Evaluation of normalized μ-calculus formulas is polynomial for fixed structure size

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

October 30, 2009

Abstract

We consider μ-calculus formulas in a normal form: after a prefix of fixed-point quantifiers follows a quantifier-free expression. We prove that the problem of evaluating (model checking) of such formula in a fixed powerset lattice (*expression complexity*) is polynomial. Assumptions about the quantifier-free part of the expression are weakest possible: it can be any monotone function given by a computational procedure.

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.

Here we consider a much easier problem of polynomial *expression complexity*, i.e. complexity for fixed size of the model. Furthermore, we restrict ourselves to expressions in a quantifier-prefix normal form, namely

$$\mu x_1.\nu x_2 \dots \mu x_{d-1}.\nu x_d.F(x_1, \dots, x_d) \tag{1}$$

For simplicity of presentation we assume that d is even, but of course the same can be done for odd d (in which case the expression would end with μ). We want to evaluate such expression in the powerset model or, equivalently, in the lattice $\{0,1\}^n$ with the order defined by $a_1 \dots a_n \leq b_1 \dots b_n$ 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 procedure which evaluates a value of the function for given arguments in time t_F.

A typical complexity, in which one can evaluate such expression, is $O(n^d \cdot t_F)$; this can be done by naive iterating [3]. 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 main result.

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 $O(d^n \cdot (d + t_F))$.*

As a side remark recall two results about parallel complexity of the modal μ-calculus model checking problem. The problem is PTIME-hard not only when considering combined complexity [8], but already for the expression complexity [2]. This somehow contradicts with the intuition that for fixed model the problem should become easy.

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. 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. Denote that the algorithm in [5], 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 [7] working in time $O(m^{d/3})$ and from [4] 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.

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 [1]. However during this conversion one also need to change the underlying structure (roughly speaking, its size becomes similar to the size of the formula). Hence even when the original model has fixed size n, after the normalization the model can become very big, and our algorithm gives exponential complexity.

Our results are very similar to that in [6]. They also get polynomial expression complexity, however using completely different techniques. Our result is slightly 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: $O(d^{2n} \cdot |F|)$.

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 the part of the expression starting from μx_i or νx_i is denoted by F_i, for example for odd i we have

$$F_i(X_1, \ldots, X_{i-1}) = \mu x_i . \nu x_{i+1} \ldots \mu x_{d-1} . \nu x_d . F(X_1, \ldots, X_{i-1}, x_i, \ldots x_d).$$

In particular $F_{d+1} = F$ and $F_1()$ is the value which we want to calculate. Recall that each F_i is a monotone function.

Below we present a general version of the 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 $F_i(X_1, \ldots, X_{i-1})$.

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

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. 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 [3]. 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 2. *Assume that the values of X_i returned by $\mathsf{Initialize}_i$ satisfy*

$$X_i \leq F_i(X_1, \ldots, X_{i-1}), \qquad \textit{and} \tag{2}$$

$$X_i \leq F_{i+1}(X_1, \ldots, X_{i-1}, X_i) \tag{3}$$

for odd i, and

$$X_i \geq F_i(X_1, \ldots, X_{i-1}), \qquad \textit{and} \tag{2'}$$

$$X_i \geq F_{i+1}(X_1, \ldots, X_{i-1}, X_i). \tag{3'}$$

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) and (3) 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) is preserved during the iterations:

$$F_{i+1}(X_1, \dots, X_i) \leq F_{i+1}(X_1, \dots, X_{i-1}, F_i(X_1, \dots, X_{i-1})) = F_i(X_1, \dots, X_{i-1}).$$

The inequality follows from monotonicity of F_{i+1} and (2) 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 $\mathsf{Calculate}_{i+1}(X_1, \dots, X_i) = F_{i+1}(X_1, \dots, X_i)$. The property (3) is even simpler,

$$F_{i+1}(X_1, \dots, X_i) \leq F_{i+1}(X_1, \dots, X_{i-1}, F_{i+1}(X_1, \dots, X_i)),$$

it follows from monotonicity of F_{i+1} and (3) before the iteration.

Property (3) guaranties that X_i is increased at each iteration. After some number of steps it has to stabilize. Then $X_i = F_{i+1}(X_1, \dots, X_i)$ is a fixed-point. From (2) we know that X_i is $\leq F_i(X_1, \dots, X_{i-1})$. Moreover it can not be strictly smaller than $F_i(X_1, \dots, X_{i-1})$, because then we would have a fixed-point smaller than the smallest fixed-point. So X_i stabilizes at $F_i(X_1, \dots, X_{i-1})$. □

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 3. *If $X_i = F_i(X'_1, \dots, X'_{i-1})$ for some $X'_1 \leq X_1, \dots, X'_{i-1} \leq X_{i-1}$ then conditions (2) and (3) hold. By symmetry, if $X_i = F_i(X'_1, \dots, X'_{i-1})$ for some $X'_1 \geq X_1, \dots, X'_{i-1} \geq X_{i-1}$ then conditions (2') and (3') hold.*

Proof
We prove the first part of the lemma (which is useful for odd i). Property (2) simply follows from monotonicity of F_i. To get (3) 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}:

$$X_i = F_{i+1}(X'_1, \dots, X'_{i-1}, X_i) \leq F_{i+1}(X_1, \dots, X_{i-1}, X_i).$$

□

Furthermore, observe that conditions (2) and (3) (respectively, (2') and (3')) trivially hold for $X_i = 00\dots0$ ($X_i = 11\dots1$).

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 tricky data structure, we do a very simple trick. We simply take

$$\mathsf{Initialize}_i(X_1, \dots, X_{i-1}) = \begin{cases} 00\dots0 & \text{for } i = 1, \\ 11\dots1 & \text{for } i = 2, \\ X_{i-2} & \text{for } i \geq 3. \end{cases} \tag{4}$$

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 $X'_1 \leq X_1, \dots, X'_{i-1} \leq X_{i-1}$ or $X_{i-2} = 00\dots0$ for odd i, and
- $X_{i-2} = F_i(X'_1, \dots, X'_{i-1})$ for some $X'_1 \geq X_1, \dots, X'_{i-1} \geq X_{i-1}$ 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, the other case is symmetric. There are two cases depending on where the current value of X_{i-2} was set:

1. We are 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, 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 first time in the repeat-until loop in $\mathsf{Calculate}_{i-2}$. Then 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 we know that $X'_{i-2} \le X_{i-2}$ (that X_{i-2} increases). Moreover, from Lemma 2, 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 $X_1, \dots, X_{i-1}$.

Till 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 4. *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 $X_{d-1} \le X_d$. In fact it follows from the above proof: in both cases we had there $X_{i-2} \le X_{i-1}$ for odd i. Although it was only for $i \le d-1$, the same proof is correct also for $i = d+1$.

All the other inequalities immediately follow from Lemma 2: 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
$$0 \le b(X_1) \le b(X_3) \le \dots \le b(X_{d-3}) \le b(X_{d-1}) \le b(X_d) \le b(X_{d-2}) \le \dots \le b(X_4) \le b(X_2) \le n. \tag{5}$$
Now look at the sequences
$$b(X_1), -b(X_2), b(X_3), -b(X_4), \dots, b(X_{d-1}), -b(X_d)$$
and notice that at each call to $\mathsf{Calculate}_{d+1}$ we get a lexicographically greater sequence. Indeed, when between two consequent calls we exit up to procedure $\mathsf{Calculate}_i$, then $b(X_1), \dots, b(X_{i-1})$ stay the same, $b(X_i)$ is increased (for odd i) or decreased (for even i), while $b(X_{i+1}), \dots, b(X_d)$

may become arbitrary. Hence at each call to $\mathsf{Calculate}_{d+1}$ we have a different sequence satisfying (5). 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 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 $t_F \geq O(nd)$, i.e. that F at least reads all its arguments, the complexity becomes $O\left(\binom{n+d}{d} \cdot t_F\right)$.

4 Concluding remarks

We have showed a polynomial expression complexity for formulas of the normalized form (1). It is very interesting if the same can be shown for arbitrary formulas, or if the problem is then equivalent to model checking in an arbitrary model.

References

[1] A. Arnold and D. Niwiński. *Rudiments of μ-calculus.* Elsevier, 2001.

[2] S. Dziembowski, M. Jurdziski, and D. Niwiński. On the expression complexity of the modal μ-calculus model checking. Unpublished manuscript.

[3] E. A. Emerson and C.-L. Lei. Efficient model checking in fragments of the propositional mu-calculus (extended abstract). In *LICS*, pages 267–278, 1986.

[4] M. Jurdzinski, M. Paterson, and U. Zwick. A deterministic subexponential algorithm for solving parity games. *SIAM J. Comput.*, 38(4):1519–1532, 2008.

[5] D. E. Long, A. Browne, E. M. Clarke, S. Jha, and W. R. Marrero. An improved algorithm for the evaluation of fixpoint expressions. In *CAV*, pages 338–350, 1994.

[6] D. Niwiński. Computing flat vectorial Boolean fixed points. Unpublished manuscript.

[7] S. Schewe. Solving parity games in big steps. In *FSTTCS*, pages 449–460, 2007.

[8] S. Zhang, O. Sokolsky, and S. A. Smolka. On the parallel complexity of model checking in the modal mu-calculus. In *LICS*, pages 154–163, 1994.