THE BLACK-BOX MODEL *

Paweł Parys[1]

Abstract. We consider μ-calculus formulas in a normal form: after a prefix of fixed-point quantifiers follows a quantifier-free expression. We are interested in the problem of evaluating (model checking) such formulas in a powerset lattice. We assume that the quantifier-free part of the expression can be any monotone function given by a black-box – we may only ask for its value for given arguments. As a first result we prove that when the lattice is fixed, the problem becomes polynomial (the assumption about the quantifier-free part strengthens this result). As a second result we show that any algorithm solving the problem has to ask at least about n^2 (namely $\Omega\left(\frac{n^2}{\log n}\right)$) queries to the function, even when the expression consists of one μ and one ν (the assumption about the quantifier-free part weakens this result).

Mathematics Subject Classification. 68Q17, 03B70.

1. Introduction

Fast evaluation of μ-calculus expressions is one of the key problems in theoretical computer science. Although it is a very important problem and many people were working on it, no one has been able to show any polynomial time algorithm. On the other hand the problem is in NP∩co-NP, so it may be very difficult to show any lower bound on the complexity. In such situation a natural direction of research is to slightly modify the assumptions and see whether the problem becomes easier.

Keywords and phrases. μ-calculus, black-box model, lower bound, expression complexity.

* *Work supported by Polish government grant no. N N206 380037.*

[1] Institute of Informatics, University of Warsaw, ul. Banacha 2, 02-097 Warszawa, Poland. parys@mimuw.edu.pl

Article published by EDP Sciences

We restrict ourselves to expressions in a quantifier-prefix normal form, namely

$$\theta_1 x_1 . \theta_2 x_2 \ldots \theta_d x_d . f(x_1, \ldots, x_d), \tag{1.1}$$

where $\theta_i = \mu$ for odd i and $\theta_i = \nu$ for even i. We denote expression (1.1) as $\mu\nu(d, f)$. We want to evaluate such expressions in the powerset model or, equivalently, in the lattice $\{0,1\}^n$ with the order defined by $a_1 \ldots a_n \leq b_1 \ldots b_n$ when $a_i \leq b_i$ for all i. The function $f\colon \{0,1\}^{nd} \to \{0,1\}^n$ is an arbitrary monotone function and is given by a black-box (oracle) which evaluates the value of the function for given arguments.

First concentrate on the problem of polynomial *expression complexity*, *i.e.* complexity for a fixed size of the model. We assume that the oracle representing the function answers in time t_f (in other words it is a computational procedure calculating the function in time t_f). To simplify the complexity formulas assume that $t_f \geq \Theta(nd)$, *i.e.* that the procedure at least reads its arguments. One can easily evaluate expression (1.1) in time $O(n^d \cdot t_f)$; this can be done by naive iterating [4]. We show that, using a slightly modified version of the naive iterating algorithm, the complexity can be $O\left(\binom{n+d}{d} \cdot t_f\right)$. For big n it does not improve anything, however for fixed n the complexity is equal to $O(d^n \cdot t_f)$, hence is polynomial in d. This is our first result, described in Sections 2 and 3.

Theorem 1.1. *There is an algorithm which for any fixed model size n calculates the value of expression* (1.1) *in time polynomial in d and t_f, namely $O(d^n \cdot t_f)$.*

Our result slightly extends an unpublished result in [7]. The authors also get polynomial expression complexity, however using completely different techniques. Our result is stronger, since they consider only expressions in which f is given by a vectorial Boolean formula, not as an arbitrary function. Moreover their complexity is slightly higher: $O(d^{2n} \cdot |f|)$.

As a side remark recall two results about parallel complexity of the modal μ-calculus model checking problem (where we allow an arbitrary formula, not necessarily of form (1.1)). The problem is PTIME-hard not only when considering combined complexity [9], but already for the expression complexity [3]. This somehow contradicts with the intuition that for fixed model the problem should become easy.

Our second result is an almost quadratic lower bound for $d = 2$. It was possible to achieve any lower bound thanks to the assumption that the algorithm may access the function f in just one way: by evaluating its value for given arguments. Moreover, we are not interested in the exact complexity, only in the number of queries to the function f. In other words we consider decision trees: each internal node of the tree is labeled by an argument, for which the function f should be checked, and each of its children corresponds to a possible value of f for that argument. The tree has to determine the value of expression (1.1): for each path from the root to a leaf there is at most one possible value of (1.1) for all functions which are consistent with the answers on that path. We are interested in the height of such trees, which justifies the following definition.

Definition 1.2. For any natural number d and finite lattice L we define $num(d, L)$ as the minimal number of queries, which has to be asked by any algorithm correctly calculating expression (1.1) basing only on queries to the function $f\colon L^d \to L$.

In this paper we consider only the case $d = 2$. We show that almost n^2 queries are necessary in that case. Precisely, we have the following result, described in Sections 4 to 6.

Theorem 1.3. *For any natural n it holds* $num(2, \{0,1\}^n) = \Omega\left(\frac{n^2}{\log n}\right)$.

This result is a first step towards solving the general question, for any d. It shows that in the black-box model something may be proved. Earlier it was unknown even if more than nd queries are needed for any d. Note that $num(1, \{0,1\}^n)$ is n (as well as in the case of d operators μ and no ν operators it is enough to do n queries). So the result gives an example of a situation where the alternation of fixed-point quantifiers μ and ν is provably more difficult than just one type of quantifiers μ or ν. Although it is widely believed that the alternation should be a source of algorithmic complexity, the author is not aware of any other result showing this phenomenon, except the result in [2].

Let us comment on the way how the function f is given. We consider very general algorithms, which make very weak assumptions: the function can be given by an arbitrary program. This is called a black-box model, and was introduced in [6]. In particular our formulation covers vectorial Boolean formulas, as well as modal formulas in a Kripke structure of size n. Moreover our framework is more general, since not every monotone function can be described by a modal formula of small size, even when it can be computed quickly by a procedure. Thus our assumptions strengthens the upper bound (Thm. 1.1) and weakens the lower bound (Thm. 1.3). Note that the algorithm in [6], working in time $O(n^{\lfloor d/2 \rfloor + 1} \cdot t_f)$, can also be applied to our setting. On the other hand the recent algorithms, from [8] working in time $O(m^{d/3})$ and from [5] working in time $m^{O(\sqrt{m})}$ (where $m \geq n$ depends on the size of f), use the parity games framework, hence require that f is given by a Boolean or modal formula of small size. This can be compared to models of sorting algorithms. One possible assumption is that the only way to access the data is to compare them (this is similar to our black-box model). Then an $\Omega(n \log n)$ lower bound can be proved. Most of the sorting algorithms work in this framework. On the other hand, when the data can be accessed directly, faster algorithms are possible (like $O(n)$ for strings and $O(n \log \log n)$ for integers).

It is known that for a given structure an arbitrary μ-calculus formula can be converted to a formula of form (1.1) in polynomial time, see Section 2.7.4 in [1]. Hence, a polynomial algorithm evaluating expressions of form (1.1) immediately gives a polynomial algorithm for arbitrary expressions. However during this conversion one also needs to change the underlying structure to one of size nd, where d is the nesting level of fixed-point quantifiers. So, even when the original model has fixed size n, after the normalization the model can become very big, and our algorithm from Theorem 1.1 gives exponential complexity.

2. The iterating algorithm

Let us first fix some notation. For $x \in \{0,1\}^n$ denote the number of bits in x which are set to 1 by $b(x)$. Moreover, let f_i be the function represented by the part of expression (1.1) starting from $\theta_i x_i$:

$$f_i(x_1, \ldots, x_{i-1}) = \theta_i x_i . \theta_{i+1} x_{i+1} \ldots \theta_{d-1} x_{d-1} . \theta_d x_d . f(x_1, \ldots, x_{i-1}, x_i, \ldots x_d).$$

In particular $f_{d+1} = f$ and $f_1()$ is the value $\mu\nu(d, f)$, which we want to calculate. Recall that each f_i is a monotone function.

Below we present a general version of the iterating algorithm evaluating expression (1.1). The algorithm can be described by a series of recursive procedures, one for each fixed-point operator; the goal of a procedure $\mathsf{Calculate}_i(x_1, \ldots, x_{i-1})$ is to calculate $f_i(x_1, \ldots, x_{i-1})$.

$\mathsf{Calculate}_i(x_1, \ldots, x_{i-1})$:
  $x_i = \mathsf{Initialize}_i(x_1, \ldots, x_{i-1})$
  repeat
    $x_i = \mathsf{Calculate}_{i+1}(x_1, \ldots, x_i)$
  until x_i stops changing
  return x_i

The most internal procedure $\mathsf{Calculate}_{d+1}(x_1, \ldots, x_d)$ simply returns $f(x_1, \ldots, x_d)$. To evaluate the whole expression we simply call $\mathsf{Calculate}_1()$.

Till now we have not specified the $\mathsf{Initialize}_i$ procedures. First assume that they always return $00\ldots0$ for odd i and $11\ldots1$ for even i. Then we simply get the naive iterating algorithm from [4]. However we would like to make use of already done computations and start an iteration from values which are closer to the fixed-point. Of course we can not start from an arbitrary value. The following standard lemma gives conditions under which the computations are correct.

Lemma 2.1. *Assume that the values of x_i returned by $\mathsf{Initialize}_i$ satisfy*

$$x_i \leq f_i(x_1, \ldots, x_{i-1}), \qquad \text{and} \tag{2.1}$$
$$x_i \leq f_{i+1}(x_1, \ldots, x_{i-1}, x_i) \tag{2.2}$$

for odd i, and

$$x_i \geq f_i(x_1, \ldots, x_{i-1}), \qquad \text{and} \tag{2'}$$
$$x_i \geq f_{i+1}(x_1, \ldots, x_{i-1}, x_i). \tag{3'}$$

for even i. Then the procedure $\mathsf{Calculate}_i(x_1, \ldots, x_{i-1})$ calculates $f_i(x_1, \ldots, x_{i-1})$. Moreover, for $1 \leq i \leq d$, at each step of the repeat-until loop x_i increases and satisfies properties (2.1) and (2.2) for odd i (decreases and satisfies properties (2') and (3') for even i).

Proof. For $i = d + 1$ the first part is obviously true. For $i \leq d$ the proof is by induction on the order in which the procedures return. As the cases of odd and even i are symmetric, assume that i is odd. Recall that in this case we calculate a μ fixed-point. First observe that property (2.1) is preserved during the iterations:

$$f_{i+1}(x_1, \ldots, x_i) \leq f_{i+1}(x_1, \ldots, x_{i-1}, f_i(x_1, \ldots, x_{i-1})) = f_i(x_1, \ldots, x_{i-1}).$$

The inequality follows from monotonicity of f_{i+1} and (2.1) before the iteration, while the equality is true because the value of f_i is a fixed-point of f_{i+1}. We have also used the induction assumption to know that $\mathsf{Calculate}_{i+1}(x_1, \ldots, x_i) = f_{i+1}(x_1, \ldots, x_i)$. Property (2.2) is even simpler,

$$f_{i+1}(x_1, \ldots, x_i) \leq f_{i+1}(x_1, \ldots, x_{i-1}, f_{i+1}(x_1, \ldots, x_i)),$$

it follows from monotonicity of f_{i+1} and (2.2) before the iteration.

Property (2.2) guarantees that x_i is increased at each iteration. After some number of steps it has to stabilize. Then $x_i = f_{i+1}(x_1, \ldots, x_i)$ is a fixed-point. From (2.1) we know that x_i is $\leq f_i(x_1, \ldots, x_{i-1})$. Moreover it can not be strictly smaller than $f_i(x_1, \ldots, x_{i-1})$, because then we would have a fixed-point smaller than the smallest fixed-point. So x_i stabilizes at $f_i(x_1, \ldots, x_{i-1})$. □

The second lemma more precisely indicates possible results of $\mathsf{Initialize}_i$: it says that it may be a previously calculated value of the expression for smaller/greater argument.

Lemma 2.2. *If $x_i = f_i(x'_1, \ldots, x'_{i-1})$ for some $x'_1 \leq x_1, \ldots, x'_{i-1} \leq x_{i-1}$ then conditions (2.1) and (2.2) hold. By symmetry, if $x_i = f_i(x'_1, \ldots, x'_{i-1})$ for some $x'_1 \geq x_1, \ldots, x'_{i-1} \geq x_{i-1}$ then conditions (2') and (3') hold.*

Proof. We prove the first part of the lemma (which is useful for odd i). Property (2.1) simply follows from monotonicity of f_i. To get (2.2) we use the fact that the x_i, as a value of f_i, is a fixed-point of f_{i+1}, and the monotonicity of f_{i+1}:

$$x_i = f_{i+1}(x'_1, \ldots, x'_{i-1}, x_i) \leq f_{i+1}(x_1, \ldots, x_{i-1}, x_i).$$ □

Furthermore, observe that conditions (2.1) and (2.2) (respectively, (2') and (3')) trivially hold for $x_i = 00\ldots0$ ($x_i = 11\ldots1$).

3. The algorithm with polynomial expression complexity

To speed up the algorithm we need to somehow remember already calculated values of expressions and use them later as a starting value, when the same expression for greater/smaller arguments is going to be calculated. Instead of remembering all the results calculated so far in some sophisticated data structure, we do a very simple trick. We just take

$$\mathsf{Initialize}_i(x_1, \ldots, x_{i-1}) = \begin{cases} 00\ldots0 & \text{for } i = 1, \\ 11\ldots1 & \text{for } i = 2, \\ x_{i-2} & \text{for } i \geq 3. \end{cases}$$

First we will argue why this is really correct. Precisely, in the light of the above lemmas, we need to prove that while initializing x_i by x_{i-2} it holds

- $x_{i-2} = f_i(x'_1, \dots, x'_{i-1})$ for some $x'_1 \le x_1, \dots, x'_{i-1} \le x_{i-1}$ or $x_{i-2} = 00\dots0$ for odd i, and
- $x_{i-2} = f_i(x'_1, \dots, x'_{i-1})$ for some $x'_1 \ge x_1, \dots, x'_{i-1} \ge x_{i-1}$ or $x_{i-2} = 11\dots1$ for even i.

The proof is by induction on the order in which instructions are executed. Take any moment in which we enter a $\mathsf{Calculate}_i$ procedure, and assume that the thesis was true before. Moreover assume that i is odd, as the other case is symmetric. There are two cases depending on where the current value of x_{i-2} was set:

1. We are for the first time in the repeat-until loop in $\mathsf{Calculate}_{i-2}$. Then there are two subcases. First, it is possible that $x_{i-2} = 00\dots0$; then the thesis trivially holds. Otherwise, by induction assumption we know that $x_{i-2} = f_{i-2}(x'_1, \dots, x'_{i-3})$ for some $x'_1 \le x_1, \dots, x'_{i-3} \le x_{i-3}$. But the value of f_{i-2} is a fixed-point of f_{i-1}, and a value of f_{i-1} is a fixed-point of f_i, so

$$x_i = x_{i-2} = f_{i-1}(x'_1, \dots, x'_{i-3}, x_{i-2}) = f_i(x'_1, \dots, x'_{i-3}, x_{i-2}, x_{i-2}).$$

   From Lemma 2.1, condition (2') we know that $x_{i-1} \ge f_{i-1}(x_1, \dots, x_{i-2})$, and by monotonicity of f_{i-1}

$$f_{i-1}(x_1, \dots, x_{i-3}, x_{i-2}) \ge f_{i-1}(x'_1, \dots, x'_{i-3}, x_{i-2}) = x_{i-2}.$$

   Hence $x_{i-2} \le x_{i-1}$, so really x_i is initialized with a value of f_i for some arguments smaller than $x_1, \dots, x_{i-1}$.
2. We are not for the first time in the repeat-until loop in $\mathsf{Calculate}_{i-2}$. Then the value of x_{i-2} was set in the previous iteration of the loop, and is equal to $f_{i-1}(x_1, \dots, x_{i-3}, x'_{i-2})$, where x'_{i-2} is the previous value of x_{i-2}. Moreover, since a value of f_{i-1} is a fixed-point of f_i, we have $x_i = x_{i-2} = f_i(x_1, \dots, x_{i-3}, x'_{i-2}, x_{i-2})$. From Lemma 2.1 we know that $x'_{i-2} \le x_{i-2}$ (that x_{i-2} increases). Moreover, from Lemma 2.1, condition (2') we know that $x_{i-1} \ge f_{i-1}(x_1, \dots, x_{i-2})$, and by monotonicity of f_{i-1}

$$f_{i-1}(x_1, \dots, x_{i-3}, x_{i-2}) \ge f_{i-1}(x_1, \dots, x_{i-3}, x'_{i-2}) = x_{i-2}.$$

   Hence $x'_{i-2} \le x_{i-2}$ and $x_{i-2} \le x_{i-1}$, so really x_i is initialized with a value of f_i for some arguments smaller than $x_1, \dots, x_{i-1}$.

By now we already know that the algorithm is correct, let now analyze its complexity. A key idea of this analysis is placed in the following lemma.

Lemma 3.1. *Arguments of each call to* $\mathsf{Calculate}_{d+1}$ *satisfy*

$$x_1 \le x_3 \le \dots \le x_{d-3} \le x_{d-1} \le x_d \le x_{d-2} \le \dots \le x_4 \le x_2.$$

Proof. First observe the inequality $x_{d-1} \leq x_d$. In fact it follows from the above proof: in both cases we had there $x_{i-2} \leq x_{i-1}$ for odd i. Although it was only for $i \leq d-1$, the same proof is correct also for $i = d+1$.

All the other inequalities immediately follow from Lemma 2.1: each x_i is initialized with x_{i-2} and then increased for odd i and decreased for even i. □

To determine the complexity it is enough to look at numbers of bits set to 1 in the variables. We have

$$0 \leq b(x_1) \leq b(x_3) \leq \cdots \leq b(x_{d-3}) \leq b(x_{d-1}) \leq \\ \leq b(x_d) \leq b(x_{d-2}) \leq \cdots \leq b(x_4) \leq b(x_2) \leq n. \tag{3.1}$$

Now look at the sequences

$$b(x_1), -b(x_2), b(x_3), -b(x_4), \ldots, b(x_{d-1}), -b(x_d)$$

and notice that at each call to $\mathsf{Calculate}_{d+1}$ we get a lexicographically greater sequence. Indeed, when between two consecutive calls we exit back to procedure $\mathsf{Calculate}_i$, then $b(x_1), \ldots, b(x_{i-1})$ stay the same, $b(x_i)$ is increased (for odd i) or decreased (for even i), while $b(x_{i+1}), \ldots, b(x_d)$ may become arbitrary. Hence at each call to $\mathsf{Calculate}_{d+1}$ we have a different sequence satisfying (3.1). There are $\binom{n+d}{d}$ such sequences, so we spend time $O\left(\binom{n+d}{d} \cdot t_f\right)$ inside $\mathsf{Calculate}_{d+1}$. Notice that as an effect of calling each $\mathsf{Calculate}_i$ we will at least once call $\mathsf{Calculate}_{d+1}$, so there are $O\left(\binom{n+d}{d} \cdot d\right)$ calls to any procedure. In each call we spend time $O(n)$ plus time of executing called subprocedures (we have $O(n)$ because we need to initialize and return x_i, which is of size n). So the total complexity is $O\left(\binom{n+d}{d} \cdot (nd + t_f)\right)$. Under a natural assumption that $t_f \geq \Theta(nd)$, *i.e.* that f at least reads all its arguments, the complexity becomes $O\left(\binom{n+d}{d} \cdot t_f\right)$.

4. Changing the lattice

Now we come to a Proof of Theorem 1.3. In this section we reduce the problem from the lattice $\{0,1\}^n$ to some more convenient lattice. In Section 5 we define a family of difficult functions f. In Section 6 we finish the proof of the theorem.

Instead of the lattice $\{0,1\}^n$ it is convenient to use a better one. Take the alphabet Γ_n consisting of letters a_i for $1 \leq i \leq \frac{n(n+1)}{2} + 1$ and the alphabet $\Sigma_n = \{0,1\} \cup \Gamma_n$. We introduce the following partial order on it: the letters a_i are incomparable; the letter 0 is smaller than all other letters; the letter 1 is bigger than all other letters. We will be considering sequences of n such letters, *i.e.* the lattice is Σ_n^n. The order on the sequences is defined as previously: $a_1 \ldots a_n \leq b_1 \ldots b_n$ when $a_i \leq b_i$ for all i.

We formulate a general lemma, which allows to change a lattice in our problem. For any two lattices L_1, L_2 we say that $h\colon L_1 \to L_2$ is an *order-preserving* map, when it preserves the order, *i.e.* $x \leq y$ implies $h(x) \leq h(y)$.

Lemma 4.1. *Let L_1, L_2 be two finite lattices and $enc\colon L_1 \to L_2$ and $dec\colon L_2 \to L_1$ two order-preserving maps such that $dec \circ enc = id_{L_1}$. Then $num(d, L_1) \leq num(d, L_2)$.*

Proof. In other words we should be able to use any algorithm calculating $\mu\nu(d, f)$ in L_2 to calculate $\mu\nu(d, f)$ in L_1. Let $f^1\colon L_1^d \to L_1$ be the unknown function in L_1. We define $f^2\colon L_2^d \to L_2$ as $f^2(x_1, \dots, x_d) = enc(f^1(dec(x_1), \dots, dec(x_d)))$. Note that f^2 is a monotone function if f^1 was monotone, since *enc* and *dec* preserve the order.

Let $\bot_1, \bot_2, \top_1, \top_2$ be the minimal and maximal elements in L_1 and L_2. For any $x \in L_1$ we have $\bot_2 \leq enc(x)$, so $dec(\bot_2) \leq dec(enc(x)) = x$, which means that $dec(\bot_2) = \bot_1$. Similarly $dec(\top_2) = \top_1$.

Note that $dec(\mu\nu(d, f^2)) = \mu\nu(d, f^1)$. This is true, because these fixpoint expressions may be replaced by a term containing applications of f and minimal and maximal elements. This is done in a classic way, we replace the fixpoint operators by an iterated nesting. The minimal required number of iterations depends on the structure. Here we have only two structures, L_1 and L_2, so we may take the bigger of the two minimal numbers. Hence we may use the same term in L_1 and L_2, the difference is if we use f^1 or f^2, $\bot_1$ or $\bot_2$, $\top_1$ or $\top_2$. Then an easy induction on the term structure shows that $dec(\mu\nu(d, f^2)) = \mu\nu(d, f^1)$, because $dec(\bot_2) = \bot_1$, $dec(\top_2) = \top_1$, $dec(f^2(x_1, \dots, x_d)) = f^1(dec(x_1), \dots, dec(x_d))$. So to find $\mu\nu(d, f^1)$ it is enough to find $\mu\nu(d, f^2)$, which may be found for any f^2 in $num(d, L_2)$ queries to f^2. To evaluate f^2 in our case it is enough to do one query to f^1. Hence $\mu\nu(d, f^1)$ may be found in $num(d, L_2)$ queries (or maybe less queries in some other way). □

For the lattice Σ_n^n we have the following result, from which Theorem 1.3 follows:

Lemma 4.2. *For any natural n it holds $num(2, \Sigma_n^n) \geq \frac{n(n+1)}{2}$.*

This lemma is proved in the next two sections. Here we show how Theorem 1.3 follows from it.

Proof (Thm. 1.3). We will show how Theorem 1.3 follows from this lemma. Take k such that $\binom{2k}{k} \geq \frac{n(n+1)}{2} + 1$. From the Stirling formula follows that $\binom{2k}{k}$ grows exponentially in k, so we may have $k = O(\log n)$. Take $m = \left\lfloor \frac{n}{2k} \right\rfloor$. From Lemma 4.2 for m we see that $num(2, \Sigma_m^m) \geq \frac{m(m+1)}{2} = \Omega\left(\frac{n^2}{\log n}\right)$.

Now it is enough to use Lemma 4.1 to see that $num(2, \{0,1\}^n) \geq num(2, \Sigma_m^m)$. We need to define functions $enc\colon \Sigma_m^m \to \{0,1\}^n$ and $dec\colon \{0,1\}^n \to \Sigma_m^m$. Each letter from Σ_m will be encoded in a sequence of $2k$ letters from $\{0,1\}$ in the following way: 0 is translated to the sequence of $2k$ zeroes, 1 to the sequence of $2k$ ones, any of the letters a_i is translated to some sequence of $2k$ bits, in which exactly k bits are equal to 1. Because $n \geq m$ we have $\binom{2k}{k} \geq \frac{m(m+1)}{2} + 1$, so there are enough different such sequences to encode all letters. We use this encoding to

define $enc(x)$: an i-th letter of x is encoded in the i-th fragment of $2k$ bits and the final $n-2km$ bits are set to zeroes. On the other hand to read an i-th letter of the value of $dec(y)$, we look at the i-th fragment of $2k$ bits: when it corresponds to one of the letters a_i, this a_i is the result; otherwise the result is 0 or 1 depending on whether there are less than k ones in the sequence or not. Note that dec is defined on all sequences, not only on results of enc. It is easy to see that $dec(enc(x)) = x$ for any $x \in \Sigma_m^m$ and that both functions are order-preserving maps (mainly because encodings of different letters a_i are incomparable). □

5. Difficult functions

In this section we define a family of functions used in the proof of Lemma 4.2. As we consider only $d = 2$, to avoid the indexes we use y instead of x_1 and x instead of x_2. A function $f_{z,\sigma}\colon \Sigma_n^{2n} \to \Sigma_n^n$ is parametrized by a sequence $z \in \Gamma_n^n$ (which will be the result of $\mu y.\nu x.f_{z,\sigma}(y,x)$) and by a permutation $\sigma\colon \{1,\ldots,n\} \to \{1,\ldots,n\}$ (which is an order in which the letters of z are uncovered). Note that z is from Γ_n^n, not from Σ_n^n, so it can not contain 0 or 1, just the letters a_i. Whenever z and σ are clear from the context, we simply write f. In the following the i-th element of a sequence $x \in \Sigma_n^n$ is denoted by $x[i]$. A pair z, σ defines a sequence of values $y_0, \ldots, y_n$:

$$y_k[i] = \begin{cases} z[i] & \text{for} \quad \sigma^{-1}(i) \le k \\ 0 & \text{otherwise.} \end{cases}$$

In other words y_k is equal to z, but with some letters covered: they are 0 instead of the actual letter of z. In y_k there are k uncovered letters; the permutation σ defines the order, in which the letters are uncovered. Using this sequence of values we define the function. In some sense the values of the function are meaningful only for $y = y_k$, we define them first (assuming $y_{n+1} = y_n$):

$$f(y_k,x)[i] = \begin{cases} 0 & \text{if } \forall_{j>i} x[j] \le y_{k+1}[j] \quad \text{and} \quad x[i] \not\ge y_{k+1}[i] \quad \text{(Case 1)} \\ y_{k+1}[i] & \text{if } \forall_{j>i} x[j] \le y_{k+1}[j] \quad \text{and} \quad x[i] \ge y_{k+1}[i] \quad \text{(Case 2)} \\ x[i] & \text{if } \exists_{j>i} x[j] \not\le y_{k+1}[j] \quad \text{(Case 3).} \end{cases}$$

For any other node y we look for the lowest possible k such that $y \le y_k$ and we put $f(y,x) = f(y_k,x)$. When such k does not exists ($y \not\le z$), we put $f(y,x)[i] = 1$.

Lemma 5.1. *The function f is monotone and $\mu y.\nu x.f(y,x) = z$.*

Proof. First see what happens when we increase x: for any x, x', y, i such that $x' \ge x$, we want to prove that $f(y,x')[i] \ge f(y,x)[i]$. When $y \not\le z$, for both x and x' we get the same result 1. Otherwise we have to compare $f(y_k,x')[i]$ and $f(y_k,x)[i]$ (for some k). Whenever for x and x' we are in the same case of the function definition, we get the required inequality. Also when for x we have an earlier case than for x' it is OK (in particular when for x we have Case 2, it holds $x'[i] \ge x[i] \ge y_{k+1}[i]$). On the other hand it is impossible, that for x' we get an

earlier case than for x (it is easy to see looking at the conditions for choosing a case).

Now see what happens, when we increase y: take $y' \geq y$. When for y' there is $y' \not\leq z$, we get a result 1, which is bigger than anything else. Otherwise the values y_k and $y_{k'}$ chosen for y and y' satisfy $y_{k'} \geq y_k$, so also $y_{k'+1} \geq y_{k+1}$. The argumentation that in such case $f(y_{k'}, x)[i] \geq f(y_k, x)[i]$ is identical as for the change of x.

To calculate the fixpoint expression, first see that $\nu x.f(y_k, x) = y_{k+1}$. It follows immediately from the definition: $f(y_k, y_{k+1}) = y_{k+1}$ and for any $x > y_{k+1}$ we get $f(y_k, x) \neq x$, because $f(y_k, x)$ differs from x on the last position i where $x[i] > y_{k+1}[i]$, we get there $y_{k+1}[i]$ instead of $x[i]$. The main fixpoint satisfies $\mu y.\nu x.f(y, x) = y_n = z$, because $y_{k+1} > y_k$ for all $k < n$ and $y_{n+1} = y_n$. □

6. The proof

Now we will show that at least $\frac{n(n+1)}{2}$ queries are needed to calculate the value of $\mu y.\nu x.f(y, x)$, even if we allow as f only functions from our family. The problem can be considered as a game between two players, we call them an algorithm and an oracle. In each round the algorithm player asks a query to the function, after which the oracle player chooses an answer (which is consistent with the previous answers). The algorithm player wins if after $\frac{n(n+1)}{2} - 1$ steps each function consistent with the answers has the same value of $\mu y.\nu x.f(y, x)$. Otherwise the oracle player wins. We have to show a winning strategy for the oracle player.

First see informally what may happen. Consider first a standard algorithm evaluating fixpoint expressions. It starts from $y = y_0 = 0 \dots 0$ and $x = 1 \dots 1$. Then it repeats $x := f(y, x)$ until x stops changing, in which case $x = \nu x.f(y, x)$. For our functions it means that in each step the last 1 in x is replaced by the corresponding letter of y_1. The loop ends after n steps with $x = y_1$. Then the algorithm does $y := x$, $x := 1 \dots 1$, and repeats the above until y stops changing. For any $y = y_k$ the situation is very similar: in each step the last 1 in x is replaced by the corresponding letter of y_{k+1} (we may say that this letter is uncovered).

In fact, by choosing appropriate x the algorithm may decide which letter of y_{k+1} he wants to uncover, but always at most one. For the algorithm only the letter on which y_{k+1} differs from y_k is important, as he already knows all letters of y_k. However the difference may be on any position on which y_k has 0 (it depends on σ). The oracle player may choose this position in the most malicious way: whenever the algorithm player uncovers some letter, the oracle decides that this is not the letter on which y_k and y_{k+1} differs. So the algorithm has to try all possibilities (all positions on which y_k has 0), which takes $\frac{n(n+1)}{2}$ steps. He may also ask for some other y. It can give him any profit only if he accidentally guesses some letters of z. However the oracle may always decide that the guess of the algorithm is incorrect (that the value of z is different).

Now we come to a more formal proof. We show a strategy for the oracle player. During the game we (the oracle player) keep a variable cur ($0 \leq cur < n$), which is

equal to 0 at the beginning and is increased during the game. Intuitively it means how many letters of z are already known to the algorithm player. By s we denote the number of queries already asked (it increases by 1 after each query) and by s_{lok} the number of queries asked for this value of cur (it increases by 1 after each query and is reset to 0 when cur changes).

At every moment we keep a set F of functions consistent with all the answers till now (there may be more consistent functions, but each function in our set has to be consistent). The set will be described by a set of permutations Π and by sets of allowed values $A_i \subseteq \Gamma_n$, one for each coordinate $1 \leq i \leq n$. The sets should satisfy the following conditions:

1. for each $i \leq cur$ there is only one value of $\sigma(i)$ for $\sigma \in \Pi$;
2. in Π there are permutations σ with at least $n - cur - s_{lok}$ different values of $\sigma(cur + 1)$;
3. for each permutation $\sigma \in \Pi$ when we take any other permutation σ' which agrees with σ on the first $cur + 1$ arguments ($\sigma(i) = \sigma'(i)$ for each $1 \leq i \leq cur + 1$), we have $\sigma' \in \Pi$ as well;
4. for each $\sigma \in \Pi$ and $i \leq cur$ there is only one value in $A_{\sigma(i)}$ (note that thanks to condition 1, the value $\sigma(i)$ does not depend on the choice of σ);
5. for each $\sigma \in \Pi$ and $i > cur$ there are at least $\frac{n(n+1)}{2} + 1 - s$ values in the set $A_{\sigma(i)}$ (note that the set $\{\sigma(i) : i > cur\}$ does not depend on the choice of σ, as $\sigma(i)$ for $i \leq cur$ are fixed).

In the set F there are all functions $f_{z,\sigma}$ for which $\sigma \in \Pi$ and $z[i] \in A_i$ for each i. We see that in particular at the beginning all functions are in the set F. Note, that at each moment the value of y_{cur} is fixed, *i.e.* is the same for all functions in F (because $\sigma(i)$ and $z[\sigma(i)]$ are fixed for $i \leq cur$).

Now we specify how the answers are done for a query x, y. Whenever $y \leq y_i$ for some $i < cur$, we answer according to all the functions in our set F. The answer of each function is the same, as it depends only on the value of y_{i+1} (for the smallest i such that $y \leq y_i$), which is already the same for all functions. Such question does not give any new knowledge to the algorithm player.

Whenever $y \not\leq y_{cur}$, we remove the value $y[i]$ from the set A_i (only if it was there, in particular only if $y[i] \in \Gamma_n$) for each i such that $\sigma^{-1}(i) > cur$ for any $\sigma \in \Pi$ (note that once again this condition is satisfied for exactly the same i for every permutation in Π). All the conditions of F are still satisfied, as we removed only one value from the sets A_i after one additional query was done. In other words we remove all functions $f_{z,\sigma}$, in which $z[\sigma(i)] = y[\sigma(i)]$ for some $i > cur$. Then for each function from F we have $y \not\leq z$ (if $y \leq z$ then $y[i] = 0$ for each i with $\sigma^{-1}(i) > cur$, which means that $y \leq y_{cur}$). So we reply to the query by a sequence of ones, which is the case for all the functions in F. Intuitively this case talks about a situation when someone tries to guess z (or its part) instead of gently asking for $y = y_{cur}$. We prefer to answer that his guess was incorrect and to eliminate all functions with z similar to the y about which he asked.

Consider now the case when $y \leq y_{cur}$ but $y \not\leq y_{cur-1}$. Let ask be the greatest number such that $x[ask] \not\leq y_{cur}[ask]$ (if there is no such number we take $ask = 0$).

Intuitively the algorithm player asks whether $\sigma(cur+1) = ask$; we prefer to answer NO, so he will have to try all the possibilities until he will discover the value of $\sigma(cur+1)$. The first case is when in Π there are permutations with $\sigma(cur+1) \neq ask$. Note that this is true at least $n-cur-1$ times for this cur due to condition 2. In such case we remove from Π all the permutations with $\sigma(cur+1) = ask$ and we answer according to all the functions left in F. We have to argue that for each of them the answer is the same. On positions $i < ask$ there is always Case 3, because $x[ask] \not\leq y_{cur}[ask] = y_{cur+1}[ask]$ (the equality is true, because $\sigma(cur+1) \neq ask$). On the positions $i \geq ask$ there is $x[j] \leq y_{cur}[j] \leq y_{cur+1}[j]$ for $j > i$, so we fall into the first two cases. For $i = ask$ the result depends only on $y_{cur+1}[ask]$ which for all functions is equal to $y_{cur}[ask]$. Consider positions $i > ask$. When $x[i] = 0$ the answer is 0 in both Cases 1 and 2 (we may get Case 2 only when $y_{cur+1}[i] = 0$). When $x[i] > 0$ it has to be $y_{cur+1}[i] = x[i]$, as $x[i] \leq y_{cur}[i] \leq y_{cur+1}[i] \neq 1$, we get Case 2 and we answer $y_{cur+1}[i] = x[i]$.

The last case is when all the permutations $\sigma \in \Pi$ have $\sigma(cur+1) = ask$. Then we choose any letter from A_{ask} and we remove all other letters from A_{ask}. In other words y_{cur+1} becomes fixed, so answers for all the functions left in F are the same. We increase cur (when cur becomes equal to n, we fail). It is easy to see, that all the conditions on the set F are still satisfied.

As already mentioned, before the last case holds there has to be $n - cur - 1$ earlier queries for this cur (we increase cur after at least $n-cur$ queries), so before $\frac{n(n+1)}{2}$ queries there is no danger that cur becomes equal to n. Moreover we are sure that in F there are functions with two different value of z (which is the result of the fixpoint expression): it is enough to take any σ from Π and then in $A_{\sigma(n)}$ there are at least two values ($z[\sigma(n)]$ may be equal to both of them).

7. Concluding remarks

There are two natural future directions of research. First, it is very interesting to study whether the polynomial expression complexity can be shown for arbitrary formulas (not being in the normalized form (1.1)), or whether the problem is then equivalent to model checking in an arbitrary model. The second goal is to get an exponential lower bound for an arbitrary number of fixed-point operator alternations in the formula. We do not see how to generalize our approach to cases of greater alternation depth. Natural generalizations of the functions from this paper are no longer monotone; some new ideas are needed. Notice also that for $d = 3$ we still have a quadratic algorithm, so a cubic lower bound can be obtained only for $d \geq 4$.

Acknowledgements. The author would like to thank Igor Walukiewicz for suggesting this topic and many useful comments.

References

[1] A. Arnold and D. Niwiński, Rudiments of μ-Calculus, *Studies in Logic and the Foundations of Mathematics*. North Holland **146** (2001).

[2] A. Dawar and S. Kreutzer, Generalising automaticity to modal properties of finite structures. *Theor. Comput. Sci.* **379** (2007) 266–285.

[3] S. Dziembowski, M. Jurdziński and D. Niwiński, On the expression complexity of the modal μ-calculus model checking, unpublished manuscript.

[4] E.A. Emerson and C.-L. Lei, Efficient model checking in fragments of the propositional mu-calculus (extended abstract), in *Proc. of 1st Ann. IEEE Symp. on Logic in Computer Science, LICS '86 Cambridge, MA, June 1986*. IEEE CS Press. (1986) 267–278.

[5] M. Jurdziński, M. Paterson and U. Zwick, A deterministic subexponential algorithm for solving parity games. *SIAM J. Comput.* **38** (2008) 1519–1532.

[6] D.E. Long, A. Browne, E.M. Clarke, S. Jha and W.R. Marrero, An improved algorithm for the evaluation of fixpoint expressions. in *Proc. of 6th Int. Conf. on Computer Aided Verification, CAV '94 Stanford, CA, June 1994*, edited by D. L. Dill, Springer, *Lect. Notes Comput. Sci.* **818** (1994) 338–350.

[7] D. Niwiński, Computing flat vectorial Boolean fixed points, unpublished manuscript.

[8] S. Schewe, Solving parity games in big steps, in *Proc. of 27th Int. Conf. on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2007 Kharagpur, Dec. 2007*, edited by V. Arvind and S. Prasad, Springer. *Lect. Notes Comput. Sci.* **4855** (2007) 449–460.

[9] S. Zhang, O. Sokolsky and S.A. Smolka, On the parallel complexity of model checking in the modal mu-calculus, in *Proc. 9th Ann. IEEE Symp. on Logic in Computer Science, LICS '94 Paris, July 1994*. IEEE CS Press. (1994) 154–163.

Communicated by Ralph Matthes and Tarmo Uustalu.
Received September 28, 2012. Accepted September 28, 2012.