The Boolean Formula Value Problem as Formal Language

Klaus-Jörn Lange

WSI, Universität Tübingen,
Sand 13, D72076 Tübingen, Germany
email: lange@informatik.uni-tuebingen.de

Abstract. The Boolean formula value problem asks for the Boolean output value of a given input formula. We code it as a formal language $\mathcal{D}_+ \subset \{a, b\}^*$. $\mathcal{D}_+$ is a nonregular, visibly pushdown language. We give automata for $\mathcal{D}_+$ which enable us to derive some of its syntactic equations. It is unknown whether the given list of equations is complete. Using these equations some algebraic properties of the syntactic monoid of $\mathcal{D}_+$ are sketched.

Keywords: Boolean formula value problem, syntactic monoid, word equations, algebraic approach, TC^0 vs NC^1

1 Introduction

The Boolean formula value problem (*BFVP*) consists in evaluating Boolean formulas. Compared to the *P*-complete circuit value problem it is of rather low complexity; depending in the coding of the input formula the evaluation can be done in NC^1 if the formula is given in a parenthesized way (or in polish normal form) or deterministically in logarithmic space if the input formula is given as a tree coded by nodes and edges. In this paper we will be interested in the first of these two possibilities. It is quite easy to show the NC^1-hardness of the Boolean evaluation problem for formulas given by parenthesized words or expressions in polish normal form. On the other hand,the construction to prove membership in NC^1 is quite involved [5].

The class NC^1 is quite attractive from the formal language viewpoint since Barrington showed, that each regular set, whose syntactic monoid contains an nonsolvable group is NC^1-complete ([3]). Thus there exist regular sets whose word problem is NC^1-complete.

The aim of this investigation is to consider the Boolean formula evaluation problem from the algebraic formal language view-point despite the fact that it is nonregular and hence its syntactic monoid is infinite. This short note comes without proofs. Just the constructions and some examples are given. Proofs for the correctness of the automata constructions can be found in the Studienarbeit of Bernd Brumm ([4]).

This paper is structured as follows: we first express (a special version of) the Boolean formula value problem via the Dyck language $\mathcal{D}$ over one pair of

parenthesis. Then we present automata constructions. Finally, some first algebraic properties of *BFVP* are given including a list of defining equations which are not known to be complete.

1.1 Preliminaries

Let $L \subseteq \{a,b\}^*$. We say that two words $x, y \in \{a,b\}^*$ are *congruent modulo* L if and only if we have

$$zxz' \in L \Longleftrightarrow zyz' \in L$$

for all $z, z' \in \{a,b\}^*$. By $[x]_L$ we denote the congruence class of x modulo L. For the resulting notions and results concerning the *syntactic monoid* of L we refer to [6].

In our investigations of the Boolean formula value problem we will use the notion of visibly pushdown automata and languages as introduced by Alur ([1]). These were known before as input-driven languages and are characterized by the restriction that the modification of the stack in terms of push or pop moves are not dependent in the state but only in the input symbol. While the one-sided Dyck languages are visibly pushdown languages, the two-sided ones are not.

2 Coding the Boolean Formula Value Problem

The coding of a problem, i.e.: its representation as formal language containing words which code problem instances, can usually be done in different ways without affecting the complexity of the problem. But the resulting formal languages will differ signifcantly in their algebraic properties expressed in their syntactic monoids. For instance will a parenthesis-language contain a zero in its syntactic monoid, while in the corresponding polish normal form language arbitrary words are subwords of valid expressions, which means that there is no zero in the syntactic monoid.

Our aim is to choose a representation of BFVP which leads to a syntactic monoid as simple as possible. That is why we will represent boolean formulas by the $NAND$-operation, we will use a polish normal form instead of using parentheses, and we will represent binary trees by dyck words and not by the more usual Lukasiewicz words.

It is well known that every Boolean function can be expressed by the (binary) $NAND$-function together with the Boolean constant $TRUE$. The conversion is possible by replacing $AND(x,y)$ by $NAND(TRUE, NAND(x,y))$, $OR(x,y)$ by $NAND(NAND(TRUE,x), NAND(TRUE,y))$, and $NOT(x)$ by $NAND(TRUE,x)$. The size of the resulting $NAND$-formula is linear in the size of the original AND, OR-formula.

Hence we will consider as input formulas, which are to be evaluated, complete binary trees labelled with NAND-function, i.e.; all inner nodes have two predecessors and are labelled by the NAND-function, while the remaining nodes are leaves of indegree zero labelled by the Boolean constant TRUE.

As mentioned before, coding these formulas as graphs, with vertices and edges, leads to evaluation problems which are hard for deterministic logarithmic space.

The well-known alternative is to use parentheses to express the tree structure. Throughout of this paper we will represent the opening paranthesis by the letter a and the closing one by the letter b.

We code complete binary trees as follows: the tree consisting of a single (root) vertex is coded by the empty word λ. If a vertex has two outgoing edges leading to its predecessors the left edge is labelled by a and the right one by b. The tree is then read in-order from left to right. Thus the word $aabb$ represents the binary tree with 3 leaves, the left subree containing 2 leaves, and the right one containing one leave. Switching the left and right subtree yields the tree represented by $abab$. This gives a one-to-one corrspondence between complete binary trees and the Dyck language $\mathcal{D} \subset \{a, b\}^*$.

We decided in favour of labelling the edges and against the more usual labelling of the vertices, which would lead to the well known representation of complete binary trees by the Lukasiewicz language.

While (contextfree) grammars are in general easier to construct for the Lukasiewicz language, in the Dyck case the resulting syntactic (bicyclic) monoid is more simple. For instance $\mathcal{D}$ is generated by the single equation $ab = \lambda$ whereas the Lukasiewicz language results in the equations $aba = a$, $abb = b$, and $aab = a$.

A tree labelled by Boolean functions and constants evaluates either to TRUE or to FALSE. In this way the Dyck set $\mathcal{D}$ is divided into the two disjoint subsets $\mathcal{D} = \mathcal{D}_+ \cup \mathcal{D}_-$ where $\mathcal{D}_+$ consists in those elements of $\mathcal{D}$ which represent a tree (labelled with the NAND-function and the constant TRUE) which evaluates to TRUE and $\mathcal{D}_-$ contain those which evaluate to FALSE. Thus $\mathcal{D}_+$ is a special formulation of the Boolean formual value problem which makes $\mathcal{D}_+$ NC^1-complete.

In the following, we are going to investigate the properties of the formal language $\mathcal{D}_+$.

3 Properties of $\mathcal{D}_+$

It is easy to see that $\mathcal{D}_+$ is a context-free language. For instance, the set $\mathcal{D}_+b$[1] is generated by the grammar with the rules $S \to aaSSS|aSaSS|aaSSaSS|b$.

Obviously, $\mathcal{D}_+$ is a *visibly push-down language* as defined by Alur([1]), i.e. for each element of the terminal alphabet it is determined whether the stack of an push-down automaton accepting $\mathcal{D}_+$ is pushed (here by the symbol a) or popped (here by the symbol b).

Alur et al. showed that a language is visibly push-down if and only if a certain congruence relation is of finite index ([2]).

A close inspection shows that the resulting congruence relation divides the set $\mathcal{D}$ into four classes

$$\mathcal{D} = F \cup N \cup P \cup T.$$

[1] This set might be regarded as the Lukasiewicz-version of $\mathcal{D}_+$

This was explicated in [4].

These four classes might be explained in the following way: T consists in those Dyck-words w which represent formulas, which evaluate to TRUE and if we add a suffix v such that wv is still a Dyck word, wv evaluates to TRUE, as well.

F consists of the dual class of words representing fomulae evaluating to FALSE regardless how they are completed by a Dyck suffix.

P and N are represent those formulas which evaluate to TRUE (respectively, FALSE) whose value can be changed by a suffix.

The shortest member of these four classes are $\lambda \in P, ab \in N, aabb \in T,$ and, $abaabb \in F$. We have

$$\mathcal{D}_+ = T \cup P \text{ and } \mathcal{D}_- = F \cup N.$$

These four classes can be characterized in the following way: every $w \in \mathcal{D}$ admits a unique decomposition $w = aw_1baw_2 \cdots aw_nb$ for some $n \geq 0$ and some $w_i \in \mathcal{D}$. We then have

- $w \in P$ iff n is even and for all $1 \leq j \leq n$ we have $w_j \in \mathcal{D}_+$,
- $w \in N$ iff n is odd and for all $1 \leq j \leq n$ we have $w_j \in \mathcal{D}_+$,
- $w \in T$ iff there exists some $i < n/2$ such that $w_{2i+1} \in \mathcal{D}_-$ and for all $1 \leq j \leq 2i$ we have $w_j \in \mathcal{D}_+, and$
- $w \in F$ iff there exists some $i < n/2$ such that $w_{2i} \in \mathcal{D}_-$ and for all $1 \leq j \leq 2i-1$ we have $w_j \in \mathcal{D}_+$.

Thus $w \in \mathcal{D}_+$ iff w consists in a concatenation of an even number of words $avb, v \in \mathcal{D}_+$, followed by a word $aub, u \in \mathcal{D}_-$, or followed by nothing. If that number is odd,we have $w \in \mathcal{D}_-$.

3.1 Automata for $\mathcal{D}_+$

Alur showed how to construct a push-down automaton out of the congruence whose classes serve both as stack alphabet and as set of states. In the case of $\mathcal{D}_+$ the resulting automaton can be simplified by keeping as set of states $\{F, N, P, T\}$ but shrinking the stack alphabet to $\Gamma := \{P, N\}$ with the pushing transitions

$$\begin{array}{l} T\ a \to F, P \\ P\ a \to P, P \\ N\ a \to P, N \\ F\ a \to F, N \end{array}$$

and the popping transitions

$$\begin{array}{l} T, P\ b \to N \\ P, P\ b \to N \\ N, P\ b \to T \\ F, P\ b \to T \\ T, N\ b \to P \\ P, N\ b \to P \\ N, N\ b \to F \\ F, N\ b \to F \end{array}.$$

The resulting automaton has a very regular structure like an infinite binary tree. It is thus possible to represent uniquely each combination of state and stack content of this automaton as a binary string in a way, that configurations connected by transitions are of a very similar shape. A pushing transition, i.e. reading an a, acts on a binary string ending in the bits $xy,\ x, y \in \{0,1\}$, by appending the second to last bit x which makes the string now ending in xyx. A popping transition, i.e. reading a b, acts on a binary string ending in the bits $xyz,\ x, y, z \in \{0,1\}$, by deleting the last bit z and then exchanging the remaining last two bits which makes the string now ending in yx. Interpreting these strings as binary numbers, the four pushing and eight popping rules can be given by the following rules: The automaton has infinitely many states labelled by natural numbers greater or equal to 4. The starting state is state 6. If the (one-way) input reads an a and the automaton is in state $4n+i$ for some n and some $i < 4$ it goes to state $8n+j$ where i and j are given by:

$$\begin{aligned} 4n+0 &\to 8n+0 \\ 4n+1 &\to 8n+2 \\ 4n+2 &\to 8n+5 \\ 4n+3 &\to 8n+7 \end{aligned}.$$

If a b is read we have the following rules:

$$\begin{aligned} 8n+0 &\to 4n+0 \\ 8n+1 &\to 4n+0 \\ 8n+2 &\to 4n+2 \\ 8n+3 &\to 4n+2 \\ 8n+4 &\to 4n+1 \\ 8n+5 &\to 4n+1 \\ 8n+6 &\to 4n+3 \\ 8n+7 &\to 4n+3 \end{aligned}.$$

Since $\mathcal{D}_+$ is NC^1-completeit is thus an NC^1-complete task to read a Dyck word and make according to the input letters the corresponding modulo computations and then to determine whether the result is 6 or 7 (corresponding to $\mathcal{D}_+$) or 4 or 5 (corresponding to $\mathcal{D}_-$).

4 The Syntactic Monoid of $\mathcal{D}_+$

We now investigate the infinite syntactic monoid of $\mathcal{D}_+$. To do so, we first consider equations fulfilled by the syntactic congruence of $\mathcal{D}_+$. After that we give a few algebraic properties of $\mathcal{D}_+$.

4.1 Equations

It is easy to check the validity of the following equations fulfilled in $\{a,b\}^*$ by $\mathcal{D}_+$ either directly or using the automata given in the previous section:

1. $abab = \lambda$,
2. $aabbb = b$,
3. $aabaabb = aabba$,
4. $abbabb = babb$,
5. $aabba^iab = aabba^i$ for all $i \geq 0$, and
6. $abb^iabaabb = b^iabaabb$ for all $i \geq 1$.

The last two (sets of) equations express that to the right of the word in $aabba^*$ respectively to the left of the word in $b^*babaabb$ the $\mathcal{D}_+$-evaluation is simply just a $\mathcal{D}$-evaluation, i.e.: the reduction of the subword ab to λ.

It is unclear, whether these equations are complete, i.e.: whether there are new equations not implied by the given ones, or independent, i.e.; whether one of them is implied by the others.

Another interesting question is, whether these equations can be directed in either way (either from left to right or from right to left) such that each sequence of applications of the bidirectional equations could be simulated by a sequence of applications of the unidirectional versions of these equations.

The last question is closely connected to the search for rewriting systems converting each $w \in \{a,b\}^*$ into a normal form w' (for instance a shortest word w.r.t. some ordering) such that w' and w are congruent modulo $\mathcal{D}_+$. An application of the Knuth-Bendix-procedure to (unidirectional versions of) the given equations didn't give new ones.

4.2 Algebraic Properties of the $\mathcal{D}_+$

We finally give a few algebraic properties of the syntactic monoid $\mathcal{M}_{\mathcal{D}_+}$ of $\mathcal{D}_+$.

A standard tool to investigate the structure of a monoid are *Green's relations* (see for instance [6]). For finite monoids the D- and the J-relation allways coincide. This can hold in the infinite case, as well; an example is the syntactic monoid of the language $\mathcal{D}$ which is the *bicyclic monoid*. In contrast to that the syntactic monoid of $\mathcal{D}_+$ has one J-class (i.e. for all $x, y \in \{a,b\}^*$ there exist $z, z' \in \{a,b\}^*$ such that x is congruent zyz' modulo $\mathcal{D}_+$), but more then one D-class, since $aabb$ and the empty word λ are not D-equivalent, i.e.: there is no $x \in \{a,b\}^*$ such that both $[aabb]_L\mathcal{M}_{\mathcal{D}_+} = [x]_L\mathcal{M}_{\mathcal{D}_+}$ and $\mathcal{M}_{\mathcal{D}_+}[x]_L = \mathcal{M}_{\mathcal{D}_+}[\lambda]_L$.

We finally mention, that the syntactic monoid of $\mathcal{D}_+$ is regular. That is, for all $x \in \{a,b\}^*$ there exists some y (called an *inverse* of x) such that xyx is congruent with x and yxy with y modulo $\mathcal{D}_+$.

If that inverse element y is uniquely determined by x, such a monoid is called inverse. While the syntactic monoid of $\mathcal{D}$ is inverse, that of $\mathcal{D}_+$ is not; for instance the word bab has the two different inverses a and $aaabb$.

These algebraic differences between $\mathcal{D}$ and $\mathcal{D}_+$ express their differences in complexity. While $\mathcal{D}$ is in TC^0 (and complete w.r.t. Turing reducibilities), $\mathcal{D}_+$ is NC^1-complete(w.r.t. many-one reducibilities).

Compared to $\mathcal{D}_+$ a totally different NC^1-complete problem is the word problem of A_5 (or of any other regular set whose syntactic monoid contains a nonsolvable group ([3])). In the proof of this fact the action of a Boolean gate with inputs

x and y is in some sense simulated by evaluating the commutator $x^{-1}y^{-1}xy$ of the algebraic simulations of x and y. The proof makes uses of the fact, that a nonsolvable group contains arbitrarily long, nonvanishing commutatorchains. This leads to the question whether we can find in the syntactic monoid of $\mathcal{D}_+$, which is regular and has only one J-class, a similar algebraic simmulation of the action of Boolean gates, which would give a new proof of the NC^1-hardness of the Boolean formula value problem.

Acknowledgement

I would like to thank the referees for their careful reading of this note.

References

1. R. Alur. Visibly Pushdown Languages. In *Proc. 36th ACM Symp. on Theory of Computing*, pages 202–211, 2004.
2. R. Alur, V. Kumar, P. Madhusudan, and M. Viswanathan. Congruences for Visibly Pushdown Languages. In *Proc. ICALP*, pages 1102–1114, 2005.
3. D.A. Barrington. Bounded-width polynomial-size branching programs can recognize exactly those languages in NC^1. *J. Comp. System Sci.*, 38:150–164, 1989.
4. B. Brumm. Das Auswertungsproblem als formale Sprache. Private Communication, 2011.
5. S. R. Buss. The Boolean formula value problem is in ALOGTIME. In *Proc. 19th Ann. ACM Symp. on Theory of Computing*, pages 123–131, 1987.
6. J. E. Pin. Varieties of Formal Languages. *Plenum*, London, 1986.