A Certified Algorithm for Translating Formulas into Polynomials

An ACL2 Approach

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 certification of an algorithm in ACL2 for translating formulas from Classical Propositional Logic (CPL) into polynomials over a Boolean ring (Boolean polynomials). The main theorem states that this translation is interpretation-preserving, i.e., that a CPL formula is equivalent to its associated polynomial w.r.t. any given assignment of Boolean values to variables. CPL formulas are represented in terms of just one Boolean function: the usual conditional construct present in programming languages. The Boolean polynomial representation used is also discussed. It is shown that the formalization chosen allows for a highly automated proof development.

1 Introduction

As early as 1936, M. Stone noticed the strong relation existing between Boolean algebras and Boolean rings. This was formally stated in Stone's seminal paper [16], but it probably dates back to 1927 by I. I. Zhegalkin [20]. This relation can be extended naturally to transform CPL formulas into Boolean polynomials.

The importance of both works was not fully understood until many years later with the development of Computer Science. Today, Stone's work is the basis for the "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 [7, 8]. Concurrently, D. Kapur and P. Narendran used Gröbner bases and Buchberger's algorithm for the same purpose [13]. See also [9, 19]. This last method has been extended to many-valued propositional logics [5, 18]. At the same time, Zhegalkin's work has also become important in logic circuit design.

In this paper we present the main results obtained through the development of an automated proof of the fact that every CPL formula has an interpretation-preserving translation into Boolean polynomials.

This proof has been carried up in ACL2 [10–12]. From a logic viewpoint, ACL2 is a untyped quantifier-free first-order logic of total recursive functions with equality. We will give a brief description of ACL2 in Sect. 2.

Representation issues are very important for the success of any certification work and this case is not an exception. We have appreciated the elegance and simplicity

of representing CPL formulas in terms of just one Boolean function symbol: the three-place conditional present in most programming languages. This is discussed in Sect. 3. Moreover, as it will be shown in Sect. 4, some considerations about Boolean polynomials should be taken into account.

Section 5 presents the translation algorithm and includes an overview of the main proof effort. As this algorithm is written in an applicative subset of COMMON LISP, it is intrinsically executable. Some examples of execution are shown in Sect. 6.

Finally, we will discuss the degree of automation achieved and we will also analyze some possible extensions of this work, including a brief overview of the aspects involved in the generalization of Boolean polynomials to obtain polynomials over different algebraic structures.

2 A Brief Overview of ACL2

ACL2 (*A Computational Logic for Applicative Common Lisp*) is the successor of NQTHM [1, 3], the Boyer-Moore theorem prover. A concise description of ACL2 can be found in [10]. In fact, it is necessary to approach ACL2 from three different perspectives to fully understand it.

2.1 ACL2 is a Computational Logic

ACL2 is a first-order quantifier-free logic with equality. Its *syntax* is that of the LISP programming language. This means that a term in the logic is a constant, a variable symbol or the application of a n-ary function symbol (or a λ-expression) to n terms.

The set of *axioms* includes those of propositional logic with equality and some basic axioms that are needed to work with the usual data types. *Inference rules* are the same as in propositional calculus with equality, adding variable instantiation, induction and functional instantiation. The *induction rule* reduces theorem proofs to finite sets of cases by a powerful form of mathematical induction on ε_0-ordinals.

The logic also includes two *extension principles*: the *definitional principle* and the *encapsulation principle*. The former permits the introduction of new function symbols with an axiomatic definition; the system only admits a function under this principle if its termination can be guaranteed under certain conditions. The latter permits the introduction of new function symbols constrained by axioms; to preserve consistence, ACL2 requires "witnesses" of the existence of these functions to be exposed.

2.2 ACL2 is an Applicative Programming Language

Every ACL2 function admitted under the definitional principle is a LISP function. The converse does not hold. The execution of a function must only depend on their arguments if we want to reason about it in ACL2.

Thus, we can think of ACL2 as an *applicative programming language*, that is, a language in which the result of the application of a function is uniquely determined by its arguments. More precisely, ACL2 can be regarded as a side-effect free subset of COMMON LISP.

2.3 ACL2 is an Automated Reasoning System

ACL2 uses several *proof techniques* when trying to prove a theorem. Each proof technique can be viewed as a "process" receiving a formula as its input and producing a set of formulas as its output. The input formula is a theorem *if* each of the output formulas is a theorem.

The *simplification* process includes decision procedures for propositional logic, equality, and linear arithmetic. It also deals with term rewriting and metafunctions [2]. The *destructor elimination* process allows one to replace variables affected by destructor operations with a term consisting of a constructor operation and fresh variables.

The three following processes have a strong heuristic component. The *cross fertilization* process decides when to use and discard equality hypothesis. The *generalization* process decides when to replace non-variable terms with fresh variables. The *irrelevance elimination* process tries to discard those hypotheses not affecting the validity of the conjecture.

The last process is *induction*. It tries to find a suitable induction scheme to prove the conjecture.

3 Formulas

Next, we develop a formalization of CPL formulas in IF-form (IF-formulas) suitable for our purposes. It is noteworthy that a normalized version of this kind of formulas is in the heart of the OBDD family of algorithms [15], because a BDD is just a graph-based representation for an IF-formula.

In fact, the NQTHM Boyer-Moore logic and its descendant ACL2 define the usual propositional connectives after axiomatizing `IF`.

3.1 Representation

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, which come from COMMON LISP and have been formalized in ACL2 by B. Brock [4], 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 recognized by the following expression: `(and (atom x) (not (booleanp x)))`, representing 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.

IF-conses. Our notion of IF-cons is captured by an ACL2 structure. An IF-cons is just a collection of three objects or *slots* (the *test*, the *then* branch, and the *else*

branch). We say that this notion is "weak" because no restrictions are imposed on the types of the elements that can be stored in each slot.

The following invocation of `defstructure` defines a new constructor operation, `if-cons`, and three destructor operations or *readers*: `test`, `then` and `else`. It also creates an extensive theory for automated reasoning about specifications defined in terms of this structure. The predicate `if-consp` will recognize terms constructed with `if-cons`.

```
(defstructure if-cons test then else
  (:options
   (:conc-name nil)
   (:weak-predicate if-consp)))
```

IF-formulas. 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)))))
```

3.2 Interpretation

An assignment of values to variables can be represented as a list of variables. Thus, the value of a variable w.r.t. an assignment is `t` if the variable belongs to the list, otherwise `nil`. The value of a formula w.r.t. an assignment is computed recursively:

1. A Boolean constant is assigned itself.
2. A variable is assigned its corresponding value in the assignment.
3. A well-formed non-atom formula is assigned:
   (a) the value of its *then* branch, if the value of its *test* is `t`, or
   (b) the value of its *else* branch, if the value of its *test* is `nil`.

To make the valuation function total, we assign an arbitrary meaning to non-formulas. The value of a variable is computed by `belongs`.

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

4 Polynomial Boolean Ring

Representation issues are especially important when dealing with multivariate polynomials, due to the great variety of possibilities and to the differences between the resultant algorithms.

Note that a dense representation would not be appropriate any more, because it is tremendously space-inefficient (especially when the number of variables is high). Consequently, we are assuming a sparse [6] representation. On the other hand, we use a unnormalized representation [17], less efficient from the algorithmic point of view, but which makes it easier to verify the properties.

It suffices with a polynomial Boolean ring for our current purposes, where monomials do not need coefficients. However, we have implemented monomials with coefficients and terms to reuse part of a major formalization effort on a polynomial ring developed in a previous work [14].

We can use the ring $\langle \mathbb{Z}_2, +, -, \cdot, 0, 1 \rangle$ as a coefficient ring for polynomials. However, an easier formalization is obtained by regarding the induced Boolean ring $\langle \mathbb{Z}_2, +, \cdot, 0, 1 \rangle$, because the inverse of the first operation is not needed.

We have represented this induced Boolean ring as $\langle$`z2p`, `+`, `*`, `null`, `identity`$\rangle$ in an ACL2 package named `Z2`. Internally, we have implemented it through the Boolean ring $\langle$`booleanp`, `xor`, `and`, `nil`, `t`$\rangle$. The function `xor` is simply the logical exclusive disjunction.

We obtain a quite simple representation of a Boolean term on a given set of variables by using the Boolean list of the exponents of the variables. Then it is proved that Boolean terms form a commutative monoid w.r.t. a suitable multiplication operation and that the lexicographical ordering on terms is well-founded.

Having chosen the set of variables, it suffices to "or" their exponents element by element to compute the product of two compatible terms. A proof of terms having a commutative monoid structure w.r.t. this operation is easily obtained.

To order terms it is only necessary to take into account their exponent lists. The obvious choice is to set up a lexicographical ordering among them. It is not difficult to prove that this relation satisfies the properties of a strict partial ordering (irreflexivity and transitivity properties hold). Trichotomy is proved too. Well-foundedness is considerably more difficult to prove: it is done by a proper embedding in ε_0-ordinals.

A Boolean monomial is the product of a Boolean coefficient and a Boolean term. Clearly, to represent a monomial it suffices to use a list whose first element is its coefficient and whose rest is the accompanying term. A multiplication operation is defined and then it is proved that monomials have a monoid commutative structure w.r.t. it.

A Boolean polynomial is a finite sum of monomials. Therefore, a polynomial is simply represented by a list of monomials. In order to decide whether two polynomials are semantically equivalent, we must check that they belong to the same equivalence class. This is done by computing their normal forms and examining whether they are syntactically equal. A polynomial is said to be in normal form if their monomials are strictly ordered by the decreasing term order and none of them is null.

Then operations on polynomials are defined. To ensure that these operations and the selected representation satisfy the fundamental properties everybody expects from them, the existence of a Boolean ring structure is proved.

5 Translation of IF-Formulas into Boolean Polynomials

Next, we present the relation between Boolean rings and Boolean algebras and we use it to derive the translation algorithm.

Theorem 1 (Stone). *Let $\langle B, \vee, \wedge, \neg, 0, 1\rangle$ a Boolean algebra, $a, b \in B$ and $+$, $\cdot$ operators defined by $a + b = (a \wedge \neg b) \vee (\neg a \wedge b)$ and $a \cdot b = a \wedge b$, then $\langle B, +, \cdot, 0, 1\rangle$ is a Boolean ring with identity. Conversely, let $\langle B, +, \cdot, 0, 1\rangle$ be a Boolean ring with identity, $a, b \in B$ and $\vee$, $\wedge$, $\neg$ operators defined by $a \vee b = a \cdot b + a + b$, $a \wedge b = a \cdot b$ and $\neg a = a + 1$, then $\langle B, \vee, \wedge, \neg, 0, 1\rangle$ is a Boolean algebra.*

Now, let us consider a Boolean algebra and the following three place Boolean function *if*, defined on it:

$$\forall a, b, c \in B \; if(a, b, c) = (a \wedge b) \vee (\neg a \wedge c) \; . \tag{1}$$

This is the conditional construct found in almost every programming language. By restricting a to 0 and 1, we have:

$$\forall b, c \in B \; if(1, b, c) = b \quad \text{and} \quad \forall b, c \in B \; if(0, b, c) = c \; . \tag{2}$$

We can build an associated *if* function in the domain of the corresponding Boolean ring by applying Th. 1 to (1).

$$if(a, b, c) = a \cdot b \cdot (a + 1) \cdot c + a \cdot b + (a + 1) \cdot c = a \cdot b + a \cdot c + c \; . \tag{3}$$

The following ACL2 function uses this fact 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`$\rangle$.

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

(defun stone-aux (f n)
  (cond ((booleanp f) (if f (identity) (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
```

Of course, ACL2 proves that the Stone polynomial associated to a formula is a polynomial. This result holds even if the formula is not well-formed, due to the completion of the function `stone-aux`.

```
(defthm polynomialp-stone
  (polynomialp (stone f)))
```

Next, we define the evaluation of a polynomial. This is a three-step process. Note that we have to adapt the assignment to a suitable form before evaluating the polynomial.

```
(defun ev (p v)
  (ev-polynomial p (assignment->valuation v)))
```

We evaluate a polynomial by evaluating each of its monomials and adding the results together. Each monomial is evaluated by multiplying its coefficient by the evaluation of its term. In order to make the evaluation total, the following functions are completed for non-polynomials and non-valuations.

```
(defun ev-polynomial (p v)
  (cond ((or (not (polynomialp p)) (not (valuationp v)))
         (Z2::null))                    ; for completeness
        ((nullp p) (Z2::null))
        (t (Z2::+ (Z2::* (coefficient (first p))
                         (ev-term (term (first p)) v))
                  (ev-polynomial (rest p) v)))))

(defun ev-term (te v)
  (cond ((or (not (TER::termp te)) (not (valuationp v)))
         (Z2::null))                    ; for completeness
        ((TER::nullp te) (Z2::identity))
        ((equal v nil) (Z2::null))
        (t (and (or (not (first te)) (first v))
                (ev-term (rest te) (rest v))))))
```

The hard part of the work is dealing with the following properties about the evaluation function and the polynomial operations. These theorems establish a morphism between the Boolean ring $\langle$`polynomialp`, `+`, `*`, `null`, `identity`$\rangle$ and the Boolean ring $\langle$`booleanp`, `xor`, `and`, `nil`, `t`$\rangle$ through the evaluation function.

```
(defthm ev-null
  (equal (ev (null) x) nil))

(defthm ev-identity
  (equal (ev (identity) x) t))

(defthm ev-+
  (implies (and (polynomialp p) (polynomialp q) (assignmentp x))
           (equal (ev (+ p q) x) (xor (ev p x) (ev q x)))))

(defthm ev-*
  (implies (and (polynomialp p) (polynomialp q) (assignmentp x))
           (equal (ev (* p q) x) (and (ev p x) (ev q x)))))
```

We also need to prove a similar property about the function that transforms a propositional variable into a polynomial.

```
(defthm ev-variable->polynomial
  (implies (and (variablep v) (assignmentp x) (variablep n) (<= v n)
           (equal (ev (variable->polynomial v n) x) (belongs v x)))))
```

Finally, we achieve the main result. It states that the translation of formulas into polynomials preserves interpretation:

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

The proof found by ACL2 is done by induction, using the intermediate theorems that we have presented. The induction scheme is derived from the recursion structure of `stone-aux`. ACL2 devised this scheme and the whole subsequent proof on its own.

6 Execution Examples

We provide here some examples of execution under ACL2 of the translation algorithm, `stone`, and the normalization function, `nf`. In order to make them more readable, we are assuming in this section the following macro definitions.

```
(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)))
```

Due to the convention of using natural numbers as variables, we can think of 0 and 1 as the representations of the variables a and b, respectively.

For example, we can write `(stone (or 0 (not 0)))` to obtain the polynomial associated to the tautology formula $a \vee \neg a$.

```
STONE !> (stone (or 0 (not 0)))
((T (T)) (T (T)) (T NIL))
```

The result stands for the polynomial $a + a + 1$ whose normal form is the identity polynomial, as we can easily check:

```
STONE !> (nf (stone (or 0 (not 0))))
((T NIL))
```

Next, the polynomial associated to the contradiction formula $a \wedge (b \wedge \neg a)$ is computed:

```
STONE !> (stone (and 0 (and 1 (not 0))))
((T (T T)) (T (T T)))
```

In this case, the result is the polynomial $a \cdot b + a \cdot b$. We can also obtain its normal form which is the null polynomial.

```
STONE !> (nf (stone (and 0 (and 1 (not 0)))))
NIL
```

7 Conclusions and Further Work

An automated certification of an algorithm for translating CPL formulas to Boolean polynomials has been presented. This algorithm is based on the strong relation existing between Boolean algebras and Boolean rings.

This work requires the formalization of polynomial Boolean rings. This includes a lexicographical ordering on monomials, a normalization function and an induced equivalence relation on which congruences can be defined.

This formalization of polynomials shares a common structure with [14], where basic polynomial arithmetic on rationals is presented. The interested reader should read this paper for many details that we have omitted here. Nevertheless, this work had to be completed with the formalization of an evaluation function for Boolean polynomials. Both works include arithmetic operations, ring properties, normalization functions and some useful congruences.

The degree of automation achieved is acceptable, though some technical and intermediate lemmas were required along the proofs. In addition, sometimes it has been necessary to devise some induction schemes.

Although a portion of our previous polynomial formalization has been reused in this work, we are working in abstracting the coefficient ring to obtain polynomials over arbitrary rings. This presents several advantages, but the most important thing is that it would allow the replacement of coefficients without affecting subsequent proofs.

This can be achieved by using the encapsulation principle of the ACL2 logic to constrain coefficient operations to the desired properties. Later, functional instantiation can be used to obtain concrete implementations (e.g., polynomials over $\mathbb{Z}_p$).

Eventually, our current aim is to obtain a certified SAT decision procedure through normalization using the basic results provided by this work. Counterexample extraction capabilities would be also desirable.

References

1. Boyer, R. S., Moore, J S.: A Computational Logic. Academic Press (1978)
2. 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)
3. Boyer, R. S., Moore, J S.: A Computational Logic Handbook. Academic Press. 2nd edn. (1998)
4. Brock, B.: `defstructure` for ACL2. Computational Logic, Inc. (1997)
5. 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)
6. Johnson, S. C.: Sparse Polynomial Arithmetic. SIGSAM Bulletin **8** (1974)
7. Hsiang, J.: Refutational Theorem Proving using Term-Rewriting Systems. Artificial Intelligence **25** (1985)
8. Hsiang, J.: Rewrite Method for Theorem Proving in First-Order Theory with Equality. J. Symbolic Computation **3** (1987)
9. 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)
10. 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)

11. Kaufmann, M., Manolios, P., Moore, J S.: Computer-Aided Reasoning: An Approach. Kluwer Academic Publishers (2000)
12. Kaufmann, M., Manolios, P., Moore, J S.: Computer-Aided Reasoning: ACL2 Case Studies. Kluwer Academic Publishers (2000)
13. Kapur, D., Narendran, P.: An Equational Approach to Theorem Proving in First-Order Predicate Calculus. Ninth International Conference on Artificial Intelligence (1985)
14. 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)
15. Moore, J S.: Introduction to the OBDD Algorithm for the ATP Community. Computational Logic, Inc. Technical Report 84 (1992)
16. Stone, M.: The Theory of Representation for Boolean Algebra. Trans. AMS **40** (1936)
17. Winkler, F.: Polynomial Algorithms in Computer Algebra. Springer-Verlag (1996)
18. 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)
19. Wu, J.: First-Order Polynomial based Theorem Proving. In: Gao, X., Wang, D. (eds.): Mathematics Mechanization and Applications. Academic Press (1999)
20. Zhegalkin, I. I.: On a Technique of Evaluation of Propositions in Symbolic Logic. Mat. Sb. **34** (1927)