Algebra-Based Synthesis of Loops and Their Invariants (Invited Paper)

Andreas Humenberger and Laura Kovács(✉)

TU Wien, Wien, Austria
laura.kovacs@tuwien.ac.at

Abstract. Provably correct software is one of the key challenges in our software-driven society. While formal verification establishes the correctness of a given program, the result of program synthesis is a program which is correct by construction. In this paper we overview some of our results for both of these scenarios when analysing programs with loops. The class of loops we consider can be modelled by a system of linear recurrence equations with constant coefficients, called C-finite recurrences. We first describe an algorithmic approach for synthesising all polynomial equality invariants of such non-deterministic numeric single-path loops. By reverse engineering invariant synthesis, we then describe an automated method for synthesising program loops satisfying a given set of polynomial loop invariants. Our results have applications towards proving partial correctness of programs, compiler optimisation and generating number sequences from algebraic relations.

1 Introduction

The two most rigorous approaches for providing correct software are given by formal program verification and program synthesis [43]. The task of formal verification is to prove correctness of a given program with respect to a given logical specification [6,9,17]. On the other hand, program synthesis aims at generating programs which adhere to a given specification [2,34]. The result of a synthesis problem is therefore a program which is correct by construction with respect to the specification. While formal verification has received considerable attention with impressive results, for example, in ensuring safety of device drivers [3] and security of web services [7], program synthesis turns out to be an algorithmically much more difficult challenge [33].

Both in the setting of verification and synthesis, one of the main challenges is to verify or synthesise programs with loops/recursion. In formal verification, solving this challenge requires for example *synthesising loop invariants* [20,30,39]. Intuitively, a loop invariant is a formal description of the behaviour of the loop, expressing loop properties that hold at arbitrary loop iterations. For the purpose of automating formal verification, synthesising loop invariants that are inductive is of critical importance, as inductive invariants describe program properties/safety assertions that hold before and after each loop iteration.

F. Henglein et al. (Eds.): VMCAI 2021, LNCS 12597, pp. 17–28, 2021.
https://doi.org/10.1007/978-3-030-67067-2_2

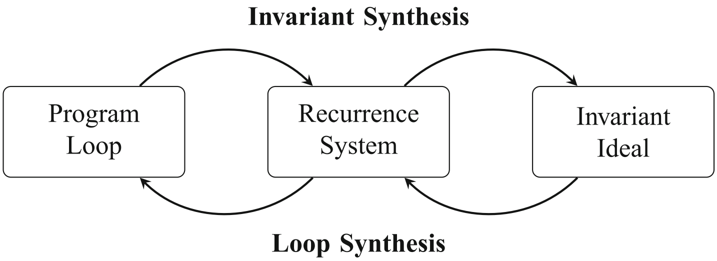

Fig. 1. Algebra-based synthesis of loops and their invariants.

In program synthesis, reasoning with loops requires answering the question whether there exists a loop satisfying a given loop invariant and synthesising a loop with respect to a given invariant. We refer to this task of synthesis as *loop synthesis*. As such, we consider loop synthesis as the reverse problem of loop invariant generation/synthesis: rather than generating invariants summarising a given loop, we synthesise loops whose summaries are captured by a given invariant property.

In this paper, we overview algebra-based algorithms for automating reasoning about loops and their invariants. The key ingredients of our work come with deriving and solving algebraic recurrences capturing the functional behaviour of loops to be verified and/or synthesised. To this end, we consider additional requirements on the loops to be verified/synthesised, in particular by imposing syntactic constraints on the form of loop expressions. The imposed constraints allow us to reduce the verification/synthesis task to the problem of solving algebraic recurrences of special forms. Here, we mainly focus on loops whose functional summaries are precisely captured by so-called C-finite recurrences [27], that is linear recurrences with constant coefficients, for which closed form solutions always exist. We use symbolic summation techniques over C-finite recurrences to compute closed forms and combine these closed forms with additional constraints to ensure that (i) algebraic relations among closed forms yield polynomial loop invariants and (ii) loops synthesised from such polynomial loop invariants implement only affine assignments.

Figure 1 overviews our approach towards synthesising loops and/or their invariants. In order to generate invariants, we extract a system of C-finite recurrence equations describing loop updates. We then compute the polynomial ideal, called the *polynomial invariant ideal*, containing all polynomial equality invariants of the loop, by using recurrences solving and Gröbner basis computation [4]. Any polynomial invariant of the given loop is then a logical consequence of the polynomials from the computed polynomial ideal basis [31]. On the other hand, for loop synthesis, we take a basis of the polynomial invariant ideal generated by given polynomial loop invariants and construct a polynomial constraint problem. This constraint problem precisely characterises the set of all C-finite recurrence systems for which the given polynomial invariants yield algebraic relations among

```
requires $N > 0$
$(x, y, z) \leftarrow (0, 0, 0)$
while $y < N$ do
    $x \leftarrow x + z + 1$
    $z \leftarrow z + 2$
    $y \leftarrow y + 1$
end
ensures $x = N^2$
```

(a) Invariant synthesis for partial correctness.

```
requires $N > 0$
$(x, y) \leftarrow (0, 0)$
while $y < N$ do
    $x \leftarrow x + 2y + 1$
    $y \leftarrow y + 1$
end
ensures $x = N^2$
```

(b) Loop synthesis to "optimize" Figure 2a.

Fig. 2. Motivating example for invariant and loop synthesis.

the induced C-finite number sequences. Every solution of the constraint problem gives thus rise to a system of C-finite recurrence equations, which is then turned into a loop for which the given polynomial relations are loop invariants [23].

In the rest of this paper, we first motivate our results on examples for invariant and loop synthesis (Sect. 2). We then report on algebra-based approaches for invariant generation (Sect. 3) and loop synthesis (Sect. 4), by summarising our main results published at [23,31].

2 Motivating Examples for Synthesising Invariants and Loops

Loop Invariant Synthesis. Verifying safety conditions and establishing partial correctness of programs is one use case of invariant generation. Consider for example the program in Fig. 2a, annotated with pre- and post-conditions specified respectively by the `requires` and `ensures` constructs. The program of Fig. 2a is clearly safe as the post-condition is satisfied when the loop is exited. However, to prove program safety we need additional loop properties, i.e. inductive loop invariants, that hold at any loop iteration. It is not hard to derive that after any iteration n of the loop (assuming $0 \leq n \leq N$), the linear invariant relation $y \leq N$ holds. It is also not hard to argue that, upon exiting the loop, the value of y is N. However, such properties do not give us much information about the (integer-valued) program variable x. For proving program safety, we need to derive loop invariants relating the values of x, y, z at an arbitrary loop iteration n. Our work in [31] generates such loop invariants by computing the polynomial ideal $I = \langle x - y^2, z - 2y\rangle$ as the so-called *polynomial invariant ideal.* The conjunction $x = y^2 \; \wedge z = 2y$ of the polynomial relations corresponding to the basis polynomials of I is an inductive loop invariant, which together with the invariant $y \leq N$ is sufficient to prove partial correctness of Fig. 2a.

Loop Synthesis. One use case of loop synthesis is program optimisation. To reduce execution time spent within loops, compiler optimisation techniques, such

```
$(x, y) \leftarrow (0, 2)$
while $y < N$ do
    $x \leftarrow x + y$
    $y \leftarrow y + 1$
end
```

(a) Erroneous loop

```
$(x, y, z) \leftarrow (0, 0, 0)$
while $y < N$ do
    $x \leftarrow x + z + 1$
    $z \leftarrow z + 2$
    $y \leftarrow y + 1$
end
```

(b) Synthesised loop

```
$(x, y) \leftarrow (0, 0)$
while $y < N$ do
    $x \leftarrow x + 2y + 1$
    $y \leftarrow y + 1$
end
```

(c) Synthesised loop

Fig. 3. Program repair via loop synthesis. Figures b–c, corresponding also to the programs of Figs. 2a–2b, are revised versions of Fig. a such that $x = y^2$ is an invariant of Figs. b–c.

as strength reduction [8], aim at replacing expensive loop operations with semantically equivalent but less expensive operations and/or reducing the number of loop variables used within loops. The burden of program optimisation in the presence of loops comes however with identifying inductive loop variables and invariants to be used for loop optimisation. Coming back to the loop in Fig. 2a, as argued before, $x = y^2 \wedge z = 2y \wedge y \leq N$ is a loop invariant of Fig. 2a. Moreover, only $x = y^2$ is already a loop invariant of Fig. 2a. Our loop synthesis procedure can be used to synthesise the affine loop of Fig. 2b from the polynomial invariant $x = y^2$, such that the synthesised loop uses less variables and arithmetic operations than Fig. 2a. Note that program repair can also be considered as an instance of program optimisation: while maintaining a given polynomial loop invariant, the task is to revise and repair a given program such that it satisfies the given invariant. Our synthesis approach therefore also provides a solution to program repair, as illustrated in Fig. 3.

3 Algebra-Based Synthesis of Loop Invariants

Overview of State-of-the-Art. One of the most related approaches to our work in automating the synthesis of polynomial loop invariants comes with the seminal work of [14], where a method for refining a user-given partial invariant was introduced to prove partial correctness of a given program. One of the first fully automatic invariant generation procedures was then given by [26] for inferring affine relations within affine programs. Since then, loop invariant generation was intensively studied and the level of automation and expressivity with respect to programs and their invariants steadily increased. Here we overview the most related techniques to our work.

The approach of [35] generalised [26] and provided a method for computing all polynomial equality relations for affine programs up to an a priori fixed degree. Recently, [18] constructively proved that the set of all polynomial equality invariants is computable for affine programs.

The works of [40] and [11] fix a polynomial template invariant and derive a constraint problem that encodes properties of loop invariants, such as inductiveness. These constraint problems are then solved by linear or polynomial algebra. The methods of [36] and [38] use abstract interpretation in combination with Gröbner bases computations for computing polynomial invariants of bounded degree. In [5], the abstract interpretation approach from [36] and the constraint-based approach from [40] is combined, yielding a procedure for computing invariants of bounded degree without resorting to Gröbner bases.

The techniques in [12,28–30] approximate an arbitrary loop by a single-path loop and then apply recurrence solving to infer nonlinear invariants. They include guards in loops and conditionals in their reasoning, and are also able to infer inequalities as loop invariant. A data-driven approach to invariant generation is given in [41] using the guess-and-check methodology. Linear algebra is used to guess candidate invariants from data generated by concrete program executions where an upper bound on the polynomial degree of the candidate is user-given. An SMT solver is then used to validate the candidates with respect to the properties of loop invariants. If this is not the case, then the candidate is refined based on the output of the SMT solver.

Our work for invariant generation does neither use abstract interpretation nor constraint solving, and does not fix an a priori bound on the degree of the polynomial invariants to be synthesised. Instead, we restrict the class of loops our work can handle to non-deterministic loops whose loop updates yield special classes of algebraic recurrences in the loop counter, and hence we cannot handle loops with arbitrary nestedness as in [30]. We rely on results of [39] proving that the set of all polynomial equality invariants for a given (non-deterministic) loop forms a polynomial ideal. In [31], we use the ideal-theoretic result of [39] and compute all polynomial invariants of the class of non-deterministic loops that can be modelled by C-finite recurrence equations. Our results can further be extended to more complex recurrences equations by allowing restricted multiplications, and hence restricted classes of linear recurrences with polynomial coefficients, among loop variables - as detailed in [20,22].

Algebra-Based Synthesis of Loop Invariants. We now summarise our algebra-based algorithm for synthesising polynomial loop invariants. To this end, we define our task of loop invariant synthesis as follows:

LOOP INVARIANT SYNTHESIS

- **Given:** A non-deterministic single-path loop $\mathcal{L}$ with program variables $\boldsymbol{x}$ such that each variable from $\boldsymbol{x}$ induces a C-finite number sequence in $\mathcal{L}$;
- **Generate:** A polynomial ideal I of all polynomials $p(\boldsymbol{x})$ such that $p(\boldsymbol{x}) = 0$ is a loop invariant of $\mathcal{L}$.

The main steps of our algorithm for loop invariant synthesis are as follows:

1. The non-deterministic single-path loop $\mathcal{L}$ is transformed into the regular expression π^*, where π is the block of assignments from the loop body of $\mathcal{L}$ and π^* denotes an arbitrary number of executions of π.
2. We extract a system of C-finite recurrence equations for π^*, by describing the C-finite number sequences for each program variable $x_i \in \boldsymbol{x}$ of $\mathcal{L}$ via a C-finite recurrence equation. To this end, we write $x_i(n)$ to denote the value of the program variable $x_i \in \boldsymbol{x}$ at an arbitrary loop iteration $n \geq 0$ as well as to refer to the number sequence $x_i(n)$ induced by the values of x_i at arbitrary loop iterations $n \geq 0$.
3. We solve the resulting C-finite recurrences of π^*, yielding a functional representation of values of $x_i(n)$ depending only on n and some initial values.
4. As a result, we derive closed forms $x_i(n) = f_i(n)$, where f_i are linear combinations of polynomial and exponential expressions in n. We also compute algebraic relations $a_i(n)$ as valid polynomial relations among exponential expressions in n.
5. A polynomial ideal I of all polynomials $p(\boldsymbol{x})$ such that $p(\boldsymbol{x}) = 0$ is a loop invariant of $\mathcal{L}$ is then computed by using Gröbner basis computation to eliminate n from the ideal generated by $\langle x_i - f_i(n), a_i(n)\rangle$. The ideal I is called the *polynomial invariant ideal* of $\mathcal{L}$.

Example 1 (Loop invariant synthesis). We illustrate our algorithm for loop invariant synthesis on the loop of Fig. 2a. The loop guard of Fig. 2a is ignored. Using matrix notation, the block π of loop body assignments induces the following coupled system of C-finite recurrence equations for π^*, with $n \geq 0$:

$$\begin{pmatrix} x(n+1) \\ z(n+1) \\ y(n+1) \end{pmatrix} = \begin{pmatrix} 2 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x(n) \\ z(n) \\ y(n) \end{pmatrix} + \begin{pmatrix} 1 \\ 2 \\ 1 \end{pmatrix}$$

The closed form solutions of the above recurrence system are given by

$$\begin{cases} x(n) = x(0) + n^2 \\ z(n) = z(0) + 2n \\ y(n) = y(0) + n \end{cases}$$

with $x(0) = 0$, $y(0) = 0$ and $z(0) = 0$ from the initial value assignments of Fig. 2a. By eliminating n from $\langle x - n^2, z - 2n, y - n\rangle$, we derive the polynomial invariant ideal $I = \langle x - y^2, z - 2y\rangle$ of π^*, yielding the polynomial loop invariant $x = y^2 \land z = 2y$.

Automation and Implementation. Our algorithm for loop invariant synthesis is fully automated within the open-source Julia package Aligator, which is available at:

https://github.com/ahumenberger/Aligator.jl.

For experimental summary and comparisons with other tools, in particular with [30], we refer to [19,21].

4 Algebra-Based Synthesis of Loops

Overview of State-of-the-Art. The classical setting of program synthesis has been to synthesise programs from proofs of logical specifications that relate the inputs and the outputs of the program [34]. Thanks to recent successful trends in formal verification based on automated reasoning [10,32], this traditional view of program synthesis has been refined to the setting of syntax-guided synthesis (SyGuS) [2]. In addition to logical specifications, SyGuS approaches consider further constraints on the program template to be synthesised, limiting thus the search space of possible solutions. A wide range of efficient applications of SyGuS have so far emerged, for example programming by examples [16], component-based synthesis [24] with learning [13] and sketc.hing [37].

Most synthesis approaches exploit counterexample-guided synthesis [2,13,37,42] within the SyGuS framework. These methods take input-output examples satisfying a given property and synthesise a candidate program that is consistent with the given inputs. Correctness of the candidate program with respect to the given property is then checked using formal verification, in particular using SMT-based reasoning. Whenever verification fails, a counterexample violating the given property is generated as an additional input and a new candidate program is generated. Our work does not use an iterative refinement of the input-output values satisfying a given property $p(\boldsymbol{x}) = 0$. Rather, we consider a precise characterisation of the solution space of loops with invariant $p(\boldsymbol{x}) = 0$ to describe all, potentially infinite input-output values of interest. Similarly to sketches [37,42], we consider loop templates restricting the search for solutions to synthesis. Yet, our templates support non-linear arithmetic, which is not yet the case in [13,37].

The programming by example approach of [15] learns programs from input-output examples and relies on lightweight interaction to refine the specification of programs to be synthesised. The approach has further been extended in [25] with machine learning, allowing to learn programs from just one (or even none) input-output example by using a simple supervised learning setup. Program synthesis from input-output examples is shown to be successful for recursive programs [1], yet synthesising loops and handling non-linear arithmetic is not yet supported by this line of research. Our work precisely characterises the solution space of all loops to be synthesised by a system of algebraic recurrences and does not use statistical models supporting machine learning.

To the best of our knowledge, existing synthesis approaches are restricted to linear invariants, see e.g. [43], whereas our work supports loop synthesis from non-linear polynomial properties. We note that many interesting program properties can be best expressed using non-linear arithmetic, for example programs implementing powers (see e.g. Fig. 2), square roots and/or Euclidean divison require non-linear invariants.

Algebra-Based Synthesis of Loops. Our work in [23] addresses the challenging task of loop synthesis, by relying on algebraic recurrence equations and constraint solving over polynomials. Following the SyGuS setting, we consider additional requirements on the loop to be synthesised and define the task of loop synthesis as follows:

LOOP SYNTHESIS

- **Given:** A polynomial ideal I containing polynomials $p(\boldsymbol{x})$ over a set $\boldsymbol{x}$ of variables;
- **Generate:** A loop $\mathcal{L}$ with program variables $\boldsymbol{x}$ such that
  (i) $p(\boldsymbol{x}) = 0$ is an invariant of $\mathcal{L}$ for every $p \in I$, and
  (ii) each variable from $\boldsymbol{x}$ in $\mathcal{L}$ induces a C-finite number sequence.

The main steps of our loop synthesis algorithm are summarised below.

1. We take a basis B of the polynomial invariant ideal I as our input.
2. We fix a non-deterministic loop template $\mathcal{T}$ whose loop updates define a C-finite recurrence system template $\mathcal{S}$, over variables $\boldsymbol{x}$ and of size s. If not specified, the size s of $\mathcal{S}$ is considered to be the number of variables in $\boldsymbol{x}$.
3. We construct a polynomial constraint problem (PCP) which can be divided into two clause sets C_1 and C_2. The first set C_1 describes the closed form solutions of the C-finite recurrence system $\mathcal{S}$. To this end, we exploit properties of C-finite recurrences and define templates for the closed forms of $\boldsymbol{x}$ by ensuring a one-to-one correspondence between the recurrence template $\mathcal{S}$ and the closed form templates of $\boldsymbol{x}$. Intuitively, the clause set C_1 mimics the procedure for computing the closed forms for the recurrence system $\mathcal{S}$. The second clause set C_2 of our PCP makes sure that, for every $p \in B$, $p(\boldsymbol{x})$ is an algebraic relation for the closed form templates of $\boldsymbol{x}$. Since B is a basis of I it follows that $p(\boldsymbol{x}) = 0$ for all $p \in I$. The solution space of our PCP $C_1 \wedge C_2$ captures thus the set of all C-finite recurrence systems of the form $\mathcal{S}$ such that $p(\boldsymbol{x}(n)) = 0$ holds for all $n \geq 0$ and for all $p \in I$, where $\boldsymbol{x}(n)$ denotes the number sequences induced by the loop variables in $\boldsymbol{x}$ (as discussed on page 5).
4. By solving our PCP, we derive C-finite recurrence systems of the form $\mathcal{S}$. These instances of $\mathcal{S}$ can however be considered as non-deterministic programs with simultaneous updates. Thus, any C-finite recurrence system solution of our PCP can directly be translated into a non-deterministic loop $\mathcal{L}$ with sequential updates, by introducing auxiliary variables. Solving our PCP yields therefore a solution to our task of loop synthesis.

In [23], we prove that our approach to loop synthesis is both sound and complete. By completeness we mean, that if there is a loop $\mathcal{L}$ with at most s variables satisfying the invariant $p(\boldsymbol{x}) = 0$ such that the loop body meets the C-finite syntactic requirements of $\mathcal{S}$, then this loop $\mathcal{L}$ is synthesised by our method. As show-cased by Fig. 3, given a loop invariant $p(\boldsymbol{x}) = 0$, one can synthesise a potentially infinite set of loops such that each loop (i) has $p(\boldsymbol{x}) = 0$ as its invariant and (ii) is "better" with respect to a user-defined preference/measure. Our loop synthesis approach can thus be used to synthesise loops with respect to some pre-defined measure.

Example 2 (Loop invariant synthesis). We illustrate our algorithm for loop synthesis on Fig. 2b. To this end, we are interested in synthesising loops from the non-linear polynomial relation $x = y^2$. The invariant we consider is therefore $p(x, y) = x - y^2 = 0$.

We start by (initially) setting $s = 2$ and defining a loop template $\mathcal{T}$ of the form

```
$(x, y) \leftarrow (a_1, a_2)$
while $true$ do
    $x \leftarrow b_{11}x + b_{12}y + b_{13}$        (1)
    $y \leftarrow b_{21}x + b_{22}y + b_{23}$
end
```

where the a_i and b_{ij} are rational-valued symbolic constants. By denoting with $n \geq 0$ the loop counter, the loop body of (1) can then be modeled by the following C-finite recurrence system:

$$\begin{pmatrix} x(n+1) \\ y(n+1) \end{pmatrix} = \begin{pmatrix} b'_{11} & b'_{12} \\ b'_{21} & b'_{22} \end{pmatrix} \begin{pmatrix} x(n) \\ y(n) \end{pmatrix} + \begin{pmatrix} b'_{13} \\ b'_{23} \end{pmatrix}, \tag{2}$$

where $x(n)$ and $y(n)$ represent the values of variables x and y at iteration n (as discussed on page 5), with $x(0) = a_1$ and $y(0) = a_2$. Note that the values of b_{ij} and b'_{ij} might differ as the sequential assignments of (1) correspond to simultaneous assignments in the algebraic representation (2) of the loop.

We next exploit properties of C-finite recurrences. For simplicity and w.l.o.g, we set up the following closed form templates for $x(n)$ and $y(n)$:

$$\begin{pmatrix} x(n) \\ y(n) \end{pmatrix} = \begin{pmatrix} c_1 \\ c_2 \end{pmatrix} \omega^n + \begin{pmatrix} d_1 \\ d_2 \end{pmatrix} \omega^n n + \begin{pmatrix} e_1 \\ e_2 \end{pmatrix} \omega^n n^2 \tag{3}$$

where c_i, d_i, e_i are rational-valued symbolic constants and ω are symbolic algebraic numbers. We then generate the clause set C_1 that ensures that we have a one-to-one correspondence between the number sequences described by the recurrence equations and the closed forms. For making sure that the equation $x - y^2 = 0$ is indeed a polynomial invariant, we plug the closed form templates (3) into the equation, and get

$$c_1\omega^n + d_1\omega^n n + e_1\omega^n n^2 - (c_2\omega^n + d_2\omega^n n + e_2\omega^n n^2)^2 = 0. \tag{4}$$

The above Eq. (4) has to hold for all $n \in \mathbb{N}$ as $x - y^2 = 0$ should be a loop invariant. That is, we want to find $c_1, c_2, d_1, d_2, e_1, e_2$ and ω such that (4) holds for all $n \in \mathbb{N}$. The properties of C-finite number sequences allow us to reduce this $\exists\forall$ problem containing exponential expressions into a finite set of polynomials

$$\begin{aligned} C_2 = \{&c_1\omega - c_2^2\omega^2 = 0, d_1\omega - 2c_2d_2\omega^2 = 0, \\ &e_1\omega - (2c_2e_2 - d_2^2)\omega^2 = 0, 2d_2e_2\omega^2 = 0, e_2^2\omega^2 = 0\} \end{aligned}$$

In summary, we get a PCP consisting of clause sets C_1 and C_2 containing 27 polynomial constraints over the unknowns $a_i, b'_{ij}, c_i, d_i, e_i, \omega$ from (1)–(3). The solution space of our PCP captures the set of all C-finite recurrence systems of the form (2) such that $x(n) - 2y(n)^2 = 0$ holds for all $n \geq 0$. That is, any solution of our PCP yields a loop with an invariant $x = y^2$.

Figures 3(b)–(c) illustrate two solutions of the PCP problem of our example: each program of Fig. 3(b)–(c) is an instance of (1), has $x - 2y^2 = 0$ as its invariant and can be synthesised using our work. The loop of Fig. 3(b), and thus of Fig. 2b, is synthesised by considering the size s of (1) to be 2, whereas Fig. 3(c) is computed by increasing the size s of (1) to 3.

Automation and Implementation. We implemented our approach to loop synthesis in the new open-source Julia package Absynth, available at

https://github.com/ahumenberger/Absynth.jl.

Our experiments using academic benchmarks on loop analysis as well as on generating number sequences in algorithmic combinatorics are available in [19,23].

5 Conclusions

We overviewed algebra-based algorithms for loop invariant synthesis and loop synthesis. The key ingredient of our work comes by modeling loops as algebraic recurrences, in particular by C-finite recurrences. To this end, we consider non-deterministic loops whose loop updates induce C-finite number sequences among loop variables. In the case of loop invariant synthesis, our work generates the polynomial ideal of all polynomial invariants of such loops by using symbolic summation in combination with properties of polynomial ideals. Extending this approach to (multi-path) loops inducing more complex recurrence equations supporting for example arbitrary multiplications among (some of the) variables is an interesting line for future work. When synthesising loops from polynomial invariants, we use symbolic summation to generate polynomial constraints whose solutions yield loops that exhibit the given invariant. Solving our constraint system requires satisfiability solving in non-linear arithmetic, opening up new directions for SMT-based reasoning with polynomial constraints. For example, we believe searching for solutions over finite domains would improve the scalability of our loop synthesis method. Extending our loop synthesis task to generate loops that are optimal with respect to a user-specified measure is another challenge to further investigate. To this end, understanding and efficiently encoding the best optimisation measures into our approach is an interesting line for future work.

Acknowledgment. We thank Maximillian Jaroschek (TU Wien) for joint work allowing to extend our invariant generation approaches to more complex loops and number sequences. We also thank Sumit Gulwani (Microsoft) and Manuel Kauers (JKU Linz) for valuable discussions on ideas leading to our loop synthesis framework. Practical aspects of our loop synthesis approach and using loop synthesis for strength reduction involve joint work with Nikolaj Bjørner (Microsoft).

We acknowledge funding from the ERC Starting Grant 2014 SYMCAR 639270, the ERC Proof of Concept Grant SYMELS 842066, the Wallenberg Academy Fellowship TheProSE, and the Austrian FWF research project W1255-N23.

References

1. Albarghouthi, A., Gulwani, S., Kincaid, Z.: Recursive program synthesis. In: CAV, pp. 934–950 (2013)
2. Alur, R., et al.: Syntax-guided synthesis. In: Dependable Software Systems Engineering, vol. 40, pp. 1–25. IOS Press (2015)
3. Ball, T., Levin, V., Rajamani, S.K.: A decade of software model checking with SLAM. Commun. ACM **54**(7), 68–76 (2011)
4. 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)
5. Cachera, D., Jensen, T.P., Jobin, A., Kirchner, F.: Inference of polynomial invariants for imperative programs: a farewell to Gröbner bases. In: SAS, pp. 58–74 (2012)
6. Clarke, E.M., Allen Emerson, E.: Design and synthesis of synchronization skeletons using branching-time temporal logic. In: Logics of Programs, pp. 52–71 (1981)
7. Cook, B.: Formal reasoning about the security of amazon web services. In: CAV, pp. 38–47 (2018)
8. Cooper, K.D., Taylor Simpson, L., Vick, C.A.: Operator strength reduction. ACM Trans. Program. Lang. Syst. **23**(5), 603–625 (2001)
9. 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)
10. De Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: TACAS, pp. 337–340 (2008)
11. de Oliveira, S., Bensalem, S., Prevosto, V.: Polynomial invariants by linear algebra. In: ATVA, pp. 479–494 (2016)
12. Farzan, A., Kincaid, Z.: Compositional recurrence analysis. In: FMCAD, pp. 57–64 (2015)
13. Feng, Y., Martins, R., Bastani, O., Dillig, I.: Program synthesis using conflict-driven learning. In: PLDI, pp. 420–435 (2018)
14. German, S.M., Wegbreit, B.: A synthesizer of inductive assertions. IEEE Trans. Software Eng. **1**(1), 68–75 (1975)
15. Gulwani, S.: Automating string processing in spreadsheets using input-output examples. In: POPL, pp. 317–330 (2011)
16. Gulwani, S.: Programming by examples: applications, algorithms, and ambiguity resolution. In: IJCAR, pp. 9–14 (2016)
17. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM **12**(10), 576–580 (1969)
18. Hrushovski, E., Ouaknine, J., Pouly, A., Worrell, J.: On strongest algebraic program invariants. J. ACM., to appear
19. Humenberger, A.: Algebra-based loop reasoning. Ph.D. thesis, TU Wien 2021
20. Humenberger, A., Jaroschek, M., Kovács, L.: Automated generation of non-linear loop invariants utilizing hypergeometric sequences. In: ISSAC, pp. 221–228 (2017)
21. Humenberger, A., Jaroschek, M., Kovács, L.: Aligator.jl - a Julia package for loop invariant generation. In: CICM, pp. 111–117 (2018)

22. Humenberger, A., Jaroschek, M., Kovács, L.:: Invariant generation for multi-path loops with polynomial assignments. In: VMCAI, pp. 226–246 (2018)
23. Humenberger, A., Kovács, L., Bjørner, N.: Algebra-based loop synthesis. In: iFM (2020, to appear)
24. Jha, S., Gulwani, S., Seshia, S.A., Tiwari, A.: Oracle-guided component-based program synthesis. In: ICSE, pp. 215–224 (2010)
25. Kalyan, A., Mohta, A., Polozov, O., Batra, D., Jain, P., Gulwani, S.: Neural-guided deductive search for real-time program synthesis from examples. In: ICLR (2018)
26. Karr, M.: Affine relationships among variables of a program. Acta Informatica **6**, 133–151 (1976)
27. Kauers, M., Paule, P.: The Concrete Tetrahedron - Symbolic Sums, Recurrence Equations, Generating Functions, Asymptotic Estimates. Texts & Monographs in Symbolic Computation. Springer, Vienna (2011). https://doi.org/10.1007/978-3-7091-0445-3
28. Kincaid, Z., Breck, J., Boroujeni, A.F., Reps, T.W.: Compositional recurrence analysis revisited. In: PLDI, pp. 248–262 (2017)
29. Kincaid, Z., Breck, J., Cyphert, J., Reps, T.W.: Closed forms for numerical loops. PACMPL **3**(POPL), 55:1–55:29 (2019)
30. Kincaid, Z., Cyphert, J., Breck, J., Reps, T.W.: Non-linear reasoning for invariant synthesis. PACMPL **2**(POPL), 541–5433 (2018)
31. Kovács, L.: Reasoning algebraically about p-solvable loops. In: TACAS, pp. 249–264 (2008)
32. Kovács, L., Voronkov, A.: First-order theorem proving and vampire. In: CAV, pp. 1–35 (2013)
33. Kuncak, V., Mayer, M., Piskac, R., Suter, P.: Software synthesis procedures. Commun. ACM **55**(2), 103–111 (2012)
34. Manna, Z., Waldinger, R.J.: A deductive approach to program synthesis. ACM Trans. Program. Lang. Syst. **2**(1), 90–121 (1980)
35. Müller-Olm, M., Seidl, H.: A note on Karr's algorithm. In: ICALP, pp. 1016–1028 (2004)
36. Müller-Olm, M., Seidl, H.: Computing polynomial program invariants. Inf. Process. Lett. **91**(5), 233–244 (2004)
37. Nye, M., Hewitt, L., Tenenbaum, J., Solar-Lezama, A.: Learning to infer program sketches. In: ICML, pp. 4861–4870 (2019)
38. Rodríguez-Carbonell, E., Kapur, D.: Automatic generation of polynomial invariants of bounded degree using abstract interpretation. Sci. Comput. Program. **64**(1), 54–75 (2007)
39. Rodríguez-Carbonell, E., Kapur, D.: Generating all polynomial invariants in simple loops. J. Symb. Comput. **42**(4), 443–476 (2007)
40. Sankaranarayanan, S., Sipma, H., Manna, Z.: Non-linear loop invariant generation using Gröbner bases. In: POPL, pp. 318–329 (2004)
41. Sharma, R., Gupta, S., Hariharan, B., Aiken, A., Liang, P., Nori, A.V.: A data driven approach for algebraic loop invariants. In: Felleisen, M., Gardner, P. (eds.) ESOP 2013. LNCS, vol. 7792, pp. 574–592. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-37036-6_31
42. Solar-Lezama, A.: The Sketching approach to program synthesis. In: APLAS, pp. 4–13 (2009)
43. Srivastava, S., Gulwani, S., Foster, J.S.: From program verification to program synthesis. In: POPL, pp. 313–326 (2010)