Symbolic Computation and Automated Reasoning for Program Analysis

Laura Kovács[1,2](✉)

[1] Chalmers University of Technology, Gothenburg, Sweden
laura.kovacs@chalmers.se
[2] TU Wien, Vienna, Austria

Abstract. This talk describes how a combination of symbolic computation techniques with first-order theorem proving can be used for solving some challenges of automating program analysis, in particular for generating and proving properties about the logically complex parts of software. The talk will first present how computer algebra methods, such as Gröbner basis computation, quantifier elimination and algebraic recurrence solving, help us in inferring properties of program loops with non-trivial arithmetic. Typical properties inferred by our work are loop invariants and expressions bounding the number of loop iterations. The talk will then describe our work to generate first-order properties of programs with unbounded data structures, such as arrays. For doing so, we use saturation-based first-order theorem proving and extend first-order provers with support for program analysis. Since program analysis requires reasoning in the combination of first-order theories of data structures, the talk also discusses new features in first-order theorem proving, such as inductive reasoning and built-in boolean sort. These extensions allow us to express program properties directly in first-order logic and hence use further first-order theorem provers to reason about program properties.

1 Introduction

The successful development and application of powerful verification tools such as model checkers [3, 18], static program analyzers [5], symbolic computation algorithms [2], decision procedures for common data structures [16], as well as theorem provers for first- and higher-order logic [17] opened new perspectives for the automated verification of software systems. In particular, increasingly common use of concurrency in the new generation of computer systems has motivated the integration of established reasoning-based methods, such as satisfiability modulo theory (SMT) solvers and first-order theorem provers, with complimentary techniques such as software testing [8]. This kind of integration has however imposed new requirements on verification tools, such as inductive reasoning [13, 15], interpolation [9], proof generation [7], and non-linear arithmetic symbolic computations [6]. Verification methods combining symbolic computation and automated reasoning are therefore of critical importance for improving software reliability.

E. Ábrahám and M. Huisman (Eds.): IFM 2016, LNCS 9681, pp. 20–27, 2016.
DOI: 10.1007/978-3-319-33693-0_2

In this talk we address this challenge by automatic program analysis. Program analysis aims to discover program properties preventing programmers from introducing errors while making software changes and can drastically cut the time needed for program development, making thus a crucial step to automated verification. The work presented in this talk targets the combination of symbolic computation techniques from algorithmic combinatorics and computer algebra with first-order theorem proving and static analysis of programs. We rely on our recent *symbol elimination method* [13]. Although the symbol elimination terminology has been introduced only recently by us, we argue that symbol elimination can be viewed as a general framework for program analysis. That is, various techniques used in software analysis and verification, such as Gröbner basis computation or quantifier elimination, can be seen as application of symbol elimination to safety verification of programs.

In a nutshell, symbol elimination is based on the following ideas. Suppose we have a program P with a set of variables V. The set V defines the language of P. We extend the language P to a richer language P_0 by adding functions and predicates, such as loop counters. After that, we automatically generate a set Π of first-order properties of the program in the extended language P_0, by using techniques from symbolic computation and theorem proving. These properties are valid properties of the program, however they use the extended language P_0. At a last step of symbol elimination we derive from Π program properties in the original language P, thus "eliminating" the symbols in $P_0 \setminus P$.

The work presented in this talk describes symbol elimination in the combination of first-order theorem proving and symbolic computation. Such a combination requires the development of new reasoning methods based on superposition first-order theorem proving [14], Gröbner basis computation [2], and quantifier elimination [4]. We propose symbol elimination as a powerful tool for program analysis, in particular for generating program properties, such as loop invariants and Craig interpolants. These properties express conditions to hold at intermediate program locations and are used to prove the absence of program errors, hence they are very important for improving automation of program analysis.

Since program analysis requires reasoning in the combination of first-order theories of data structures, the talk also presents new features in first-order theorem proving, such as inductive reasoning and built-in boolean sort. These extensions allow us to express program properties directly in first-order logic and hence use further first-order theorem provers to reason about program properties.

The algorithms described in this talk are supported by the development of the world-leading theorem prover Vampire [14], and its extension to support program analysis. Thanks to the full automation and tool support of our work, researchers and software engineers/developers are able to use our results in their work, without the need to become experts in first-order theorem proving and symbolic computation.

The work presented here is structured as follows. We first describe the use of symbol elimination in symbolic computation for generating polynomial program properties (Sect. 3). We then extend symbol elimination to its use in first-order theorem proving and present how arbitrarily quantified program properties can be inferred using symbol elimination (Sect. 4).

2 Motivating Example

Let us first motivate the work described in this talk on a small example. Consider the program given in Fig. 1, written in a C-like syntax. The program fills an integer-valued array B by the positive values of a source array A added to the values of a function call h, and an integer-valued array C with the non-positive values of A. In addition, it computes the sum s of squares of the visited positions in A. A safety assertion, in first-order logic, is specified at the end of the loop, using the `assert` construct. The program of Fig. 1 is clearly safe as the assertion is satisfied when the loop is exited. However, to prove program safety we need additional loop properties, i.e. invariants, that hold at any loop iteration. It is not hard to derive that after any iteration k of the loop (assuming $0 \leq k \leq n$), the linear invariant relation $a = b + c$ holds. It is also not hard to argue that, upon exiting the loop, the value of a is n. However, such properties do not give us much information about the arrays A, B, C and the integer s. For proving program safety, we need to derive that each $B[0], \ldots, B[b-1]$ is the sum of a strictly positive element in A and the value of f at the corresponding position of B. We also need to infer that s stores the sum of squares of the first n non-negative integers, corresponding to the visited positions in A. Formulating these properties in first-order logic yields the loop invariant:

```
a := 0; b := 0; c := 0; s :=0;
while (a < n) do
   if A[a] > 0
      then B[b] := A[a] + h(b);b := b + 1;
      else C[c] := A[a];c := c + 1;
   a := a + 1; s:=s + a * a;
end do
assert((∀p)(0 ≤ p < b ⟹ B[p] − h(p) > 0) ∧
            6 * s = n * (n + 1) * (2 * n + 1))
```

Fig. 1. Motivating example.

$$\begin{aligned} &(\forall p)(0 \leq p < b \implies \\ &\quad (\exists q)(0 \leq q < a \ \wedge\ A[q] > 0 \ \wedge\ B[p] = A[q] + h(p)) \ \wedge \\ &\quad 6 * s = a * (a+1) * (2 * a + 1)) \end{aligned} \tag{1}$$

The above property requires quantifier alternations and polynomial arithmetic and can be used to prove the safety assertion of the program. This loop property in fact describes much of the intended behavior of the loop and can be used to analyze properties of programs in which this loop is embedded. Generating such loop invariants requires however reasoning in full first-order logic with theories, in our example in the first-order theory of arrays, polynomial arithmetic and uninterpreted functions. Our work addresses this problem and proposes symbol elimination for automating program analysis.

3 Symbol Elimination in Symbolic Computation

The first part of this talk concerns the automatic generation of loop invariants over scalar variables. This line of research implements the general idea of symbol

$$\begin{cases} a^{(k+1)} = a^{(k)} + 1 \\ s^{(k+1)} = s^{(k)} + a^{(k)} * a^{(k)} \end{cases} \qquad \begin{cases} a^{(k)} = a^{(0)} + k \\ s^{(k)} = s^{(0)} + \\ \qquad \frac{k*(k+1)*(2*k+1)}{6} \end{cases} \qquad \begin{array}{l} 6 * s^{(k)} = \\ a^{(k)} * (a^{(k)} + 1) * (2 * a^{(k)} + 1) \end{array}$$

(i) (ii) (iii)

Fig. 2. Symbol Elimination in Symbolic Computation on Fig. 1.

elimination by using techniques from symbolic computation, as follows. Given a loop, we first extend the loop language by a new variable n, called the loop counter. Program variables are then considered as functions of n. Next, we apply methods from algorithmic combinatorics and compute the values of loop variables at arbitrary loop iterations as functions of n. Finally, we eliminate n using computer algebra algorithms, and derive polynomial relations among program variables as loop invariants.

In our work, we identified a certain family of loops, called P-solvable loops (to stand for polynomial-solvable) with sequencing, assignments and conditionals, where test conditions are ignored [11]. For these loops, we developed a new algorithm for generating polynomial loop invariants. Our method uses algorithmic combinatorics and algebraic techniques, namely solving linear recurrences with constant coefficients (so-called C-finite recurrences) or hypergeometric terms, computing algebraic relations among exponential sequences, and eliminating variables from a system of polynomial equations. More precisely, the key steps of using symbol elimination in symbolic computation are as follows. Given a P-solvable loop with nested conditionals, we first rewrite the loop into a collection of P-solvable loops with assignments only. Next, polynomial invariants of all sequences of P-solvable loops with assignments only are derived. These invariants describe polynomial relations valid after the first iteration of the P-solvable loop with nested conditionals, however they might not be valid after an arbitrary iteration of the P-solvable loop with nested conditionals. Therefore, from the ideal of polynomial relations after the first iteration of a P-solvable loop with nested conditionals, we keep only those polynomial relations that are polynomial invariants of the P-solvable loop with nested conditionals. In the process of deriving polynomial invariants for a (sequence of) P-solvable loop(s) with assignments only, we proceed as follows. We introduce a new variable n denoting the loop counter. Next, recurrence equations over the loop counter are constructed, describing the behavior of the loop variables at arbitrary loop iterations. These recurrence relations are solved, and closed forms of loop variables are computed as polynomials of the initial values of variables, the loop counter, and some new variables in the loop counter so that we infer polynomial relations among the new variables. The loop counter and variables in the loop counter are then eliminated by Gröbner basis computation to derive a finite set of polynomial identities among the program variables as invariants. From this finite set any other polynomial identity that is an invariant of the P-solvable loop with assignments only can be derived.

To illustrate the workflow proposed above, consider Fig. 2. Figure 2(i) describes the system of recurrence equations corresponding to the updates over a and s in Fig. 1, where $s^{(k)}$ and $a^{(k)}$ denote the values of s and a at the kth loop iteration of Fig. 1. That is, program variables become functions of loop iterations k. The closed form solutions of Fig. 2(i) is given in Fig. 2(ii). After substituting the initial values of a and s, Fig. 2(iii) shows a valid polynomial identity among the values of a and s at any loop iteration k.

Our invariant generation method using symbol elimination in symbolic computation is proved to be complete in [12]. By completeness we mean that our method generates the basis of the polynomial invariant ideal, and hence any other polynomial invariant of the P-solvable loop can be derived from the basis of the invariant ideal. For doing so, we generalised the invariant generation algorithm of [11] for P-solvable loops by iteratively computing the polynomial invariant ideal of the loop. We proved that this generalisation is sound and complete. That is, our method infers a basis for the polynomial invariant ideal of the P-solvable loop in a finite number of steps. Our proof relies on showing that the dimensions of the prime ideals from the minimal decomposition of the ideals generated at an iteration of our algorithm either remained the same or decreased at the next iteration of the algorithm. Since dimensions of ideals are positive integers, our algorithm terminates after a finite number of iterations.

4 Symbol Elimination in First-Order Theorem Proving

In the second part of our talk, we describe the use of symbol elimination in first-order theorem proving. The method of symbol elimination using a first-order theorem prover has been introduced in [13]. Unlike all previously known techniques, our method allows one to generate first-order invariants containing alternations of quantifiers for programs with arrays.

When using symbol elimination for generating loop invariants of programs with arrays, the method is based on automatic analysis of the so-called update predicates of loops. An update predicate for an array expresses updates made to the array. We observe that many properties of update predicates can be extracted automatically from the loop description and loop properties obtained by other methods such as a simple analysis of counters occurring in the loop, recurrence solving and quantifier elimination over loop variables. In the first step of loop analysis we introduce a new variable n denoting the loop counter, and use the symbolic computation framework from Sect. 3 to generate polynomial invariants over the scalar loop variables. Scalar and array variables of the loop are considered as functions of n and the language P of the loop is extended by these new function symbols. Further, the loop language is also extended by the update predicates for arrays and their properties are added to the extended language too. The update predicates make use of n and essentially describe positions at which arrays are updated, iterations at which the updates occur and the update values of the arrays. For example, we may write $upd(B, k, p, x)$ to express that an array B was updated at loop iteration k and array position p by the value x. For our running example from Fig. 1, $upd(B, k, p, x)$ is defined as:

$$upd(B, k, p, x) \iff 0 \le k \le n \wedge A^{(k)}[a^{(k)}] > 0 \wedge p = b^{(k)} \wedge x = A^{(k)}[a^{(k)}] + h(b^{(k)}),$$

expressing that k is a loop iteration value at which the array B was updated (the true-branch of the conditional of Fig. 1 was visited). As before, $A^{(k)}$ denotes the value of the array A at the kth loop iteration (A is actually unchanged throughout the loop of Fig. 1).

As a result of this step of symbol elimination, a new, extended loop language P_0 is obtained, and a collection Π of valid first-order loop properties expressed in P_0 is derived. For example, a first-order property of Fig. 1 in the extended loop language P_0 derived by our work is:

$$(\forall i, j, p, x)\big(upd(B, i, p, x) \wedge (upd(B, j, p, x) \implies j = i) \implies B^{(n)}[p] = x\big) \quad (2)$$

Property (2) expresses that if the array B is updated only once at a position p, the value x associated with this update is the final value in B.

Formulas in Π cannot be used as loop invariants, since they use symbols not occurring in the loop, and even symbols whose semantics is described by the loop itself. Note that, while the property (2) is a valid loop property, it is not yet a loop invariant as it uses the update predicates $upd(B, i, p, x)$ and $upd(B, j, p, x)$ and $B^{(n)}$ to express the final value of array B as function of n. These symbols are in P_0 but are not part of the loop language P; and hence a loop invariant expressed in the loop language P cannot make use of them, Nevertheless, the formulas in Π, such as (2), are valid properties of the loop and have a useful property: all their consequences are valid loop properties too. The second phase of symbol elimination therefore tries to generate logical consequences of Π in the original language of the loop. Any such consequence is also a valid property of the loop, and hence an invariant of the loop. Logical consequences of Π are generated by running a first-order saturation theorem prover on Π in a way that it tries to eliminate the newly introduced symbols $P_0 \setminus P$ from the extended loop language P_0. As a result of symbol elimination, a loop invariant generated for Fig. 1 is the first-order formula expressed in (1).

The main obstacle to the experimental evaluation of symbol elimination lied in the fact that all existing first-order theorem provers lacked several features essential for implementing our procedure for invariant generation. These features included reasoning with various theories and procedures for eliminating symbols. In our work we addressed these limitations as follows: we changed the term ordering used by theorem provers to make symbol elimination generate loop invariants and we added incomplete, but sound axiomatizations of first-order theories to first-order theorem provers, in particular to the Vampire theorem prover. To this end, Vampire has now built-in support for the first-order theories of integers, rationals and rationals. We have also extended Vampire with the polymorphic theory of arrays with extensionality and added the boolean sort as first-class sort in Vampire [10]. With these new features at hand, Vampire now supports automatic program analysis and invariant generation for programs with arrays [1]. We believe the new extension in Vampire increase the expressivity of first-order reasoners and facilitate reasoning-based program analysis and verification.

Acknowledgments. The work described in this talk is based on joint work with a number of authors, including Tudor Jebelean (RISC-Linz), Evgeny Kotelnikov and Simon Robillard (Chalmers University of Technology), and Andrei Voronkov (The University of Manchester and Chalmers University of Technology).

The author acknowledges funding from the ERC Starting Grant 2014 SYMCAR 639270, the Wallenberg Academy Fellowship 2014, the Swedish VR grant D0497701 and the Austrian research project FWF S11409-N23

References

1. Ahrendt, W., Kovács, L., Robillard, S.: Reasoning About Loops Using Vampire in KeY. In: Davis, M., Fehnker, A., McIver, A., Voronkov, A. (eds.) LPAR-20 2015. LNCS, vol. 9450, pp. 434–443. Springer, Heidelberg (2015). doi:10.1007/978-3-662-48899-7_30
2. Buchberger, B.: An algorithm for finding the basis elements of the residue class ring of a zero dimensional polynomial ideal. J. Symbolic Comput. **41**(3–4), 475–511 (2006)
3. Clarke, E.M., Emerson, E.A.: Design and synthesis of synchronization skeletons using branching-time temporal logic. In: Kozen, D. (ed.) Logic of Programs. LNCS, pp. 52–71. Springer, Heidelberg (1981)
4. Collins, G.E.: Quantifier elimination for real closed fields by cylindrical algebraic decomposition. In: Brakhage, H. (ed.) ATFL. LNCS, pp. 134–183. Springer, Heidelberg (1975)
5. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL, pp. 238–252 (1977)
6. de Moura, L., Passmore, G.O.: Computation in real closed infinitesimal and transcendental extensions of the rationals. In: Bonacina, M.P. (ed.) CADE 2013. LNCS, vol. 7898, pp. 178–192. Springer, Heidelberg (2013)
7. de Moura, L.M., Bjorner, N.: Proofs and refutations, and z3. In: CEUR Workshop Proceedings (2008)
8. Hamon, G., de Moura, L., Rushby, J.M.: Generating efficient test sets with a model checker. In: SEFM, pp. 261–270 (2004)
9. Jhala, R., McMillan, K.L.: A practical and complete approach to predicate refinement. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006. LNCS, vol. 3920, pp. 459–473. Springer, Heidelberg (2006)
10. Kotelnikov, E., Kovács, L., Reger, G., Voronkov, A.: The Vampire and the FOOL. In: Proceedings of CPP, pp. 37–48. ACM (2016)
11. Kovács, L.: Reasoning algebraically about P-solvable loops. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 249–264. Springer, Heidelberg (2008)
12. Kovács, L.: A complete invariant generation approach for P-solvable loops. In: Pnueli, A., Virbitskaite, I., Voronkov, A. (eds.) PSI 2009. LNCS, vol. 5947, pp. 242–256. Springer, Heidelberg (2010)
13. Kovács, L., Voronkov, A.: Finding loop invariants for programs over arrays using a theorem prover. In: Chechik, M., Wirsing, M. (eds.) FASE 2009. LNCS, vol. 5503, pp. 470–485. Springer, Heidelberg (2009)
14. Kovács, L., Voronkov, A.: First-order theorem proving and vampire. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 1–35. Springer, Heidelberg (2013)

15. McMillan, K.L.: Quantified invariant generation using an interpolating saturation prover. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 413–427. Springer, Heidelberg (2008)
16. Nelson, G., Oppen, D.C.: Fast decision procedures based on congruence closure. J. ACM **27**(2), 356–364 (1980)
17. Robinson, J.A., Voronkov, A.: Handbook of Automated Reasoning (in 2 Volumes). Elsevier and MIT Press, Cambridge (2001)
18. Sifakis, J.: A unified approach for studying the properties of transition systems. Theor. Comput. Sci. **18**, 227–258 (1982)

http://www.springer.com/978-3-319-33692-3

Integrated Formal Methods
12th International Conference, IFM 2016, Reykjavik, Iceland, June 1-5, 2016, Proceedings
Ábrahám, E.; Huisman, M. (Eds.)
2016, XIV, 538 p. 145 illus., Softcover
ISBN: 978-3-319-33692-3