Some Results on Complexity of μ-calculus Evaluation in the Black-box Model

Paweł Parys*

University of Warsaw
ul. Banacha 2, 02-097 Warszawa, Poland
parys@mimuw.edu.pl

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) of such formula in a powerset lattice. Assumptions about the quantifier-free part of the expression are the weakest possible: it 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. 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.

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 could 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.

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

$$\theta_1 x_1.\theta_2 x_2 \dots \theta_d x_d.F(x_1,\dots,x_d), \tag{1}$$

where $\theta_i = \mu$ for odd i and $\theta_i = \nu$ for even i. We want to evaluate such expression in the powerset model or, equivalently, in the lattice $\{0,1\}^n$ with the order defined by $a_1 \dots a_n \le b_1 \dots b_n$ when $a_i \le b_i$ 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 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 $t_F \ge O(nd)$, i.e. that the procedure at least reads its arguments. A typical complexity, in which one can evaluate the expression (1) is $O(n^d \cdot t_F)$; this can be done by naive iterating [1]. 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 Section 2.

Theorem 1. *There is an algorithm which for any fixed model size n calculates the value of expression (1) in time polynomial in d and t_F, namely $O(d^n \cdot t_F)$.*

Our result slightly extends an unpublished result in [2]. The authors also get polynomial expression complexity, however using completely different techniques. Our result is stronger, since they consider only

* Work supported by Polish government grant no. N206 008 32/0810.

expressions in which F is given by a vectorial Boolean formula, not as an arbitrary function. Moreover their complexity is slightly higher: $O(d^{2n} \cdot |F|)$.

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 its child corresponds to a possible value of F for that argument. The tree has to determine the value of the expression (1): for each path from the root to a leaf there is at most one possible value of (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. *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) 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 Section 3.

Theorem 2. *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 for any d there are needed more than nd queries. Note that $num(1, \{0,1\}^n)$ is n and that in the case when all d fixed-point operators are μ (instead of alternating μ and ν) 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 [3].

Let us comment the way how the function F is given. We make the weakest possible assumptions: the function can be given by an arbitrary program. This is called a black-box model, and was introduced in [4]. 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. Note that the algorithm in [4], 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 [5] working in time $O(m^{d/3})$ and from [6] 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. 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) in polynomial time, see Section 2.7.4 in [7]. Hence, a polynomial algorithm evaluating expressions of form (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 gives exponential complexity.

2 The algorithm with polynomial expression complexity

Below we present a general version of the well known iterating algorithm. 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 $\theta_i x_i . \theta_{i+1} x_{i+1} \ldots \theta_d x_d . F(X_1, \ldots, X_{i-1}, x_i, \ldots, x_d)$.

$\mathsf{Calculate}_i(X_1, \ldots, X_{i-1})$:
  $X_i = \mathsf{Initialize}_i(X_1, \ldots, X_{i-1})$
  repeat
    $X_i = \mathsf{Calculate}_{i+1}(X_1, \ldots, X_i)$
  until X_i stops changing
  return X_i

Moreover 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. When they always return $00\ldots0$ for odd i and $11\ldots1$ for even i, we simply get the naive iterating algorithm from [1]. However we would like to make use of already done computations and start a 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 1. *Assume that* $\mathsf{Initialize}_i(X_1, \ldots, X_{i-1})$ *for odd* i *returns either* $00\ldots0$ *or a result of a previous call to* $\mathsf{Calculate}_i(X'_1, \ldots, X'_{i-1})$ *for some* $X'_1 \le X_1, \ldots, X'_{i-1} \le X_{i-1}$ *and for even* i *either* $11\ldots1$ *or a result of a previous call to* $\mathsf{Calculate}_i(X'_1, \ldots, X'_{i-1})$ *for some* $X'_1 \ge X_1, \ldots, X'_{i-1} \ge X_{i-1}$. *Then the function* $\mathsf{Calculate}_i(X_1, \ldots, X_{i-1})$ *returns the correct result.*

So 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 simply take

$$\mathsf{Initialize}_i(X_1, \ldots, X_{i-1}) = \begin{cases} 00\ldots0 \text{ for } i = 1, \\ 11\ldots1 \text{ for } i = 2, \\ X_{i-2} \quad \text{for } i \ge 3. \end{cases} \tag{2}$$

It turns out that $\mathsf{Initialize}_i$ defined this way satisfies assumptions of Lemma 1, so the algorithm is correct. The complexity bound follows from a simple observation that arguments of each call to $\mathsf{Calculate}_{d+1}$ satisfy

$$X_1 \le X_3 \le \cdots \le X_{d-3} \le X_{d-1} \le X_d \le X_{d-2} \le \cdots \le X_4 \le X_2.$$

The same chain of inequalities is true for the numbers b_i of bits of X_i set to 1. Moreover the sequence $b_1, \ldots, b_d$ during each call to $\mathsf{Calculate}_{d+1}$ differs from stage to stage, it always increases in some appropriately defined order. There are $\binom{n+d}{d}$ such sequences, hence the complexity is $O\left(\binom{n+d}{d} \cdot t_F\right)$.

3 Quadratic lower bound

In order to prove Theorem 2 we first introduce a lattice which is more convenient than $\{0,1\}^n$. Take the alphabet Γ_n consisting of letters γ_i for $1 \le i \le n^2$ and the alphabet $\Sigma_n = \{0,1\} \cup \Gamma_n$, with the following partial order on it: the letters γ_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: $a_1 \ldots a_n \le b_1 \ldots b_n$ when $a_i \le b_i$ for all i. The idea is that one letter of Σ_n^n may be encoded in $O(\log n)$ bits of $\{0,1\}^m$. Hence to show Theorem 2 it is enough to prove $num(d, \Sigma_n^n) \ge \Omega(n^2)$.

To prove it we define a family of monotone functions, which will be difficult to distinguish by the algorithm. A function $F_{z,\sigma} \colon \Sigma_n^{2n} \to \Sigma_n^n$ is parametrized by a sequence $z \in \Gamma_n^n$ and by a permutation $\sigma \colon \{1, \ldots, n\} \to \{1, \ldots, n\}$ (note that z is from Γ_n^n, not from Σ_n^n, so it can not contain 0 or 1, just the letters γ_i). The result of $\mu y.\nu x.F_{z,\sigma}(y,x)$ will be z. 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 $y_0, \ldots, y_n$:

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

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 $y_{n+1} = y_n$):

$$F_{z,\sigma}(y_k, x)[i] = \begin{cases} 0 & \text{if } \forall_{j>i} x[j] \leq y_{k+1}[j] \text{ and } x[i] \not\geq y_{k+1}[i] \text{ (case 1)} \\ y_{k+1}[i] & \text{if } \forall_{j>i} x[j] \leq y_{k+1}[j] \text{ and } x[i] \geq y_{k+1}[i] \text{ (case 2)} \\ x[i] & \text{if } \exists_{j>i} x[j] \not\leq y_{k+1}[j] \text{ (case 3).} \end{cases}$$

For any other value y we look for the lowest possible k such that $y \leq y_k$ and we put $F_{z,\sigma}(y, x) = F_{z,\sigma}(y_k, x)$. When such k does not exists ($y \not\leq z$), we put $F_{z,\sigma}(y, x)[i] = 1$.

The intuition behind the functions is as follows. At each moment the algorithm knows only some k letters of z (at the beginning it do not know any letter). Then it may decide which letter of z it want to uncover in the next step. When it tries to uncover a letter at position $\sigma(k)$, it is successful; otherwise it has to try again at another position. For the worst function it has to try all possible $n - k$ positions to uncover one letter of z. This gives a quadratic bound. The algorithm may also try to guess a value of letter on some position; however there are n^2 different γ_i, so in the worst case it has to guess n^2 times until it will discover a correct letter. A more detailed analysis shows that the algorithm is not able to do anything else.

4 Concluding remarks

The detailed proofs of the results are contained in [8, 9], available on the author's web page.

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)), 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.

References

1. Emerson, E.A., Lei, C.L.: Efficient model checking in fragments of the propositional mu-calculus (extended abstract). In: LICS. (1986) 267–278
2. Niwiński, D.: Computing flat vectorial Boolean fixed points. Unpublished manuscript
3. Dawar, A., Kreutzer, S.: Generalising automaticity to modal properties of finite structures. Theor. Comput. Sci. **379**(1-2) (2007) 266–285
4. Long, D.E., Browne, A., Clarke, E.M., Jha, S., Marrero, W.R.: An improved algorithm for the evaluation of fixpoint expressions. In: CAV. (1994) 338–350
5. Schewe, S.: Solving parity games in big steps. In: FSTTCS. (2007) 449–460
6. Jurdzinski, M., Paterson, M., Zwick, U.: A deterministic subexponential algorithm for solving parity games. SIAM J. Comput. **38**(4) (2008) 1519–1532
7. Arnold, A., Niwiński, D.: Rudiments of μ-calculus. Elsevier (2001)
8. Parys, P.: Evaluation of normalized μ-calculus formulas is polynomial for fixed structure size. Unpublished manuscript
9. Parys, P.: Lower bound for evaluation of $\mu\nu$ fixpoint. In: FICS. (2009) 86–92