Towards an Automated Reasoning Tool for Complexity Analysis of Automated Reasoners

Louis Rustenholz
Universidad Politécnica de Madrid (UPM), Spain
IMDEA Software Institute, Spain

Manuel V. Hermenegildo
Universidad Politécnica de Madrid (UPM), Spain
IMDEA Software Institute, Spain

Pedro López-García
Spanish Council for Scientific Research (CSIC), Spain
IMDEA Software Institute, Spain

Alessio Mansutti
IMDEA Software Institute, Spain

Félix Ridoux[1]
Computer Science Laboratory of Sorbonne University (LIP6), France
STMicroelectronics, Crolles, France

Niki Vazou
IMDEA Software Institute, Spain

Abstract

We present the theory underpinning a complexity analysis tool (currently under development) that aims to automate tedious parts of the analysis of complex algorithms originating in the field of automated reasoning. Examples are given by super-exponential quantifier elimination procedures in real and integer arithmetic.

Our tool implements the following pipeline:

1. Together with the algorithm to be analysed, the user (an expert, e.g. the algorithm designer) can provide key metrics to track and lemmas to guide and improve the analysis. In pen-and-paper proofs, these correspond to the "non-tedious" and "creative" parts of the complexity analysis, that require human ingenuity.
2. The second step consists in the extraction of (generalised) recurrence equations. Here, we rely on a novel higher-order abstract interpretation technique, based on operator semantics. It enables (optimal) abstract compilation of symbolic programs into different kinds of purely numerical recursive representations, such as recurrence equations on interval-valued functions or numerical logic programs.
3. Finally, our tool solves the recurrence equations. We propose going beyond the direct use of computer algebra systems (CAS) by employing pre/postfixpoint-based techniques to discover and verify candidate bounds on the solutions. This approach, in turn, leverages recent progress in SMT solvers, and could benefit from techniques originating in termination-analysis research.

2012 ACM Subject Classification Theory of computation → Automated reasoning

Keywords and phrases Complexity Analysis, Abstract Interpretation, Recurrence Equations

Category Extended abstract

Acknowledgements Partially funded by MICIU projects CEX2024-001471-M *María de Maeztu*, by the European Union ERC GA 101039196 CRETE, and by the European Union MSCA GA 101154447 NEAT. We also thank the anonymous reviewers for their useful and constructive feedback.

[1] Work performed under previous IMDEA affiliation.

1 Introduction

In current practice, the complexity analysis of algorithms in computational logic is carried out almost exclusively by hand. Such analyses typically require pen-and-paper proofs spanning tens of pages, involving the derivation of recurrence equations capturing the evolution of relevant metrics during execution, and the proof of bounds on the solutions to these recurrences, often by induction. Moreover, such analysis must be repeated if the established bound is unsatisfactory, after updating the set of metrics or invariants and/or the algorithm itself. At scale, this workflow can become tedious and time-consuming.

▶ **Example.** *The history of quantifier-elimination procedures for Presburger arithmetic (PrA) offers an example of the above picture. Following Oppen's first triply-exponential-time upper bound [14], the complexity analysis was revisited and refined in a series of papers [15, 21, 22, 8], and (prompted by further algorithmic improvements) once more in recent work [2, 12]. All these works follow the blueprint described above, starting from the definition of a suitable set of metrics (e.g., the largest bit size of an integer occurring in the formula).*

The substantial existing bodies of work in automated cost analysis and symbolic computation, as well as recent advances, suggest that large parts of the above workflow are amenable to automation. We are currently developing a tool with precisely this goal.

In this extended abstract, we sketch the envisioned pipeline, and introduce some of the theory underlying the tool—including new techniques in recurrence-based static cost analysis.

Challenges. The algorithms we consider present interesting difficulties for automated analyses. Indeed, they require reasoning about highly non-linear, non-polynomial invariants, out of scope of classical numerical analyses, motivating recurrence-based approaches. They also involve broad classes of metrics and relations between them, while making it important to keep extracted recurrences as precise as possible; both touch on open problems in recurrence-based cost analysis. Finally, in their case, full automation is unrealistic, which encourages the development of interactive workflows between theorem provers and automated analysers.

Tool architecture. The pipeline of our tool, described in the abstract of page 1, is illustrated in Fig. 1. The following sections describe interesting features of each of its main components.

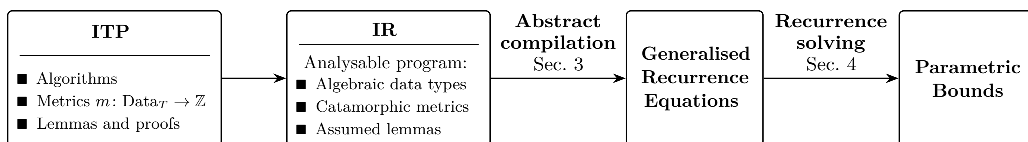

Figure 1 The pipeline of our tool.

Input programs. The first step of the tool consists in translating the algorithm to be analysed from an Interactive Theorem Prover (ITP) into an Intermediate Representation (IR) that is designed for analysis convenience. Besides the algorithm, the user provides *metrics*, and optionally *lemmas* directly in the ITP environment. The auxiliary lemmas are used to improve the analysis; they are *assumed true* (and their proof erased) in the IR.

At the time of writing this abstract, we are implementing this translation starting from `LiquidHaskell` [20] as a frontend. The IR is an imperative language featuring (first-order, monomorphic) recursive functions, ADTs, metric definitions, and non-deterministic primitives (`top`, `assume`) that reflect underspecified objects and user lemmas.

2 (Non-)catamorphic metrics and lemmas

Metrics. The central object of the first step of the pipeline is the notion of *(size) metric.*

▶ **Definition 1** (Size metric). *Given a datatype T, a* (size) metric *on T is any integer-valued function $m : T \to \mathbb{Z}$. For $t \in T$, we say that $m(t)$ is the* size of t (under the metric m).

The *recurrences* built in recurrence-based cost analysis relate sizes of program variables at different program points—e.g., for functions, relations between input and output sizes. Bounds on the *cost* of a function are then expressed as functions of the *sizes* of its inputs.[2]

Familiar examples include list length, integer value, or tree height: these arise both in elementary (manual) complexity analysis, and well-known automated complexity analyses. While Definition 1 is extremely general, precise analyses may require more sophisticated metrics than the examples above. For example, *potentials* in amortised complexity analysis [19] may be seen as instances of size metrics. One may also need to consider exotic metrics, such as the total number of elements in a data structure satisfying some logical property, the *cardinality* of elements of a particular shape, or number-theoretical properties of integers.

Compared to previous work, our contribution supports fully-automated recurrence extraction for a broader class of metrics, which we call *catamorphic* (collections of) *metrics.*

▶ **Definition 2** (Catamorphic collection of metrics). *Consider a collection $\mathcal{T} = (T_1, ..., T_n)$ of algebraic data types (with only base type $\mathbb{Z}$), possibly mutually recursive, given categorically as the initial algebra $in : F\mathcal{T} \overset{\simeq}{\rightrightarrows} \mathcal{T}$ of some endofunctor F in the n-fold (cartesian) product* $\mathtt{Types}^n$ *of the appropriate category of types. A collection $\mathcal{M}$ of metrics, composed of d_i metrics of type $T_i \to \mathbb{Z}$ on each T_i, may be viewed as a morphism $\mathcal{M} : \mathcal{T} \to \prod_i \mathbb{Z}^{d_i}$.*

We say that $\mathcal{M}$ is a catamorphic collection of metrics *whenever there exists a morphism $rm : F\left(\prod_i \mathbb{Z}^{d_i}\right) \to \prod_i \mathbb{Z}^{d_i}$ such that $\mathcal{M} \circ in = rm \circ F(\mathcal{M})$.*

Intuitively, $\mathcal{M}$ is catamorphic whenever the size of a term depends only on the sizes of the arguments of its constructor.

▶ **Example.** *Consider the type* `ListInt` *of list of integers, with corresponding functor $F : X \to 1 + \mathbb{Z} \times X$, where the two components of the coproduct respectively correspond to the* `Nil` *and* `Cons` *constructors. The number of distinct integers in a list,* `card`*, is a* non-*catamorphic metric. Indeed,* $\mathtt{card}(\mathtt{Cons}(\mathtt{hd}, \mathtt{tl}))$ *does not depend only of* $\mathtt{card}(\mathtt{tl})$ *and the value of* `hd`*, but also of the actual elements of* `tl`*.*

In contrast, the length *of a list is a trivial catamorphic metric, where $rm_{\mathtt{len},\mathtt{Nil}}() = 0$ and $rm_{\mathtt{len},\mathtt{Cons}}(_, n) = 1 + n$ for $n \in \mathbb{Z}$. More subtle catamorphic metrics include the number of elements satisfying some arbitrary property or the product of all elements.*

Lemmas. Our tool also supports user-defined *non-catamorphic metrics.* These metrics are automatically translated into *catamorphic overapproximations*, a step that causes the automated analysis to lose precision. In this case, we let the user intervene by providing lemmas in the ITP environment, which are used to recover lost precision.

▶ **Example.** *Consider a program function* $\mathtt{f} : \mathtt{ListInt} \to \mathtt{ListInt}$ *concatenating a list with itself. The tool easily infers that* $\mathtt{len}(\mathtt{f}(\ell)) = 2 \cdot \mathtt{len}(\ell)$*, but, working with catamorphisms alone, it can only infer that* $\mathtt{card}(\mathtt{f}(\ell)) \leq 2 \cdot \mathtt{card}(\ell)$*.*

Were the user to provide a lemma stating $\mathtt{card}(\mathtt{f}(\ell)) \leq \mathtt{card}(\ell)$*, the tool would exploit this information to sharpen the analysis of all procedures calling* `f`*.*

---

[2] We only discuss equations on *sizes* here; *costs* are similar after introducing *cost models* (e.g. [13]).

3 Recurrence extraction by abstract compilation

When extracting recurrences from programs, it is important to ensure that they produce solutions that are *sound*, and it is desirable that these are *precise*. Indeed, intuitively, several sets of relations may be inferred between sizes at different program points, but different sets may provide different information on the concrete behaviour of the program. Order theory and abstract interpretation [4] are convenient frameworks to study these notions.

Compared to previous work, we refine the notion of *soundness* of an equation with respect to the program it abstracts: our formalisation compares *operators* instead of their fixed points ("solutions"). This makes equations reflect the *control flow* of the original program. Moreover, via Galois connections, this provides a satisfying notion of "**best** equation that can be extracted from that program", under given signatures and sets of metrics, and a methodology to inductively overapproximate it.

3.1 Programs and recurrences as operators and their abstractions

We build upon [18]: (generalised) recurrence equations are represented by operators on the function space of their unknowns. Similarly, we introduce *operator semantics of programs*, which assigns to a program `P` an operator $[\![\mathtt{P}]\!]_\Phi$, where $\operatorname{lfp} [\![\mathtt{P}]\!]_\Phi$ is its usual concrete semantics.

▶ **Definition 3** (Operator semantics (sketch)). *Usual denotational semantics of a program* `P` *that recursively defines functions* $f \in \mathcal{F}$*, with non-deterministic primitives, can be given as* $\operatorname{lfp} [\![\mathtt{P}]\!]_\Phi$*, where* $[\![\mathtt{P}]\!]_\Phi \in \operatorname{End}\Big(\prod_{\mathcal{F} \ni f : \Pi_i T_i \to T_o} \left(\prod_i T_i \to \mathcal{P}(T_o)\right)\Big)$.

We take the operator $[\![\mathtt{P}]\!]_\Phi$ *as our concrete operator semantics (delaying the fixed point).*

$[\![\mathtt{P}]\!]_\Phi$ may be viewed as a semantic representation *of the program itself*, control-flow and recursive structure included, with only syntactic features hidden away. Because of this, Galois connections in operator space may be viewed as *abstractions of program themselves*.

▶ **Proposition 4** ([17]). *Let* $\mathcal{M}_{T_i}$ *be (any) collection of metrics on each* T_i*. There is a "size abstraction" Galois connection from concrete programs (viewed as operators* $[\![\mathtt{P}]\!]_\Phi$ *on symbolic functions) to (non-deterministic) purely numerical programs, represented as* $[\![\mathtt{P}]\!]^\sharp_\Phi \in \operatorname{End}\Big(\prod_{f \in \mathcal{F}} \left(\prod_i \mathbb{Z}^{|\mathcal{M}_{T_i}|} \to \mathcal{P}(\mathbb{Z}^{|\mathcal{M}_{T_o}|})\right)\Big)$*. These can be further abstracted, e.g. to recurrence equations on interval-valued functions (where* $\mathcal{I}$ *replaces* $\mathcal{P}$ *above). By including* $\mathcal{I}(\mathbb{Z})$ *in* $\mathbb{Z}^2$*, we recover recurrence equations on usual integer-valued functions.*

We call *abstract compilation* such transformations (adding back syntax), that turn a concrete program (e.g. symbolic IR programs) into a simplified program on a new signature (e.g. numerical logic programs, recurrence equations, etc.). Our contribution solves, for catamorphic metrics, one of the challenges raised by [17]: the *computation* of such $[\![\mathtt{P}]\!]^\sharp_\Phi$.

3.2 An abstract compiler for IR

We have implemented these ideas in an abstract interpreter for IR. It contains higher-order abstract domains, as well as a generic "size lifter" (a domain functor) that applies numerical abstractions to symbolic programs through user-provided catamorphic metrics.

This allows fully automated abstract compilation of IR programs to precise recurrence equations, reflecting control flow and piecewise behaviour.

We also support compilation to first-order logic programs over extended integer arithmetic, which is a more precise representation than interval recurrence equations. This enables, e.g., reuse of abstract interpreters for logic programs, as well as SMT-based invariant verification.

4 Recurrence solving: search and verification of (inductive) bounds

In favourable cases, the obtained equations may be directly solvable by a mainstream computer algebra system (CAS) or fall within the scope of classical methods such as the master theorem. However, this need not be the case in general, and the precise equations extracted by abstract compilation may not admit any closed-form solution: indeed, they closely mirror program themselves, which can present complex features.

In such cases, we adopt the order-theoretical approach of [18], and search instead for pre/postfixpoints of the operator Φ, i.e. *inductive* bounds of the solution.

▶ **Proposition 5** ([18], Knaster-Tarski corollary)**.** *Let* $\Phi \in \mathrm{End}(\mathfrak{F})$ *be a* monotone *equation, i.e. an order-preserving operator on the underlying function space* $(\mathfrak{F}, \dot{\leq})$*, assumed to be a complete lattice. Then,* $\mathrm{lfp}\,\Phi$ *exists, and is what we call the* solution f_{sol} *to the equation.*

For any $\hat{f}$*,* $\Phi\hat{f} \dot{\leq} \hat{f} \implies f_{sol} \dot{\leq} \hat{f}$*, i.e. all postfixpoints are upper bounds.*

Moreover, under termination assumptions, we show that $\mathrm{lfp}\,\Phi = \mathrm{gfp}\,\Phi$*, so that we also have* $\hat{f} \dot{\leq} \Phi\hat{f} \implies \hat{f} \dot{\leq} f_{sol}$*.*

As observed by several recent works [18, 7, 17, 10], various pre/postfixpoint search techniques can be applied to discover inductive bounds. These include abstract interpretation, constrained optimisation, and templates mixed with quantifier elimination, among others.

4.1 The function comparison problem and transcendental SMT

Some of these techniques produce *candidate* bounds $\hat{f}$, and require additional work to *verify* the inductivity condition $\forall \vec{n}\, \Phi(\hat{f})(\vec{n}) \leq \hat{f}(\vec{n})$. This is a non-trivial problem, even in the single-quantifier case (*inference* with templates, instead, has one quantifier alternation).

Previous work has mainly focused on *incomplete* techniques based on CAS and rewriting. However, when $\hat{f}$ and $\Phi(\hat{f})$ are both (piecewise) polynomials, the comparison can be decided, at least under real relaxation, thanks to the decidability of real polynomial arithmetic. Moreover, the algorithmic theory of transcendental real and integer arithmetic, e.g. extended with exponentials and logarithms, has seen significant progress in recent years [3, 1, 2, 5, 6]. This motivates us to explore the use of a recent SMT tool, `Yices-TRA`[3], an extension of the `Yices` solver supporting nonlinear and transcendental arithmetic, which we are currently integrating as a backend to our pipeline.

4.2 Outlook: integration of termination-analysis techniques

These approaches suggest several research directions at the interface with termination analysis.

For example, as mentioned above and explored in subtler cases by [10], *termination hypotheses* on equations themselves are key to obtaining lower bounds on least fixed points.

Termination of equations is also relevant to their *execution*, which is a building block in some optimisation-based techniques [16, 18]. However, even for terminating programs, artificial non-termination may be introduced by non-determism of abstractions. Examples indicate possible ways to handle these situations, but no general theory is available yet.

Finally, *ranking functions* are central to several complexity analysers [11], and support, e.g., the common counter-based approach to equation setup (see [9] and related work). In a more mysterious sense, they also seem central to the *geometry of sets of postfixpoints* (see [18], Theorem 3), and suggest repair/refine approaches to function space exploration.

---

[3] `https://github.com/arith-lab/yices-tra`

References

1. Michael Benedikt, Dmitry Chistikov, and Alessio Mansutti. The Complexity of Presburger Arithmetic with Power or Powers. In *ICALP*, 2023. doi:10.4230/LIPIcs.ICALP.2023.112.
2. Dmitry Chistikov, Alessio Mansutti, and Mikhail R. Starchak. Integer Linear-Exponential Programming in NP by Quantifier Elimination. In *ICALP*, 2024. doi:10.4230/LIPICS.ICALP.2024.132.
3. Alessandro Cimatti, Alberto Griggio, Ahmed Irfan, Marco Roveri, and Roberto Sebastiani. Incremental Linearization for Satisfiability and Verification Modulo Nonlinear Arithmetic and Transcendental Functions. *TOCL*, August 2018. doi:10.1145/3230639.
4. Patrick Cousot and Radhia Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proc. of POPL'77*, pages 238–252. ACM Press, 1977. doi:10.1145/512950.512973.
5. Florian Frohn and Jürgen Giesl. Satisfiability Modulo Exponential Integer Arithmetic. In *IJCAR 2024*, July 2024. doi:10.1007/978-3-031-63498-7_21.
6. Jorge Gallego-Hernández, Enrico Lipparini, and Alessio Mansutti. MCSAT Modulo Transcendental Arithmetics, 2026. arXiv:2606.00697.
7. Amir K. Goharshady, S. Hitarth, and Sergei Novozhilov. Efficient Synthesis of Tight Polynomial Upper-Bounds for Systems of Conditional Polynomial Recurrences. In *ESOP 2025*, May 2025. doi:10.1007/978-3-031-91121-7_1.
8. Christoph Haase. Subclasses of Presburger Arithmetic and the Weak EXP Hierarchy. In *CSL-LICS*, 2014. doi:10.1145/2603088.2603092.
9. Bishoksan Kafle, John P. Gallagher, Manuel V. Hermenegildo, Maximiliano Klemen, Pedro Lopez-Garcia, and José F. Morales. Regular Path Clauses and their Application in Solving Loops. In *HCVS*, 2021. doi:10.4204/EPTCS.344.3.
10. Satoshi Kura, Hiroshi Unno, and Takeshi Tsukada. Supermartingales for Unique Fixed Points: A Unified Approach to Lower Bound Verification, 2026. arXiv:2504.04132.
11. Nils Lommen and Jürgen Giesl. Modular Automatic Complexity Analysis of Recursive Integer Programs. In *ESOP*, 2026. doi:10.1007/978-3-032-22723-2_1.
12. Alessio Mansutti and Mikhail R. Starchak. One-parametric Presburger Arithmetic has Quantifier Elimination. In *MFCS*, 2025. doi:10.4230/LIPICS.MFCS.2025.72.
13. Jorge Navas, Edison Mera, Pedro Lopez-Garcia, and Manuel V. Hermenegildo. User-Definable Resource Bounds Analysis for Logic Programs. In *ICLP*, September 2007. doi:10.1007/978-3-540-74610-2_24.
14. Derek C. Oppen. A $2^{2^{2^{pn}}}$ Upper Bound on the Complexity of Presburger Arithmetic. *J. Comput. Syst. Sci.*, 1978. doi:10.1016/0022-0000(78)90021-1.
15. Cattamanchi R. Reddy and Donald W. Loveland. Presburger Arithmetic with Bounded Quantifier Alternation. In *STOC*, 1978. doi:10.1145/800133.804361.
16. Louis Rustenholz, Maximiliano Klemen, Miguel Ángel Carreira-Perpiñán, and Pedro López-García. A Machine Learning-based Approach for Solving Recurrence Relations and its use in Cost Analysis of Logic Programs. *TPLP*, 2024. doi:10.1017/S1471068424000413.
17. Louis Rustenholz, Pedro Lopez-Garcia, and Manuel V. Hermenegildo. Abstractions of Sequences, Functions and Operators. *STTT*, March 2026. Special issue on CSV'25. doi:10.1007/s10009-026-00843-3.
18. Louis Rustenholz, Pedro Lopez-Garcia, José F. Morales, and Manuel V. Hermenegildo. An Order Theory Framework of Recurrence Equations for Static Cost Analysis - Dynamic Inference of Non-Linear Inequality Invariants. In *SAS*, 2024. Extended (preprint) version: arXiv:2406.18260. doi:10.1007/978-3-031-74776-2_14.
19. Robert Endre Tarjan. Amortized Computational Complexity. *SIAM Journal on Algebraic Discrete Methods*, 6(2):306–318, 1985. doi:10.1137/0606031.
20. Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones. Refinement Types for Haskell. In *ICFP '14*, 2014. doi:10.1145/2628136.2628161.
21. Volker Weispfenning. The Complexity of Almost Linear Diophantine Problems. *J. Symb. Comput.*, 1990. doi:10.1016/S0747-7171(08)80051-X.
22. Volker Weispfenning. Complexity and Uniformity of Elimination in Presburger Arithmetic. In *ISSAC*, pages 48–53, 1997. doi:10.1145/258726.258746.