Boolean Programs and Quantified Propositional Proof Systems

Stephen Cook and Michael Soltys

July 22, 2003

Abstract

We introduce the notion of Boolean programs, which provide more concise descriptions of Boolean functions than Boolean circuits. We characterize nonuniform PSPACE in terms of polynomial size families of Boolean programs. We then show how to use Boolean programs to witness quantifiers in the subsystems G_1 and G_1^* of the proof system G for the quantified propositional calculus.

1 Introduction

A Boolean circuit can be explained as a straight line program in which each line defines a new Boolean variable in terms of input variables and previously defined variables. In this paper we introduce an extension of this idea: A *Boolean program* is a straight line program in which each line defines a Boolean function in terms of previously defined Boolean functions. Just as the evaluation problem for Boolean circuits is complete for polynomial time, the evaluation problem for Boolean programs is complete for polynomial space.

In section 3 we show how a family of Boolean programs provides a natural way of describing the language accepted by a polynomial space Turing machine. Since the validity problem for quantified Boolean formulas is complete for PSPACE, Boolean programs can be used to witness the existential quantifiers in a valid quantified Boolean formula. In sections 4 and 5 we begin the study of how this idea can be used to shed light on the computational power of proof systems involving quantified Boolean formulas.

2 Boolean Programs

A *Boolean program* P is specified by a finite sequence $\{f_1, \ldots, f_m\}$ of function symbols, where each symbol f_i has an associated arity k_i, and an associated defining equation

$$f_i(\bar{p}_i) := A_i$$

where $\bar{p}_i$ is a list $p_1, ..., p_{k_i}$ of variables and A_i is a formula (see below) all of whose variables are among $\bar{p}_i$ and all of whose function symbols are among $f_1, ..., f_{i-1}$.

The *formulas* of P are defined as follows:

(i). 0, 1, and p are formulas, for any variable p.

(ii). If f is a k-ary function symbol in P and $B_1, ..., B_k$ are formulas, then $f(B_1, ..., B_k)$ is a formula.

(iii). If A and B are formulas, then $(A \wedge B)$, $(A \vee B)$, and $\neg A$ are formulas.

We give the obvious semantics to Boolean programs. We identify 0 with false and 1 with true. Each function symbol f_i in P is interpreted by a Boolean function (also denoted f_i) according to its defining equation $f_i(\bar{p}_i) := A_i$, where $f_i : \{0,1\}^{k_i} \longrightarrow \{0,1\}$. We say that P *computes* each such function f_i.

Example 1 The following is a Boolean program computing the parity of n variables using $\lceil \lg(\lg n) \rceil$ function definitions, the longest of size $O(n)$. We assume that $n = 2^{2^m}$ for some integer m, so that $\lceil \lg(\lg n) \rceil = \lg(\lg n) = m$. This is not an unreasonable assumption since we can always pad the input with zeroes to put it in this form. Note that the original function has only two variables.

$$\begin{aligned}
&f_0(p_1, p_2) := (\neg p_1 \wedge p_2) \vee (p_1 \wedge \neg p_2)\\
&f_1(p_1, p_2, p_3, p_4) := f_0(f_0(p_1, p_2), f_0(p_3, p_4))\\
&f_2(p_1, \ldots, p_{16}) :=\\
&\qquad f_1(f_1(p_1, \ldots, p_4), f_1(p_5, \ldots, p_8), f_1(p_9, \ldots, p_{12}), f_1(p_{13}, \ldots, p_{16}))\\
&\qquad\quad \vdots\\
&f_n(p_1, \ldots, p_{2^{2^n}}) :=\\
&\qquad f_{n-1}(f_{n-1}(p_1, \ldots, p_{2^{2^{n-1}}}), \ldots, f_{n-1}(p_{(2^{2^n} - 2^{2^{n-1}} + 1)}, \ldots, p_{2^{2^n}}))
\end{aligned}$$

We are interested in the complexity of the following problem: Given a Boolean program P computing a k-ary function f and given an argument $\bar{a} \in \{0,1\}^k$, evaluate $f(\bar{a})$.

For example, we can evaluate $f_1(0,1,0,0)$ from the parity program above as follows:

$$\begin{aligned} f_1(0,1,0,0) &= f_0(f_0(0,1), f_0(0,0)) \\ &= f_0(1,0) \\ &= 1 \end{aligned}$$

The *size* $|P|$ of a Boolean program $P = \{f_1, ..., f_m\}$ is the total number of symbols in the sequence of definitions $f_1(\bar{p_1}) := A_1, ..., f_m(\bar{p_m}) := A_m$.

Theorem 1 The following problem can be solved by a multitape Turing machine in space $\mathrm{O}(|P|)$: Given a Boolean program P computing a function f, and given an argument $\bar{a}$ for f, find the value $f(\bar{a})$.

Proof of Theorem 1. Suppose P has function definitions

$$f_1(\bar{p_1}) := A_1, ..., f_m(\bar{p_m}) := A_m$$

We describe a recursive procedure for evaluating $f_i(\bar{b})$ given i and the argument $\bar{b}$, which proceeds by evaluating the leftmost innermost unevaluated subterm of A_i. In the general case, certain subterms of A_i will have been evaluated. Let A_i' be the simplification of A_i resulting by replacing each subterm of A_i by its value, whenever the value is known. (We assume that all the connectives $\wedge, \vee, \neg$ are evaluated whenever their arguments are known.) Let $f_j(\bar{c})$ be the leftmost innermost unevaluated subterm of A_i', where $\bar{c} \in \{0,1\}^{k_j}$. Then call the procedure recursively to evaluate $f_j(\bar{c})$. Note that $j < i$.

Observe that the depth of the recursion is at most m, and at each point in the computation each activation record on the stack corresponds to an evaluation $f_i(\bar{b})$, where all the indices i are distinct. Further, note that the number of bits of space required for each such record, say to evaluate $f_i(\bar{b})$, is bounded by a constant times the number of symbols in the original definition $f_i(\bar{p_i}) := A_i$. This is true, even taking into account that the number of bits needed to express one symbol f_j or p_j could be $\log m$. The reason is that the original definition $f_i(\bar{p_i}) := A_i$ is available on the (read-only) input tape, and need not be repeated at each level. Thus the total number of bits of working space needed to evaluate $f_m(\bar{a})$ is $O(|P|)$. $\square$

3 Uniform and Non-uniform PSPACE

Recall that a language $L \subseteq \{0,1\}^*$ is in PSPACE iff some multitape Turing machine decides L using space at most $p(n)$, for some polynomial p. The corresponding nonuniform class is PSPACE/poly. We say that $L \in$ PSPACE/poly iff there exists an advice function $A(n)$ and a polynomial space Turing Machine M such that $x \in L$ iff $M(x, A(|x|)) =$ "yes", where M has two read-only input tapes, one containing x and the other containing the advice $A(|x|)$. We say that a family $P_1, P_2, ...$ of Boolean programs *computes* a language L if P_n computes the characteristic function of $L \cap \{0,1\}^n$, for all n. The family is *polynomial size* if $|P_n| \leq p(n)$ for some polynomial p.

The following result is the analog for space of the standard characterization of nonuniform polytime: A language L is in P/poly iff L can be computed by some polynomial size family of Boolean circuits.

Theorem 2 A language L is in PSPACE/poly iff L is computed by some polynomial size family of Boolean programs.

Proof of Theorem 2. $\Longleftarrow$ Let $A(n)$ be the encoding of the Boolean program that computes $f(x_1, \ldots, x_n)$. Design M with advice $A(n)$ as follows: on input $x \in \{0,1\}^n$, M evaluates $f(x_1, \ldots, x_n)$. By Theorem 1, this can be done in PSPACE.

$\Longrightarrow$ Suppose $L \in$ PSPACE/poly. Then there exists an advice function $A(n)$ and a polynomial space TM M such that $x \in L$ iff $M(x, A(|x|)) =$ "yes". For each n and each input x of length n, each configuration in the computation of M with input $(x, A(n))$ can be coded in a standard way by a bit string $\bar{p} = p_1, ..., p_r$, where $r = n^k + k$, for some k. We define the Boolean program

$$P_n = \{\bar{g_0}, \bar{g_1}, ..., \bar{g_r}, \overline{Init}, Out, f\}$$

where

$$\bar{g_0}(\bar{p}) = \bar{q} = \text{ the successor configuration to } \bar{p}$$

$$\overline{g_{i+1}}(\bar{p}) = \bar{g_i}(\bar{g_i}(\bar{p})), 0 \leq i < r$$

$$\overline{Init}(x) = \bar{p} = \text{ the initial configuration for input } x$$

$$Out(\bar{p}) = 1 \text{ iff } \bar{p} \text{ codes an accepting configuration}$$

$$f(x) = Out(\bar{g_r}(\overline{Init}(x)))$$

Notice that $\bar{g}_i(\bar{p})$ is the configuration 2^i steps after configuration $\bar{p}$, and 2^r is an upper bound on the total number of steps in the computation, since no configuration can repeat. Thus $f(x) = 1$ iff M accepts x with advice $A(n)$. $\square$

We say that a family $P_1, P_2, ...$ of Boolean programs is *uniform* if some Turing machine can, given n, output a code for P_n in time polynomial in n. The following result is proved by a slight modification of the above proof, and holds true for a variety of definitions of *uniform*.

Theorem 3 A language L is in PSPACE iff L is computed by some uniform polynomial size family of Boolean programs.

The *Boolean program value problem* is: Given a Boolean program P computing a function f, and given an assignment $\bar{a}$ to the arguments of f, determine whether $f(\bar{a}) = 1$. The following result follows easily from Theorem 1 and the proof of Theorem 2.

Theorem 4 The Boolean program value problem is log-space complete for PSPACE.

We close this section with one other characterization of nonuniform PSPACE. It is well known [3, pg 456] that the set of valid quantified Boolean formulas is complete for PSPACE. It is not hard to modify the proof of this to prove the following:

Theorem 5 $L \in$ PSPACE/poly iff there exists a polynomial size family of quantified Boolean formulas $\{\alpha_1(x_1), \alpha_2(x_1, x_2), \alpha_3(x_1, x_2, x_3), \ldots\}$ such that $x \in L \cap \{0,1\}^n$ iff $\alpha_n(x)$ is valid.

4 Witnessing G_1 Proofs

In this section we consider G_1, a restricted version of the quantified propositional proof system G (see [2, pg 57] for a detailed definition) and constructively show how to witness the existential quantifiers with Boolean programs of size polynomial in the size of the proof. For our purposes it suffices to know that G is based on the propositional part of Gentzen's system LK (for $\wedge, \vee, \neg$,0,1), with rules to introduce $\wedge, \vee, \neg$ and the propositional quantifiers $\exists p, \forall p$. The system G_1 is G restricted to Σ_1^q formulas; that is formulas equivalent to existential formulas. Here we consider a slight restriction of G_1 in which formulas must be *strict* Σ_1^q; that is formulas beginning with a block of existential quantifiers followed by a quantifier-free formula. Our result readily generalizes to G_1 itself.

Our purpose is to give a poly-time procedure α which does the following: given as input a G_1 proof π of a sequent S of the form

$$\ldots, A_i(\bar{p}), \ldots, \exists \bar{x}_j B_j(\bar{p}, \bar{x}_j), \ldots \to \ldots, C_s(\bar{p}), \ldots, \exists \bar{y}_t D_t(\bar{p}, \bar{y}_t), \ldots \tag{1}$$

where the A's, B's C's, and D's are quantifier-free formulas and $\bar{p}$ is a list of all free variables, the procedure outputs a sequent

$$\ldots, A_i(\bar{p}), \ldots, B_j(\bar{p}, \bar{q}_j), \ldots \to \ldots, C_s(\bar{p}), \ldots, D_t(\bar{p}, \bar{f}_t(\bar{p}, \bar{q})), \ldots \tag{2}$$

together with a Boolean program P that computes the functions $\bar{f}$ such that (2) is a valid sequent (using the semantics of P). Here the function list $\bar{f}$ has distinct function symbols for distinct formulas D_t and the argument list $\bar{q}$ of each function symbol includes places for all existentially quantified variables on the left side of (1) (with distinct variables for distinct formulas B_j).

It follows from results presented in [2] that the problem of transforming a G_1 proof π and values for the variables into witnesses for the quantifiers has the complexity of Papadimitriou's NP search class PLS (Polynomial Local Search). Thus apparently the full power of Boolean programs is not needed, and it would be interesting to find a natural restriction that does the job.

The procedure α converts each sequent (1) in the proof π in turn to the form (2), building a Boolean program P as it goes, where P computes all function symbols in all sequents that have been converted so far. The axioms for G_1 are the quantifier-free sequents $p \to p$, $0 \to$, $\to 1$ (where p is a variable) and are unchanged in the conversion. The other sequents in π follow from earlier sequents by rules of LK and they are converted in a manner depending on the rule. Here we explain the conversion for the more interesting cases.

In what follows we omit mention of the free variables $\bar{p}$ for clarity, but they are there.

contraction left:

$$\frac{\Gamma\, A, A \to \Delta}{\Gamma, A \to \Delta}$$

Let $A = \exists \bar{z} B(\bar{z})$. Suppose the top has been converted to

$$\Gamma'(\bar{q}), B(\bar{u}_1), B(\bar{u}_2) \to \Delta'(\bar{f}(\bar{q}, \bar{u}_1, \bar{u}_2))$$

Then the procedure converts the bottom to

$$\Gamma'(\bar{q}), B(\bar{u}) \to \Delta'(\bar{f}'(\bar{q}, \bar{u}))$$

where we add the function definitions $\bar{f}'(\bar{q}, \bar{u}) := \bar{f}(\bar{q}, \bar{u}, \bar{u})$. to the Boolean program P.

contraction right:

$$\frac{\Gamma \to \Delta, A, A}{\Gamma \to \Delta, A}$$

Let $A = \exists \bar{z} B(\bar{z})$ and suppose that the top has been converted to

$$\Gamma'(\bar{q}) \to \Delta', B(\bar{g}(\bar{q})), B(\bar{h}(\bar{q}))$$

The procedure converts the bottom to

$$\Gamma'(\bar{q}) \to \Delta', B(\bar{f}(\bar{q}))$$

where $\bar{f}$ has function definitions

$$\bar{f}(\bar{q}) := \begin{cases} \bar{g}(\bar{q}) & \text{if } B(\bar{g}(\bar{q})) \\ \bar{h}(\bar{q}) & \text{otherwise} \end{cases}$$

(This is easily changed to the format of a Boolean program.)

$\wedge$-intro right:

$$\frac{\Gamma \to \Delta, A \quad \Gamma \to \Delta, B}{\Gamma \to \Delta, A \wedge B}$$

Suppose the procedure has converted the top two sequents to

$$\Gamma'(\bar{q}_1) \to \Delta'(\bar{g}(\bar{q}_1)), A'(\bar{s}(\bar{q}_1)) \quad \text{and} \quad \Gamma'(\bar{q}_2) \to \Delta'(\bar{h}(\bar{q}_2)), B'(\bar{r}(\bar{q}_2))$$

with a Boolean program P to compute the functions. Then the sequent

$$\Gamma'(\bar{q}_1) \to \Delta'(\bar{g}(\bar{q}_1)), \Delta'(\bar{h}(\bar{q}_1)), A'(\bar{s}(\bar{q}_1)) \wedge B'(\bar{r}(\bar{q}_1))$$

is valid. Now proceed as in **contraction right** to combine the two versions of Δ'.

cut rule:

$$\frac{\Gamma \to \Delta, A \quad A, \Gamma \to \Delta}{\Gamma \to \Delta}$$

Suppose the top two sequents have been converted to

$$\Gamma'(\bar{q}_1) \to \Delta'(\bar{g}(\bar{q}_1)), A'(\bar{r}(\bar{q}_1)) \quad \text{and} \quad A'(\bar{q}), \Gamma'(\bar{q}_2) \to \Delta'(\bar{h}(\bar{q}, \bar{q}_2))$$

with a Boolean program P. The idea is to substitute $\bar{r}(\bar{q}_1)$ for $\bar{q}$ and substitute $\bar{q}_1$ for $\bar{q}_2$ in the second sequent to match the two versions of A. Thus we introduce the function definitions

$$\bar{h}'(\bar{q}_1) := \bar{h}(\bar{r}(\bar{q}_1), \bar{q}_1)$$

so the sequent

$$\Gamma'(\bar{q}_1) \to \Delta'(\bar{g}(\bar{q}_1)), \Delta'(\bar{h}'(\bar{q}_1))$$

is valid when P is augmented with $\bar{h}'$. Now we proceed to combine the two versions of Δ' as in the case **contraction right**.

∃-intro right:

$$\frac{\Gamma \to \Delta, A(B)}{\Gamma \to \Delta, \exists x A(x)}$$

Suppose the top has been converted to $\Gamma'(\bar{q}) \to \Delta', (A'(B))(\bar{f}(\bar{q}))$. Then the procedure converts the bottom to

$$\Gamma'(\bar{q}) \to \Delta', (A'(g(\bar{q}))(\bar{f}(\bar{q}))$$

where g has defining equation $g(\bar{q}) := B(\bar{f}(\bar{q}))$

5 Witnessing G_1^* Proofs

The system G_1^* is identical to G_1, except that proofs are required to be treelike (i.e. each sequent in the proof can be used at most once as a hypothesis for a later sequent). We show that the existential quantifiers in G_1^* proofs can be witnessed by functions defined using a restricted form of Boolean program. These Restricted Boolean programs are like extension definitions in Extended Frege Proofs [1, 2], and hence they correspond to Boolean circuits. Thus the existential quantifiers in a G_1^* proof can be witnessed in time polynomial in the length of the proof.

We define a *Restricted Boolean program* to be the same as a Boolean program, with the restriction that in each function definition $f_i(\bar{p}_i) := A_i$, each occurrence of a function symbol f_j in A_i must be in the context $f_j(\bar{p_j})$ (i.e. the left side of the definition of f_j).

Thus all functions in a Restricted Boolean program can be evaluated at a given assignment to its variables by a single pass through the program (which is certainly a poly-time algorithm). If each term $f_i(\bar{p_i})$ in the program is replaced by f_i, and if each symbol $:=$ is replaced by the Boolean connective $\leftrightarrow$, then the program becomes a sequence of extension definitions defining values for the variables f_i. The f_i can also be thought of as gates in a Boolean circuit.

We now describe a polytime procedure β which does the same thing as the procedure α in the previous section, except the input proof π must be a G_1^* proof instead of a G_1 proof, and the Boolean program P computing the functions $\bar{f}$ is a Restricted Boolean program. Our proof can be easily modified to show that Extended Frege systems p-simulate G_1^*, thus providing an alternative proof to the more interesting half of Lemma 4.6.3 in [2].

Our new procedure β is similar to procedure α, except in converting each successive sequent S in the proof π, we will be free to modify (and not just extend) the Restricted Boolean programs generated for the previous conversions. This modification was not done in procedure α because the programs might be needed to convert later sequents (and their duplication could cause an exponential increase). However since the present proof π is treelike, each Boolean program is needed only once in its original form.

We now explain for each case considered in the previous section how to modify the argument to produce a Restricted Boolean program P.

contraction left:

The function definition $\bar{f}'(\bar{q}, \bar{u}) := \bar{f}(\bar{q}, \bar{u}, \bar{u})$ is not allowed in a Restricted Boolean program. Instead we modify the Restricted Boolean program P computing $\bar{f}(\bar{q}, \bar{u_1}, \bar{u_2})$ by identifying corresponding variables in the two lists $\bar{u_1}, \bar{u_2}$ and renaming them $\bar{u}$ throughout P. The result is a Restricted Boolean program computing $\bar{f}'(\bar{q}, \bar{u})$.

contraction right:

No modification is needed.

$\wedge$-intro right:

Simply modify the programs computing $\bar{h}(\bar{q_2})$ and $\bar{r}(\bar{q_2})$ by renaming $\bar{q_2}$ to $\bar{q_1}$, so that the programs compute $\bar{h}(\bar{q_1})$ and $\bar{r}(\bar{q_1})$

cut rule:

We form a restricted program P computing $\bar{h}'(\bar{q_1})$ by concatenating the restricted program P_1 computing $\bar{r_1}(\bar{q_1})$ with a modification P_2' of the restricted program P_2 computing $\bar{h}(\bar{q}, \bar{q_2})$. We may assume that the function symbols in P_1 and P_2 are disjoint, and P_2 has no variables other than $\bar{q}$ and $\bar{q_2}$ (by deleting them from function terms and substituting 0 for them in other terms). The modification consists of first renaming $\bar{q_2}$ by $\bar{q_1}$ throughout P_2, second replacing each variable q_i in the list $\bar{q}$ by the corresponding term $r_i(\bar{q_1})$ in the list $\bar{r}(\bar{q_1})$, and finally replacing each function term $f(...)$ by $f'(\bar{q_1})$. Thus semantically $f'(\bar{q_1}) = f(...)$. In particular $\bar{h}'(\bar{q_1}) = \bar{h}((\bar{r}(\bar{q_1}), \bar{q_1})$ as required.

∃-intro right:

No modification is needed.

References

[1] Stephen Cook and Robert Reckhow. The relative efficiency of propositional proof systems. *Journal of Symbolic Logic*, 44(1), 1979.

[2] Jan Krajicek. *Bounded Arithmetic, Propositional Logic, and Complexity Theory.* Cambridge, 1995.

[3] Christos H. Papadimitriou. *Computational Complexity.* Addison-Wesley, 1994.

[4] Alasdair Urquhart. The complexity of propositional proofs. *Bulletin of Symbolic Logic*, 1(4):425–467, 1995.