A Certified Polynomial-Based Decision Procedure for Propositional Logic

Inmaculada Medina-Bulo[1], Francisco Palomo-Lozano[1],
and José A. Alonso-Jiménez[2]

[1] Department of Computer Languages and Systems. University of Cádiz
Esc. Superior de Ingeniería de Cádiz. C/ Chile, s/n. 11003 Cádiz. Spain
{francisco.palomo,inmaculada.medina}@uca.es

[2] Department of Comp. Sciences and Artificial Intelligence, University of Sevilla
Fac. de Informática y Estadística. Avda. Reina Mercedes, s/n. 41012 Sevilla, Spain
jalonso@cica.es

Abstract. In this paper we present the formalization of a decision procedure for Propositional Logic based on polynomial normalization. This formalization is suitable for its automatic verification in an applicative logic like ACL2. This application of polynomials has been developed by reusing a previous work on polynomial rings [19], showing that a proper formalization leads to a high level of reusability. Two checkers are defined: the first for contradiction formulas and the second for tautology formulas. The main theorems state that both checkers are sound and complete. Moreover, functions for generating models and counterexamples of formulas are provided. This facility plays also an important role in the main proofs. Finally, it is shown that this allows for a highly automated proof development.

1 Introduction

In this paper we present the main results obtained through the development of an automated proof of the correctness of a polynomial-based decision procedure for Propositional Logic in ACL2 [14,15,16]. ACL2 [1] is the successor of NQTHM [3,5], the Boyer-Moore theorem prover. A concise description of ACL2 can be found in [14]. In order to understand ACL2, it is necessary to consider it under three different perspectives:

1. From a logic viewpoint, ACL2 is a untyped quantifier-free first-order logic of total recursive functions with equality. However, its *encapsulation* principle allows for some kind of higher-order reasoning.
2. From a programming language viewpoint, ACL2 is an applicative programming language in which the result of the application of a function is uniquely determined by its arguments. Every ACL2 function admitted under the definitional principle is a LISP function, so you can obtain both verified and executable software.

[1] A Computational Logic for Applicative Common Lisp.

3. From a reasoning system viewpoint, ACL2 is an automated reasoning system and it behaves as a heuristic theorem prover.

Representation issues play a major role in this work. We have represented Propositional Logic formulas in terms of just one Boolean function symbol: the three-place conditional construct present in most programming languages. This is discussed in Sect. 2.1. Definitions related with Boolean polynomials are presented in Sect. 2.2.

Surprisingly, polynomial-based theorem proving has a long history. According to H. Zhang [28], Boole himself [2] was the first to use Boolean polynomials to represent logical formulas and Herbrand described a polynomial-based decision procedure in his thesis. Later, in 1936, M. Stone [23] stated the strong relation existing between Boolean algebras and Boolean rings. Analogous results had been discovered, independently, in 1927 by I. I. Zhegalkin [29].

This relation is at the basis of the modern "algebraic methods" of logical deduction. The algebraic approach began with the development by J. Hsiang of a canonical term-rewriting system for Boolean algebras with applications to first-order theorem proving [11,12]. Concurrently, D. Kapur and P. Narendran used Gröbner bases and Buchberger's algorithm for the same purpose [17].[2] This last method has been extended to many-valued propositional logics [7,26] and it has been recently applied to knowledge based systems verification [18].

Several decision procedures for propositional logic that produce a verifiable proof log have been implemented. For example, [9,10] report the development of BDDs and Stłmarck's algorithm as HOL derived rules. On the other hand, actual formal verifications of decision procedures are less common. The classical work from [3] contains a verified decision procedure in NQTHM using IF-expressions. A similar procedure has been extracted from a COQ proof in [22]. Another decision procedure obtained via proof extraction in NUPRL is described in [6]. However, none of them is based on polynomial normalization.

We have not considered the possibility of integrating the decision procedure into the theorem prover via reflection, though this is feasible in ACL2 thanks to its metatheoretical extensibility capabilities [4]. A reflected decision procedure has been developed in [1] with NUPRL. See also [8] for a critical survey of reflection in theorem proving from a theoretical and practical viewpoint.

Section 2.3 presents a translation algorithm from formulas into polynomials. Once that suitable evaluation functions have been defined, this translation is shown to be interpretation-preserving. In Sect. 3, we review Hsiang's canonical term-rewriting system (TRS) for Boolean algebras. A normalization algorithm that is not based in term-rewriting is also presented. In Sect. 4, we prove the correctness of the decision procedure for Propositional Logic. As the involved algorithms are written in an applicative subset of COMMON LISP, they are intrinsically executable. Some examples of execution are shown in Sect. 5.

Finally, we will discuss the degree of automation achieved and we will also analyze some possible extensions of this work.

[2] See also [13,28,27].

2 IF-Formulas and Boolean Polynomials

In [20] an ACL2 formalization of IF-Formulas and Boolean polynomials is proposed. Next, the notion of Stone polynomial of an IF-formula is easily defined. We review here the main results obtained with some improvements.

As the conditional construct IF is functionally complete, we can regard our Propositional Logic formulas as IF-formulas without loss of generality. In fact, the NQTHM Boyer-Moore logic and its descendant ACL2 define the usual propositional connectives after axiomatizing IF. IF-formulas are also related with the OBDD algorithm as can be seen in [21]. A BDD manager has been recently formalized in ACL2 [24].

2.1 IF-Formulas

The underlying representation of IF-formulas is based on the notion of IF-cons. IF-conses are weaker than IF-formulas in the sense that they may not represent well-formed formulas. We use record structures to represent IF-conses. This provides us with a weak recognizer predicate that we strengthen to develop a recognizer for well-formed formulas.

Boolean constants, `nil` and `t`, are recognized by the ACL2 `booleanp` predicate. The set of propositional variables could be then represented by the set of atoms not including the Boolean constants. However, if we represent variables using natural numbers then it is easier to share the same notion of variable in formulas and polynomials. Thus, we define our variable recognizer, `variablep`, to recognize just natural numbers.

Our notion of IF-cons is captured by an ACL2 structure. An IF-cons is just a collection of three objects (the *test*, and the *then* and *else* branches). The predicate `if-consp` will recognize terms constructed with `if-cons`, while the functions `test`, `then` and `else` act as destructors. Well-formed IF-formulas can be recognized by the following total recursive ACL2 predicate:

```
(defun formulap (f)
  (or (booleanp f) (variablep f)
      (and (if-consp f)
           (formulap (test f))
           (formulap (then f))
           (formulap (else f)))))
```

An assignment of values to variables can be represented as a list of Booleans.[3] Thus, the value of a variable with respect to an assignment is given by the element which occupies its corresponding position.

```
(defun truth-value (v a)
  (nth v a))
```

[3] Remember each Boolean variable is represented as a natural number.

The value of a formula under an assignment is defined recursively by the following function. To make the valuation function total, we assign an arbitrary meaning to non-formulas.

```
(defun value (f a)
  (cond ((booleanp f) f)
        ((variablep f) (truth-value f a))
        ((if-consp f)
         (if (value (test f) a)
             (value (then f) a)
           (value (else f) a)))
        (t nil)))                      ; for completeness
```

The following theorem states a simple but important property. It says that the value of a formula under an assignment is true if and only if the value of the negation of that formula under the same assignment is false. Why this property is important will become clear in Sect. 4.

```
(defthm duality
  (implies (and (formulap f) (assignmentp a))
           (iff (equal (value f a) t)
                (equal (value (if-cons f nil t) a) nil))))
```

2.2 Boolean Polynomials

In order to represent polynomials with Boolean coefficients, we can use the Boolean ring given by $\langle\{0,1\},\oplus,\wedge,0,1\rangle$ where $\oplus$ is the logical exclusive disjunction (exclusive-or), $\wedge$ is the logical conjunction and 0 and 1 are regarded as truth-values (false and true). In the following definitions, let $B=\{0,1\}$ and $\neg$, $\vee$ stand for logical negation and logical disjunction, respectively.

Although it suffices with a polynomial Boolean ring for our current purposes, where monomials do not need coefficients, we have implemented monomials with coefficients and terms to reuse part of a previous work on polynomial rings [19].

Definition 1. *A Boolean term on a finite set $V=\{v_1,\ldots,v_n\}$ of Boolean variables with an ordering relation $<_V=\{(v_i,v_j):1\leq i<j\leq n\}$ is a finite product of the form:*

$$\bigwedge_{i=1}^{n}(v_i\vee\neg a_i)\qquad\forall i\ a_i\in B\ . \tag{1}$$

We obtain a quite simple representation of a Boolean term on a given set of variables by using the Boolean sequence $\langle a_1,\ldots,a_n\rangle$, namely, v_i appears in the term if and only if $a_i=1$. The main results that we have proved in ACL2 on our Boolean term formalization may be summed up in the following points:

1. Boolean terms form a commutative monoid with respect to a suitable multiplication operation.
2. Lexicographical ordering on terms is well-founded.

As we usually work with Boolean terms defined on the same set of variables, their sequences will have the same length. In this case they are said to be *compatible*.

Definition 2. *We define the multiplication of two compatible terms as the following operation:*

$$\bigwedge_{i=1}^{n}(v_i \vee \neg a_i) \cdot \bigwedge_{i=1}^{n}(v_i \vee \neg b_i) = \bigwedge_{i=1}^{n}(v_i \vee \neg(a_i \vee b_i)) \ . \tag{2}$$

Having chosen the set of variables, it suffices to "or" their sequences element by element to compute the multiplication of two compatible terms. A proof of terms having a commutative monoid structure with respect to the previous operation is easily obtained.

To order terms it is only necessary to take into account their associated sequences. The obvious choice is to set up a *lexicographical ordering* among them. In the case of compatible terms, this definition is straightforward, since the sequences involved have the same length.

Definition 3. *The lexicographical ordering on compatible Boolean terms is defined as the following relation:*

$$\langle a_1, \ldots, a_n \rangle < \langle b_1, \ldots, b_n \rangle \equiv \exists i \ (\neg a_i \wedge b_i \wedge \forall j < i \ a_j = b_j) \ . \tag{3}$$

Definition 4. *A Boolean monomial on V is the product of a Boolean coefficient and a Boolean term.*

$$c \wedge \bigwedge_{i=1}^{n}(v_i \vee \neg a_i) \qquad c \in B \quad \forall i \ a_i \in B \ . \tag{4}$$

In the same way as happened to terms, it is suitable to define a compatibility relation on monomials. We say that two monomials are *compatible* when their underlying terms are compatible.

A multiplication operation is defined and then it is proved that monomials have a monoid commutative structure with respect to it.

Due to technical reasons it is convenient to extend compatibility of monomials to polynomials. To achieve this we first say that a polynomial is *uniform* if all of its monomials are compatible each other. Henceforth, we will assume uniformity.

Definition 5. *A Boolean polynomial on V is a finite sum of monomials.*

$$\bigoplus_{i=1}^{m}\left[c_i \wedge \bigwedge_{j=1}^{n}(v_j \vee \neg a_{ij})\right] \qquad \forall i,j \ c_i, a_{ij} \in B \ . \tag{5}$$

Now, the definition of compatibility between polynomials arises in a natural way. Two polynomials are *compatible* if their monomials are compatible too.

Finally, we have proved that Boolean polynomials have a ring structure. To achieve this, only coefficients and terms had to be changed from the formalization described in [19]. These changes are reported in [20].

2.3 Interpretation Preserving Translation

Next, we use the relation between Boolean rings and Boolean algebras to derive the translation algorithm. Let us consider a Boolean algebra and the following three place Boolean function *if*, defined on it:

$$\forall a, b, c \in B \;\; \mathit{if}(a, b, c) = (a \wedge b) \vee (\neg a \wedge c) \ . \tag{6}$$

We can build an associated *if* function in the corresponding Boolean ring:

$$\mathit{if}(a, b, c) = a \cdot b \cdot (a + 1) \cdot c + a \cdot b + (a + 1) \cdot c = a \cdot b + a \cdot c + c \ .$$

The following ACL2 functions use this to compute the polynomial associated to a formula (Stone polynomial). The function `variable->polynomial` transforms a propositional variable into a suitable polynomial. The underlying polynomial Boolean ring is represented by $\langle$`polynomialp`, `+`, `*`, `null`, `identity`$\rangle$. The argument of the function `identity` is a technical detail that guarantees the uniformity of the resulting polynomial.

```
(defun stone (f)
  (stone-aux f (max-variable f)))

(defun stone-aux (f n)
  (cond ((booleanp f) (if f (identity (LISP::+ n 1)) (null)))
        ((variablep f) (variable->polynomial f n))
        ((if-consp f)
         (let ((s-test (stone-aux (test f) n))
               (s-then (stone-aux (then f) n))
               (s-else (stone-aux (else f) n)))
           (+ (* s-test (+ s-then s-else)) s-else)))
        (t (null))))                    ; for completeness
```

Then, a function, `ev`, to evaluate a polynomial with respect to an assignment is defined. Finally, it is proved that the translation of formulas into polynomials preserves the interpretation:

```
(defthm interpretation-preserving-translation
  (implies (and (formulap f) (assignmentp a))
           (iff (value f a) (ev (stone f) a))))
```

The hard part of the work is dealing with the theorems about the evaluation function and polynomial operations.

3 Normalization

In this section, we review the Hsiang's Canonical TRS and develop a straightforward normalization procedure for Boolean polynomials. Unlike disjunctive and conjunctive normal forms, polynomial normalization allows us to associate a unique polynomial to each Propositional Logic Formula.

3.1 Hsiang's Canonical TRS for Boolean Algebras

A Boolean ring with identity $\langle B, +, \cdot, 0, 1\rangle$ is a ring that is idempotent with respect to $\cdot$. It is a known fact that every Boolean ring is nilpotent with respect to $+$ and commutative. Hsiang [11,12] derives his canonical term-rewriting system for Boolean algebras by first generating a canonical system for Boolean rings. Firstly, he considers the Boolean ring axioms:[4]

A1. $a + (b + c) = (a + b) + c$ (associativity of $+$).
A2. $a + b = b + a$ (commutativity of $+$).
A3. $a + 0 = a$ (right identity of $+$).
A4. $a + (-a) = 0$ (right inverse of $+$).
A5. $(a \cdot b) \cdot c = a \cdot (b \cdot c)$ (associativity of $\cdot$).
A6. $a \cdot (b + c) = a \cdot b + a \cdot c$ (distributivity of $\cdot$ over $+$).
A7. $a \cdot 1 = a$ (right identity of $\cdot$).
A8. $a \cdot a = a$ (idempotency of $\cdot$).
T1. $a + a = 0$ (nilpotency of $+$).
T2. $a \cdot b = b \cdot a$ (commutativity of $\cdot$).

By executing the AC-completion procedure on these rules, he obtains the BR canonical TRS for Boolean rings. Then, BR can be completed[5] by adding rules for transforming the usual Boolean algebraic operations into Boolean ring operations, obtaining the BA canonical TRS for Boolean algebras.

BR:

$$\begin{aligned} a + 0 &\longrightarrow a, \\ a \cdot (b + c) &\longrightarrow a \cdot b + a \cdot c, \\ a \cdot 0 &\longrightarrow 0, \\ a \cdot 1 &\longrightarrow a, \\ a \cdot a &\longrightarrow a, \\ a + a &\longrightarrow 0, \\ -a &\longrightarrow a \ . \end{aligned}$$

BA:

$$\begin{aligned} a \vee b &\longrightarrow a \cdot b + a + b, \\ a \wedge b &\longrightarrow a \cdot b, \\ \neg a &\longrightarrow a + 1, \\ a \implies b &\longrightarrow a \cdot b + a + 1, \\ a \iff b &\longrightarrow a \cdot b \cdot 1, \\ a + 0 &\longrightarrow a, \\ a \cdot (b + c) &\longrightarrow a \cdot b + a \cdot c, \\ a \cdot 0 &\longrightarrow 0, \\ a \cdot 1 &\longrightarrow a, \\ a \cdot a &\longrightarrow a, \\ a + a &\longrightarrow 0 \ . \end{aligned}$$

[4] Note that, T1 and T2 are not axioms, but theorems that are added so that the AC-unification algorithm can be used.

[5] The $-a \longrightarrow a$ rule is discarded since the inverse of $+$ has no significant meaning in Boolean algebras.

Therefore, the irreducible form of any Boolean algebra term is the normal expression defined by the BA TRS above, and it is unique (since BA is a canonical TRS). This implies that a formula from Propositional Logic is a tautology if and only if its irreducible expression is 1, and it is a contradiction if and only if its irreducible expression is 0.

3.2 A Straightforward Normalization Algorithm

An algorithm can be developed to avoid the overhead associated to Hsiang's TRS. Instead of rewriting modulo BA, formulas are translated to polynomials and then polynomial normalization is used. Once we have defined an order on terms, we can say that a polynomial is in normal form if and only if their monomials are strictly ordered by the decreasing term order and none of them is null. This definition implies the absence of identical monomials in a normalized uniform polynomial. We divide the specification of the normalization function in two steps:

1. A function capable of adding a monomial to a polynomial. This must be a normalization-preserving function.
2. A normalization function stable for normalized null polynomials that adds the first monomial to the normalization of the remaining monomials by using the previous function.

The normalization function is easy to define: if the polynomial is null, it is already in normal form, otherwise, it suffices to normalize the rest of the polynomial and then add the first monomial to the result.

```
(defun nf (p)
  (cond ((or (not (polynomialp p)) (nullp p)) (null))
        (t (+-monomial (first p) (nf (rest p))))))
```

In order to make `+-monomial` total we need to complete, taking the utmost care, the values that it returns when it is not applied to a polynomial.

Next, we show the most important part of the definition of `+-monomial` function. It takes a monomial `m` and a polynomial `p` as its arguments.

1. If `m` is null, `p` is returned.
2. If `p` is null, the polynomial composed of `m` is returned.
3. If `m` and the first monomial of `p` have the same term, both monomials are added. If the result is null then the rest of `p` is returned, otherwise a polynomial consisting of the resulting monomial and the rest of `p` is returned.
4. If `m` is greater than the first monomial of `p`, a polynomial consisting of `m` and `p` is returned.
5. Otherwise, a polynomial consisting of the first monomial of `p` and the result of recursively adding `m` to the rest of `p` is returned.

Important properties of the normalization function have been proved, such as that it meets its specification,

```
(defun nfp (p)
  (equal (nf p) p))

(defthm nfp-nf
  (nfp (nf p)))
```

and that polynomial uniformity is preserved under normalization.

```
(defun uniformp (p)
  (or (nullp p) (nullp (rest p))
      (and (MON::compatiblep (first p) (first (rest p)))
           (uniformp (rest p)))))

(defthm uniformp-nf
  (implies (uniformp p)
           (uniformp (nf p))))
```

One relevant result states that the normal form of a polynomial is strictly decreasingly ordered with respect to the lexicographical order defined on terms.

```
(defthm orderedp-nf
  (orderedp (nf p)))
```

In order to obtain this, we define the function `orderedp` by using the lexicographical order defined on terms.

```
(defun term-greater-than-leader (m p)
  (or (nullp p) (TER::< (MON::term (first p)) (MON::term m))))

(defun orderedp (p)
  (and (polynomialp p)
       (or (nullp p)
           (and (not (MON::nullp (first p)))
                (term-greater-than-leader (first p) (rest p))
                (orderedp (rest p))))))
```

4 A Decision Procedure

In this section, our main aim is to construct a polynomial-based procedure for deciding whether a propositional logic formula is a tautology and prove its correctness.

A formula is a tautology if and only if the value of the formula under every possible assignment of values to variables is true. So, the following first-order formula states the correctness of a tautology-checker:

$$\forall f \; [\texttt{(tautology-checker } f\texttt{)} \iff \forall a \; \texttt{(value } f \; a\texttt{)} = \texttt{t}] \tag{7}$$

However, it is not possible to write directly this theorem in ACL2, due to the lack of quantifiers. For example, the following “theorem” does not capture our idea:

```
(defthm flawed-tautology-checker-correctness
  (iff (tautology-checker f) (equal (value f a) t)))
```

The problem is that its first-order interpretation is the following:

$$\forall f, a\ [(\texttt{tautology-checker}\ f) \iff (\texttt{value}\ f\ \ a) = \texttt{t}] \qquad (8)$$

which is rather different from (7).

A possible solution to this problem in ACL2 is to use `defun-sk` to introduce an intermediate function whose body has an outermost quantifier. Internally, `defun-sk` uses `defchoose` which is implemented by using the encapsulation principle.

An equivalent approach is to divide the proof in two parts: soundness and completeness. This is the approach used in [3].

$$\forall f, a\ [(\texttt{tautology-checker}\ f) \implies (\texttt{value}\ f\ \ a) = \texttt{t}] \quad \text{(sound)}$$
$$\forall f\ [\neg(\texttt{tautology-checker}\ f) \implies \exists a\ (\texttt{value}\ f\ \ a) = \texttt{nil}] \quad \text{(complete)}$$

The existential quantifier in the second formula can be relieved by substituting a proper function for a. This enforces the constructive character of the ACL2 logic: an explicit function providing a counterexample for a non-tautological formula is constructed in order to prove that the tautology-checker is complete.

On the other hand, it is a bit easier to formalize a contradiction-checker than a tautology-checker when using the polynomial-based approach. Thus, we begin by defining a contradiction-checker. Soundness and completeness for this kind of checker are defined analogously to the tautological case.

4.1 Contradiction-Checker

The contradiction-checker proceeds by transforming a formula into a polynomial, then computing its normal form and, finally, checking if the resulting normal form is the null polynomial.

So, we are trying to prove that a formula is a contradiction if the polynomial in normal form associated to the formula is the null polynomial. First, we introduce the function `contradiction-checker`.

```
(defun contradiction-checker (f)
  (equal (nf (stone f)) (null)))
```

Second, we prove that the contradiction-checker is sound.

```
(defthm contradiction-checker-is-sound
  (implies (and (formulap f) (assignmentp a) (contradiction-checker f))
           (equal (value f a) nil)))
```

The proof outline is as follows:

1. The translation from formulas to polynomials is interpretation-preserving allowing us to transform `(value f a)` into `(ev (stone f) a)`.

2. The evaluation of a polynomial is stable under normalization, so we can replace `(ev (stone f) a)` by `(ev (nf (stone f)) a)`.
3. The term `(nf (stone f))` is known to be the null polynomial by the hypothesis `(contradiction-checker f)`. But the evaluation of the null polynomial is `nil` under every possible assignment.

Therefore, we only need to show that the evaluation of a polynomial is equal to the evaluation of its normal form.[6]

```
(defthm ev-nf
  (implies (and (polynomialp p) (assignmentp a))
           (equal (ev p a) (ev (nf p) a))))
```

In order to prove completeness, we have to compute an explicit model for the formula in case of the formula not being a contradiction. We construct the model from the associated polynomial because it is simpler to find than from the formula itself.

In fact, it suffices to take an assignment such that the least term of the normalized polynomial evaluates to true. It is clear that each of the greater terms must be false, because the least term lacks (at least) a variable appearing in the remaining terms.

As we use the same representation for terms and assignments, this observation is supported by the following theorem. Therefore, the value of the formula is true with respect to the assignment given by the least term of its associated normalized polynomial.

```
(defthm ev-term-<
  (implies (and (TER::termp t1) (TER::termp t2)
                (TER::compatiblep t1 t2) (TER::< t2 t1))
           (equal (ev-term t1 t2) nil)))
```

Next, we define the function that computes such a term. Recall that normalized polynomials remain ordered with respect to the lexicographical order defined on terms. The null polynomial is a special case: it corresponds to a contradiction, which has no models.

```
(defun least-term (p)
  (cond ((nullp p) (TER::null 0))                ; for completeness
        ((nullp (rest p)) (MON::term (first p)))
        (t (least-term (rest p)))))

(defun model (f)
  (least-term (nf (stone f))))
```

Then, it is proved that the contradiction-checker is complete.

[6] In fact, the definition of this theorem is completed with syntactic restrictions to prevent the infinite application of its associated rewrite rule.

```
(defthm contradiction-checker-is-complete
  (implies (and (formulap f) (not (contradiction-checker f)))
           (equal (value f (model f)) t)))
```

The proof outline is as follows:

1. Let `m` be `(model f)`.
2. The translation from formulas to polynomials is interpretation-preserving allowing us to transform `(value f m)` into `(ev (stone f) m)`.
3. The evaluation of a polynomial is stable under normalization, so we can replace `(ev (stone f) m)` by `(ev (nf (stone f)) m)`.
4. But the evaluation of `(nf (stone f))` is `t` under `m`, by induction on the structure of `(nf (stone f))`, which is known to be an ordered polynomial.

Some lemmas are needed for the last step. The main lemma asserts that whenever `least-term` is applied to a non-null ordered uniform polynomial, it computes an assignment that makes its evaluation `t`.

```
(defthm ev-least-term
  (implies (and (polynomialp p) (uniformp p) (orderedp p)
                (not (equal p (null))))
           (equal (ev p (least-term p)) t)))
```

4.2 Tautology-Checker

Once we have certified the contradiction-checker, the definition and certification of a tautology-checker is considerably easier. We proceed by constructing the IF-formula corresponding to the negation of the input formula, then it is only necessary to check whether the resulting IF-formula is a contradiction.

```
(defun tautology-checker (f)
  (equal (nf (stone (if-cons f nil t))) (null)))
```

Let us consider the `duality` property stated in Sect. 2.1. This important result reduces the problem of determining whether a formula is a tautology to the dual problem of determining whether the negation of this formula is a contradiction. Using this result, we can easily show that the tautology-checker is sound.

```
(defthm tautology-checker-is-sound
  (implies (and (formulap f) (assignmentp a) (tautology-checker f))
           (equal (value f a) t)))
```

But counterexamples of a formula are just models of its negation. Therefore, we can state the completeness of the tautology-checker in the following way:

```
(defun counterexample (f)
  (model (if-cons f nil t)))

(defthm tautology-checker-is-complete
  (implies (and (formulap f) (not (tautology-checker f)))
           (equal (value f (counterexample f)) nil)))
```

Consequently, the proof of this theorem is simply reduced to the completeness of the contradiction-checker, which we have proven before.

5 Execution Examples

For the sake of simplicity, we are assuming in this section the following macro definitions. In order to prevent name conflicts, we do this in a new package, EX.

```
(defmacro not (a)   `(if-cons ,a nil t))
(defmacro and (a b) `(if-cons ,a (if-cons ,b t nil) nil))
(defmacro or (a b)  `(if-cons ,a t (if-cons ,b t nil)))
(defmacro imp (a b) `(if-cons ,a (if-cons ,b t nil) t))
(defmacro iff (a b) `(if-cons ,a (if-cons ,b t nil) (if-cons ,b nil t)))
```

This is just a bit of syntactic sugar to avoid cumbersome IF-notation when writing formulas from Classical Propositional Logic. In fact, these macros are proved to do the correct thing, though we omit the details here. Basically, the proof consist of stating that the interpretation of the formula built by each macro agree with its corresponding truth-table.

Next, we are going to enumerate some formulas as we discuss their characters by means of the execution of the corresponding functions in an ACL2 session. Let us recall that "false" and "true" are represented by 0 and 1, but they are implemented with `nil` and `t`. Variables are represented by natural numbers so that, for example, we can think of p_0, p_1 and p_2 as `0`, `1` and `2`, respectively.

Although we have not discussed it, we have specified and verified suitable guards for every presented function. Thanks to guard verification we can be sure that execution will not abort[7] if functions are applied to data in their intended (guarded) domain.

◇ The formula $\neg((p_0 \implies p_1) \iff (\neg p_1 \implies \neg p_0))$ is a contradiction.

```
EX !> (contradiction-checker (not (iff (imp 0 1) (imp (not 1) (not 0)))))
T
```

◇ The formula $\neg(p_0 \implies p_1) \vee (p_1 \implies p_0)$ is not a tautology ($p_0 = 0$, $p_1 = 1$ is a counterexample), nor a contradiction ($p_0 = p_1 = 0$ is a model). Its corresponding Boolean polynomial in normal form is $(p_0 \wedge p_1) \oplus p_1 \oplus 1$.

```
EX !> (tautology-checker (or (not (imp 0 1)) (imp 1 0)))
NIL
EX !> (counterexample (or (not (imp 0 1)) (imp 1 0)))
(NIL T)
EX !> (contradiction-checker (or (not (imp 0 1)) (imp 1 0)))
NIL
EX !> (model (or (not (imp 0 1)) (imp 1 0)))
(NIL NIL)
EX !> (nf (stone (or (not (imp 0 1)) (imp 1 0))))
((T (T T)) (T (NIL T)) (T (NIL NIL)))
```

[7] As long as there are enough resources to do the computation.

◇ The formula $((p_0 \vee p_1) \implies (p_0 \vee p_2)) \iff (p_0 \vee (p_1 \implies p_2))$ is a tautology.

```
EX !> (tautology-checker (iff (imp (or 0 1) (or 0 2)) (or 0 (imp 1 2))))
T
```

◇ The formula $(p_0 \vee (p_1 \wedge p_2)) \wedge ((p_0 \vee p_1) \wedge (p_0 \vee p_2))$ is not a tautology ($p_0 = p_1 = p_2 = 0$ is a counterexample), nor a contradiction ($p_0 = 0$, $p_1 = p_2 = 1$ is a model). Its corresponding Boolean polynomial in normal form is $(p_0 \wedge p_1 \wedge p_2) \oplus p_0 \oplus (p_1 \wedge p_2)$.

```
EX !> (tautology-checker (and (or 0 (and 1 2)) (and (or 0 1) (or 0 2))))
NIL
EX !> (counterexample (and (or 0 (and 1 2)) (and (or 0 1) (or 0 2))))
(NIL NIL NIL)
EX !> (contradiction-checker (and (or 0 (and 1 2)) (and (or 0 1) (or 0 2))))
NIL
EX !> (model (and (or 0 (and 1 2)) (and (or 0 1) (or 0 2))))
(NIL T T)
EX !> (nf (stone (and (or 0 (and 1 2)) (and (or 0 1) (or 0 2)))))
((T (T T T)) (T (T NIL NIL)) (T (NIL T T)))
```

◇ The formula $((p_0 \iff p_1) \iff p_2) \iff (p_0 \iff (p_1 \iff p_2))$ is a tautology.

```
EX !> (tautology-checker (iff (iff (iff 0 1) 2) (iff 0 (iff 1 2))))
T
```

6 Conclusions and Further Work

A decision procedure for Propositional Logic in ACL2 has been presented. This includes a contradiction-checker and a tautology-checker together with their proofs of soundness and completeness. Functions for finding counterexamples and models for formulas are also provided. They are useful not only to compute but also to prove the main theorems stating the correctness of the checkers. These results are by no means trivial and we think that this work is testimonial to the high level of automation that can be reached in ACL2 when a proper formalization is used.

All the functions and theorems presented here have been collected in ACL2 books to increase their reusability. Moreover, we have specified and verified suitable guards for every presented function. So, we can be sure that execution will not abort if functions are applied to data in their intended (guarded) domain.

This decision procedure is based in Boolean polynomial normalization, but instead of applying Hsiang's canonical term-rewriting system for Boolean algebras, a straightforward normalization algorithm is used. This application of polynomials has been developed by reusing a previous work on polynomial rings [19]. The formalization presented there is modified to accommodate polynomials with rational coefficients to Boolean polynomials. These modifications are presented in [20].

Previously, formulas are translated into polynomials by using the relation between Boolean algebras and Boolean rings. This translation is interpretation-preserving with respect to a suitable valuation function for formulas and a suitable evaluation function for polynomials. This and other properties were formalized in [20].

The whole formalization consists of (roughly) 40 pages of ACL2 source code. 23 pages are devoted to the formalization of Boolean polynomials and 17 to the formalization of the decision procedures. The automation degree that we have obtained is high, though some technical lemmas and hints were required.

As polynomial formalization has been the most time-consuming task, some of our future work will be devoted to the study of better formalization techniques for generic polynomials.

One important method of logical deduction is the "algebraic method" which also uses the idea of translating formulas into polynomials. This technique transforms the logical problem into an algebraic problem, polynomial ideal membership, what reduces the problem to the computation of Gröbner bases.

The first step to achieve this would consist of certifying Buchberger's algorithm for Gröbner bases computation in ACL2. A work from L. Théry [25], achieves this goal in COQ. Nevertheless, ACL2 and COQ logics differ in many aspects. Automated certification of Buchberger's algorithm in ACL2 remains a challenging project.

The reduction relation defined for Buchberger's algorithm is a subset of the ordering on polynomials. Once we have stated an ordering between terms, it can be extended to polynomials. Therefore, defining a term order is required prior to the definition of the concepts associated with Buchberger's algorithm and, particularly, to its termination. We have already formalized a suitable lexicographical order on terms and proved its well-foundedness in ACL2.

References

1. Aitken, W. E., Constable, R. L., Underwood, J. L.: Metalogical Frameworks II: Developing a Reflected Decision Procedure. J. Automated Reasoning **22**(2) (1999)
2. Boole, G. The Mathematical Analysis of Logic. Macmillan (1847)
3. Boyer, R. S., Moore, J S.: A Computational Logic. Academic Press (1978)
4. Boyer, R. S., Moore, J S.: Metafunctions: Proving Them Correct and Using Them Efficiently as New Proof Procedures. In: Boyer, R. S., Moore, J S. (eds.): The Correctness Problem in Computer Science. Academic Press (1981)
5. Boyer, R. S., Moore, J S.: A Computational Logic Handbook. Academic Press. 2nd edn. (1998)
6. Caldwell, J. L.: Classical Propositional Decidability via Nuprl Proof Extraction. 11th International Conference on Theorem Proving in Higher Order Logics. LNCS **1479** (1998)
7. Chazarain, J., Riscos, A., Alonso, J. A., Briales, E.: Multi-Valued Logic and Gröbner Bases with Applications to Modal Logic. J. Symbolic Computation **11** (1991)
8. Harrison, J.: Metatheory and Reflection in Theorem Proving: A Survey and Critique. SRI International Cambridge Computer Science Research Centre. Technical Report CRC–053 (1995)

9. Harrison, J.: Binary Decision Diagrams as a HOL Derived Rule. The Computer Journal **38** (1995)
10. Harrison, J.: Stłmarck's Algorithm as a HOL Derived Rule. 9th International Conference on Theorem Proving in Higher Order Logics. LNCS **1125** (1996)
11. Hsiang, J.: Refutational Theorem Proving using Term-Rewriting Systems. Artificial Intelligence **25** (1985)
12. Hsiang, J.: Rewrite Method for Theorem Proving in First-Order Theory with Equality. J. Symbolic Computation **3** (1987)
13. Hsiang, J., Huang, G. S.: Some Fundamental Properties of Boolean Ring Normal Forms. DIMACS series on Discrete Mathematics and Computer Science: The Satisfiability Problem. AMS (1996)
14. Kaufmann, M., Moore, J S.: An Industrial Strength Theorem Prover for a Logic Based on Common Lisp. IEEE Trans. on Software Engineering **23**(4) (1997)
15. Kaufmann, M., Manolios, P., Moore, J S.: Computer-Aided Reasoning: An Approach. Kluwer Academic Publishers (2000)
16. Kaufmann, M., Manolios, P., Moore, J S.: Computer-Aided Reasoning: ACL2 Case Studies. Kluwer Academic Publishers (2000)
17. Kapur, D., Narendran, P.: An Equational Approach to Theorem Proving in First-Order Predicate Calculus. 9th International Conference on Artificial Intelligence (1985)
18. Laita, L. M., Roanes-Lozano, E., Ledesma, L., Alonso, J. A.: A Computer Algebra Approach to Verification and Deduction in Many-Valued Knowledge Systems. Soft Computing **3**(1) (1999)
19. Medina-Bulo, I., Alonso-Jiménez, J. A., Palomo-Lozano, F.: Automatic Verification of Polynomial Rings Fundamental Properties in ACL2. ACL2 Workshop 2000 Proceedings, Part A. The University of Texas at Austin, Department of Computer Sciences. Technical Report TR–00–29 (2000)
20. Medina-Bulo, I., Palomo-Lozano, F., Alonso-Jiménez, J. A.: A Certified Algorithm for Translating Formulas into Polynomials. An ACL2 Approach. International Joint Conference on Automated Reasoning (2001)
21. Moore, J S.: Introduction to the OBDD Algorithm for the ATP Community. Computational Logic, Inc. Technical Report 84 (1992)
22. Paulin-Mohring, C., Werner, B.: Synthesis of ML Programs in the System Coq. J. Symbolic Computation **15**(5–6) (1993)
23. Stone, M.: The Theory of Representation for Boolean Algebra. Trans. AMS **40** (1936)
24. Sumners, R.: Correctness Proof of a BDD Manager in the Context of Satisfiability Checking. ACL2 Workshop 2000 Proceedings, Part A. The University of Texas at Austin, Department of Computer Sciences. Technical Report TR–00–29 (2000)
25. Théry, L. A Machine-Checked Implementation of Buchberger's Algorithm. J. Automated Reasoning **26** (2001)
26. Wu, J., Tan, H.: An Algebraic Method to Decide the Deduction Problem in Propositional Many-Valued Logics. International Symposium on Multiple-Valued Logics. IEEE Computer Society Press (1994)
27. Wu, J.: First-Order Polynomial Based Theorem Proving. In: Gao, X., Wang, D. (eds.): Mathematics Mechanization and Applications. Academic Press (1999)
28. Zhang, H.: A New Strategy for the Boolean Ring Based Approach to First Order Theorem Proving. Department of Computer Science. University of Iowa. Technical Report (1991)
29. Zhegalkin, I. I.: On a Technique of Evaluation of Propositions in Symbolic Logic. Mat. Sb. **34** (1927)