Aligator.jl – A Julia Package for Loop Invariant Generation

Andreas Humenberger[1(✉)], Maximilian Jaroschek[1,2], and Laura Kovács[1,3]

[1] TU Wien, Vienna, Austria
ahumenbe@forsyte.at
[2] JKU Linz, Linz, Austria
[3] Chalmers, Gothenburg, Sweden

Abstract. We describe the `Aligator.jl` software package for automatically generating all polynomial invariants of the rich class of extended P-solvable loops with nested conditionals. `Aligator.jl` is written in the programming language Julia and is open-source. `Aligator.jl` transforms program loops into a system of algebraic recurrences and implements techniques from symbolic computation to solve recurrences, derive closed form solutions of loop variables and infer the ideal of polynomial invariants by variable elimination based on Gröbner basis computation.

1 Introduction

In [2] we described an automated approach for generating loop invariants as a conjunction of polynomial equalities for a family of loops, called extended P-solvable loops. For doing so, we abstract loops to a system of algebraic recurrences over the loop counter and program variables and compute polynomial equalities among loop variables from the closed form solutions of the recurrences.

Why Julia? Our work was previously implemented in the `Aligator` software package [4], within the Mathematica system [8]. While Mathematica provides high-speed implementations of symbolic computation techniques, it is a proprietary software which is an obstacle for using `Aligator` in applications of invariant generation. The fact that Mathematica provides no possibility to parse and modify program code was also a reason to move to another environment. To make `Aligator` better suited for program analysis, we decided to redesign `Aligator` in the Julia programming language [3]. Julia provides a simple and efficient interface for calling C/C++ and Python code. This allows us to resort to already existing computer algebra libraries, such as Singular [1] and SymPy [5]. Julia also provides a built-in package manager that eases the use of other packages and enables

All authors are supported by the ERC Starting Grant 2014 SYMCAR 639270. We also acknowledge funding from the Wallenberg Academy Fellowship 2014 TheProSE, the Swedish VR grant GenPro D0497701, and the Austrian FWF research projects RiSE S11409-N23 and W1255-N23. Maximilian Jaroschek is also supported by the FWF project Y464-N18.

F. Rabe et al. (Eds.): CICM 2018, LNAI 11006, pp. 111–117, 2018.
https://doi.org/10.1007/978-3-319-96812-4_10

others to use Julia packages, including our `Aligator.jl` tool. Before committing to Julia, we also considered the computer algebra system SageMath [7] and an implementation directly in C/C++ as options for redesigning `Aligator`. The former hosts its own Python version which makes the installation of other Python packages (e.g. for parsing source code) tedious and error-prone. While C/C++ is very efficient and provides a large ecosystem on existing libraries, developing C/C++ projects requires more effort than Julia packages. We therefore believe that Julia provides the perfect mix between efficiency, extensibility and convenience in terms of programming and symbolic computations.

`Aligator.jl`. This paper overviews `Aligator.jl` and details its main components. The code of `Aligator.jl` is available open-source at:

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

All together, `Aligator.jl` consists of about 1250 lines of Julia code. We evaluated `Aligator.jl` on challenging benchmarks on invariant generation. Our experimental results are available at the above mentioned link and demonstrate the efficiency of `Aligator.jl`.

Contributions. Our new tool `Aligator.jl` significantly extends and improves the existing software package `Aligator` as follows:

- Unlike `Aligator`, `Aligator.jl` is open-source and easy to integrate into other software packages.
- `Aligator.jl` implements symbolic computation techniques directly in Julia for extracting and solving recurrences and generates polynomial dependencies among exponential sequences.
- Contrarily to `Aligator`, `Aligator.jl` handles not only linear recurrences with constant coefficients, called C-finite recurrences. Rather, `Aligator.jl` also supports hypergeometric sequences and sums and term-wise products of C-finite and hypergeometric recurrences [2].
- `Aligator.jl` is complete. That is, a finite basis of the polynomial invariant ideal is always computed.

2 Background and Notation

`Aligator.jl` computes polynomial invariants of so-called extended P-solvable loops [2]. Loop guards and test conditions are ignored in such loops and denoted by ... or `true`, yielding non-deterministic loops with sequencing and conditionals. Program variables $V = \{v_1, \ldots, v_m\}$ of extended P-solvable loops have numeric values, abstracted to be rational numbers. The assignments of extended P-solvable loops are of the form $v_i := \sum_{j=0}^{m} c_i v_j + c_{m+1}$ with constants $c_0, \ldots, c_{m+1}$, or $v_i := r(n)v_i$, where $r(n)$ is a rational function in the loop counter n. We give an example of an extended P-solvable loops in Fig. 1.

In correspondence to V, the initial values of the variables are given by the set $V_0 := \{v_1(0), \dots, v_m(0)\}$; that is, $v_i(0)$ is the initial value of v_i. In what follows, we consider V and V_0 fixed and state all definitions relative to them. Given an extended P-solvable loop as input, `Aligator.jl` generates all its polynomial equality invariants. By a polynomial equality invariant, in the sequel simply polynomial invariant, we mean the equality:

$$p(v_1, \dots, v_m, v_1(0), \dots, v_m(0)) = 0, \tag{1}$$

where p is a polynomial in $V \cup V_0$ with rational number coefficients. In what follows, we also refer to the polynomial p in (1) as a polynomial invariant. For $n \in \mathbb{N} \setminus \{0\}$ and a loop variable v_i, we write $v_i(n)$ to denote the value of v_i after the nth loop iteration. As (1) is a loop invariant, we have:

$$p(v_1(n), \dots, v_m(n), v_1(0), \dots, v_m(0)) = 0 \text{ for } n > 0.$$

As shown in [2,6], the set of polynomial invariants in V, w.r.t. the initial values V_0, forms a polynomial ideal, called the polynomial invariant ideal. Given an extended P-solvable loop, `Aligator.jl` computes *all* its polynomial invariants as it computes a basis of the polynomial invariant ideal, a finite set of polynomials $\{b_1, \dots, b_k\}$. Any polynomial invariant can be written as a linear combination $p_1 b_1 + \cdots + p_k b_k$ for some polynomials $p_1, \dots, p_k$.

while ... **do**
  if ... **then**
    $r := r - v;\ v := v + 2$
  else
    $r := r + u;\ u := u + 2$
  end if
end while

Fig. 1. An extended P-solvable loop.

3 System Description of `Aligator.jl`

Inputs to `Aligator.jl` are extended P-solvable loops and are fed to `Aligator.jl` as `String` in the Julia syntax. We illustrate the use of `Aligator.jl` on our example from Fig. 1:

Example 1. Fig. 1 is specified as a Julia string as follows:

```
julia> loopstr = """
           while true
               if true
                   r = r - v; v = v + 2
               else
                   r = r + u; u = u + 2
               end
           end
       """
```

Polynomial loop invariants are inferred using `Aligator.jl` by calling the function `aligator(str::String)` with a string input containing the loop as its argument.

```
julia> aligator(loopstr)
Singular Ideal over Singular Polynomial Ring (QQ),(r_0,v_0,u_0,r,v,u)
  with generators (v_0^2-u_0^2-v^2+u^2+4*r_0-2*v_0+2*u_0-4*r+2*v-2*u)
```

The result of `Aligator.jl` is a Gröbner basis of the polynomial invariant ideal. It is represented as an object of type `Singular.sideal` that is defined in the Singular package. For Fig. 1, `Aligator.jl` reports that the polynomial invariant ideal is generated by the polynomial invariant $\{v_0^2 - u_0^2 - v^2 + u^2 + 4r_0 - 2v_0 + 2u_0 - 4r + 2v - 2u = 0\}$ in variables r_0, v_0, u_0, r, v, u, where r_0, v_0, u_0 denote respectively the initial values of r, v, u.

We now overview the main parts of `Aligator.jl`: (i) extraction of recurrence equations, (ii) recurrence solving and (iii) computing the polynomial invariant ideal.

Extraction of Recurrences. Given an extended P-solvable loop as a Julia string, `Aligator.jl` creates the abstract syntax tree of this loop. This tree is then traversed in order to extract loop paths (in case of a multi-path loop) and the corresponding loop assignments. The resulting structure is then flattened in order to get a loop with just one layer of nested loops. Within `Aligator.jl` this is obtained via the method `extract_loop(str::String)`. As a result, the extracted recurrences are represented in `Aligator` by an object of type `Aligator.MultiLoop`, in case the input is a multi-path loop; otherwise, the returned object is of type `Aligator.SingleLoop`.

Example 2. Using Example 1, `Aligator.jl` derives the loop and its corresponding systems of recurrences:

```
julia> loop = extract_loop(loopstr)
2-element Aligator.MultiLoop:
 [r(n1+1) = r(n1) - v(n1), v(n1+1) = v(n1) + 2, u(n1+1) = u(n1)]
 [r(n2+1) = r(n2) + u(n2), u(n2+1) = u(n2) + 2, v(n2+1) = v(n2)]
```

As loop paths are translated into single-path loops, `Aligator.jl` introduces a loop counter for each path and computes the recurrence equations of the loop variables r, v, u with respect to the loop counters n_1 and n_2.

Recurrence Solving. For each single-path loop, its system of recurrences is solved. `Aligator.jl` performs various simplifications on the extracted recurrences, for example by eliminating cyclic dependencies introduced by auxiliary variables and uncoupling mutually dependent recurrences. The resulting, simplified recurrences represent sums and term-wise products of C-finite or hypergeometric sequences. `Aligator.jl` computes closed forms solutions of such recurrences by calling the method `closed_forms` and using the symbolic manipulation capabilities of `SymPy.jl`:

Example 3. For Example 2, we get the following systems of closed forms:

```
julia> cforms = closed_forms(loop)
2-element Array{Aligator.ClosedFormSystem,1}:
 [v(n1) = 2*n1+v(0), u(n1) = u(0), r(n1) = -n1^2-n1*(v(0)-1)+r(0)]
 [u(n2) = 2*n2+u(0), v(n2) = v(0), r(n2) = n2^2+n2*(u(0)-1)+r(0)]
```

The returned value is an array of type `Aligator.ClosedFormSystem`.

Invariant Ideal Computation. Using the closed form solutions for (each) single-path loop, `Aligator.jl` next derives a basis of the polynomial invariant ideal of the (multi-path) extended P-solvable loop. To this end, `Aligator.jl` uses the `Singular.jl` package for Gröbner basis computations in order to eliminate variables in the loop counter(s) from the system of closed forms. For multi-path loops, `Aligator.jl` relies on iterative Gröbner basis computations until a fixed point is derived representing a Gröbner basis of the polynomial invariant ideal – see [2] for theoretical details.

Computing polynomial invariants within `Aligator.jl` is performed by the function `invariants(cforms::Array{ClosedFormSystem,1})`. The result is an object of type `Singular.sideal` and represents a Gröbner basis of the polynomial invariant ideal in the loop variables.

Example 4. For Example 3, `Aligator.jl` generates the following Gröbner basis, as already described on page 4:

```
julia> ideal = invariants(cforms)
Singular Ideal over Singular Polynomial Ring (QQ),(r_0,v_0,u_0,r,v,u)
  with generators (v_0^2-u_0^2-v^2+u^2+4*r_0-2*v_0+2*u_0-4*r+2*v-2*u)
```

4 Experimental Evaluation

Our approach to invariant generation was shown to outperform state-of-the-art tools on invariant generation for multi-path loops with polynomial arithmetic [2]. In this section we focus on the performance of our new implementation in `Aligator.jl` and compare results to `Aligator` [4]. In our experiments, we used benchmarks from [2]. Our experiments were performed on a machine with a 2.9 GHz Intel Core i5 and 16 GB LPDDR3 RAM. When using `Aligator.jl`, the invariant ideal computed by `Aligator.jl` was non-empty for each example; that is, for each example we were able to find non-trivial invariants.

Tables 1(a) and (b) show the results for a set of single- and multi-path loops respectively. In both tables the first column shows the name of the instance, whereas columns two and three depict the running times (in seconds) of `Aligator` and `Aligator.jl`, respectively.

By design, `Aligator.jl` is at least as strong as `Aligator` concerning the quality of the output. When it comes to efficiency though, we note that `Aligator.jl` is slower than `Aligator`. We expected this result as `Aligator` uses the highly optimized algorithms of Mathematica. When taking a closer look at how much time is spent in the different parts of `Aligator.jl`, we observed that the most time in `Aligator.jl` is consumed by symbolic manipulations. Experiments indicate that

Table 1. Experimental evaluation of `Aligator.jl`.

(a)

Single-path	`Aligator`	`Aligator.jl`
`cohencu`	0.072	2.879
`freire1`	0.016	1.159
`freire2`	0.062	2.540
`petter1`	0.015	0.876
`petter2`	0.026	1.500
`petter3`	0.035	2.080
`petter4`	0.042	3.620

(b)

Multi-path	`Aligator`	`Aligator.jl`
`divbin`	0.134	1.760
`euclidex`	0.433	3.272
`fermat`	0.045	2.159
`knuth`	55.791	12.661
`lcm`	0.051	2.089
`mannadiv`	0.022	1.251
`wensley`	0.124	1.969

we can improve the performance of `Aligator.jl` considerably by using the Julia package `SymEngine.jl` instead of `SymPy.jl`. We believe that our initial experiments with `Aligator.jl` are promising and demonstrate the use of our efforts in making our invariant generation open-source.

5 Conclusion

We introduced the new package `Aligator.jl` for loop invariant generation in the programming language Julia. Our `Aligator.jl` tool is an open-source software package for invariant generation using symbolic computation and can easily be integrated with other libraries and tools.

References

1. Decker, W., Greuel, G.M., Pfister, G., Schönemann, H.: SINGULAR 4-1-0—a computer algebra system for polynomial computations (2016). http://www.singular.uni-kl.de
2. Humenberger, A., Jaroschek, M., Kovács, L.: Invariant generation for multi-path loops with polynomial assignments. In: Dillig, I., Palsberg, J. (eds.) VMCAI. LNCS, vol. 10747, pp. 226–246. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-73721-8_11
3. Julia. https://julialang.org/
4. Kovács, L.: Aligator: a mathematica package for invariant generation (system description). In: Armando, A., Baumgartner, P., Dowek, G. (eds.) IJCAR 2008. LNCS (LNAI), vol. 5195, pp. 275–282. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-71070-7_22
5. Meurer, A., Smith, C.P., Paprocki, M., Čertík, O., Kirpichev, S.B., Rocklin, M., Kumar, A., Ivanov, S., Moore, J.K., Singh, S., Rathnayake, T., Vig, S., Granger, B.E., Muller, R.P., Bonazzi, F., Gupta, H., Vats, S., Johansson, F., Pedregosa, F., Curry, M.J., Terrel, A.R., Roučka, S., Saboo, A., Fernando, I., Kulal, S., Cimrman, R., Scopatz, A.: SymPy: symbolic computing in Python. PeerJ Comput. Sci. **3**, e103 (2017). https://doi.org/10.7717/peerj-cs.103

6. Rodríguez-Carbonell, E., Kapur, D.: Generating all polynomial invariants in simple loops. J. Symb. Comput. **42**(4), 443–476 (2007). https://doi.org/10.1016/j.jsc.2007.01.002
7. SageMath. http://www.sagemath.org/
8. Wolfram, S.: An Elementary Introduction to the Wolfram Language. Wolfram Media Inc. (2017). https://www.wolfram.com/language/elementary-introduction/2nd-ed/