Acta Informatica, 32(2):117–187, 1995.

Generating Boolean μ-Expressions

Thomas Eiter

Information Systems Department
Technical University of Vienna
Paniglgasse 16, A-1040 Vienna, Austria
E-mail: eiter@vexpert.dbai.tuwien.ac.at

Abstract

In this paper, we consider the class of Boolean μ-functions, which are the Boolean functions definable by μ-expressions (Boolean expressions in which no variable occurs more than once). We present an algorithm which transforms a Boolean formula E into an equivalent μ-expression–if possible–in time linear in $\|E\|$ times 2^{n_m}, where $\|E\|$ is the size of E and n_m is the number of variables that occur more than once in E. As an application, we obtain a polynomial time algorithm for Mundici's problem of recognizing μ-functions from k-formulas [17]. Furthermore, we show that recognizing Boolean μ-functions is co-**NP**-complete for functions essentially dependent on all variables and we give a bound close to co-**NP** for the general case.

Keywords: Boolean functions, μ-functions, propositional logic, formulas, μ-formulas, recognition problem, polynomial algorithms, NP-completeness

1 Introduction

Boolean functions and algorithms for manipulating Boolean formulas (or as well, propositional formulas) are important in many areas of computer science, e.g. in control theory, artificial intelligence, database theory etc. A number of classes of Boolean functions have been studied extensively in the literature (cf. [23]), of which the monotone Boolean functions (i.e., those definable without negation) are probably most well-known.

In this paper, we consider the class of Boolean μ-functions, which are the Boolean functions definable by μ-expressions (Boolean expressions in which no variable occurs more than once). This class has attracted much attention during the last years, cf. [4, 5, 21, 9, 19, 17, 16, 10, 6], but had been considered–in a more general context–at least more than 30 years earlier [14, 20] (cf. also [6]); it is well-known that monotone μ-expressions correspond to series-parallel graphs and networks, cf. [16, 18].

Gurvich considered μ-functions and their applications in game theory [3, 4], graph theory [5, 7] and, with Libkin, in lattice theory [8]; see [6] for several criteria for μ-functions and references. Valiant and Pitt studied the class of μ-expressions in learning theory [21, 19]; Hunt and Stearns derived complexity results for μ-expressions and described a number of applications to logic, algebra, computational provability, and binary decision programs [9, 10]. Mundici provided in [16] an important semantical characterization of μ-functions, which he fruitfully applied to derive complexity results for recognizing μ-functions [17]. Essentially the same characterization –a trivially equivalent formulation – was independently published by Gurvich [3] (cf. [4, 5] for English translations of texts using this result). Furthermore, Mundici pointed out an interesting application of μ-expressions in a nontraditional method for solving hard problems (cf. [15]); in particular, he described a nontraditional (in the sense of [15]) method to solve an **NP**-complete problem on monotone μ-expressions by using an electric circuit [18].

While several complexity results for problems on μ-formula have been derived [9, 17, 1], no concrete algorithm of transforming a given Boolean expression E into an equivalent μ-expression (provided one exists) has been formulated so far. In this paper, we present such an algorithm, which is linear in $\|E\|$ times 2^{n_m}, where $\|E\|$ is the size of E and n_m is the number of variables that occur more than once in E; notice that a polynomial algorithm is unlikely to exist, since deciding whether E defines a μ-function is co-**NP**-hard.

An immediate application of our algorithm sharpens a result by Mundici on recognizing μ-functions from k-formulas, which are Boolean combinations of μ-expressions that are "almost" μ-expressions, i.e., formulas E in which at most $k \cdot \log \|E\|$ variables occur more than once, where k is a constant; we obtain that this problem is polynomial time decidable. Even more, our algorithm provides an effective method for rewriting k-formulas to μ-expressions in polynomial time.

We further derive new complexity results for recognizing Boolean μ-functions from general Boolean formulas. In particular, we show that this problem is co-**NP**-complete for functions essentially depending on all variables and we give a bound close to co-**NP** for the general case.

The rest of this paper is organized as follows. Section 2 introduces notation and reviews previous results. In Section 3, we present our algorithm; Section 4 is devoted to issues concerning efficient implementation and complexity. Section 5 gives new complexity results for recognizing Boolean μ-functions, and Section 6 concludes the paper.

2 Preliminaries and previous results

We assume that the reader is familiar with Boolean functions (BFs; for a background, cf. [23]) and the basic concepts of **NP**-completeness (cf. [2, 11]).

Boolean expressions (BEs) are built using the connectives $\neg, \wedge, \vee$ from Boolean variables $x_1, \ldots, x_n$ and constants $0, 1$. A BE E is *monotone* iff it is built only with $\wedge, \vee$; and it is in *negation normal form* (*NNF*) if negation occurs only in front of variables. The *size* of E, denoted by $\|E\|$, is measured by the number of symbols in E. Each BE E represents an n-ary Boolean function (BF) $f_E : \{0,1\}^n \to \{0,1\}$. A BF f is *monotone* iff $f = f_E$ for some monotone BE E. For $v \in \{0,1\}$ we denote by $f|_{x_i=v}$ the BF f' defined by $f'(x_1,\ldots,x_n) = f(x_1,\ldots,x_{i-1},v,x_{i+1},\ldots,x_n)$.

We can think of a *prime implicant* (resp. *prime clause*) of $f(x_1,\ldots,x_n)$ as a minimal subset $\{y_1,\ldots,y_k\} \subseteq \{x_i, \neg x_i : 1 \leq i \leq n\}$ (in terms of inclusion) not containing both x_i and $\neg x_i$, for any i in $1,\ldots,n$ such that for any BE E representing f, the BE $\neg(y_1 \wedge \cdots \wedge y_k) \vee E$ (resp. $\neg E \vee y_1 \vee \cdots \vee y_k$) is a tautology (as usual, we identify the empty conjunction with 1 and the empty disjunction with 0). By $\mathcal{PI}(f)$ and $\mathcal{PC}(f)$ we denote the sets of all prime implicants and all prime clauses of f, respectively. It is well-known that for monotone f, it holds that I and C are subsets of $\{x_1,\ldots,x_n\}$ for each $I \in \mathcal{PI}(f)$ and $C \in \mathcal{PC}(f)$.

The class of (Boolean) μ-expressions is defined as follows.

Definition 2.1 (cf. [21, 17]) *The class of μ-expressions is the smallest set that contains*

(*a*) 0, 1 *and each variable x_i, and*

(*b*) *if E, F are μ-expressions $\neq 0,1$ on disjoint variables, the expressions $\neg E$, $E \wedge F$, and $E \vee F$.*

*A BE E is called μ-equivalent iff E is equivalent to some μ-expression, and a BF f is a μ-*function *iff $f = f_E$ for some μ-expression E.*

Mundici studied the properties of monotone μ-expressions [16, 17, 18]; several results on general μ-expressions have been derived by Hunt and Stearns [9, 10]. Independently, Gurvich and other researchers in the Soviet Union developed characterizations of μ-expressions and investigated applications, cf. [3, 4, 5, 6].

To our knowledge, no algorithm has been explicitly described for generating an equivalent μ-expression for a given BE E (if possible). If E is known to be μ-equivalent, the algorithm for finding μ-expressions [21] can be used for this purpose. Thus, in the general case, one has to decide first whether E defines a μ-function before applying this algorithm. Moreover, this algorithm calls subroutines for solving problems on Boolean functions that are co-**NP**-hard in general (e.g., a prime implicant test) and that, assuming $\mathbf{R} \neq \mathbf{NP}$, are still hard for μ-functions (cf. [11]). We present a *constructive* algorithm for deciding if E is μ-equivalent in Section 3, which immediately outputs an equivalent μ-expression and calls no subroutines for hard problems.

Mundici [16] considered the problem of recognizing μ-equivalent expressions for two subclasses of Boolean expressions, namely the classes of monotone BEs and monotone k-expressions, which consist of all BEs E in which at most $k \cdot \log \|E\|$ variables occur more than once. He obtained the following results.

Proposition 2.1 [17] (*i*) *Given a monotone BE E, deciding if f_E is a μ-function is in co-***NP**. (*ii*) *Given a monotone k-expression E, deciding if f_E is a μ-function is in* **NP** $\cap$ *co-***NP**.

The key for these nontrivial results is the following semantical characterization of monotone μ-functions.

Proposition 2.2 ([16]; cf. also [3, 5]) *A monotone BF f is a μ-function iff for each $I \in \mathcal{PI}(f)$ and $C \in \mathcal{PC}(f)$, $|I \cap C| = 1$.*

As mentioned above, this result was derived independently in [16] and in [3] (cf. [5]), where it is formulated by referring to duality.

A proof of co-**NP**-hardness of Mundici's first problem, i.e., recognizing a μ-function from a monotone BE, can be found in [1]. A similar result for recognizing a *particular* μ-function has been derived in [10].

In this paper, we will show that Mundici's second problem is decidable in polynomial time. This follows from the runtime of our algorithm for generating μ-expressions; in fact, our algorithm finds for an arbitrary k-expression an equivalent μ-expression (if one exists) in polynomial time.

3 An algorithm to construct μ-expressions

In this section, we develop our algorithm for generating μ-expressions. After explaining the basic principle, we describe the algorithm in detail and consider several examples.

We need some additional concepts and definitions. The *and-or trees* over variables $x_1, \ldots, x_n$ are defined as the collection of trees T whose nodes are labeled in the following way: Each terminal node is labeled with $0, 1$ or x_i, $\neg x_i$ for some i in $1, \ldots, n$ and each nonterminal node of T is labeled with either "$\wedge$" (called an *and-node*) or "$\vee$" (an *or-node*), such that labels alternate on every path from the root to a nonterminal node. (We use the terms root, son, father, subtree etc. in the usual sense.) By $Term(T)$ we denote the set of labels of the terminal nodes of T, which, if unambiguous, we identify with their labels.

Formally, we represent and-or trees as follows. $0, 1$ and every x_i, $\neg x_i$ are and-or trees; if $T_1, \ldots, T_n$, $n \geq 2$, are and-or trees such that no T_i's root is labeled with $\wedge$ (resp. $\vee$), then $\langle \wedge, T_1, \ldots, T_n \rangle$ (resp. $\langle \vee, T_1, \ldots, T_n \rangle$) represents the and-or tree whose root is an and-node (resp. or-node) and has sons $T_1, \ldots, T_n$, where the order of the sons does not matter. An example of an and-or tree is shown in Figure 1; horizontal bars correspond to and connections.

Proposition 3.1 *Every and-or tree T naturally corresponds to a set of BEs in NNF which is closed under the laws of associativity and commutativity.*

An and-or T is a *μ-tree* if it represents some μ-expression. μ-trees correspond to μ-functions as follows.

Lemma 3.2 *Every Boolean μ-function $f(x_1, \ldots, x_n)$ can be identified with a unique μ-tree T_f, which is given by the and-or-tree T_E of any μ-expression E for f.*

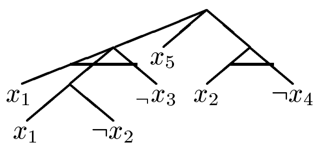


Figure 1: And-or tree $T = \langle \vee, \langle \wedge, x_1 \langle \vee, x_1, \neg x_2 \rangle, \neg x_3 \rangle, x_5, \langle \wedge, x_2, \neg x_4 \rangle \rangle$.

Proof. As argued in [9, 10], any equivalent μ-expressions E and F in NNF can be transformed into each other by applying the laws of commutativity and associativity. $\square$

Our method for generating μ-expressions relies on this property, together with the following decomposition property.

Lemma 3.3 *Let $f = f(x_1, \ldots, x_n)$ be a μ-function. Then for every $i = 1, \ldots, n$, both $f|_{x_i=0}$ and $f|_{x_i=1}$ are μ-functions.*

The basic idea for our algorithm is to utilize this decomposition lemma for generating the μ-tree of a Boolean μ-function f bottom up, i.e., to apply this decomposition rule in the reverse direction. More precisely, since for any variable x,

$$f = xf|_{x=1} \vee \neg x f|_{x=0},$$

the idea is to obtain the μ-tree T_f of f from the μ-trees T_{f_0} and T_{f_1} for $f|_{x=0}$ and $f|_{x=1}$, respectively. This is possible since the trees T_f, T_{f_0}, and T_{f_1} are structurally very similar. In fact, T_{f_0} and T_{f_1} can be obtained from T_f by substituting 1 resp. 0 for x and subsequently reorganizing the tree into a μ-tree; this process touches only a small, contiguous part of the tree, which is merely locally changed. This makes it possible to efficiently decide, given the μ-trees T_{f_0} and T_{f_1}, whether f is a μ-function by looking at the possible patterns for T_{f_0} and T_{f_1}. If the patterns match, output the μ-tree T_f for f.

After this rough exposition, we describe the algorithm in detail. The possible patterns for T_{f_1} and T_{f_0}, where $T_{f_1} \neq T_{f_0}$, are derivable from Tables 1–3.

Table 1 lists all the cases for decomposing f where $T_f = x$ or x occurs as the son of the root of T_f. Notice that in all those cases, at least one of T_{f_0} and T_{f_1} is 1 or 0. The analogous decomposition patterns $bl1'$, $bo1'$–$bo3'$ and $ba1'$–$ba3'$, where $\neg x$ occurs in place of x, are obtained from $bl1$, $bo1$–$bo3$ and $ba1$–$ba3$ respectively by exchanging the trees for T_{f_0} and T_{f_1}.

Tables 2 and 3 list the remaining cases if x occurs as a terminal node in T_f; Table 2 covers the case in which x is son of an or-node and Table 3 the case in which x is the son of an and-node. Each of the cases $o1$–$o4$ and $a1$–$a4$ represents all situations in

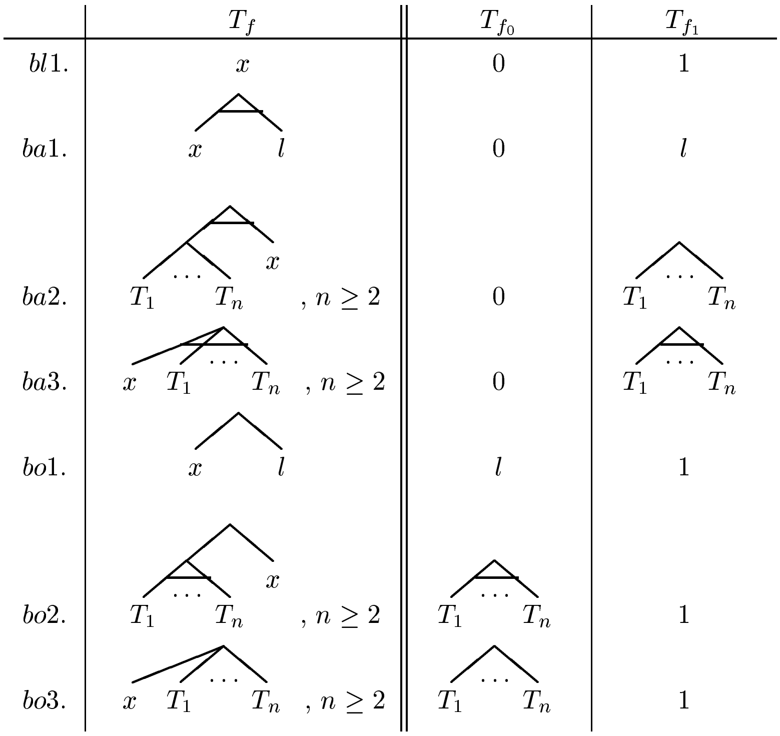


Table 1: μ-trees for the decomposition $f = xf|_{x=1} \vee \neg xf|_{x=0}$ of a μ-function f: all cases where $T_f = x$ or x is son of the root of T_f. The trees T_{f_0} and T_{f_1} are unique. ($l \in \{x_i, \neg x_i : 1 \leq i \leq n\}$.)

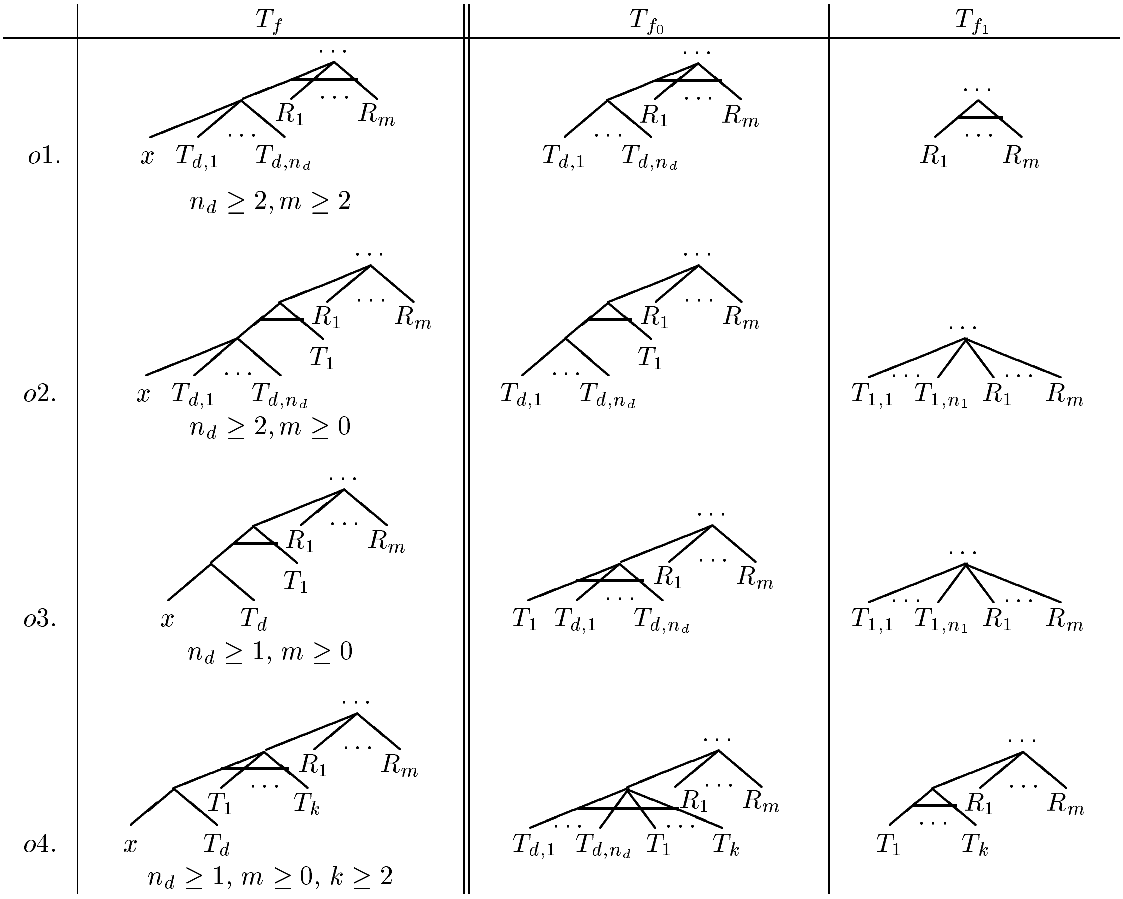


Table 2: μ-tree patterns for the decomposition $f = xf|_{x=1} \vee \neg x f|_{x=0}$ of a μ-function f: all possible cases for x to be in the tree T_f if it is son of an or-node and not son of the root. The trees T_{f_0} and T_{f_1} are unique. (For $\alpha \in \{d, 1\}$, $T_\alpha = \langle \cdot, T_{\alpha,1}, \ldots, T_{\alpha,n_\alpha} \rangle$ if $n_\alpha \geq 2$, else $T_\alpha = T_{\alpha,1}$ and $T_\alpha = l$ for some $l \in \{x_i, \neg x_i : 1 \leq i \leq n\}$.)

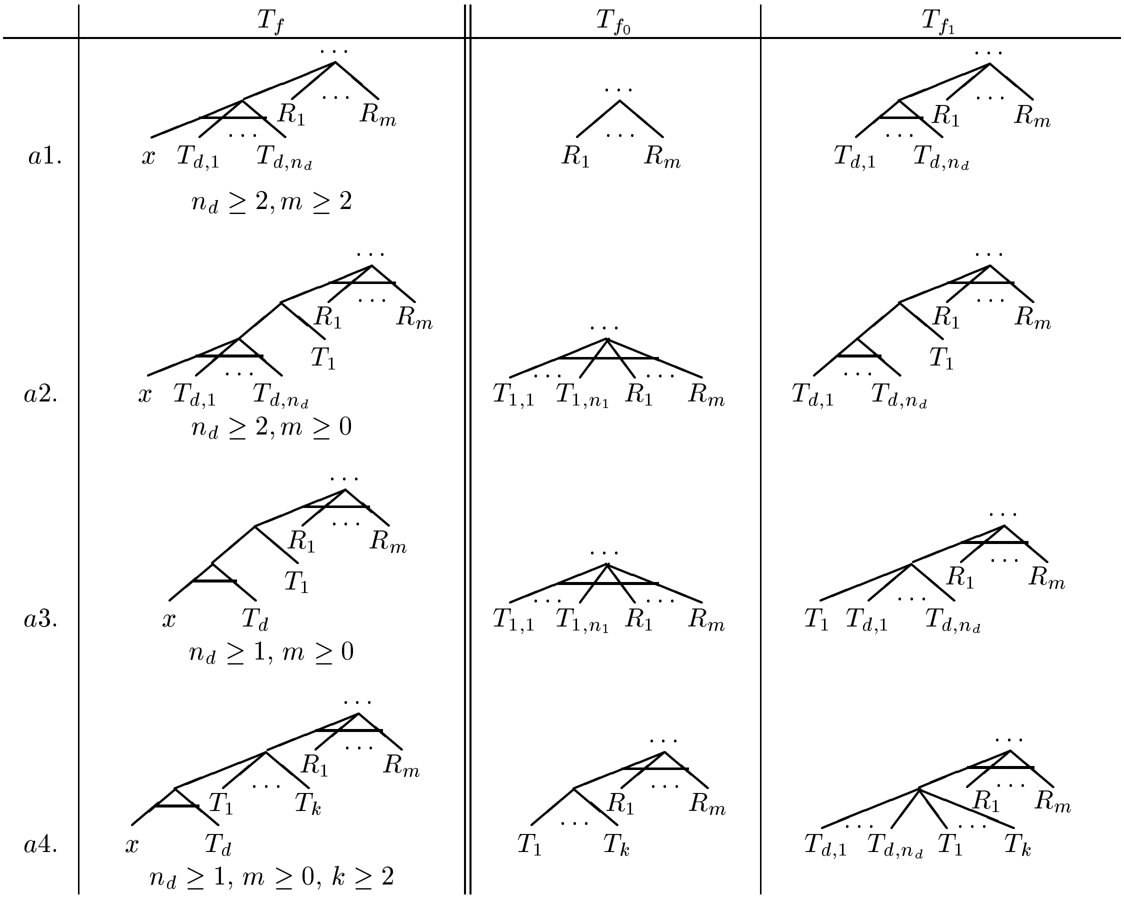
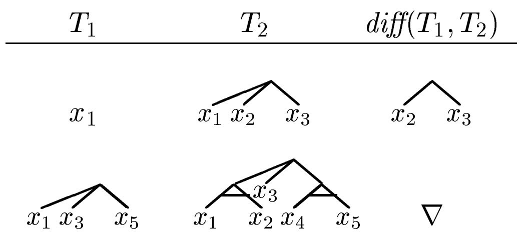


Table 3: μ-tree patterns for the decomposition $f = xf|_{x=1} \vee \neg xf|_{x=0}$ of a μ-function f: all possible cases for x to be in the tree T_f if it is son of an and-node and not son of the root. The trees T_{f_0} and T_{f_1} are unique. (For $\alpha \in \{d, 1\}$, $T_\alpha = \langle \cdot, T_{\alpha,1}, \ldots, T_{\alpha,n_\alpha} \rangle$ if $n_\alpha \geq 2$, else $T_\alpha = T_{\alpha,1}$ and $T_\alpha = l$ for some $l \in \{x_i, \neg x_i : 1 \leq i \leq n\}$.)

which T_f, T_{f_0} and T_{f_1}, are obtained from some μ-tree by replacing the same terminal node with μ-trees as depicted. (Thus, each entry represents a pattern for a family of decomposition cases.) The patterns for the analogous cases $o1'$–$o4'$ and $a1'$–$a4'$ where $\neg x$ occurs in place of x are obtained by exchanging T_{f_0} and T_{f_1} in $o1$–$o4$ and $a1$–$a4$.

Some properties of the decomposition patterns stated by the next lemma justify unambiguous reverse application of decomposition. But first we need for our purpose to define the difference of two μ-trees.

Informally, we regard the μ-tree T_d as the difference $\mathit{diff}(T_1,T_2)$ of the μ-trees T_1 and T_2 if T_d is a subtree of T_d whose terminal nodes are $Term(T_2) - Term(T_1)$, and such that T_d consists of a node r in T_2 plus, in case r is not a terminal node, a subset of at least two of the trees whose root is a son of r. If such a tree T_d does not exist, then $\mathit{diff}(T_1,T_2)$ is undefined. The following examples should clarify this notion.



Formally, $\mathit{diff}(T_1,T_2)$ is defined as follows.

Definition 3.1 *For μ-trees T_1 and T_2, let $\mathit{diff}(T_1,T_2)$ be defined as follows.*

$$\mathit{diff}(T_1,T_2) = \begin{cases} T_d, & \text{if } Term(T_1) \subset Term(T_2) \text{ and the following holds:} \\ & \quad (a)\ T_d \text{ is a } \mu\text{-tree that is a subtree of } T_2, \\ & \quad (b)\ Term(T_d) = Term(T_2) - Term(T_1), \\ & \quad (c)\ \text{for every node } s \text{ from } T_2 \text{ that is a son of the root} \\ & \qquad \text{of } T_d, \text{ all descendants of } s \text{ in } T_2 \text{ belong to } T_d; \\ \nabla, & \text{otherwise.} \end{cases}$$

Notice that $\mathit{diff}(T_1,T_2)$ is well defined.

Lemma 3.4 *Let f be a BF and let x be a variable.*

(i) If f is a μ-function, then T_f, T_{f_0}, and T_{f_1} are identical or match exactly one of the possible decomposition patterns for T_f where $T_d = \mathit{diff}(T_{f_0},T_{f_1})$ or $T_d = \mathit{diff}(T_{f_1},T_{f_0})$.

(ii) If $T_{f_1} = T_{f_0}$ are μ-trees for $f|_{x=1}$ and $f|_{x=0}$, then f is a μ-function with μ-tree T_{f_1}.

(iii) If $T_{f_1} \neq T_{f_0}$ are μ-trees for $f|_{x=1}$ and $f|_{x=0}$ that match one of the decomposition patterns where $T_d = \mathit{diff}(T_{f_0},T_{f_1})$ or $T_d = \mathit{diff}(T_{f_1},T_{f_0})$, then f is a μ-function with μ-tree T_f from the matched pattern. Moreover, in this case T_{f_0} and T_{f_1} match exactly one of the decomposition patterns.

(iv) If $f|_{x=1}$ or $f|_{x=0}$ is not a μ-function, or the respective μ-trees T_{f_1} and T_{f_0} are different and do not match any decomposition pattern, then f is not a μ-function.

Proof. The proof is based on Lemmas 3.2 and 3.3.
(i): If $f = f_E$ for some BE E in which x does not occur, then $f|_{x=1} = f|_{x=0} = f_E$; it follows from Lemmas 3.2 and 3.3 that $T_f = T_{f_0} = T_{f_1}$.

Otherwise, let T be the unique μ-tree for f. Since x or $\neg x$ must occur in T, it follows that T must match at least one of the tree patterns for T_f derivable from Tables 1–3. Inspection of all pairs of patterns for T_f–which we omit for brevity–yields that T cannot match two patterns simultaneously; hence, it follows that T matches exactly one of the patterns for T_f.

For each possible pattern for T_f, the attached patterns for T_{f_0} and T_{f_1} are uniquely determined. Indeed, the μ-trees for T_{f_0} and T_{f_1} can be obtained from T_f as follows:

1. replace x (resp. $\neg x$) in T_f by 0 and 1 (resp. 1 and 0);
2. if 0 (resp. 1) is son of an and-node n, then replace the subtree with root n by 0 (resp. remove 1 from the tree; if n has now merely a single son s left, remove n and the subtree with root s and add the sons of s to the father of n). This amounts to replacing $0 \wedge F$ (resp. $1 \wedge F$) in a BE for T_f by the equivalent BE 0 (resp. E). An analogous operation is carried out if 0 (resp. 1) is son of an or-node.
3. repeat the previous step as long as possible.

It is easy to see that step 2. is carried out at most twice. The resulting μ-trees are μ-trees for $f_{x=0}$ and $f_{x=1}$, respectively, which are unique by Lemma 3.2. The subtree T_d of T is cut off in this process from exactly one of the trees T_{f_0} and T_{f_1}, but remains in the other; it is easy to see that the μ-tree T_d from the patterns satisfies either $T_d = \mathit{diff}(T_{f_0}, T_{f_1})$ or $T_d = \mathit{diff}(T_{f_1}, T_{f_0})$. This concludes the proof of part (i).

(ii): Let E be a μ-expression such that $T_E = T_{f_1}$. Then, f is defined by the BE $(\neg x \wedge E) \vee (x \wedge E)$, which is equivalent to E. Thus, f is a μ-function and, by Lemma 3.2, $T_f = T_E = T_{f_1}$.

(iii): Let T' be the μ-tree T_f according to the matched pattern, and let f' be the μ-function defined by T'. Then, along the lines of the proof of part (i) and from Lemma 3.2, we obtain that $T_{f'_0} = T_{f_0}$ and $T_{f'_1} = T_{f_1}$. Thus, $f' = xf'_{x=1} \vee \neg x f'_{x=0} = xf_{x=1} \vee \neg x f_{x=0} = f$. Inspection of all pairs of possible patterns yields that T_{f_0} and T_{f_1} can not match two different patterns simultaneously.

(iv): If $f_{x=0}$ or $f_{x=1}$ is not a μ-function, it follows from Lemma 3.3 that f is not a μ-function. The remaining part follows immediately from part (i). $\Box$

Thus, one can construct the μ-tree for a BF f, given by a BE E, using a recursive algorithm, which first constructs the μ-trees for $f_{x=1}$ and $f_{x=0}$ for some variable x and then looks up the decomposition patterns for constructing the desired tree. Parts (iii) and (iv) in Lemma 3.4 imply that this step is deterministic and yields the correct result.

Algorithm muform(F)

input: a Boolean expression F.
output: a μ-expression G equivalent to F or ∇ if no such G exists.

```
begin   G :=nnf(F); T := mutree(G);
  if T ≠ ∇ then return form(T)
  else return ∇;
end.
```

Algorithm mutree(F)

input: a Boolean expression F in NNF.
output: a μ-tree for F, or ∇ if none exists.

```
begin    G := simp(F);
  if G is a μ-expression then return tree(G);
      /* decompose expression into subexpressions */
  select a variable x that occurs in G more than once;
  for i = 0,1 do begin
      G_i := G|_{x=i};   T_{f_i} := mutree(G_i);
      if T_{f_i} = ∇ then return ∇;
  end;
      /* try to combine T_{f_0} and T_{f_1} to a μ-tree for G */
  if T_{f_0} = T_{f_1} then return T_{f_1};
  if T_{f_0} ∈ {0,1} or T_{f_1} ∈ {0,1} then
      return bmatch(T_{f_0}, T_{f_1}, x);
  T_d := diff(T_{f_0}, T_{f_1});
  if T_d ≠ ∇ then
      return match(T_{f_0}, T_{f_1}, T_d, 0, x);
  T_d := diff(T_{f_1}, T_{f_0});
  if T_d ≠ ∇ then
      return match(T_{f_0}, T_{f_1}, T_d, 1, x);
  return ∇;
end.
```

Table 4: Algorithm **muform**(F) for transforming a Boolean expression F into an equivalent μ-expression.

We are now ready to describe our algorithm; we start by specifying several subroutines.

The function **simp**(F) returns a BF equivalent to F, which is obtained by simplifying subexpressions $\neg 0$, $\neg 1$, $1 \wedge F'$, $1 \vee F'$, $0 \wedge F'$, and $0 \vee F'$ until no longer possible.

The function **bmatch**(T_{f_0}, T_{f_1}, x) matches T_{f_0} and T_{f_1} with one of the decomposition cases $bl1$, $bl1'$, etc. (cf. Table 1) and returns the μ-tree T_f indexed by T_{f_0} and T_{f_1};

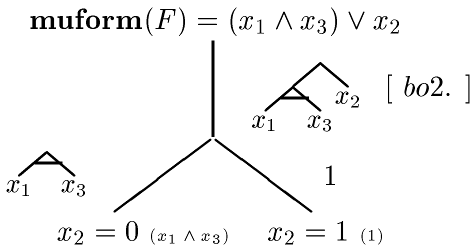
The function **match**$(T_{f_0}, T_{f_1}, T_d, v, x)$ tries to match the μ-trees T_{f_0}, T_{f_1}, and T_d, where $T_d = \mathit{diff}\,(T_{f_0}, T_{f_1})$ if $v = 0$ and $T_d = \mathit{diff}\,(T_{f_1}, T_{f_0})$ if $v = 1$, with the decomposition patterns. If matching is successful, then the function returns the μ-tree T_f from the matched pattern; otherwise, the function returns ∇.

Furthermore, we employ two functions for converting μ-trees into BEs and vice versa: **tree**(F) returns the μ-tree T naturally corresponding to the μ-expression F in NNF, and **form**(T) returns a BE corresponding to T.

The main algorithm, **mutree**, is embedded into the algorithm **muform**, which puts the input BE into NNF and converts the output of **mutree** into a BE (see Table 4).

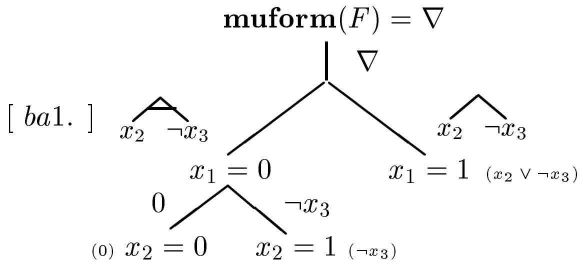
Before we discuss correctness and implementation of the algorithm in the next section, we consider some examples.

Example 1: Let $F = (x_1 \vee x_2) \wedge (x_2 \vee x_3)$. The schematic execution tree of **muform**(F) is as follows. (Node labels show the values assigned to the decomposition variables and the simplified input expressions; edge labels show the returned μ-tree and the decomposition rule used for synthesis; and, each μ-expression at a terminal node is equivalent to the formula obtained from F by substituting for every variable x_i the value assigned to it on the path from the root to that node.)



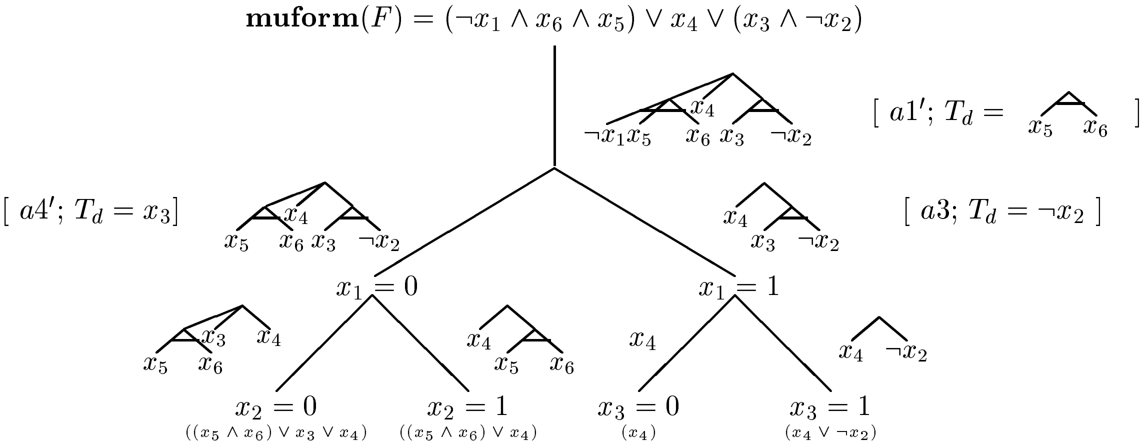
After decomposition by x_2, recursion of **mutree** stops both for $x_2 = 0$ and $x_2 = 1$ since the simplified input BEs ($x_1 \wedge x_3$ resp. 1) are μ-formulas. The trees $T_{f|_{x_2=0}}$ and $T_{f|_{x_2=1}}$ match with pattern $bo2$; hence, the corresponding tree $T_f = \langle \vee, \langle \wedge, x_1, x_3 \rangle, x_2 \rangle$ is returned, which is converted into the BE $(x_1 \wedge x_3) \vee x_2$. Notice that $F \equiv (x_1 \wedge x_3) \vee x_2$. $\square$

Example 2: Let $F = (x_1 \vee x_2) \wedge (x_2 \vee (\neg x_3 \vee 0)) \wedge ((\neg x_3 \wedge 1) \vee x_1)$. The schematic execution tree of **muform**(F) is as follows.



Here it is assumed that the algorithm selects first x_1 and later x_2 for decomposition. Note that the trees $T_{f|_{x_1=0}} = \langle \wedge, x_2, \neg x_3 \rangle$ and $T_{f|_{x_1=1}} = \langle \vee, x_2, \neg x_3 \rangle$ do not match any decomposition pattern; therefore, **muform**(F) returns ∇. In fact, it can be shown that F is not μ-equivalent. $\square$

Example 3: Let $F = [\,(x_5 \wedge x_6) \vee x_1 \vee (x_2 \wedge x_4) \vee [\neg x_2 \wedge (x_3 \vee x_4)]\,] \wedge [\,\neg x_1 \vee [(\neg x_3 \vee x_4) \wedge (\neg x_3 \vee (x_4 \wedge \neg x_2))]\,]$. The schematic execution tree of **muform**(F) is as follows.



Here the algorithm selects first x_1 and then x_2 resp. x_3 for decomposition. $\square$

The following theorem states the correctness of our algorithm.

Theorem 3.5 *Given a Boolean expression F,* **muform**(F) *returns an equivalent μ-expression G if f_F is μ-equivalent and ∇ otherwise.*

Proof. Let F be a BE in NNF. By induction on the number m of variables occurring in F, we show that **mutree**(F) returns the unique μ-tree for F if F is μ-equivalent and ∇ otherwise.

The case $m = 0$ is trivial (note that either **simp**(F) = 0 or **simp**(F) = 1).

Assume the statement holds for all m', $0 \le m' \le m$, and consider the case $m+1$. If G is a μ-expression, then, by the specification of **simp**, obviously the correct result is returned. Otherwise, there must exist a variable x that occurs in G at least two times. Let the computation select an arbitrary such x. Clearly, the number of variables occurring in $G_0 = G|_{x=0}$ and $G_1 = G|_{x=1}$ is bounded by m. Thus, by the induction hypothesis, T_{f_0} is assigned the μ-tree for G_0 if one exists and ∇ otherwise, and similarly T_{f_1} is assigned the μ-tree for G_1 if one exists and ∇ otherwise. If the computation returns ∇ because of $T_{f_0} = \nabla$ or $T_{f_1} = \nabla$, by (iv) of Lemma 3.4 the correct result is returned.

Otherwise, if $T_{f_0} = T_{f_1}$, then by (ii) of Lemma 3.4 the correct result is returned.

If $T_{f_0} \in \{0,1\}$ or $T_{f_1} \in \{0,1\}$, then, as one can see, T_{f_0} and T_{f_1} can only match a pattern derivable from Table 1. From the specification of **bmatch** and (iii) and (iv) of Lemma 3.4, it follows that the correct result is returned.

In the remaining case, the computation tries to match any decomposition pattern for T_{f_0} and T_{f_1} in which T_d occurs, which are the decomposition patterns derivable from Tables 2 and 3. It is immediate from the definition of *diff* that $\mathit{diff}(T_{f_0}, T_{f_1}) \neq \nabla$ and $\mathit{diff}(T_{f_1}, T_{f_0}) \neq \nabla$ can not hold simultaneously; hence, it follows from the specification of **match** and (i), (iii), and (iv) of Lemma 3.4 that the correct result is returned.

Consequently, the claimed statement holds for all m', $0 \leq m' \leq m+1$. This concludes the proof of the claim on the result of **mutree**.

Since the result of **mutree**(F) is a μ-tree for F if one exists and ∇ otherwise, the theorem clearly follows. □

We remark that the algorithms **muform** and **mutree** can be easily modified in order to return a more meaningful result if no μ-expression exists. In fact, at the point in the computation where the failure occurs, one has a counterexample, which is the two sets of variable assignments that generated the mismatching μ-trees. In Example 2, the counterexample is the pair of assignments $\langle x_1 = 0 \rangle$ and $\langle x_1 = 1 \rangle$.

4 Implementation and complexity

In this section, we sketch details for an efficient implementation of algorithm **muform**.

We store BEs in NNF as binary trees in the usual way. Notice that transforming a BE into NNF and building the formula tree is possible in linear time. Furthermore, it is easy to see that computing **simp**(F) is possible in linear time, as well as deciding if G is a μ-formula and computing $G|_{x=i}$, i.e., substituting value i for x in G.

From a μ-expression G in NNF represented by a binary tree, the corresponding μ-tree T_G can be constructed in linear time by a simple procedure. For efficient equality and matching tests, each μ-tree T is augmented as follows. For each possible terminal label l from 0, 1 and $x_i, \neg x_i$, $1 \leq i \leq n$, we provide a pointer $p(l,T)$ to the terminal node with label l in T, which has value *nil* if $l \notin \mathit{Term}(T)$; furthermore, we store at each node n of T a pointer $f(n)$ to its father and the distance $d(n)$ to the root. Clearly, T can be augmented in linear time. Thus,

Lemma 4.1 *Function* **tree**(G) *can be computed in linear time.*

Using the described data structure, equality of μ-trees can be decided quickly.

Lemma 4.2 *Checking μ-tree equality $T_1 = T_2$ for μ-trees T_1 and T_2 is possible in linear time.*

Proof. Consider the following procedure, which tries to match T_1 with T_2 bottom up. If $Term(T_1) \neq Term(T_2)$, output *false*; otherwise, proceed as follows. Initialize lists $L_0, L_1, \ldots$ such that L_i contains all terminal nodes n of T_1 with $d(n) = i$, and associate with each node n of T_1 a pointer $p(n)$, which, if n is a terminal node, is initialized with the terminal node in T_2 that has the same label and with value *nil* otherwise.

Then, repeat the following steps until only one node in a single list is left, in which case output *true*:

1. Remove the first node, say n, from the list with highest index (i.e., largest root distance) that is nonempty, say L_k.
2. If for $f(n)$ (i.e., the father of n), $p(f(n)) \neq nil$ (this means that $f(n)$ is in list L_{k-1}), check if $f(p(n)) = p(f(n))$; if not, output *false*, otherwise continue.
3. If $f(n)$ is not already in L_{k-1}, then, if $f(n)$ and $f(p(n))$ have the same label, add $f(n)$ to L_{k-1} and set $p(f(n))$ to the node $f(p(n))$ in T_2, otherwise output *false*.

It is not difficult to see that this procedure correctly decides $T_1 = T_2$; moreover, using the terminal label pointers $p(l, T)$, it can be computed in linear time. □

Lemma 4.3 *Function* **bmatch**(T_{f_0}, T_{f_1}, x) *can be computed in linear time.*

Proof. Indeed, checking whether T_{f_0} and T_{f_1} match a particular pattern $bl1$, $bl1'$ etc. derivable from Table 1 is possible in constant time, and T_f (resp. its augmented version) can be constructed from T_{f_0} and T_{f_1} in linear time. Since the number of patterns is constant, the linear time bound follows. □

Next we consider the remaining matching cases. First we note that *diff*(T_1, T_2) can be computed fast. We choose an implementation which returns a pointer to the root r of T_d in T_2 and a list of pointers to the sons $s_1, \ldots, s_n$ of r in case r is a nonterminal.

Lemma 4.4 *diff*(T_1, T_2) *can be computed in linear time.*

Proof. Consider the following algorithm. Check if $Term(T_1) \subset Term(T_2)$. (Using terminal label pointers, this is easy.) If not, output ∇, otherwise put the terminal nodes of T_2 labeled with some $t \notin Term(T_1)$ into lists $L_0, L_1, \ldots$ where all nodes with root distance i are placed in list L_i. Furthermore, initialize a flag for each node of T_2 indicating whether it is placed in some list L_i. Then, repeat the following step until only one node, say v, in a single list is left. Remove the first node, say n, from the list with highest index that is nonempty, say L_k; if for $f(n)$, the flag is not set (this means that $f(n)$ is not in L_{k-1}), set the flag and put $f(n)$ into L_{k-1}.

It is easy to see that this procedures terminates and that v is the only candidate for the root r of the tree T_d in the definition of $\mathit{diff}(T_1, T_2)$.

Then, compute the collection $s_1, \ldots, s_n$ of those sons d_i of v such that $Term(T_{d_i}) \subseteq Term(T_2) - Term(T_1)$, where T_{d_i} is the μ-tree with root s_i, $i = 1, \ldots, n$; notice that trees rooted at distinct sons of v have no common terminal labels. Finally, check if

$$\bigcup_{i=1}^{n} Term(T_{d_i}) = Term(T_2) - Term(T_1),$$

or, in case v is a terminal node, whether $\{v\} = Term(T_2) - Term(T_1)$; if so, return a pointer to v and, if $n > 0$, a list of pointers to $s_1, \ldots, s_n$, otherwise return ∇.

It is not hard to see that this algorithm correctly computes $\mathit{diff}(T_1, T_2)$. Moreover, by using terminal label pointers $p(l, T)$, it can be implemented to run in linear time. □

Lemma 4.5 *Function* $\mathbf{match}(T_{f_0}, T_{f_1}, T_d, v, x)$ *can be computed in linear time.*

Proof. Testing if a particular pattern $\mathcal{P}$ $a1$, $a1'$ etc. derivable from Tables 2–3 is matched is possible as follows. Check if the tree in which T_d occurs (i.e., T_{f_v}) matches its pattern; if so, modify T_{f_v} (resp. a copy thereof) by cutting off T_d, and restructure the tree according to T_{1-v} from $\mathcal{P}$. Then, check if the obtained μ-tree T satisfies $T = T_{f_{1-v}}$. Lemma 4.2 implies that this test is possible in linear time. Furthermore, the augmented μ-tree T_f can be constructed from T_{f_v} and T_d in linear time. Since the number of patterns to check is a constant, the lemma follows. □

We obtain from the above lemmas that the body of **mutree** can be executed in linear time. Hence, we obtain the following bound on the runtime of **muform**. Recall that $\|F\|$ denotes the size of the Boolean expression F.

Theorem 4.6 *Let F be a BE and let n_m be the number of variables occurring more than once in F. Then, in the described implementation* $\mathbf{muform}(F)$ *returns in time $O(\|F\| 2^{n_m})$ an μ-expression equivalent to F if one exists and ∇ otherwise.*

Proof. By Theorem 3.5 and Lemmas 4.1–4.5. □

Notice that 2^{n_m} is cautious for the number of calls to **muterm**, which can be smaller in many situations. In Example 3, for instance, this number is 7 while $2^{n_m} = 16$. The greedy strategy, which selects the variable with the maximum number of occurrences in G, is promising for keeping the recursion tree small in general, but will not achieve polynomiality, cf. Proposition 2.1. Furthermore, the factor 2^{n_m} can be reduced at the cost of a more sophisticated function $\mathbf{simp}(F)$, e.g. by simplifying also subexpressions $E \vee E$, $E \wedge E$, etc.

5 Complexity of recognizing μ-functions

The worst case complexity of a best algorithm for finding μ-expressions is closely tied to the complexity of recognizing μ-functions from BEs. For monotone BEs, this issue has been thoroughly dealt with in [17, 1]. In this section we derive new results for k-expressions [17] as well as general BEs.

Our algorithm has an immediate application to Mundici's problem of recognizing μ-functions from a monotone k-expression [17] (cf. Section 2). Since $O(\|F\| 2^{k log \|F\|}) = O(\|F\|^{k+1})$ we immediately obtain that Mundici's problem can be solved in polynomial time. In fact, we have from Theorem 4.6 the following more general result.

Theorem 5.1 *Let E be a k-expression. Then,* **muform**(E) *constructs a μ-expression F equivalent to E or outputs ∇ if none exists in time $O(\|F\|^{k+1})$.*

Now let us consider more general BEs. Of course, the results on monotone BEs entail the lower bound of co-**NP**-hardness for recognizing a μ-function f from a BE. We show co-**NP**-membership for the case where f essentially depends on all variables, and we give a close upper bound for the general case. First, we need some additional concepts.

Call any vector $r = (r_1, \ldots, r_n)$ of literals r_i, i.e., unary BFs $r_i(x)$ where $r_i(x) \equiv x$ or $r_i(x) \equiv \neg x$ for $i = 1, \ldots, n$ a *renaming.* For any BF $f(x_1, \ldots, x_n)$ and any n-ary renaming r, the renaming of f by r, f^r, is the BF defined by $f(r_1(x_1), \ldots, r_n(x_n))$. We say that a BF f is *renamable monotone* iff there exists a renaming r such that f^r is a monotone BF. Notice that every monotone BF is renamable monotone.

Lemma 5.2 *Every Boolean μ-function f is renamable monotone. Furthermore, f is a μ-function iff every renaming f^r of f is a μ-function.*

Proof. Let E be any μ-expression E in NNF such that $f_E = f$. Define the renaming r such that $r_i(x) \equiv \neg x$ if $\neg x_i$ occurs in E and $r_i(x) \equiv x$ otherwise; then, f^r is clearly monotone. The second part of the proposition is similar. □

Note that not every renamable monotone function is a μ-function. For example, consider f_E for $E = (x_1 \wedge x_2) \vee (x_2 \wedge x_3) \vee (x_1 \wedge x_3)$. We remark that testing whether a BE E defines a renamable monotone BF is co-**NP**-complete.

Call any subset $I \subseteq \{x_i, \neg x_i : 1 \leq i \leq n\}$ such that $\{x_i, \neg x_i\} \not\subseteq I$ for every i in $1, \ldots, n$ a *pseudo prime-implicant* of a BF $f(x_1, \ldots, x_n)$ if the following holds: (i) $f(\underline{a}) = 1$, where $\underline{a}(i) = 0$ if $x_i \notin I$ and $\underline{a}(i) = 1$ if $x_i \in I$, for $i = 1, \ldots, n$, and (ii) $f(\underline{b}) = 0$ for every $\underline{b}$ that differs from $\underline{a}$ only on the component j and $\{x_j, \neg x_j\} \cap I \neq \emptyset$, for j in $1, \ldots, n$.

Recall that a prime implicant of f (cf. Section 2) is a minimal subset $\{y_1, \ldots, y_k\} \subseteq \{x_i, \neg x_i : 1 \leq i \leq n\}$ not containing both x_i and $\neg x_i$, for any i in $1, \ldots, n$ such that for any BE E representing f, the BE $\neg(y_1 \wedge \cdots \wedge y_k) \vee E$ is a tautology. Clearly, every prime implicant of f is a pseudo prime-implicant of f. Notice that it can be tested in polynomial time, given a BE E and $I \subseteq \{x_i, \neg x_i : 1 \leq i \leq n\}$, whether I is a pseudo prime-implicant of f_E. The following can be easily shown.

Lemma 5.3 *If a BF f is monotone, then I is a pseudo prime-implicant of f iff I is a prime implicant of f.*

Using these lemmas, we show the next result. Say that a BE E resp. the represented BF $f = f_E(x_1, \ldots, x_n)$ *essentially depends on variable* x_i iff $f(\underline{a}) \neq f(\underline{b})$ for some n-vectors $\underline{a}$, $\underline{b} \in \{0,1\}^n$ that differ only on the i-th component, cf. [16]. The result captures the complexity of deciding whether a given BE E that depends on all its variables defines a μ-function. Notice that, strictly speaking, we assume that the input consists of the pair (E, C) where C is a certificate showing that E actually depends on all its variables. A polynomial-time testable certificate of size polynomial in $\|E\|$ exists, since deciding if a BE E depends on all its variables is in **NP**.

Theorem 5.4 *Let E be a BE on $x_1, \ldots, x_n$ which essentially depends on all its variables. Then, deciding whether f_E is a μ-function is co-**NP**-complete.*

Proof. The hardness part follows from the proof of [1, Theorem 5.7]. There it was shown that deciding whether a monotone BE $E = (x_1 \wedge x_{n+1}) \vee \cdots \vee (x_n \wedge x_{2n}) \vee E'$ on $x_1, \ldots, x_{2n}$ defines a μ-function is co-**NP**-hard, where E has no prime implicant of size less than 3. Clearly, E essentially depends on all its variables. Hence, our problem is co-**NP**-hard.

It remains to show the membership part. Notice that, as well-known, a BF $f(x_1, \ldots, x_n)$ is monotone iff $f(\underline{a}) = 1$ implies $f(\underline{b}) = 1$, for every $\underline{a}, \underline{b} \in \{0,1\}^n$ identical up to one component, say i, for which $\underline{a}(i) = 0$. Furthermore, if a BE F represents f, then the dual BF $d(f)$ is represented by the dualized expression $d(F)$ (obtained from F by replacing 0, 1, $\wedge$, and $\vee$ with 1, 0, $\vee$, and $\wedge$, respectively), and $\mathcal{PC}(f) = \mathcal{PI}(d(f))$ for any BF f.

Now let $\mathcal{S} = \langle w_1, \ldots, w_n, p, I, J \rangle$ be a structure as follows:

- each w_i, $i = 1, \ldots, n$ is a pair $\langle \underline{a}_i, \underline{b}_i \rangle$ of tuples $\underline{a}_i, \underline{b}_i \in \{0,1\}^n$ differing only on the i-th component such that $\underline{a}_i(i) = 0$, $\underline{b}_i(i) = 1$, and $f_E(\underline{a}_i) \neq f_E(\underline{b}_i)$;
- p is a triple $\langle j, \underline{c}, \underline{d} \rangle$ of $j \in \{1, \ldots, n\}$ and $\underline{c}, \underline{d} \in \{0,1\}^n$ such that $\underline{c}$ and $\underline{d}$ differ only on the j-th component, $\underline{c}(j) = 0$, $\underline{d}(j) = 1$, and $f_E(\underline{c}) \neq f_E(\underline{d})$;
- $I, J \subseteq \{x_1, \ldots, x_n\}$ such that $|I \cap J| \neq 1$ and the following holds: If $f_E(\underline{c}) \neq f_E(\underline{a}_j)$, then I is a pseudo prime-implicant of f_E^r and J is a pseudo prime-implicant of $d(f_E^r)$ where the renaming $r = (r_1, \ldots, r_n)$ is defined by $r_i(x) \equiv x$ if $f_E(\underline{a}_i) = 0$ and $r_i(x) \equiv \neg x$ otherwise, for $i = 1, \ldots, n$.

First note that, by our assumptions, tuples $w_1, \ldots, w_n$ as required always do exist.
Claim: E defines a μ-function iff no structure $\mathcal{S}$ as described exists.
Proof of the claim:
($\Rightarrow$) Assume that f is a μ-function. Then, f is renamable monotone. It follows that no triple p as required can exist such that $f_E(\underline{c}) \neq f_E(\underline{a}_j)$, for any admissible $w_1, \ldots, w_n$.

Since f_E depends on all its variables, it is easy to see that there exists a unique monotone renaming r of f_E, and that $r = (r_1, \ldots, r_n)$ where, for any admissible $w_1, \ldots, w_n$, we have $r_i(x) \equiv x$ if $f_E(\underline{a}_i) = 0$ and $r_i(x) \equiv \neg x$ otherwise, for $i = 1, \ldots, n$. By Lemma 5.2 we have that f^r is a μ-function. Hence, by Proposition 2.2 and Lemma 5.3 we obtain that I and J as required for the structure $\mathcal{S}$ do not exist. It follows that no structure $\mathcal{S}$ as described exists.

($\Leftarrow$) Assume that f_E is not a μ-function. If f_E is not renamable monotone, then $w_1, \ldots, w_n$ and p exist such that $f_E(\underline{c}) \neq f(\underline{a}_j)$; hence, a structure $\mathcal{S}$ as required clearly exists in that case. Otherwise, assume that f_E is renamable monotone. Let $w_1, \ldots, w_n$ be as required. Then the renaming $r = (r_1, \ldots, r_n)$ defined by $r_i(x) \equiv x$ if $f_E(\underline{a}_i) = 0$ and $r_i(x) \equiv \neg x$ otherwise, for $i = 1, \ldots, n$ is a monotone renaming of f. By Lemma 5.2 we have that f_E^r is not a μ-function. Hence, by Proposition 2.2 and Lemma 5.3 it follows that there exist pseudo prime-implicants I and J of f_E^r and $d(f_E^r)$, respectively, such that $|I \cap J| \neq 1$. Consequently, the structure $\mathcal{S} = \langle w_1, \ldots, w_n, \langle 1, \underline{a}_1, \underline{b}_1 \rangle, I, J \rangle$ satisfies the required properties. This completes the proof of our claim.

To conclude the proof of the theorem, notice that the size of any structure $\mathcal{S}$ satisfying the required properties is polynomial in $\|E\|$. Furthermore, it can be checked in polynomial time whether a structure $\mathcal{S}$ satisfies the required properties. Consequently, a guess for $\mathcal{S}$ proving that f_E is not a μ-function can be verified in polynomial time. It follows that our problem is in co-**NP**. $\Box$

This result can be generalized by a modified proof to the case where E depends on exactly k variables and k is part of the problem input.

It is an interesting question whether this result still holds if k is not provided for input. Note that deciding if a BE E essentially depends on a variable x is **NP**-complete; thus, k can be computed with n calls to an oracle in **NP** (where n is the number of variables). This can be improved to $O(\log n)$ oracle calls by doing a binary search over $[0, .., n]$, since deciding if E depends on at least m variables is in **NP**. Thus we have the following result. Notice that Θ_2^P (also $\Delta_2^P[O(\log n)]$, $\mathbf{P}^{\mathbf{NP}[O(\log n)]}$) denotes the class of problems decidable in polynomial time with a logarithmic number (measured in the input size) of calls to an oracle in **NP**, which was studied excessively in [13, 12, 22].

Theorem 5.5 *Deciding if a BE E on $x_1, \ldots, x_n$ defines a μ-function is in Θ_2^P.*

It remains open whether also completeness for Θ_2^P holds. However, it is not clear how to transform the satisfiability problem, SAT, in polynomial time to this problem. This suggests to suspect that the problem is not Θ_2^P-complete, cf. [12].

6 Conclusion

We considered the problem of generating a μ-expression, i.e., a Boolean expression in which no variable occurs more than once, equivalent to a given Boolean expression.

Such expressions have applications in learning theory, for instance [21, 19] and correspond to series-parallel graphs and networks [16, 18]. We presented an algorithm that solves this problem in time linear in the input size times 2^{n_m}, where n_m is the number of repeatedly occurring variables, knowing that a polynomial time algorithm is not possible unless $\mathbf{P} = \mathbf{NP}$. However, the algorithm performs well for subclasses of Boolean expressions. In particular, our algorithm works in polynomial time for Mundici's class of k-expressions considered in [17]. Thus, we obtained that the problem of deciding whether for a given k-expression an equivalent μ-expression exists [17] is decidable in polynomial time.

Furthermore, we have derived new complexity results for recognizing μ-functions from Boolean expressions. We determined the exact complexity for expressions dependent on all variables, and we gave fairly close bounds for the general case.

It remains future work to design polynomial-time algorithms for generating μ-expressions for further relevant classes of Boolean expressions.

Acknowledgements. The author would like to thank the referees for their valuable comments and Vladimir Gurvich and Helmut Veith for their help in translating Russian; the remark on the modified version of **muform** is due to a referee. In particular, the author is indebted to one of the referees for pointing out to him work on repetition-free formulas (μ-formulas) which has been carried out in the Soviet Union, and for kindly providing copies of relevant papers.

References

[1] T. Eiter. Exact Transversal Hypergraphs and Application to Boolean μ-Functions. Technical Report CD-TR 92/38, CD-Laboratory for Expert Systems, TU Vienna, Austria, June 1992. To appear in *Journal of Symbolic Computation.*

[2] M. Garey and D. S. Johnson. *Computers and Intractability – A Guide to the Theory of NP-Completeness.* W. H. Freeman, New York, 1979.

[3] V. A. Gurvich. On Repetition-Free Boolean Functions. *Uspekhi Math. Nauk*, 32:183–184, 1977. [Russian].

[4] V. A. Gurvich. On the Normal Form of Positional Games. *Soviet Math. Dokl.*, 25:572–575, 1982. English translation of Dokl. Akad. Nauk SSSR, 264:30–33 (1982).

[5] V. A. Gurvich. Some Properties and Applications of Complete Edge-Chromatic Graphs and Hypergraphs. *Soviet Math. Dokl.*, 30:803–807, 1984. English translation of Dokl. Akad. Nauk SSSR, 279:1306–1310 (1984).

[6] V. A. Gurvich. Criteria for Repetition-Freeness of Functions in the Algebra of Logic. *Soviet Math. Dokl.*, 43:721–726, 1991. English translation of Dokl. Akad. Nauk SSSR, 318 (1991).

[7] V. A. Gurvich. Positional Structures and Chromatic Graphs. *Soviet Math. Dokl.*, 45:168–172, 1992. English translation of Dokl. Akad. Nauk SSSR, 322:828–831 (1992).

[8] V. A. Gurvich and L. Libkin. Trees as Semilattices. *Discrete Mathematics*, 1994. To appear.

[9] H. Hunt III and R. Stearns. Monotone Boolean Functions, Distributive Lattices, and the Complexities of Logics, Algebraic Structures, and Computation Structures (Preliminary Report). In *Proceedings STACS-86*, volume 210 of *LNCS*, pages 277–287, 1986.

[10] H. Hunt III and R. Stearns. The Complexity of Very Simple Boolean Formulas With Applications. *SIAM J. Comp.*, 19(1):44–70, 1990.

[11] D. S. Johnson. A Catalog of Complexity Classes. volume A of *Handbook of Theoretical Computer Science*, chapter 2. Elsevier Science Publishers B.V. (North-Holland), 1990.

[12] J. Kadin. $P^{NP[O(\log n)]}$ and Sparse Turing-Complete Sets for NP. *Journal of Computer and System Sciences*, 39:282–298, 1989.

[13] M. Krentel. The Complexity of Optimization Problems. *Journal of Computer and System Sciences*, 36:490–509, 1988.

[14] A. V. Kuznetsov. On Repetition-Free Contact Schemes and Repetition-Free Superpositions of the Functions in the Algebra of Logic. *Trudy Mat. Inst. Steklov.*, 51:186–225, 1958. [Russian].

[15] Y. Matiyasewich. Possible Nontraditional Methods for Establishing the Deducibility of Propositional Formulas. *Voprosy Kibernet.* (*Moscow*), 113:87–90, 1987. [Russian].

[16] D. Mundici. Functions Computed by Monotone Boolean Formulas With No Repeated Variables. *Theoretical Computer Science*, 66:113–114, 1989.

[17] D. Mundici. Reducibility of Monotone Formulas to μ-Formulas. In *3rd Workshop Computer Science Logic (CSL)*, LNCS 440, pages 267–270, 1989.

[18] D. Mundici. Solution of Rota's Problem on the Order of Series-Parallel Networks. *Advances in Applied Mathematics*, 12:455–463, 1991.

[19] L. Pitt and L. Valiant. Computational Limitations on Learning from Examples. *Journal of the ACM*, 35:965–984, 1988.

[20] B. A. Trakhtenbrot. On the Theory of Repetition-Free Contact Schemes. *Trudy Mat. Inst. Steklov.*, 51:226–269, 1958. [Russian].

[21] L. Valiant. A Theory of the Learnable. *Commun. ACM*, 27(11):1134–1142, 1984.

[22] K. Wagner. Bounded query classes. *SIAM J. Comp.*, 19(5):833–846, 1990.

[23] I. Wegener. *The Complexity of Boolean Functions.* Wiley - Teubner series in computer science. Wiley & Sons Ltd and Teubner, Stuttgart, 1987.