Series Logo
Volume 00, 19xx

Some Fundamental Properties of Boolean Ring Normal Forms

Jieh Hsiang and Guan Shieng Huang

ABSTRACT. Boolean ring is an algebraic structure which uses *exclusive* − *or* instead of the usual *or*. It yields a unique normal form for every Boolean function. In this paper we present several fundamental properties concerning Boolean rings. We present a simple method for deriving the Boolean ring normal form directly from a truth table. We also describe a notion of normal form of a Boolean function with a don't-care condition, and show an algorithm for generating such a normal form. We then discuss two Boolean ring based theorem proving methods for propositional logic. Finally we give some arguments on why the Boolean ring representation had not been used more extensively, and how it can be used in computing.

1. Introduction

Boolean ring is an algebraic structure which is equivalent to Boolean algebra. The major representational differences are that Boolean ring uses *exclusive-or* (+) instead of *or* (∨) to represent Boolean functions, and that there is no need for negation in Boolean ring. Furthermore, there is a *unique* Boolean ring normal form for every Boolean function. It is curious, however, that in spite of its long history and elegant algebraic properties, the Boolean ring representation has rarely been used in the computational context.

In this paper we present several fundamental results concerning Boolean rings. Some of the results are so elementary and simple that they can (and should) be taught in a first-year logic design course. It is surprising that most of them seem not known, at least to our knowledge.

In Section 2 we give an overview of some known results about Boolean ring normal form, notably the term rewriting method for producing the normal form through simplification.

In Section 3 we show a method for generating the Boolean ring normal form directly from the truth table of a Boolean function. This method is conceptually very simple and is straightforward to implement. Unlike the well-known Karnaugh

---

1991 *Mathematics Subject Classification.* Primary 54C40, 14E20; Secondary 46E25, 20C20.
This research is supported in part by grant NSC 85-2213-E-002-052 of the National Science Council of the Republic of China.

method, which may produce different conjunctive normal forms depending on which prime implicants are chosen, our method is completely deterministic.

We then present a notion of normal form for a Boolean function with a don't-care condition, and an effective way for generating this normal form.

A Boolean function with a don't-care condition A is a (partial) Boolean function whose values on truth assignments in A are undefined. This concept is useful in circuit design in which the task of specification can be simplified by ignoring truth assignments that are inconsequential. However, during the actual design and fabrication of such a specification, the truth assignments in A must be given some value. Depending on the designs, the values may be filled differently. Thus, even if two designs satisfy the same specification, they may represent different functions. This makes the circuit verification problem significantly more complicated.

From an algebraic point of view, a truth table with a con't-care condition represents a class of Boolean functions. Thus, if there is a mechanical way of choosing a "normal form" out of the entire class of functions, then checking the equivalence of two functions (under the same don't-care condition) becomes reducing the two functions into the same normal form. we present such a method in section 4. Our method is based on the Buchberger algorithm for generating the Gröbner basis of the ideal defined by A, and uses this Gröbner basis to produce a unique normal form for each equivalent class of Boolean functions. Our method should make verifying correct implementations of specification with a don't-care condition considerably easier.

In Section 5 we show two ways of performing automated theorem proving for propositional logic in the Boolean ring framework. The first one is a resolution-style procedure based on Buchberger algorithm. This procedure was first described in [**KN**][1], and is different from resolution in several ways. First the input is not restricted to clausal form. Second it employs the inference rule of *simplification*, which has no natural counterpart in the resolution framework. Simplification is a powerful way of reducing the search space in theorem proving [**BH**].

The second method we present is a Davis-Putnam like procedure. In addition to *splitting*, an inference rule employed in Davis-Putnam [**DP, DLL**], it also utilizes the inference rule of *simplification*. Since simplification is more natural and has much more reduction power than the unit clause rule of Davis-Putnam, we feel that it may have some advantage over Davis-Putnam as a basis for an efficient satisfiability checker.

In the last section of the paper, we give some reasons on why Boolean ring has not been used more extensively in logic or computer science. We also point out areas where it can be used productively.

2. The Boolean ring normal form

A *Boolean ring* is a commutative ring $(B; +, *, 0, 1)$ in which $*$ is *idempotent* (i.e., $x * x = x$) and $+$ is *nilpotent* (i.e, $x + x = 0$)[2]. The operator $+$ is known in

[1] The first approach to automated theorem proving using the Boolean ring representation is, to our knowledge, [**H85**]

[2] It is easy to show that the nilpotence of $+$ is a consequence of the idempotence of $*$.

logic design as *exclusive-or*. By introducing the relationship

$$\begin{aligned} x \wedge y &\equiv x * y \\ x \vee y &\equiv x + y + xy \\ \neg x &\equiv 1 + x \end{aligned}$$

one can show that the corresponding algebraic structure $(B; \wedge, \vee, \neg, 0, 1)$ is a Boolean algebra [**S**]. The isomorphic relationship between Boolean algebras and Boolean rings was found in 1936 by Stone [**S**] and could probably date back to 1927 by Zhegalkin [**Z**].

A Boolean function of n variables is a mapping from $\{0,1\}^n$ to $\{0,1\}$. In the rest of the paper we reserve $\mathcal{B}$ for the set $\{0,1\}$ and $\mathcal{F}$ for the set of Boolean functions with n variables.

The operators $*$ and $+$ can be extended to the functional level. Let $f, g \in \mathcal{F}$, we define $f * g$ as a function h such that $h(x) = f(x) * g(x)$ for all $x \in \mathcal{B}^n$. The operator $+$ is defined similarly. Since a Boolean ring is also a field, we know that $(\mathcal{F}; +, *, 0, 1)$ is a commutative ring, where 0 and 1 are the constant functions 0 and 1.

Let $x_1, \cdots, x_n \in \mathcal{F}$ be the projection functions such that the value of x_i depends only on the i^{th} argument, it is easy to see that the set $\{x_1, \cdots, x_n, 1\}$ generates the entire ring $\mathcal{F}$. In other words, $\mathcal{F}$ can also be regarded as a *polynomial ring* $\mathcal{B}[x_1, \cdots, x_n]$.

A Boolean function m is a *monomial* if it can be represented as a conjunction:

$$\prod_{x \in \mathcal{V}} x \text{ where } \mathcal{V} \subseteq \{x_1, x_2, ..., x_n\}.$$

If $\mathcal{V}$ is empty, then m is the unity function 1. In this definition, as in the rest of the paper, we use $\prod$ as a shorthand for a chain of conjunctions and $\sum$ for a chain of "+". Note that since $*$ is idempotent, each Boolean variable appears in a monomial only once.

A Boolean function can be expressed as a sum of monomials. Such a representation is called a *Boolean polynomial*. By the nilpotence of $+$, an identical pair of monomials in a Boolean polynomial can be deleted. Thus, each monomial can appear at most once in a Boolean polynomial. With these simplifications, a Boolean function can be represented by a unique Boolean polynomial normal form which is either 1, 0, or a sum of distinct monomials. This normal form is the *Boolean ring normal form* ($BRNF$). We emphasize that unlike the well-known disjunctive normal form (DNF), BRNF is unique for any Boolean function.

THEOREM 2.1. (Stone 1936) *There exists a unique BRNF for each Boolean function with n variables.*

Given a Boolean function, we can derive its Boolean ring normal form by reduction using a canonical set of rewrite rules. This method was first presented in [**H85**]. We describe it here briefly.

A *rewrite rule* is an oriented equation. A rewrite rule can be applied to reduce a term, via equational replacement, in the left-to-right fashion. This process is called *simplification*. We require that the simplification relation be *well-founded*. That is, no term can be simplified indefinitely. The well-foundedness requirement can usually be ensured by imposing a simplification ordering when orienting equations into rules [**D**]. If a term cannot be simplified by a set of rewrite rules R, then we

say that the term is *R-irreducible*. If a term s is simplified by R to a term t and t is R-irreducible, then we say that t is an *R-normal form* of s. Note that the normal form of a term may not be unique. A set of rewrite rules R is called a *canonical rewrite system* if the R-normal form of every term is unique.

The following is a canonical rewrite system, called BA, for Boolean algebra:

$$\begin{aligned}
x \vee y &\rightarrow x * y + x + y \\
x \equiv y &\rightarrow x + y + 1 \\
\neg x &\rightarrow x + 1 \\
x \supset y &\rightarrow x * y + x + 1 \\
x * 1 &\rightarrow x \\
x * 0 &\rightarrow 0 \\
x * x &\rightarrow x \\
x + x &\rightarrow 0 \\
x + 0 &\rightarrow x \\
x * (y + z) &\rightarrow x * y + x * z
\end{aligned}$$

We remark that the operators $+$ and $*$ are commutative and associative.

Given a Boolean function, one can apply the rules of BA to simplify it (in arbitrary order) until no more simplification is possible. The resulting (unique) normal form is the BRNF of the Boolean function.

For example, given $p \wedge (p \vee q)$, it can be transformed into it BRNF as follows:

$$\begin{aligned}
p \wedge (p \vee q) &\rightarrow p(pq + p + q) \\
&\rightarrow ppq + pp + pq \\
&\rightarrow pq + pq + p \\
&\rightarrow p
\end{aligned}$$

3. Generating BRNF from a truth table

In this section we describe a method for generating the Boolean ring normal form of a Boolean function represented by a truth table. Our method works on the truth table directly and does not need auxiliary notions such as prime implicants in Karnaugh map. Furthermore, since BRNF is unique, our method is also more deterministic.

Let D denote $\{0,1\}^n$, the domain of the Boolean function, where n is assumed to be a fixed integer throughout this section. Given a truth assignment s (of the n variables $x_1, \cdots, x_n$) we use s_i to denote the value of x_i in s.

Definition 3.1. Let s and t be two truth assignments. We say that s is a *positive extension* of t if

1. for all i such that $t_i = 1$, $s_i = 1$,
2. there exists an i such that $t_i = 0$ and $s_i = 1$.

The set of positive extensions of s is denoted $pex(s)$.

Definition 3.2. Let s be a truth assignment. We use

- $|s|$ to denote the number of s_i which is 1,
- $br(s)$ to denote the monomial $\prod_{s_i=1} x_i$. For the truth assignment s which assigns 0 to all Boolean variables, $br(s)$ is 1.

For example, if $s = (0, 0, 1)$, then $|s| = 1$, $pex(s) = \{(1, 0, 1), (0, 1, 1), (1, 1, 1)\}$, and $br(s) = z$.

We are now ready to give the *flip-tag algorithm* which produces the BRNF from a truth table.

Flip-tag algorithm

Input: a truth table f

Output: the BRNF of f

1. For each $s \in D$, let $tag(s) := f(s)$.
2. For i from 0 to n, for each s such that $|s| = i$, if $tag(s) = 1$, then for every t in $pex(s)$, $tag(t) := tag(t) + 1$.
3. output $\sum_{tag(s)=1} br(s)$.

EXAMPLE 3.3. Consider the following truth table:

x	y	z	f
0	0	0	0
0	0	1	1
0	1	0	1
1	0	0	0
0	1	1	0
1	0	1	1
1	1	0	0
1	1	1	0

At the beginning the tag function of the truth assignments is the same as f. The Flip-tag algorithm dictates that the tag be examined, from top to bottom. Whenever a 1 is encountered, the tag of all the positive extensions of that truth assignment is reversed. For the given function, the algorithm works as the following table shows:

x	y	z	$tag(s)$				$final - tag(s)$	$br(s)$
0	0	0	0				0	1
0	0	1	1	1			1	z
0	1	0	1		1		1	y
1	0	0	0				0	x
0	1	1	0	1	0		0	yz
1	0	1	1	0			0	xz
1	1	0	0		1	1	1	xy
1	1	1	0	1	0	1	1	xyz

By collecting $br(s)$ of those truth assignments s whose final tags are 1, we get $y + z + xy + xyz$ as the BRNF of f.

For the ease of demonstrating the example, we created a new tag column whenever a 1 in the tag of a truth assignment s is encountered. In this case, the tag of each the positive extension of s is changed in the same column.

We now show the correctness of the flip-tag algorithm. Before we start, we need a few definitions and simple lemmas.

DEFINITION 3.4. Let s be a truth assignment, we use $rep(s)$ to denote the Boolean expression $\prod_{s_i=1} x_i \prod_{s_j=0} \bar{x}_j$, where $\bar{x}$ is $\neg x$ or, equivalently, $x + 1$.

For example, if $s = (0, 0, 1)$, then $rep(s) = \bar{x}\bar{y}z$.

PROPOSITION 3.5. *Let V be a set of variables and f be a Boolean function (represented by a truth table), then*

1. $1 + \sum_{\emptyset \neq U \subseteq V} \prod_{x_j \in U} x_j = \prod_{x_j \in V} (x_j + 1)$.
2. $f = \sum_{s \ where \ f(s)=1} rep(s)$.

These are two basic Boolean properties and we skip the proofs.

LEMMA 3.6. *Let s be a truth assignment, then $rep(s) = br(s) + \sum_{t \in pex(s)} br(t)$.*

PROOF. We first observe that for each $t \in pex(s)$,

$$\begin{aligned} br(t) &= (\prod_{s_i=1} x_i) \prod_{t_j=1 \& s_j=0} x_j \\ &= br(s) \prod_{t_j=1 \& s_j=0} x_j. \end{aligned}$$

Let V be the set of Boolean variables such that $s_j = 0$. Then

$$\begin{aligned} rep(s) &= \prod_{s_i=1} x_i \prod_{s_j=0} (x_j + 1) \\ &= br(s) \prod_{s_j=0} (x_j + 1) \\ &= br(s)(1 + \sum_{\emptyset \neq U \subseteq V} \prod_{x_j \in U} x_j) \\ &= br(s)(1 + \sum_{t \in pex(s)} \prod_{t_j=1 \ \& s_j=0} x_j) \\ &= br(s) + br(s)(\sum_{t \in pex(s)} \prod_{t_j=1 \ \& s_j=0} x_j) \\ &= br(s) + \sum_{t \in pex(s)} br(s) \prod_{t_j=1 \ \& s_j=0} x_j \\ &= br(s) + \sum_{t \in pex(s)} br(t). \end{aligned}$$

□

Thus, by Proposition 3.5 and Lemma 3.6, we have

LEMMA 3.7. *Let f be a Boolean function given as a truth table, then $f = \sum_{s, \ where \ f(s)=1} (br(s) + \sum_{t \in pex(s)} br(t))$.*

The correctness of the $flip - tag$ algorithm follows from Lemma 3.7. For each of truth assignment s whose value is 1, a copy of $br(s)$ and one of each of $br(t)$, where $t \in pex(s)$, need to be added to the Boolean expression of f. However, since $+$ is nilpotent, any two identical copies of monomials can be eliminated. By working the main loop from the less defined (fewest 1's) to the more defined truth assignments, we are ensured that the monomial representing less defined truth assignments will not be reconsidered later in the algorithm. The tag function is then used to keep track of whether a truth assignment s really has the value 1 when all truth assignments less defined than s have already been considered.

4. Normal form of a Boolean function with a don't-care condition

Let A be a subset of $\mathcal{B}^n$. A (partial) Boolean function f is said to be *with the don't-care condition* A if the values of f on truth assignments in A are undefined. Such a partial Boolean function can also be regarded as a class of (total) Boolean functions with identical values for truth assignments not in the set A. From this viewpoint, we say that two (total) Boolean functions f and g are *equivalent under* A, denoted $f \approx_A g$, if $f(s) = g(s)$ for all $s \notin A$. It is obvious that $\approx_A$ defines an equivalent relation. We use $[f]_A$ to denote the equivalent class of f under $\approx_A$.

In this section we describe a method for deriving a unique normal form for an equivalent class of $\approx_A$. This problem is interesting at least for the following reason: In circuit design one often encounters a circuit specified as a truth table with a don't-care condition (say A), since one may not be concerned with some of the output values. During actually design, one needs to assign values to the truth assignments in A in order to have a working circuit. The values assigned, however, may differ due to different design techniques. Thus, two different designers may produce circuits representing different functions although both are correct with respect to the original design. In a mathematical formulation, it simply means that the two functions designed, f and g, belong to the same equivalent class of $\approx_A$. A challenge that arises is how one may verify the correctness of such two circuits. Most of the known methods do not apply since they usually assume that the two circuits under investigation represent the same function.

The method which we are going to present here provides a solution to this problem.

4.1. Generating set of an ideal. Let $\mathcal{F} = \mathcal{B}[x_1, \cdots, x_n]$ be a polynomial ring, and A be a don't-care condition. Let $\mathcal{F}_\mathcal{A} = \{f | f \approx_A 0\}$. It is easy to see that $\mathcal{F}_\mathcal{A}$ is an ideal of $\mathcal{F}$.

The equivalence of the two relations $\mathcal{F}/\approx_A$ and $\mathcal{F}/\mathcal{F}_\mathcal{A}$ is established by the following lemma, whose proof is trivial.

LEMMA 4.1. *$f \approx_A g$ if and only if $f + g \in \mathcal{F}_\mathcal{A}$.*

This lemma suggests a scenario for solving our problem. That is, if there is an effective way for checking the membership of $\mathcal{F}_\mathcal{A}$, then the equivalence of f and g can be easily decided.

We present such a method based on a notion of generating set dcalled *Gröbner basis*. Informally, a Gröbner basis is a set of rewrite rules which reduces all members of the same equivalent class to a unique normal form. (The normal form for $\mathcal{F}_\mathcal{A}$ under a Gröbner basis is, obviously, 0.)

DEFINITION 4.2. Given a polynomial ring $\mathcal{F} = \mathcal{B}[x_1, \cdots, x_n]$, a don't-care condition A and its associated ideal $\mathcal{F}_\mathcal{A}$, a *generating set*, $\mathcal{G}$, of $\mathcal{F}_\mathcal{A}$ is a set of polynomials $\{g_1, \cdots, g_n\}$ such that

- for every $p \in \mathcal{F}_\mathcal{A}$, there exists polynomials $p_1, \cdots, p_n$ such that $\sum_i p_i g_i = p$, and
- for every $p_1, \cdots, p_n$, $\sum_i p_i g_i \in \mathcal{F}_\mathcal{A}$.

LEMMA 4.3. *The singleton set $\{g | g(s) = 1 \; iff \; s \in A\}$ is a generating set of $\mathcal{F}_\mathcal{A}$.*

The proof is obvious.

EXAMPLE 4.4. Let $A = \{(1,0,0),(1,1,0),(1,0,1)\}$. Then the set $\{xyz + x\}$ is a generating set of $\mathcal{F}_{\mathcal{A}}$. (The expression $xyz + x$ can be generated using the flip-tag algorithm given in the previous section.)

4.2. Gröbner basis. Most of the results in this section apply to any polynomial ring over a field. However for simplicity, we present them only in terms of Boolean polynomial rings, which is sufficient for our purpose.

Continuing from Example 4.4, since $xyz + x \approx_A 0$, we can treat it as a rewrite rule $xyz \to x$. By imposing a total ordering on the set of monomials, we can orient all polynomials into rewrite rules of the form $m \to \alpha$ where m is a monomial and α is the rest of the polynomial. Such an ordering can *always* be obtained by first imposing a total ordering on the set of Boolean variables, say $x_1 > \cdots > x_n$, then compare two monomials first by their sizes, then by the members of the monomials[3]. We remark that the resulting ordering is well-founded.

A polynomial p, when treated as a rule $m \to \alpha$, induces a reduction $\longrightarrow_p$ defined as follows: Given two Boolean polynomials p_1 and p_2, $p_1 \longrightarrow_p p_2$ if $p_1 = m_1 m + \beta$ for some monomial m_1 and polynomial β and that $p_2 = m_1\alpha + \beta$. For simplicity we shall drop the subscript p unless confusion may occur. Since the ordering used to orient polynomials into rules is well-founded, the associated reduction relation $\to$ is irreflexive, antisymmetric, and well-founded. We call such a reduction relation a *noetherian relation*.

Let R be a set of (Boolean polynomial) rewrite rules and $\to$ be its associated reduction relation, a Boolean polynomial p is said to be in *R-normal form* or *R-irreducible* if p is a BRNF which is not reducible using rules in R. Let $\xrightarrow{*}$ and $\xleftrightarrow{*}$ be the trnasitive and reflexive-transitive closures of $\to$, then $\to$ is *Church-Rosser* if for every polynomials p and q such that $p \xleftrightarrow{*} q$, there is an r such that $p \xrightarrow{*} r \xleftarrow{*} q$. It is well-known that if $\to$ is noetherian and Church-Rosser, then every p has a unique normal form (see, e.g., [**DJ**]).

DEFINITION 4.5. A *Gröbner basis* of an ideal $\mathcal{F}_{\mathcal{A}}$ is a generating set $\mathcal{G}_{\mathcal{A}}$ such that, when oriented as rules, the reduction relation is noetherian and Church-Rosser.

THEOREM 4.6 (Buchberger). *For every ideal $\mathcal{F}_{\mathcal{A}}$ of a Boolean polynomial ring, there is a Gröbner basis $\mathcal{G}_{\mathcal{A}}$. Furthermore, a polynomial p is in $\mathcal{F}_{\mathcal{A}}$ if and only if $p \xrightarrow{*} 0$.*

An immediate consequence of the theorem is that

COROLLARY 4.7. *Let $\mathcal{F}_{\mathcal{A}}$ be an ideal and $[f]_A$ be an equivalent class. Then all Boolean functions in $[f]_A$ have the same $\mathcal{G}_{\mathcal{A}}$-normal form.*

In other words, once we find the Gröbner basis of $\mathcal{F}_{\mathcal{A}}$, we can produce all the intended normal forms.

4.3. Buchberger algorithm for generating Gröbner bases. An algorithm for generating the Gröbner basis of an ideal was given by Buchberger [**B**]. As is consistent with the rest of the paper, we simplify the algorithm to work only with Boolean polynomials.

[3]Other total orderings can be obtained in similar ways. For instance, by comparing two monomials using the multiset ordering [**DM**], one can order monomials in a different way.

Let $\mathcal{R}$ be a set of rewrite rules (converted from Boolean polynomials) and let $m_1 m \to \alpha$ and $m_2 m \to \beta$ be two rules in $\mathcal{R}$, where $m \neq 1$, then $m_1\beta + m_2\alpha$ is an $\mathcal{R}$-*critical polynomial*. An $\mathcal{R}$-critical polynomial is *non-trivial* if its $\mathcal{R}$-normal form is not 0. We use $CP(\mathcal{R})$ to denote the set of rewrite rules obtained by orienting all non-trivial $\mathcal{R}$-critical polynomials derived from rules in $\mathcal{R}$.

A set of rewrite rules $\mathcal{R}$ is *inter-reduced* if every rule $r \in \mathcal{R}$ is irreducible with respect to $\mathcal{R} \setminus \{r\}$. By reducing rules in $\mathcal{R}$ among each other, one can always making $\mathcal{R}$ into an inter-reduced one. We call the resulting (inter-reduced) set of rules $reduced(\mathcal{R})$.

With these definitions in mind, we introduce a version of the Buchberger algorithm for generating a Gröbner basis for a set of polynomials.

Input: A set of polynomials $\mathcal{P}$ and a total ordering on the set of variables.
Output: a Gröbner basis $\mathcal{G}$ of $\mathcal{P}$.

1. Initiate $\mathcal{R}$ to be the set of rules converted from the polynomials in $\mathcal{P}$.
2. While $CP(\mathcal{R} \cup \{xx \to x\}_{x \in \mathcal{V}}) \neq \emptyset$, do
   (a) $\mathcal{R} := \mathcal{R} \cup CP(\mathcal{R} \cup \{xx \to x\}_{x \in \mathcal{V}})$
   (b) $\mathcal{R} := reduced(\mathcal{R})$
3. $\mathcal{G} := \mathcal{R}$

This version of the algorithm is clearly not the most efficient, but it is sufficient for expository purposes.

Including the idempotence rules ($xx \to x$) is essential for the completeness of the method[4].

EXAMPLE 4.8. Take the ideal in Example 4.4, in which $A = \{(1,0,0),(1,1,0),(1,0,1)\}$ and $\{xyz + x\}$ is a generating set of $\mathcal{F}_{\mathcal{A}}$. Assume that $x > y > z$, then the starting rewrite system is

$$xx \to x \tag{4.1}$$

$$yy \to y \tag{4.2}$$

$$zz \to z \tag{4.3}$$

$$xyz \to x \tag{4.4}$$

Rules (4.3) and (4.4) produces a critical polynomial $xyz + xz$ which, after reduction, becomes $xz + x$. It is then made into a rule

$$xz \to x \tag{4.5}$$

The new rule (4.5) then reduces rule (4.4) into

$$xy \to x \tag{4.6}$$

Since there is no more non-trivial critical polynomials, we obtain $\mathcal{G}(I) = \{xy \to x, xz \to x\}$.

4.4. Normal form of a Boolean function with a don't-care condition. Now we are ready to give a procedure for producing the normal form of a Boolean function with a don't-care condition.

Let f be a Boolean function with the don't-care condition A. We may assume that f is represented by a truth table. We proceed as follows:

[4]We should note that we could not find any literature on Gröbner basis in which the idempotence rules are explicitly included in the generation process, although it may be due to our unfamiliarity with the literature.

1. Find a generating set of $\mathcal{F}_{\mathcal{A}}$, and call it I. (This can be easily done by applying Lemma 4.3.)
2. Construct the Gröbner basis of $\mathcal{F}_{\mathcal{A}}$, $\mathcal{G}_{\mathcal{A}}$, using the aforementioned Buchberger algorithm and I.
3. Choose a (total) Boolean function g from the equivalence class $[f]_A$. (This can be done by assigning 0 to all truth-assignments in A, and apply the flip-tag algorithm.)
4. Reduce g to its Boolean ring normal form with respect to $\mathcal{G}_{\mathcal{A}}$. The resulting normal form is a unique normal form for f.

EXAMPLE 4.9. Consider the function f defined by the following truth table:

x	y	z	f
0	0	0	0
0	0	1	1
0	1	0	1
1	0	0	X
0	1	1	1
1	0	1	X
1	1	0	X
1	1	1	0

In Example 4.8 we have already derived the Gröbner basis for the don't-care condition in this example: $\mathcal{G}_{\mathcal{A}} = \{xy \to x, xz \to x\}$. By replacing the X in the definition of f by 0, we obtain a function g whose BRNF, by flip-tag algorithm, is $y + z + xy + xz + yz + xyz$. A final stage of normalization using $\mathcal{G}_{\mathcal{A}}$ yields $x + y + z + yz$, which is the unique normal form of f.

5. Boolean ring-based propositional reasoning

5.1. Buchberger algorithm for SAT/UNSAT. The Buchberger algorithm described above can be easily adopted to become a theorem proving procedure for propositional logic.

Given a set of propositional formulas $S = \{\varphi_1, \cdots, \varphi_n\}$. Since logically it means that each of the formulas φ_i is true, we convert the $\varphi_i = 1$'s into Boolean ring rewrite rules and carry out the Buchberger algorithm. If S is unsatisfiable, then the ideal represented by S will be the ring itself. In other words, the equation $1 = 0$ will be produced by the Buchberger algorithm with $S \cup \{x_i x_i \to x_i\}_i$ if and only if S is unsatisfiable.

When converting formulas into rules heuristics can be applied to produce shorter rules. For instance an equation of the form

$$\varphi_1 \wedge \cdots \wedge \varphi_n = 1$$

can be transformed into n equations

$$\varphi_1 = 1, \cdots, \varphi_n = 1,$$

before converting into rules. Similarly, an equation

$$\varphi_1 \vee \cdots \vee \varphi_n = 0$$

can be transformed into equations

$$\varphi_1 = 0, \cdots, \varphi_n = 0.$$

We remark that the procedure we are described here does not require that the input formulas are in clausal form.

EXAMPLE 5.1. Given a set of clauses $S = \{p \vee q, t \vee s, \neg p \vee \neg t, \neg p \vee \neg s, \neg q \vee \neg t, \neg q \vee \neg s\}$. S can be transformed into

$$pq \rightarrow p + q + 1 \tag{5.1}$$
$$st \rightarrow s + t + 1 \tag{5.2}$$
$$pt \rightarrow 0 \tag{5.3}$$
$$ps \rightarrow 0 \tag{5.4}$$
$$qt \rightarrow 0 \tag{5.5}$$
$$qs \rightarrow 0. \tag{5.6}$$

Rules (5.1) and (5.3) produce a critical equation $pt + qt + t = 0$ which, after further simplification, becomes rule

$$t \rightarrow 0. \tag{5.7}$$

Rule (5.7) immediately deletes rules (5.3) and (5.5), and simplifies rule (5.2) into

$$s \rightarrow 1. \tag{5.8}$$

Rule (5.8) then simplifies rule (5.4) into

$$p \rightarrow 0 \tag{5.9}$$

and rule (5.6) into

$$q \rightarrow 0. \tag{5.10}$$

As the last step of the procedure, rules (5.9) and (5.10) simplifies rule (5.1) into

$$1 \rightarrow 0, \tag{5.11}$$

which means that S is unsatisfiable.

Applying Boolean ring formalism to theorem proving was first described in [**H82, H85**], in which a complete procedure for inputs in clausal form was presented. The main purpose there was aimed at studying complete theorem proving methods for first order logic, and the propositional procedure came as a side result.

The first theorem proving method which used both Boolean ring and Buchberger algorithm was given in [**KN**]. This method, although complete for propositional logic, was not complete for first order logic. It was later made into a complete method in [**BD**].

5.2. Davis-Putnam à la Boolean ring. We may regard the procedures mentioned in the previous section as resolution-like procedures in the framework of Boolean ring. While critical polynomial generation roughly corresponds to resolution, these procedures have the advantage of powerful simplification inferences which have no equivalent notion in resolution[5]. However, the procedure has its shortcoming. For instance, one needs to include the idempotence rules when generating critical polynomials.

[5]In resolution framework there are simplification inference rules such as subsumption and clausal simplification, as described in, e.g., [**L**] , but they are not as natural as simplification with rewrite rules.

In this section we introduce a Davis-Putnam like procedure, which does not need the generation of critical polynomials. This procedure has two inference rules. The first one is *reduce*, which reduces a set of Boolean rules in an inter-reduced set, $reduced(R)$. This inference rule is the same as the one in the Buchberger algorithm. The second inference rule is *split*, which chooses a Boolean variable, say x, and split R into two sets, $R_1 = R \cup \{x \to 1\}$ and $R_2 = R \cup \{x \to 0\}$.

The procedure works as follows: Given an input set of rules R, we first *reduce* it to $reduced(R)$. If the resulting set contains contradiction $(1 = 0)$, then R is inconsistent. Otherwise choose a Boolean variable and *split* $reduced(R)$ accordingly. The two resulting sets of rules can be treated recursively. The input set R is inconsistent if and only if all of the resulting sets lead to contradiction. If R is consistent, then each resulting set that does not contain contradiction will contain a truth assignment which satisfies R.

In the following we put the above informal description into a recursive procedure. We call a Boolean variable x *splittable in* R if neither $x \to 0$ nor $x \to 1$ is a rule in R.

procedure $DPBR(R, S)$
input: a set of Boolean rules R
output: a collection of truth assignments S
1. $R := reduced(R)$,
2. **if** $1 = 0 \in R$,
   then stop,
   else if there is a splittable variable x in R
   then $R_1 := R \cup \{x \to 1\}$,
   call $DPBR(R_1, S)$,
   $R_2 := R \cup \{x \to 0\}$,
   call $DPBR(R_2, S)$,
   else add R as a truth assignment to S.

If the input set of rules is R, then the procedure starts from $DPBR(R, \emptyset)$.

It is obvious that this procedure is complete. It is different from Davis-Putnam in several aspects. First, the input set needs not be in clausal form. Thus, it may have some advantage if the input formulas cannot be readily transformed into clausal form. Second, *reduce* is strictly more powerful than the *unit clause rule* of Davis-Putnam in terms of reduction power. Therefore we feel that the number of splits necessary in the average case should be smaller than Davis-Putnam, although we have not yet done any rigorous studies.

EXAMPLE 5.2. Now we re-do Example 5.1 using the "Davis-Putnam" approach.

Since no reduction can be done, we choose an arbitrary variable, say p, to split. Then the two new sets are:

$$
\begin{aligned}
pq &\to p+q+1 && (5.12)\\
st &\to s+t+1 && (5.13)\\
pt &\to 0 && (5.14)\\
ps &\to 0 && (5.15)\\
qt &\to 0 && (5.16)\\
qs &\to 0 && (5.17)\\
p &\to 1. && (5.18)
\end{aligned}
$$

and

$$
\begin{aligned}
pq &\to p+q+1 && (5.19)\\
st &\to s+t+1 && (5.20)\\
pt &\to 0 && (5.21)\\
ps &\to 0 && (5.22)\\
qt &\to 0 && (5.23)\\
qs &\to 0 && (5.24)\\
p &\to 0. && (5.25)
\end{aligned}
$$

It is easy to see that in both cases the set produces $1 = 0$ after one round of simplification.

As another example, we demonstrate that our method may use fewer applications of *splitting* than Davis-Putnam.

EXAMPLE 5.3. Let S be the set of clause

$$
\begin{aligned}
\neg p \vee \neg q \vee \neg r,\\
\neg p \vee \neg q \vee r,\\
p \vee \neg q \vee \neg r,\\
\neg p \vee q \vee \neg r,\\
\neg p \vee q \vee r,\\
p \vee \neg q \vee r,\\
p \vee q \vee \neg r\\
p \vee q \vee r.
\end{aligned}
$$

It is easy to see that Davis-Putnam needs two steps of splitting. In our method the clauses are transformed into

$$\begin{aligned} pqr &\rightarrow 0 \\ pqr &\rightarrow pq \\ pqr &\rightarrow qr \\ pqr &\rightarrow pr \\ pqr &\rightarrow pq + pr + p \\ pqr &\rightarrow pq + qr + q \\ pqr &\rightarrow pr + qr + r \\ pqr &\rightarrow pq + pr + qr + p + q + r + 1. \end{aligned}$$

Note that the contraction $1 = 0$ through simplification alone without using *any* splitting.

6. Discussion

Boolean ring is an alternative representation to Boolean algebra. Instead of $\vee$, it uses $+$, exclusive-or. Consequently, there is a unique normal form for every Boolean function, in which negation ($\neg$) is not necessary.

In this paper we presented several procedures for various operations based on Boolean rings. We described a simple method for deriving the Boolean ring normal form directly from a truth table. We also described a notion of normal form of a Boolean function with a don't-care condition, and showed an algorithm based on Gröbner basis for generating such a normal form. Finally we discussed two Boolean ring based theorem proving methods for propositional logic. Although the two methods, to some extent, resemble ground resolution and Davis-Putnam, they have more simplification power and seem to be quite effective.

Despite its extreme simplicity, the Boolean ring representation has not been used extensively both in logical reasoning and in computation. It is interesting to investigate why it is the case.

Operationally the main difference between $exclusive - or$ and or is that the former is nilpotent. Consequently negation does not appear in the normal form. This makes Boolean ring formulas hard to read for human, since one cannot tell which predicate symbol is negated and which is not. When a formula is long, it becomes impossible to make a natural interpretation of its meaning. This problem may partially explain why logicians have not used Boolean ring in actual reasoning.

The same problem may also explain why Boolean ring has not been more widely used in automated deduction. Boolean ring based first order theorem proving methods (e.g., [**H85**]) have been demonstrated to be quite favorable when compared with other methods [**HJ, PR, WS, BB, KZ**]. However, due to the normalization process, it is not likely that one can reconstruct the generated proof back to human-readable form once the proof is found. Thus, when one is interested in deriving a convincing proof rather than just demonstrating that the theorem is correct, then Boolean ring is not a good choice.

Boolean ring is not used more widely in circuit design for a similar reason. One of the more popular circuit design methodologies is programmable logic array (PLA). It is conceivable that the OR gates in PLA can be replaced by XOR. However, once an input to the OR gate is true, the output is decided. For XOR,

on the other hand, all inputs to XOR must be evaluated before an output can be decided. The reason is, once again, due to the nilpotence of XOR.

The question, then, is whether the Boolean ring representation is good for *any* practical applications. We feel that the answer is a resounding yes. Basically one can regard Boolean ring as an efficient internal data structure that provides a uniform representation and fast basic operations. Thus, for any problem for which one only cares about the input/output relationship but not how the computation is performed, Boolean ring is a feasible candidate. Satisfiabilty problems (see [GPFW] for a survey) such as constraint solving is an example. In addition to its effectiveness, our method also allows a more flexible input format. Proof-checking is another. In proof-checking one often only cares about having a flexible and efficient procedure to check the correctness of simple to moderately difficult theorems but not what the proofs look like. A third example is circuit verification, in particular when don't-care conditions are involved, since Boolean algebra cannot provide a satisfactory algebraic framework for effectively handle these problems.

References

[B] B. Buchberger, *Ein algorithmisches Kriterium für die Lösbarkeit eines algebraischen Gleichungssystems*, Aequ. Math. **4**/3 (1970), 374–383.

[BB] F. Baj, M.P. Bonacina, M. Bruschi and A. Zanzi, *Another term rewriting-based proof of the non-obvious theorem* Association of Automated Reasoning Newsletter, **13** (1989), 4–8.

[BD] L. Bachmair and N. Dershowitz, *Inference rules for rewrite-based first-order theorem proving*, Second Symposium on Logic in Computer Science, Ithaca, NY (1987), 331–337.

[BH] M.P. Bonacina and J. Hsiang, *Towards a Foundation of Completion Procedures as Semidecision Procedures*, Theoretical Computer Science, **146** (1995), 199–242.

[D] N. Dershowitz, *Orderings for term-rewriting systems*, Theoretical Computer Science, **17** (1982), 279–301.

[DJ] N. Dershowitz and J.-P. Jouannaud, *Rewrite systems*, Handbook of Theoretical Computer Science, volume B, Jan van Leeuwen (ed.) (1990), 243–320.

[DLL] M. Davis, G. Logeman and D. Loveland, *A machine program for theorem proving*, Comm. of ACM, **5** (1962), 394–397.

[DM] N. Dershowitz and Z. Manna, *Proving termination with multiset orderings* Comm. ACM, **22** (1979), 465–476.

[DP] M. Davis and H. Putnam, *A computing procedure for quantification theory*, J. ACM, **7** (1960), 201–215.

[GPFW] J. Gu, P.W. Purdom, J. Franco and B.W. Wah, *Algorithms for Satisfiability (SAT) Problem: A Survey*, DIMACS Volume Series on Discrete Mathematics and Theoretical Computer Science: *The Satisfiability (SAT) Problem*, American Mathematical Society (1996).

[H82] J. Hsiang, *Topics in automated theorem proving and program generation*, PhD. Thesis, Univ. of Illinois, Urbana, IL, 1982.

[H85] J. Hsiang, *Refutational theorem proving using term-rewriting systems* Artificial Intelligence, (1985), 255–300.

[HJ] J. Hsiang and N.A. Josephson, *TeRSe: A Term Rewriting Theorem prover*, The Rewrite Rule Laboratory Workshop, General Electric Research and Development Center, Schenectady, NY, 1983

[KN] D. Kapur and P. Narendran, *An equational approach to theorem proving in first-order predicate calculus*, Ninth International Joint Conference on Artificial Intelligence, L.A., (1985), 1146–1153.

[KZ] D. Kapur and H. Zhang, *Non-obviousness – last time?* Association of Automated Reasoning Newsletter, **14** (1989), 2–4.

[L] D.W. Loveland, *Automated theorem proving: a logical basis*, North Holland, New York, 1978.

[PR] F.J. Pelletier and Piotr Rudnicki, *Non-obviousness*, Association of Automated Reasoning Newsletter, **6** (1986), 4–5.

[S] M. Stone, *The theory of representation for Boolean algebra*, Trans. Amer. Math. Soc., **40** (1936), 37–111.

[WS] R. Wilkerson and B. Smith, *Non-obviousness – again* Association of Automated Reasoning Newsletter, **11** (1989), 3–5.

[Z] I.I. Zhegalkin, *On a technique of evaluation of propositions in symbolic logic*, Mat. Sb. **34** (1927), 9–27.

Department of Computer Science and Information Engineering, National Taiwan University, Taipei, TAIWAN

E-mail address: hsiang@csie.ntu.edu.tw