Aligator.jl – A Julia Package for Loop Invariant Generation *

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

[1] TU Wien
[2] JKU Linz
[3] Chalmers

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. We describe the main components and functionalities of `Aligator.jl` and report on some experimental results.

1 Introduction

Automating the analysis and verification of programs with loops comes with the burden of automatically inferring loop invariants. In [3,4] 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. Such loops contain conditionals and sequencing, with test conditions ignored. In a nutshell, our method works as follows. (i) The assignments of extended P-solvable loops are translated into a system of algebraic recurrences over the loop counter and program variables. (ii) Closed form solutions of these recurrence equations are computed as functions of the loop counter, by using symbolic computation algorithms for holonomic sequences [6]. (iii) Using Gröbner basis computation, algebraic relations among the closed form solutions are computed and the loop counter is eliminated, yielding a finite set of polynomial equalities among the program variables. These polynomial equalities hold at every loop iteration and are thus loop invariants. Moreover, they describe the set of all polynomial invariants as they generate the ideal of polynomial invariants of the given extended P-solvable loop.

* 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.

Our work on invariant generation was initially implemented in the `Aligator` software package [8], within the Mathematica system [12]. Mathematica provides high-speed implementations of symbolic computation techniques and was therefore a perfect choice for a proof-of-concept implementation of `Aligator`. However, there are a number of disadvantages when using `Aligator`. First, as Mathematica is a proprietary software, `Aligator` is not open source; this prevents the integration and use of `Aligator` in open-source verification frameworks. Second, `Aligator` does not provide any capabilities for directly processing source code which should be supported when analyzing programs and inferring invariants.

To make `Aligator` better suited for program analysis and invariant generation, we decided to redesign `Aligator` in the Julia programming language [5]. We believe Julia provides the perfect mix between efficiency, extensibility and convenience in terms of programming and symbolic computations. The extensibility of Julia is given by its 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 [9]. Julia also provides a built-in package manager that eases the use of other packages and enables others to use Julia packages, including our `Aligator.jl` tool.

The purpose of this paper is to overview `Aligator.jl` and detail 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.

Contributions. We present the new tool `Aligator.jl` for analyzing program loops and inferring their invariants. `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` is implemented in Julia, replacing the Mathematica framework of `Aligator`.
- `Aligator.jl` automatically extracts loops from Julia source code, replacing thus the existing burden of `Aligator` to manually analyze and translate source code programs into a custom Mathematica input. `Aligator.jl` also extends `Aligator` by handling a richer class of loops where multiplication with the loop counter is allowed.
- `Aligator.jl` implements symbolic computation techniques for extracting and solving recurrences over the loop counter and program variables and generates polynomial dependencies among exponential sequences. All these methods are implemented directly in Julia, without relying on external computer algebra systems. 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 termwise products of C-finite and hypergeometric recurrences [3].
- `Aligator.jl` performs variable elimination using Gröbner basis computation by relying on the computer algebra system Singular. Unlike `Aligator`,

`Aligator.jl` is complete. That is, a finite basis of the polynomial invariant ideal is always computed.

We experimentally evaluate `Aligator.jl` in Section 4.

2 Background and Notation

`Aligator.jl` computes polynomial invariants of so-called extended P-solvable loops. 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 refer to [4] for a precise definition of extended P-solvable loops and give an example of such a loop in Figure 1.

In correspondence to V, the initial values of the variables are given by the set $V_0 := \{v_1(0), \ldots, v_m(0)\}$; 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` will generate all its polynomial equality invariants. By a polynomial equality invariant, in the sequel simply polynomial invariant, we mean the equality:

```
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.

$$p(v_1, \ldots, v_m, v_1(0), \ldots, v_m(0)) = 0, \quad (1)$$

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:

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

As shown in [11,3], 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 $\{b_1, \ldots, b_k\}$. Any polynomial invariant can be written as a linear combination $p_1 b_1 + \cdots + p_k b_k$ for some polynomials $p_1, \ldots, p_k$ and it can be algorithmically tested whether a given polynomial is an element of the invariant ideal.

3 System Description of `Aligator.jl`

In this section, we give installation instructions, showcase the use of `Aligator.jl` and describe the main components of our tool.

3.1 Installation

Using `Aligator.jl` requires installing Julia. Further, one also needs to install the packages `Cxx`, `Nemo` and `Singular`; this can be done by simply executing the following commands in the built-in Julia command line:

```
julia> Pkg.add("Cxx")
julia> Pkg.add("Nemo")
julia> Pkg.checkout("Nemo")
julia> Pkg.clone("https://github.com/oscar-system/Singular.jl")
julia> Pkg.build("Singular")
```

Our `Aligator.jl` tool can now be installed by executing the command:

```
julia> Pkg.clone("https://github.com/ahumenberger/Aligator.jl")
```

`Aligator.jl` makes use of the following Julia packages.

`SymPy.jl`	for symbolic manipulations;
`Nemo.jl`	for number theory;
`Singular.jl`	for Gröbner basis computation.

The three packages above provide wrappers for existing libraries written in Python and C/C++. The first package provides a wrapper for the Python library SymPy. `Nemo.jl` is a wrapper of various libraries from number theory [2], whereas `Singular.jl` wraps the core functionalities of the computer algebra system Singular.

3.2 A Quick Start

With `Aligator.jl` installed, we can compute our first invariants. To this end, the module `Aligator` has to be loaded via the command `using`.

```
julia> using Aligator
```

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 a simple loop below:

```
julia> loop = """
           while true
             x = 2x
             y = 1/2y
           end
       """
```

Given a loop as input, 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(loop)
Singular Ideal over Singular Polynomial Ring (QQ),(x_0,y_0,x,y)
  with generators (x_0*y_0-x*y)
```

The result of `Aligator.jl` is a basis of the polynomial invariant ideal. It is represented as an object of type `Singular.sideal` that is defined in the Singular package. In our example, `Aligator.jl` reports that the polynomial invariant ideal is generated by the polynomial invariant $\{x_0 \cdot y_0 - x \cdot y = 0\}$ in variables x_0, y_0, x, y, where x_0 and y_0 represent the initial values of x and y, respectively.

3.3 Architecture

We now overview the main parts of `Aligator.jl`: (i) extraction of recurrence equations, (ii) recurrence solving and (iii) computing the polynomial invariant ideal. We do so by showcasing `Aligator.jl` on the loop from Figure 1:

Example 1. Figure 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
       """
```

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. That is, in the case of a multi-path loop, `Aligator.jl` translates the loop into a sequence of single-path loops, each single-path loop corresponding to one path of the multi-path loop – see [4] for details. 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 the loop `loopstr` from 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. For solving such recurrences, `Aligator.jl` implements C-finite recurrence solving methods [6] and Petkovšek's algorithm [10] for handling hypergeometric sequences. We implemented these recurrence solving techniques directly in Julia using the symbolic manipulation capabilities of `SymPy.jl`. As a result, `Aligator.jl` computes closed forms solutions of recurrences by calling the method `closed_forms`.

Example 3. Continuing with the `loop` from 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. We note however that the closed forms computed in the previous step might contain exponential sequences in the loop counter. For example, the recurrence equation $x(n+1) = 2 \cdot x(n)$ corresponding to the loop assignment $x = 2 \cdot x$ from Section 3.2 yields the closed form solution $x(n) = 2^n \cdot x(0)$. For handling exponential sequences, `Aligator.jl` implements Ge's algorithm [7] for computing the ideal of algebraic dependencies among the exponential sequences and uses the Julia package `Nemo.jl` to access various number theoretical algorithms. The basis of this ideal is further used to eliminate variables in the loop counter(s) from the closed form solutions. To this end, `Aligator.jl` uses Gröbner basis computations 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. We note that `Aligator.jl` always terminates and refer to [4] for theoretical details.

Internally, `Aligator.jl` uses the `Singular.jl` package for computing Gröbner bases. 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 in the loop variables. This Gröbner basis generates the polynomial invariant ideal of the loop whose closed form solutions are expressed by `cforms`.

Example 4. For the closed form system of Example 3, `Aligator.jl` generates the following Gröbner basis:

```
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)
```

`Aligator.jl` reports that the polynomial invariant ideal of Figure 1 is generated by the single polynomial loop invariant

$$v_0^2 - u_0^2 - v^2 + u^2 + 4r_0 - 2v_0 + 2u_0 - 4r + 2v - 2u = 0$$

where r_0, v_0, u_0 denote the initial values of r, v, 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 [4]. In this section we focus on the performance of our new implementation in `Aligator.jl` and compare results to `Aligator` [8].

In our experiments, we used benchmarks from [4]. Our experiments were performed on a machine with a 2.9 GHz Intel Core i5 and 16 GB LPDDR3 RAM; for each example, a timeout of 300 seconds was set. 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 (1a) and (1b) 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.

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

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 its recurrence solving step, due to its initial, not optimized implementation with unnecessary external function calls to the

Python library SymPy. We are confident that recurrence solving in `Aligator.jl` can be improved, bringing a significant performance speed-up. 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. Our experiments with `Aligator.jl` are promising and we believe that the performance of `Aligator.jl` can further be improved by optimizing our algorithms for symbolic computation.

References

1. Decker, W., Greuel, G.M., Pfister, G., Schönemann, H.: SINGULAR 4-1-0 — A computer algebra system for polynomial computations. `http://www.singular.uni-kl.de` (2016)
2. Fieker, C., Hart, W., Hofmann, T., Johansson, F.: Nemo/Hecke: Computer Algebra and Number Theory Packages for the Julia Programming Language. In: ISSAC. pp. 157–164. ACM (2017)
3. Humenberger, A., Jaroschek, M., Kovács, L.: Automated Generation of Non-Linear Loop Invariants Utilizing Hypergeometric Sequences. In: ISSAC. pp. 221–228. ACM (2017)
4. Humenberger, A., Jaroschek, M., Kovács, L.: Invariant Generation for Multi-Path Loops with Polynomial Assignments. In: VMCAI. Lecture Notes in Computer Science, vol. 10747, pp. 226–246. Springer (2018)
5. Julia: https://julialang.org/
6. Kauers, M., Paule, P.: The Concrete Tetrahedron. Text and Monographs in Symbolic Computation, Springer Wien, 1st edn. (2011)
7. Kauers, M.: Algorithms for Nonlinear Higher Order Difference Equations. Ph.D. thesis, RISC-Linz (October 2005)
8. Kovács, L.: Aligator: A Mathematica Package for Invariant Generation (System Description). In: IJCAR. Lecture Notes in Computer Science, vol. 5195, pp. 275–282. Springer (2008)
9. 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, v., Saboo, A., Fernando, I., Kulal, S., Cimrman, R., Scopatz, A.: SymPy: symbolic computing in Python. PeerJ Computer Science 3, e103 (Jan 2017)
10. Petkovšek, M.: Hypergeometric solutions of linear recurrences with polynomial coefficients. Journal of Symbolic Computation 14(2–3), 243 – 264 (1992)
11. Rodríguez-Carbonell, E., Kapur, D.: Generating all polynomial invariants in simple loops. Journal of Symbolic Computation 42(4), 443 – 476 (2007)
12. Wolfram, S.: An Elementary Introduction to the Wolfram Language. Wolfram Media Inc. (2017)