Proving and Computing: Applying Automated Reasoning to the Verification of Symbolic Computation Systems (Invited Talk)

José-Luis Ruiz-Reina

Computational Logic Group
Dept. of Computer Science and Artificial Intelligence, University of Seville
E.T.S.I. Informática, Avda. Reina Mercedes, s/n. 41012 Sevilla, Spain
jruiz@us.es

Abstract. The application of automated reasoning to the formal verification of symbolic computation systems is motivated by the need of ensuring the correctness of the results computed by the system, beyond the classical approach of testing. Formal verification of properties of the implemented algorithms require not only to formalize the properties of the algorithm, but also of the underlying (usually rich) mathematical theory.

We show how we can use ACL2, a first-order interactive theorem prover, to reason about properties of algorithms that are typically implemented as part of symbolic computation systems. We emphasize two aspects. First, how we can override the apparent lack of expressiveness we have using a first-order approach (at least compared to higher-order logics). Second, how we can execute the algorithms (efficiently, if possible) in the same setting where we formally reason about their correctness.

Three examples of formal verification of symbolic computation algorithms are presented to illustrate the main issues one has to face in this task: a Gröbner basis algorithm, a first-order unification algorithm based on directed acyclic graphs, and the Eilenberg-Zilber algorithm, one of the central components of a symbolic computation system in algebraic topology.

1 Introduction

Formal verification of the correctness properties of computing systems is one of the main applications of mechanized reasoning. This is applied in any situation where correctness is so important that one has to verify the system beyond the classical approach of testing. For example, safety critical systems or those where failures may produce high economic losses (these include hardware, microprocessors, microcode and software systems or, more precisely, models of them). In these cases, to increase confidence in the system, a theorem stating its main properties is mechanically proved using a theorem prover.

Symbolic computation systems are software systems, so this idea can be applied to formally verify the correctness of the algorithms implemented in them.

Since in this case they are usually based on a rich mathematical theory, formal verification require not only to formalize the properties of the algorithms, but also of the underlying theory. Another important aspect to take into account is that the models implemented have to be executable and, if possible, efficient.

ACL2 [7,8] is a theorem prover that uses a first-order logic to reason about properties of programs written in an applicative programming language. It has been successfully applied in a number of industrial–strength verification projects [7]. In this talk, we argue that it can be also applied to the verification of symbolic computation systems. We emphasize two aspects. First, how we deal with the apparent lack of expressiveness of a first–order logic (at least compared to higher-order logics). Second, how we can execute the algorithms modeled (efficiently, if possible) in a setting where we also formally reason about them.

We illustrate the main issues one has to deal with when facing this task, by means of three examples. First, Buchberger algorithm for computing a Gröbner basis of a given polynomial ideal. Second, an algorithm, based on directed acyclic graphs, for computing most general unifiers of two given first-order terms. Finally the Eilenberg-Zilber theorem, a central theorem in algebraic topology.

2 The ACL2 System

ACL2 is both a programming language, a logic for reasoning about programs in the language and a theorem prover to assist in the development of proofs of theorems in the logic.

As a programming language, ACL2 is an extension of an applicative subset of Common Lisp. This means that it contains none of Common Lisp that involve side effects like global variables or destructive updates. In this way, functions in the programming language behave as functions in mathematics, and thus one can reason about them using a first-order logic.

The ACL2 logic is a quantifier-free, first-order logic with equality. The logic includes axioms for propositional logic and for a number of Lisp functions and data types, describing the programming language. Rules of inference of the logic include those for propositional calculus, equality and instantiation. But maybe the main rule of inference is the *principle of induction*, that permits proofs by well-founded induction on the ordinal ε_0. This include induction on natural numbers and structural induction.

From the logical point of view, ACL2 functions are total, in the sense that they are defined on every input, even if it is not an intended input. By the *principle of definition*, new function definitions are admitted as definitional axioms only if there exists a measure in which the arguments of each recursive call decrease with respect to a well-founded relation, ensuring in this way that no inconsistencies are introduced by new definitions.

The ACL2 theorem prover mechanizes that logic, being particularly well suited for obtaining mechanized proofs based on simplification and induction. ACL2 is automatic in the sense that once a conjecture is submitted to the prover, the attempt is carried out without interaction with the user. But for non-elementary

results, is often the case that the prover fails to find a proof in its first attempt. Thus, we can say that ACL2 is interactive in the sense that the role of the user is essential for a successful use of the prover: usually, she has to provide a number of definition and lemmas (what is called the *logical world*) that the system will use as simplification (rewrite) rules. The proper definitions and lemmas needed for the proof are obtained first from a preconceived hand proof, but also from inspection of failed proof attempts.

3 Gröbner Basis Computation

In [13], a formal verification of a Common Lisp implementation of Buchberger's algorithm [4] for computing Gröbner bases of polynomial ideals is presented. This needed to formalize a number of previous mathematical theories, including results about coefficient fields, polynomial rings and ideals, abstract reductions, polynomial reductions, ordinal measures and of course Gröbner bases. All these notions fit quite well in the first–order ACL2 logic.

It is worth mentioning that this formal project benefited from previous works done in the system. In particular, an ACL2 theory about term rewriting systems had been previously developed [14]. It turns out that the notions of critical pair and of complete rewrite system [2] are closely related to the notions of *s-polynomial* and Gröbner basis, respectively. In fact, some of the results needed in both formalizations are concrete instances of general results about abstract reductions. Thus, once proved the abstract results, they can applied in any concrete context. *Encapsulation* and *functional instantiation* [9] in ACL2 are a good abstraction mechanism that provides some kind of second–order reasoning in this first–order logic, allowing to reuse previous general results in a convenient way.

Another key concept in this formalization is the notion of *polynomial proof*. In this context, a polynomial ideal basis is seen as a rewriting system that reduces polynomial in some sense, generalizing the notion of polynomial division. In [13], the concept of polynomial proof is introduced, as a data structure the contains explicitly all the components of a sequence of polynomial reductions. It turns out that the correctness properties of the Buchberger algorithm can be described as properties of certain functions that transform polynomial proofs.

4 A Dag–Based Quadratic Unification Algorithm

A *unification algorithm* [1] receives as input a pair of first-order terms and returns, whenever it exists, a most general substitution of its variables for terms, such that when applied to both terms, they become equal. Unification is a key component in automated reasoning and in logic programming, for example.

A naive implementation of unification may have exponential complexity in worst cases. Nevertheless, using more sophisticated data structures and algorithms, it is possible to implement unification quadratic in time and linear in space complexity. In [15], the ACL2 implementation and formal verification of

such an efficient algorithm is reported. The key idea is to use a data structure based on directed acyclic graphs (*dags*), which allows *structure sharing*. This implementation can be executed in ACL2 at a speed comparable to a similar C implementation, but in addition its correctness properties are formally verified.

Two main issues where encountered in this formalization project:

- For the execution efficiency of the implementation, it is fundamental that the substitution computed by the algorithm is built by iteratively applying destructive updates to the dag representing the terms to unify. In principle, as said before, ACL2 is an applicative programming language, so destructive updates are not allowed. Nevertheless, ACL2 provides *single-threaded objects (stobjs)* [3], which are data structures with a syntactically restricted use, so that only one instance of the object needs ever exist. This means that when executing an algorithm that uses a stobj, its fields can be updated by destructive assignments, while maintaining the applicative semantics for reasoning about it. In this case, using a stobj to store the input terms as a dag (implementing structure sharing by means of pointers), allows to clearly separate reasoning about the logic of the unification process (which is independent of how terms are represented) from the details related to the efficient data structures used.
- In ACL2, functions are total, and before being admitted in the logic, their termination for every possible input has to be proved. Thus, in principle, this unification algorithm cannot be defined in the ACL2 logic, because a possible input could be, for example, a stobj storing a cyclic graph, which could lead the algorithm to a non-terminating execution. Nevertheless, we know that the intended inputs to the algorithm will always be an acyclic graph and that for those intended inputs, it can be proved that the algorithm terminates. Thus, to accept the definition of the algorithm in the logic, we need to introduce in its logical definition a condition checking that the structure stored in the stobj is acyclic and that it represents well-formed terms. Nevertheless, from the efficiency point of view this is unacceptable, since this expensive check would be evaluated in every iteration of the algorithm. Fortunately, the combination of the *defexec* feature [6], together with the *guard verification mechanism* allows to safely skip this expensive check when executing, provided that the function has received a well-formed input and taking into account that it is previously proved that in every iteration the well-formedness condition is preserved.

5 The Eilenberg-Zilber Theorem

The Eilenberg–Zilber theorem [12] is a fundamental theorem in Simplicial Algebraic Topology, establishing a bridge between a geometrical concept (cartesian product) and a algebraic concept (tensor product). Concretely, it states homological equivalence between the cartesian product and the tensor product of two chain complexes. The Eilenberg-Zilber theorem, expressed as a reduction, has a

correspondent algorithm that it is a central component of the computer algebra system *Kenzo* [5], devoted to computing in Algebraic Topology.

Since Kenzo is implemented in Common Lisp, it seems that ACL2 is a good choice to verify some of its components, as it is in this case of the Eilenberg-Zilber algorithm. Although Kenzo is far from being implemented using only applicative features, we can formally verify an ACL2 version of the algorithm, and use it as a *verified checker* for the results obtained using Kenzo.

In [11], it is reported a complete ACL2 formal proof of the Eilenberg-Zilber theorem. In fact, the formalization presented was developed reusing part of a previous formal proof of a normalization theorem needed as a preprocessor justifying the way Kenzo works [10].

The formal proof of the Eilenberg-Zilber theorem is not trivial at all. The first issue encountered was that the existing (informal) proofs were not suitable for being formalized in a first–order logic, so a new informal proof had to be carried out by hand, and then formalized in ACL2. In this proof, the key component is a new structure called *simplicial polynomial*, which represent linear combinations of composition of simplicial operators. Although those linear combinations of functions are in principle second–order objects, it turns out that they can be represented as a first-order object. Moreover, the set of simplicial polynomials, together with addition and composition operations, has a ring structure. It turns out that the new proof developed is carried out mainly by establishing a number of lemmas that, although being non-trivial, can be proved by induction and simplification using the ring properties and the properties given by the simplicial identities (the identities that define simplicial sets). Induction and simplification is the kind of reasoning that is suitable for the ACL2 theorem prover.

References

1. Baader, F., Snyder, W.: Unification Theory. In: Handbook of Automated Reasoning. Elsevier Science Publishers (2001)
2. Baader, F., Nipkow, T.: Term Rewriting and All That. Cambridge University Press (1998)
3. Boyer, R., Moore, J S.: Single-Threaded objects in ACL2 (1999), `http://www.cs.utexas.edu/users/moore/publications/stobj/main.pdf`
4. Buchberger, B.: Bruno Buchberger's PhD thesis 1965: An algorithm for finding the basis elements of the residue class ring of a zero dimensional polynomial ideal. Journal of Symbolic Computation 41(3-4), 475–511 (2006)
5. Dousson, X., Rubio, J., Sergeraert, F., Siret, Y.: The Kenzo Program, Institut Fourier (1999), `http://www-fourier.ujf-grenoble.fr/~sergerar/Kenzo/`
6. Greve, D., Kaufmann, M., Manolios, P., Moore, J S., Ray, S., Ruiz-Reina, J.-L., Sumners, R., Vroon, D., Wilding, M.: Efficient execution in an automated reasoning environment. Journal of Functional Programming 1, 15–46 (2008)
7. Kaufmann, M., Moore, J S.: ACL2 home page. University of Texas at Austin (2014), `http://www.cs.utexas.edu/users/moore/acl2`
8. Kaufmann, M., Manolios, P., Moore, J S.: Computer-Aided Reasoning: An Approach. Kluwer Academic Publishers (2000)

9. Kaufmann, M., Moore, J S.: Structured Theory Development for a Mechanized Logic. Journal of Automated Reasoning 26(2), 161–203 (2001)
10. Lambán, L., Martín-Mateos, F.-J., Rubio, J., Ruiz-Reina, J.-L.: Formalization of a normalization theorem in simplicial topology. Annals of Mathematics and Artificial Intelligence 64, 1–37 (2012)
11. Lambán, L., Martín-Mateos, F.-J., Rubio, J., Ruiz-Reina, J.-L.: Verifying the bridge between simplicial topology and algebra: the Eilenberg-Zilber algorithm. Logic Journal of the IGPL 22(1), 39–65 (2014)
12. May, J.P.: Simplicial objects in Algebraic Topology. Van Nostrand (1967)
13. Medina–Bulo, I., Palomo–Lozano, F., Ruiz–Reina, J.-L.: A verified Common Lisp implementation of Buchberger's algorithm in ACL2. Journal of Symbolic Computation 45(1), 96–123 (2010)
14. Ruiz-Reina, J.L., Alonso, J.A., Hidalgo, M.J., Martín-Mateos, F.J.: Formal proofs about rewriting using ACL2. Ann. Math. and Artificial Intelligence 36(3), 239–262 (2002)
15. Ruiz-Reina, J.-L., Alonso, J.-A., Hidalgo, M.-J., Martín-Mateos, F.J.: Formal correctness of a quadratic unification algorithm. Journal of Automated Reasoning 37(1-2), 67–92 (2006)