A Machine Learning-based Approach for Solving Recurrence Relations and its use in Cost Analysis of Logic Programs † *

LOUIS RUSTENHOLZ, MAXIMILIANO KLEMEN
Technical University of Madrid (UPM) & IMDEA Software Institute

MIGUEL Á. CARREIRA-PERPIÑÁN
University of California, Merced

PEDRO LOPEZ-GARCIA
Spanish Council for Scientific Research & IMDEA Software Institute

Abstract

Automatic static cost analysis infers information about the resources used by programs without actually running them with concrete data, and presents such information as functions of input data sizes. Most of the analysis tools for logic programs (and many for other languages), as CiaoPP, are based on setting up recurrence relations representing (bounds on) the computational cost of predicates and solving them to find closed-form functions. Such recurrence solving is a bottleneck in current tools: many of the recurrences that arise during the analysis cannot be solved with state-of-the-art solvers, including Computer Algebra Systems (CASs), so that specific methods for different classes of recurrences need to be developed. We address such a challenge by developing a novel, general approach for solving arbitrary, constrained recurrence relations, that uses machine learning (sparse-linear and symbolic) regression techniques to *guess* a candidate closed-form function, and a combination of an SMT-solver and a CAS to *check* whether such function is actually a solution of the recurrence. Our prototype implementation and its experimental evaluation within the context of the CiaoPP system show quite promising results. Overall, for the considered benchmark set, our approach outperforms state-of-the-art cost analyzers and recurrence solvers, and can find closed-form solutions, in a reasonable time, for recurrences that cannot be solved by them.

Under consideration in Theory and Practice of Logic Programming (TPLP).

KEYWORDS: Cost Analysis, Recurrence Relations, Static Analysis, Machine Learning, Sparse Linear Regression, Symbolic Regression.

† Extended, revised version of our work published in ICLP (Klemen et al. 2023) (see Appendix A).

* Research partially supported by MICINN projects PID2019-108528RB-C21 *ProCode*, TED2021-132464B-I00 *PRODIGY*, and FJC2021-047102-I, and the Tezos foundation. The authors would also like to thank the anonymous reviewers for their insightful comments, which really help improve this paper. Special thanks go to ThanhVu Nguyen for his assistance with the related work discussion, and to John Gallagher, Manuel V. Hermenegildo and José F. Morales for their valuable discussions, feedback, and their work as developers of the CiaoPP system.

1 Introduction and Motivation

The motivation of the work presented in this paper stems from automatic static cost analysis and verification of logic programs (Debray et al. 1990; Debray and Lin 1993; Debray et al. 1997; Navas et al. 2007; Serrano et al. 2014; Lopez-Garcia et al. 2016; 2018). The goal of such analysis is to infer information about the resources used by programs without actually running them with concrete data, and present such information as functions of input data sizes and possibly other (environmental) parameters. We assume a broad concept of resource as a numerical property of the execution of a program, such as number of *resolution steps*, *execution time*, *energy consumption*, *memory*, number of *calls* to a predicate, number of *transactions* in a database, etc. Estimating in advance the resource usage of computations is useful for a number of applications, such as automatic program optimization, verification of resource-related specifications, detection of performance bugs, helping developers make resource-related design decisions, security applications (e.g., detection of side channel attacks), or blockchain platforms (e.g., smart-contract gas analysis and verification).

The challenge we address originates from the established approach of setting up recurrence relations representing the cost of predicates, parameterized by input data sizes (Wegbreit 1975; Rosendahl 1989; Debray et al. 1990; Debray and Lin 1993; Debray et al. 1997; Navas et al. 2007; Albert et al. 2011; Serrano et al. 2014; Lopez-Garcia et al. 2016), which are then solved to obtain *closed forms* of such recurrences (i.e., functions that provide either exact, or upper/lower bounds on resource usage in general). Such approach can infer different classes of functions (e.g., polynomial, factorial, exponential, summation, or logarithmic).

The applicability of these resource analysis techniques strongly depends on the capabilities of the component in charge of solving (or safely approximating) the recurrence relations generated during the analysis, which has become a bottleneck in some systems.

A common approach to automatically solving such recurrence relations consists of using a Computer Algebra System (CAS) or a specialized solver. However, this approach poses several difficulties and limitations. For example, some recurrence relations contain complex expressions or recursive structures that most of the well-known CASs cannot solve, making it necessary to develop ad-hoc techniques to handle such cases. Moreover, some recurrences may not have the form required by such systems because an input data size variable does not decrease, but increases instead. Note that a decreasing-size variable could be implicit in the program, i.e., it could be a function of a subset of input data sizes (a ranking function), which could be inferred by applying established techniques used in termination analysis (Podelski and Rybalchenko 2004). However, such techniques are usually restricted to linear arithmetic.

In order to address this challenge we have developed a novel, general method for solving arbitrary, constrained recurrence relations. It is a *guess and check* approach that uses machine learning techniques for the *guess* stage, and a combination of an SMT-solver and a CAS for the *check* stage (see Figure 1). To the best of our knowledge, there is no other approach that does this. The resulting closed-form function solutions can be of different kinds, such as polynomial, factorial, exponential or logarithmic.

Our method is parametric in the *guess* procedure used, providing flexibility in the kind of functions that can be inferred, trading off expressivity with efficiency and theoretical

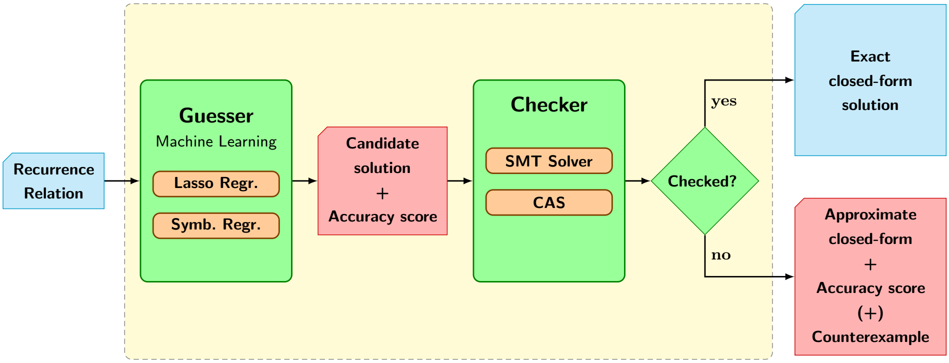
Recurrence Relation
Guesser
Machine Learning
Lasso Regr.
Symb. Regr.
Candidate solution + Accuracy score
Checker
SMT Solver
CAS
Checked?
yes
no
Exact closed-form solution
Approximate closed-form + Accuracy score (+) Counterexample

Fig. 1: Architecture of our novel machine learning-based recurrence solver.

guarantees. We present the results of two instantiations, based on sparse linear regression (with non-linear templates) and symbolic regression. Solutions to our equations are first evaluated on finite samples of the input space, and then, the chosen regression method is applied. In addition to obtaining exact solutions, the search and optimization algorithms typical of machine learning methods enable the efficient discovery of good approximate solutions in situations where exact solutions are too complex to find. These approximate solutions are particularly valuable in certain applications of cost analysis, such as granularity control in parallel/distributed computing. Furthermore, these methods allow exploration of model spaces that cannot be handled by complete methods such as classical linear equation solving.

The rest of this paper is organized as follows. Section 2 gives and overview of our novel *guess and check* approach. Then, Section 3 provides some background information and preliminary notation. Section 4 presents a more detailed, formal and algorithmic description of our approach. Section 5 describes the use of our approach in the context of static cost analysis of (logic) programs. Section 6 comments on our prototype implementation as well as its experimental evaluation and comparison with other solvers. Finally, Section 7 discusses related work, and Section 8 summarizes some conclusions and lines for future work. Additionally, Appendix B provides complementary data to the evaluation of Section 6 and Table 2.

2 Overview of our Approach

We now give an overview of our approach and its two stages already mentioned, illustrated in Figure 1: *guess* a candidate closed-form function, and *check* whether such function is actually a solution of the recurrence relation.

Given a recurrence relation for a function $f(\vec{x})$, solving it means to find a closed-form expression $\hat{f}(\vec{x})$ defining a function, on the appropriate domain, that satisfies the relation. We say that $\hat{f}$ is a closed-form expression whenever it does not contain any subexpression built using $\hat{f}$ (i.e., $\hat{f}$ is not recursively defined), although we will often additionally aim for expressions built only on elementary arithmetic functions, e.g., constants, addition, subtraction, multiplication, division, exponential, or perhaps rounding operators and factorial. We will use the following recurrence as an example to illustrate our approach.

$$\begin{aligned} f(x) &= 0 && \text{if } x = 0 \\ f(x) &= f(f(x-1)) + 1 && \text{if } x > 0 \end{aligned} \tag{1}$$

2.1 The "guess" stage

As already stated, our method is parametric in the guess procedure utilized, and we instantiate it with both sparse linear and symbolic regression in our experiments. However, for the sake of presentation, we initially focus on the former to provide an overview of our approach. Subsequently, we describe the latter in Section 3. Accordingly, any possible model we can obtain through sparse linear regression (which constitutes a candidate solution) must be an affine combination of a predefined set of terms that we call *base functions*. In addition, we aim to use only a small number of such *base functions*. That is, a candidate solution is a function $\hat{f}(\vec{x})$ of the form

$$\hat{f}(\vec{x}) = \beta_0 + \beta_1\ t_1(\vec{x}) + \beta_2\ t_2(\vec{x}) + \ldots + \beta_p\ t_p(\vec{x}),$$

where the t_i are arbitrary functions on $\vec{x}$ from a set $\mathcal{F}$ of candidate *base functions*, which are representative of common complexity orders, and the β_i's are the coefficients (real numbers) that are estimated by regression, but so that only a few coefficients are nonzero.

For illustration purposes, assume that we use the following set $\mathcal{F}$ of base functions:

$$\mathcal{F} = \{\lambda x.x, \lambda x.x^2, \lambda x.x^3, \lambda x.\lceil \log_2(x) \rceil, \lambda x.2^x, \lambda x.x \cdot \lceil \log_2(x) \rceil\},$$

where each base function is represented as a lambda expression. Then, the sparse linear regression is performed as follows.

1. Generate a training set $\mathcal{T}$. First, a set $\mathcal{I} = \{\vec{x}_1, \ldots, \vec{x}_n\}$ of input values to the recurrence function is randomly generated. Then, starting with an initial $\mathcal{T} = \emptyset$, for each input value $\vec{x}_i \in \mathcal{I}$, a training case s_i is generated and added to $\mathcal{T}$. For any input value $\vec{x} \in \mathcal{I}$ the corresponding training case s is a pair of the form

$$s = \big(\langle b_1, \ldots, b_p \rangle,\ r\big),$$

where $b_i = [\![t_i]\!]_{\vec{x}}$ for $1 \leq i \leq p$, and $[\![t_i]\!]_{\vec{x}}$ represents the result (a scalar) of evaluating the base function $t_i \in \mathcal{F}$ for input value $\vec{x}$, where $\mathcal{F}$ is a set of p base functions, as already explained. The (dependent) value r (also a constant number) is the result of evaluating the recurrence $f(\vec{x})$ that we want to solve or approximate, in our example, the one defined in Equation 1. Assuming that there is an $\vec{x} \in \mathcal{I}$ such that $\vec{x} = \langle 5 \rangle$, its corresponding training case s in our example will be

$$\begin{aligned} s &= \big(\langle [\![x]\!]_5, [\![x^2]\!]_5, [\![x^3]\!]_5, [\![\lceil \log_2(x) \rceil]\!]_5, \ldots \rangle,\ \mathbf{f(5)}\big) \\ &= \big(\langle 5, 25, 125, 3, \ldots \rangle,\ \mathbf{5}\big). \end{aligned}$$

2. Perform sparse regression using the training set $\mathcal{T}$ created above in order to find a small subset of base functions that fits it well.

   We do this in two steps. First, we solve an ℓ_1-regularized linear regression to learn an estimate of the non-zero coefficients of the base functions. This procedure, also called lasso (Hastie et al. 2015), was originally introduced to learn interpretable models by selecting a subset of the input features. This happens because the ℓ_1 (sum of absolute values) penalty results in some coefficients becoming exactly zero (unlike an ℓ_2^2 penalty, which penalizes the magnitudes of the coefficients but typically results in none of them being zero). This will typically discard most of the base functions in $\mathcal{F}$, and only those that are really needed to approximate our target function will be kept. The level of penalization is controlled by a hyperparameter $\lambda \geq 0$. As commonly done in machine learning (Hastie et al. 2015), the value of λ that generalizes optimally on unseen (test) inputs is found via cross-validation on a separate validation set (generated randomly in the same way as the training set). The result of this first sparse regression step are coefficients $\beta_1, \ldots, \beta_p$ (typically many of which are zero), and an independent coefficient β_0. In a second step, we keep only those coefficients (and their corresponding terms t_i) for which $|\beta_i| \geq \epsilon$ (where the value of $\epsilon \geq 0$ is determined experimentally). We find that this post-processing results in solutions that better estimate the true non-zero coefficients.

3. Finally, our method performs again a standard linear regression (without ℓ_1 regularization) on the training set $\mathcal{T}$, but using only the base functions selected in the previous step. In our example, with $\epsilon = 0.05$, we obtain the model

$$\hat{f}(x) = 1.0 \cdot x.$$

A final test set $\mathcal{T}_{\text{test}}$ with input set $\mathcal{I}_{\text{test}}$ is then generated (in the same way as the training set) to obtain a measure R^2 of the accuracy of the estimation. In this case, we obtain a value $R^2 = 1$, which means that the estimation obtained predicts exactly the values for the test set. This does not prove that the $\hat{f}$ is a solution of the recurrence, but this makes it a candidate solution for verification. If R^2 were less than 1, it would mean that the function obtained is not a candidate (exact) solution, but an approximation (not necessarily a bound), as there are values in the test set that cannot be exactly predicted.

Currently, the set of base functions $\mathcal{F}$ is fixed; nevertheless, we plan to automatically infer better, more problem-oriented sets by using different heuristics, as we comment on in Section 8. Alternatively, as already mentioned, our guessing method is parametric and can also be instantiated to symbolic regression, which mitigates this limitation by creating new expressions, i.e., new "templates", beyond linear combinations of $\mathcal{F}$. However, only shallow expressions are reachable in acceptable time by symbolic regression's exploration strategy: timeouts will often occur if solutions can only be expressed by deep expressions.

2.2 The "check" stage

Once a function that is a candidate solution for the recurrence has been guessed, the second step of our method tries to verify whether such a candidate is actually a solution. To do so, the recurrence is encoded as a first order logic formula where the references to the target function are replaced by the candidate solution whenever possible. Afterwards, we use an SMT-solver to check whether the negation of such formula is satisfiable, in which case we can conclude that the candidate is not a solution for the recurrence. Otherwise, if such formula is unsatisfiable, then the candidate function is an exact solution. Sometimes, it is necessary to consider a precondition for the domain of the recurrence, which is also included in the encoding.

To illustrate this process, Expression 2 below

$$f(x) = \begin{cases} 0 & \text{if } x = 0 \\ f(f(x-1)) + 1 & \text{if } x > 0 \end{cases} \tag{2}$$

shows the recurrence we target to solve, for which the candidate solution obtained previously using (sparse) linear regression is $\hat{f}(x) = x$ (for all $x \geq 0$). Now, Expression 3 below shows the encoding of the recurrence as a first order logic formula.

$$\forall x \left((x = 0 \implies \underline{f(x)} = 0) \land (x > 0 \implies \underline{f(x)} = \underline{f(\underline{f(x-1)})} + 1) \right) \tag{3}$$

Finally, Expression 4 below shows the negation of such formula, as well as the references to the function name substituted by the definition of the candidate solution. We underline both the subexpressions to be replaced, and the subexpressions resulting from the substitutions.

$$\exists x \, \neg(((x = 0 \implies \underline{x} = 0) \land (x > 0 \implies \underline{x} = \underline{x-1} + 1))) \tag{4}$$

It is easy to see that Formula (4) is unsatisfiable. Therefore, $\hat{f}(x) = x$ is an exact solution for $f(x)$ in the recurrence defined by Equation 1.

For some cases where the candidate solution contains transcendental functions, our implementation of the method uses a CAS to perform simplifications and transformations, in order to obtain a formula supported by the SMT-solver. We find this combination of CAS and SMT-solver particularly useful, since it allows us to solve more problems than only using one of these systems in isolation.

3 Preliminaries

Notations. We use the letters x, y, z to denote variables, and a, b, c, d to denote constants and coefficients. We use f, g to represent functions, and e, t to represent arbitrary expressions. We use φ to represent arbitrary boolean constraints over a set of variables. Sometimes, we also use β to represent coefficients obtained with (sparse) linear regression. In all cases, the symbols can be subscribed. $\mathbb{N}$ and $\mathbb{R}^+$ denote the sets of non-negative integer and non-negative real numbers respectively, both including 0. Given two sets A and B, B^A is the set of all functions from A to B. We use $\vec{x}$ to denote a finite sequence $\langle x_1, x_2, \ldots, x_p \rangle$, for some $p > 0$. Given a sequence S and an element x, $\langle x | S \rangle$ is a new sequence with first element x and tail S. We refer to the classical finite-dimensional 1-norm (Manhattan norm) and 2-norm (Euclidean norm) by ℓ_1 and ℓ_2 respectively, while ℓ_0 denotes the "norm" (which we will call a pseudo-norm) which counts the number of non-zero coordinates of a vector.

Recurrence relations. In our setting, a *recurrence relation* (or *recurrence equation*) is just a functional equation on $f : \mathcal{D} \to \mathbb{R}$ with $\mathcal{D} \subset \mathbb{N}^m$, $m \geq 1$, that can be written as

$$\forall \vec{x} \in \mathcal{D}, \; f(\vec{x}) = \Phi(f, \vec{x}),$$

where $\Phi : \mathbb{R}^{\mathcal{D}} \times \mathcal{D} \to \mathbb{R}$ is used to define $f(\vec{x})$ in terms of other values of f. In this paper we consider functions f that take natural numbers as arguments but output real values, e.g., corresponding to costs such as energy consumption, which need not be measured as integer values in practice. Working with real values also makes optimizing for the regression coefficients easier. We restrict ourselves to the domain $\mathbb{N}^m$ because in our motivating application, cost/size analysis, the input arguments to recurrences represent data sizes, which take non-negative integer values. Technically, our approach may be easily extended to recurrence equations on functions with domain $\mathbb{Z}^m$.

A system of recurrence equations is a functional equation on multiple $f_i : \mathcal{D}_i \to \mathbb{R}$, i.e., $\forall i$, $\forall \vec{x} \in \mathcal{D}_i,\ f_i(\vec{x}) = \Phi_i(f_1, \ldots, f_r, \vec{x})$. Inequations and non-deterministic equations can also be considered by making Φ non-deterministic, i.e., $\Phi : \mathbb{R}^{\mathcal{D}} \times \mathcal{D} \to \mathcal{P}(\mathbb{R})$ and $\forall \vec{x} \in \mathcal{D},\ f(\vec{x}) \in \Phi(f, \vec{x})$. In general, such equations may have any number of solutions. In this work, we focus on deterministic recurrence equations, as we will discuss later.

Classical recurrence relations of order k are recurrence relations where $\mathcal{D} = \mathbb{N}$, $\Phi(f, n)$ is a constant when $n < k$ and $\Phi(f, n)$ depends only on $f(n-k), \ldots, f(n-1)$ when $n \geq k$. For example, the following recurrence relation of order $k = 1$, where $\Phi(f, n) = f(n-1)+1$ when $n \geq 1$, and $\Phi(f, n) = 1$ when $n < 1$ (or equivalently, $n = 0$), i.e.,

$$f(n) = \begin{cases} 1 & \text{if } n = 0, \\ f(n-1) + 1 & \text{if } n \geq 1, \end{cases} \tag{5}$$

has the closed-form function $f(n) = n + 1$ as a solution.

Another example, with order $k = 2$, is the historical Fibonacci recurrence relation, where $\Phi(f, n) = f(n-1) + f(n-2)$ when $n \geq 2$, and $\Phi(f, n) = 1$ when $n < 2$:

$$f(n) = \begin{cases} 1 & \text{if } n = 0 \text{ or } n = 1, \\ f(n-1) + f(n-2) & \text{if } n \geq 2. \end{cases} \tag{6}$$

Φ may be viewed as a *recursive definition* of "the" solution f of the equation, with the caveat that the definition may be non-satisfiable or partial (degrees of freedom may remain). We define below an *evaluation strategy* EvalFun of this recursive definition, which, when it terminates for all inputs, provides *a* solution to the equation. This will allow us to view recurrence equations as programs that can be evaluated, enabling us to easily generate input-output pairs, on which we can perform regression to attempt to guess a symbolic solution to the equation.

This setting can be generalized to *partial solutions* to the equation, i.e., partial functions $f : \mathcal{D} \rightharpoonup \mathbb{R}$ such that $f(\vec{x}) = \Phi(f, \vec{x})$ whenever they are defined at $\vec{x}$.

We are interested in *constrained recurrence relations*, where Φ is expressed piecewise as

$$\Phi(f, \vec{x}) = \begin{cases} e_1(\vec{x}) & \text{if } \varphi_1(\vec{x}) \\ e_2(\vec{x}) & \text{if } \varphi_2(\vec{x}) \\ \vdots & \vdots \\ e_k(\vec{x}) & \text{if } \varphi_k(\vec{x}), \end{cases} \tag{7}$$

with $f : \mathcal{D} \to \mathbb{R}$, $\mathcal{D} = \{\vec{x} \mid \vec{x} \in \mathbb{N}^k \wedge \varphi_{\text{pre}}(\vec{x})\}$ for some boolean constraint φ_{pre}, called the *precondition of* f, and $e_i(\vec{x})$, $\varphi_i(\vec{x})$ are respectively arbitrary expressions and constraints over both $\vec{x}$ and f. We further require that Φ is always defined, i.e., $\varphi_{\text{pre}} \models \vee_i\, \varphi_i$. A case such that e_i, φ_i do not contain any call to f is called a *base case*, and those that do are called *recursive cases*. In practice, we are only interested in equations with at least one base case and one recursive case.

A challenging class of recurrences that can be tackled with our approach is that of "nested" recurrences, where recursive cases may contain nested calls $f(f(...))$.

We assume that the φ_i are mutually exclusive, so that Φ must be deterministic. This is not an important limitation for our motivating application, cost analysis, and in particular the one performed by the CiaoPP system. Such cost analysis can deal with a large class of non-deterministic programs by translating the resulting non-deterministic recurrences into deterministic ones. For example, assume that the cost analysis generates the following recurrence (which represents an input/output size relation).

$$\begin{array}{rcll} f(x) & = & 0 & \text{if } x = 0 \\ f(x) & = & f(x-1)+1 & \text{if } x > 0 \\ f(x) & = & f(x-1)+2 & \text{if } x > 0 \end{array}$$

Then, prior to calling the solver, the recurrence is transformed into the following two deterministic recurrences, the solution of which would be an upper or lower bound on the solution to the original recurrence. For upper bounds:

$$\begin{array}{rcll} f(x) & = & 0 & \text{if } x = 0, \\ f(x) & = & \max(f(x-1)+1, f(x-1)+2) & \text{if } x > 0, \end{array}$$

and for lower bounds:

$$\begin{array}{rcll} f(x) & = & 0 & \text{if } x = 0, \\ f(x) & = & \min(f(x-1)+1, f(x-1)+2) & \text{if } x > 0. \end{array}$$

Our regression technique correctly infers the solution $f(x) = 2x$ in the first case, and $f(x) = x$ in the second case. We have access to such program transformation, that recovers determinism by looking for worst/best cases, under some hypotheses, outside of the scope of this paper, on the kind of non-determinism and equations that are being dealt with.

We now introduce an evaluation strategy of recurrences that allows us to be more consistent with the termination of programs than the more classical semantics consisting only of maximally defined partial solutions. Let $\mathsf{def}(\Phi)$ denote a sequence $\langle (e_1(\vec{x}), \varphi_1(\vec{x})), \ldots, (e_k(\vec{x}), \varphi_k(\vec{x})) \rangle$ defining a (piecewise) constrained recurrence relation Φ on f, where each element of the sequence is a pair representing a case. The evaluation of the equation for a concrete value $\vec{d}$, denoted $\mathsf{EvalFun}(f(\vec{d}))$, is defined as follows.

$$\mathsf{EvalFun}\big(f(\vec{d})\,\big) = \mathsf{EvalBody}(\mathsf{def}(\Phi), \vec{d}\,)$$

$$\mathsf{EvalBody}\big(\langle (e,\varphi) | \mathsf{Ps} \rangle, \vec{d}\,\big) = \begin{cases} [\![e]\!]_{\vec{d}} & \text{if } \varphi(\vec{d}\,) \\ \mathsf{EvalBody}(\mathsf{Ps}, \vec{d}) & \text{if } \neg\varphi(\vec{d}\,) \end{cases}$$

The goal of our regression strategy is to find an expression $\hat{f}$ representing a function $\mathcal{D} \to \mathbb{R}$ such that, for all $\vec{d} \in \mathcal{D}$,

- If $\mathsf{EvalFun}(f(\vec{d}))$ terminates, then $\mathsf{EvalFun}(f(\vec{d})) = [\![\hat{f}]\!]_{\vec{d}}$, and
- $\hat{f}$ does not contain any recursive call in its definition.

If the above conditions are met, we say that $\hat{f}$ is a *closed form* for f. In the case of (sparse) linear regression, we are looking for expressions

$$\hat{f}(\vec{x}) = \beta_0 + \beta_1\ t_1(\vec{x}) + \beta_2\ t_2(\vec{x}) + \ldots + \beta_p\ t_p(\vec{x}), \qquad (8)$$

where $\beta_i \in \mathbb{R}$, and t_i are expressions over $\vec{x}$, not including recursive references to $\hat{f}$.

For example, consider the following Prolog program which does not terminate for a call `q(X)` where `X` is bound to a non-zero integer.

```
q(X) :- X > 0, X1 is X + 1, q(X1).
q(X) :- X = 0.
```

The following recurrence relation for its cost (in resolution steps) can be set up.

$$\begin{array}{rcll} \mathtt{C_q}(x) & = & 1 & \text{if } x = 0 \\ \mathtt{C_q}(x) & = & 1 + \mathtt{C_q}(x+1) & \text{if } x > 0 \end{array} \tag{9}$$

A CAS will give the closed form $\mathtt{C_q}(x) = 1 - x$ for such recurrence, however, the cost analysis should give $\mathtt{C_q}(x) = \infty$ for $x > 0$.

(Sparse) Linear Regression. Linear regression (Hastie et al. 2009) is a statistical technique used to approximate the linear relationship between a number of explanatory (input) variables and a dependent (output) variable. Given a vector of (input) real-valued variables $X = (X_1, \ldots, X_p)^T$, we predict the output variable Y via the model

$$\hat{Y} = \beta_0 + \sum_{i=1}^{p} \beta_i X_i, \tag{10}$$

which is defined through the vector of coefficients $\vec{\beta} = (\beta_0, \ldots, \beta_p)^T \in \mathbb{R}^{p+1}$. Such coefficients are estimated from a set of observations $\{(\langle x_{i1}, \ldots, x_{ip} \rangle, y_i)\}_{i=1}^{n}$ so as to minimize a loss function, most commonly the sum of squares

$$\vec{\beta} = \underset{\vec{\beta} \in \mathbb{R}^{p+1}}{\arg\min} \sum_{i=1}^{n} \left(y_i - \beta_0 - \sum_{j=1}^{p} x_{ij} \beta_j \right)^2 . \tag{11}$$

Sometimes (as is our case) some of the input variables are not relevant to explain the output, but the above least-squares estimate will almost always assign nonzero values to all the coefficients. In order to force the estimate to make exactly zero the coefficients of irrelevant variables (hence removing them and doing *feature selection*), various techniques have been proposed. The most widely used one is the lasso (Hastie et al. 2015), which adds an ℓ_1 penalty on $\vec{\beta}$ (i.e., the sum of absolute values of each coefficient) to Expression 11, obtaining

$$\vec{\beta} = \underset{\vec{\beta} \in \mathbb{R}^{p+1}}{\arg\min} \sum_{i=1}^{n} \left(y_i - \beta_0 - \sum_{j=1}^{p} x_{ij} \beta_j \right)^2 + \lambda \sum_{j=1}^{p} |\beta_j|, \tag{12}$$

where $\lambda \geq 0$ is a hyperparameter that determines the level of penalization: the greater λ, the greater the number of coefficients that are exactly equal to 0. The lasso has two advantages over other feature selection techniques for linear regression. First, it defines a convex problem whose unique solution can be efficiently computed even for datasets where either of n or p are large (almost as efficiently as a standard linear regression). Second, it has been shown in practice to be very good at estimating the relevant variables. In fact, the regression problem we would really like to solve is that of Expression 11 but subject to the constraint that $\left\| (\beta_1, \ldots, \beta_p)^T \right\|_0 \leq K$, i.e., that at most K of the p coefficients are non-zero, an ℓ_0-constrained problem. Unfortunately, this is an NP-hard problem. However, replacing the ℓ_0 pseudo-norm with the ℓ_1-norm has been observed to produce good approximations in practice (Hastie et al. 2015).

Symbolic Regression. Symbolic regression is a regression task in which the model space consists of all mathematical expressions on a chosen signature, i.e., expression trees with variables or constants for leaves and operators for internal nodes. To avoid overfitting, objective functions are designed to penalize model complexity, in a similar fashion to sparse linear regression techniques. This task is much more ambitious: rather than searching over the vector space spanned by a relatively small set of base functions as we do in sparse linear regression, the search space is enormous, considering any possible expression which results from applying a set of mathematical operators in any combination. For this reason, heuristics such as evolutionary algorithms are typically used to search this space, but runtime still remains a challenge for deep expressions.

The approach presented in this paper is parametric in the regression technique used, and we instantiate it with both (sparse) linear and symbolic regression in our experiments (Section 6). We will see that symbolic regression addresses some of the limitations of (sparse) linear regression, at the expense of time. Our implementation is based on the symbolic regression library `PySR` (Cranmer 2023), a multi-population evolutionary algorithm paired with classical optimization algorithms to select constants. In order to avoid overfitting and guide the search, `PySR` penalizes model complexity, defined as a sum of individual node costs for nodes in the expression tree, where a predefined cost is assigned to each possible type of node.

In our application context, (sparse) linear regression searches for a solution to the recurrence equation in the affine space spanned by candidate functions, i.e., $\hat{f} = \beta_0 + \sum_i \beta_i t_i$ with $t_i \in \mathcal{F}$, while symbolic regression may choose any expression built on the chosen operators. For example, consider equation `exp3` of Table 1, whose solution is $(x, y) \mapsto 2^{x+y}$. This solution cannot be expressed using (sparse) linear regression and the set of candidates $\{\lambda xy.x, \lambda xy.y, \lambda xy.x^2, \lambda xy.y^2, \lambda xy.2^x, \lambda xy.2^y\}$, but can be found with symbolic regression and operators $\{+(\cdot,\cdot), \times(\cdot,\cdot), 2^{(\cdot)}\}$.

4 Algorithmic Description of the Approach

In this section we describe our approach for generating and checking candidate solutions for recurrences that arise in resource analysis.

4.1 A first version

Algorithms 1 and 2 correspond to the *guesser* and *checker* components, respectively, which are illustrated in Figure 1. For the sake of presentation, Algorithm 1 describes the instantiation of the guess method based on lasso linear regression. It receives a recurrence relation for a function f to solve, a set of base functions, and a threshold to decide when to discard irrelevant terms. The output is a closed-form expression $\hat{f}$ for f, and a *score* $\mathcal{S}$ that reflects the accuracy of the approximation, in the range $[0, 1]$. If $\mathcal{S} \approx 1$, the approximation can be considered a candidate solution. Otherwise, $\hat{f}$ is an approximation (not necessarily a bound) of the solution.

Generate. In line 1 we start by generating a set $\mathcal{I}$ of random inputs for f. Each input $\vec{x_i}$ is an m-tuple verifying precondition φ_{pre}, where m is the number of arguments of f. In line 2 we produce the training set $\mathcal{T}$. The (explanatory) inputs are generated by

Algorithm 1: Candidate Solution Generation (*Guesser*).

```
Input  : Target recurrence relation on $f : \mathcal{D} \to \mathbb{R}$.
         $\varphi_{\mathrm{pre}}$: precondition defining $\mathcal{D}$.
         $n$: number of (random) inputs for evaluating $f$.
         $\mathcal{F} \subseteq \mathcal{D} \to \mathbb{R}$: set of base functions (represented as a tuple).
         $\Lambda$: range of values to automatically choose a lasso hyperparameter $\lambda \in \mathbb{R}^+$ that
         maximizes the performance of the model via cross-validation.
         $k$: indicates performing $k$-fold cross-validation, $k \geq 2$.
         $\epsilon \in \mathbb{R}^+$: threshold for term ($t_i \in \mathcal{F}$) selection.
Output: $\hat{f} \in \mathsf{Exp}$: a candidate solution (or an approximation) for $f$.
         $\mathcal{S} \in [0, 1]$: score, indicating the accuracy of the estimation ($R^2$).
1 $\mathcal{I} \leftarrow \{\vec{x_i} \mid \vec{x_i} \in \mathbb{N}^m \wedge \varphi_{\mathrm{pre}}(\vec{x_i})\}_{i=1}^{n}$ ;          // $n$ random inputs for $f$
2 $\mathcal{T} \leftarrow \{(E(\mathcal{F}, \vec{x}), f(\vec{x})) \mid \vec{x} \in \mathcal{I}\}$ ;                          // Training set
3 $(\vec{\beta'}, \beta'_0) \leftarrow$ CVLassoRegression$(\mathcal{T}, \Lambda, k)$;
4 $(\mathcal{F}', \mathcal{T}') \leftarrow$ RemoveTerms$(\mathcal{F}, \mathcal{T}, \vec{\beta'}, \beta'_0, \epsilon)$ ;                      // $\epsilon$-pruning
5 $(\vec{\beta}, \beta_0, \mathcal{S}) \leftarrow$ LinearRegression$(\mathcal{T}')$;
6 $\hat{f} \leftarrow \beta_0 + \lambda\vec{x} \cdot \vec{\beta}^T \times E(\mathcal{F}', \vec{x})$;
7 return $(\hat{f}, \mathcal{S})$;
```

evaluating the base functions in $\mathcal{F} = \langle t_1, t_2, \ldots, t_p \rangle$ with each tuple $\vec{x} \in \mathcal{I}$. This is done by using function E, defined as follows.

$$E(\langle t_1, t_2, \ldots, t_p \rangle, \vec{x}) = \langle t_1(\vec{x}), t_2(\vec{x}), \ldots, t_p(\vec{x}) \rangle$$

We also evaluate the recurrence equation for input $\vec{x}$, and add the observed output $f(\vec{x})$ as the last element in the corresponding training case.

Regress. In line 3 we generate a first linear model by applying function CVLassoRegression to the generated training set, which performs a sparse linear regression with lasso regularization. As already mentioned, lasso regularization requires a hyperparameter λ that determines the level of penalization for the coefficients. Instead of using a single value for λ, CVLassoRegression uses a range of possible values, applying cross-validation on top of the linear regression to automatically select the best value for that parameter, from the given range. In particular, k-fold cross-validation is performed, which means that the training set is split into k parts or *folds*. Then, each fold is taken as the validation set, training the model with the remaining $k-1$ folds. Finally, the performance measure reported is the average of the values computed in the k iterations. The result of this function is the vector of coefficients $\vec{\beta'}$, together with the intercept β'_0. These coefficients are used in line 4 to decide which base functions are discarded before the last regression step. Note that RemoveTerms removes the base functions from $\mathcal{F}$ together with their corresponding output values from the training set $\mathcal{T}$, returning the new set of base functions $\mathcal{F}'$ and its corresponding training set $\mathcal{T}'$. In line 5, standard linear regression (without regularization or cross-validation) is applied, obtaining the final coefficients $\vec{\beta}$ and β_0. Additionally, from this step we also obtain the score $\mathcal{S}$ of the resulting model. In line 6 we set up the resulting closed-form expression, given as a function on the variables in $\vec{x}$. Note that we use the function E to bind the variables in the base functions to the arguments of the closed-form expression. Finally, the closed-form expression and its corresponding score are returned as the result of the algorithm.

Algorithm 2: Solution Checking (*Checker*).

```
   Input  : Target recurrence relation on $f : \mathcal{D} \to \mathbb{R}$.
            $\varphi_{\text{pre}}$: precondition defining $\mathcal{D}$.
            $\hat{f} \in \mathsf{Exp}$: a candidate solution for $f$.
   Output: true if $\hat{f}$ is a solution for $f$, false otherwise.
 1 $\varphi_{\text{previous}} \leftarrow$ true ;
 2 Formula $\leftarrow$ true ;
 3 foreach $(e, \varphi) \in \mathsf{def}(f)$ do
 4     Eq $\leftarrow$ replaceCalls("$f(\vec{x}) - e = 0$", $f(\vec{x}), \hat{f}, \varphi_{\text{pre}}, \varphi$);
 5     if $\neg$ containsCalls(Eq, $f$) then
 6         Eq $\leftarrow$ simplifyCAS(inlineCalls(Eq, $\hat{f}$, def($\hat{f}$)));
 7         if supportedSMT(Eq) then
 8             Formula $\leftarrow$ "Formula $\wedge$ ($\varphi_{\text{pre}} \wedge \varphi_{\text{previous}} \wedge \varphi \implies$ Eq)";
 9             $\varphi_{\text{previous}} \leftarrow$ "$\varphi_{\text{previous}} \wedge \neg\varphi$" ;
10         else
11             return false;
12         end
13     else
14         return false;
15     end
16 end
17 return ($\not\models_{SMT} [\![\neg\text{Formula}]\!]_{SMT}$);
```

Verify. Algorithm 2 mainly relies on an SMT-solver and a CAS. Concretely, given the constrained recurrence relation on $f : \mathcal{D} \to \mathbb{R}$ defined by

$$f(\vec{x}) = \begin{cases} e_1(\vec{x}) & \text{if } \varphi_1(\vec{x}) \\ e_2(\vec{x}) & \text{if } \varphi_2(\vec{x}) \\ \vdots & \vdots \\ e_k(\vec{x}) & \text{if } \varphi_k(\vec{x}) \end{cases}$$

our algorithm constructs the logic formula

$$\left[\!\!\left[\bigwedge_{i=1}^{k} \left(\left(\bigwedge_{j=1}^{i-1} \neg\varphi_j(\vec{x}) \right) \wedge \varphi_i(\vec{x}) \wedge \varphi_{\text{pre}}(\vec{x}) \implies \mathsf{Eq}_i \right) \right]\!\!\right]_{\text{SMT}} \tag{13}$$

where operation $[\![e]\!]_{\text{SMT}}$ is the translation of any expression e to an SMT-LIB expression, and Eq_i is the result of replacing in $f(\vec{x}) = e_i(\vec{x})$ each occurrence of f (*a priori* written as an uninterpreted function) by the definition of the candidate solution $\hat{f}$ (by using replaceCalls in line 4), and performing a simplification (by using simplifyCAS in line 6). The function replaceCalls(expr, $f(\vec{x}')$, $\hat{f}$, φ_{pre}, φ) replaces every subexpression in expr of the form $f(\vec{x}')$ by $\hat{f}(\vec{x}')$, if $\varphi_{\text{pre}}(\vec{x}') \wedge \varphi \implies \varphi_{\text{pre}}(\vec{x}')$. When checking Formula 13, all variables are assumed to be integers. As an implementation detail, to work with Z3 and access a large arithmetic language (including rational division), variables are initially declared as reals, but integrality constraints ($\bigwedge_i \vec{x}_i = \lfloor \vec{x}_i \rfloor$) are added to the final formula. Note that this encoding is consistent with the evaluation (EvalFun) described in Section 3.

The goal of simplifyCAS is to obtain (sub)expressions supported by the SMT-solver. This typically allows simplifying away differences of transcendental functions, such as exponentials and logarithms, for which SMT-solvers like Z3 currently have extremely limited support, often dealing with them as if they were uninterpreted functions. For example, `log2` is simply absent from SMT-LIB, although it can be modelled with exponentials, and reasoning abilities with exponentials are very limited: while Z3 can check that $2^x - 2^x = 0$, it cannot check (without further help) that $2^{x+1} - 2 \cdot 2^x = 0$. Using $\mathsf{simplifyCAS}$, the latter is replaced by $0 = 0$ which is immediately checked.

Finally, the algorithm asks the SMT-solver for models of the negated formula (line 17). If no model exists, then it returns true, concluding that $\hat{f}$ is an exact solution to the recurrence, i.e., $\hat{f}(\vec{x}) = f(\vec{x})$ for any input $\vec{x} \in \mathcal{D}$ such that $\mathsf{EvalFun}(f(\vec{x}))$ terminates. Otherwise, it returns false. Note that, if we are not able to express $\hat{f}$ using the syntax supported by the SMT-solver, even after performing the simplification by $\mathsf{simplifyCAS}$, then the algorithm finishes returning false.

4.2 Extension: domain splitting

For the sake of exposition, we have only presented a basic combination of Algorithms 1 and 2, but the core approach *generate, regress and verify* can be generalized to obtain more accurate results. Beyond the use of different regression algorithms (replacing lines 3–6 in Algorithm 1, e.g., with symbolic regression as presented in Section 3), we can also decide to apply Algorithm 1 separately on multiple subdomains of $\mathcal{D}$: we call this strategy *domain splitting*. In other words, rather than trying to directly infer a solution on $\mathcal{D}$ by regression, we do the following.

- *Partition* $\mathcal{D}$ into subdomains $\mathcal{D}_i$.
- Apply (any variant of) Algorithm 1 on each $\mathcal{D}_i$, i.e., *generate* input-output pairs, and *regress* to obtain candidates $\hat{f}_i : \mathcal{D}_i \to \mathbb{R}$.
- This gives a global candidate $\hat{f} : x \mapsto \{\hat{f}_i \text{ if } x \in \mathcal{D}_i\}$, that we can then attempt to *verify* (Algorithm 2).

A motivation for doing so is the observation that it is easier for regression algorithms to discover expressions of "low (model) complexity", i.e., expressions of low depth on common operators for symbolic regression and affine combinations of common functions for (sparse) linear regression (note that *model complexity of the expression* is not to be confused with the *computational complexity of the algorithm* whose cost is represented by the expression, i.e., the asymptotic rate of growth of the corresponding function). We also observe that our equations often admit solutions that can be described piecewise with low (model) complexity expressions, where equivalent global expressions are somehow artificial: they have high (model) complexity, making them hard or impossible to find. In other words, equations with "piecewise simple" solutions are more common than those with "globally simple" solutions, and the domain splitting strategy is able to decrease complexity for this common case by reduction to the more favorable one.

For example, consider equation `merge` in Table 1, representing the cost of a merge function in a merge-sort algorithm. Its solution is

$$\begin{aligned} &f : \mathbb{N}^2 \to \mathbb{R} \\ &(x, y) \mapsto \begin{cases} x + y - 1 & \text{if } x > 0 \wedge y > 0, \\ 0 & \text{if } x = 0 \vee y = 0. \end{cases} \end{aligned}$$

It admits a piecewise affine description, but no simple global description as a polynomial, although we can admittedly write it as $\min(x, 1) \times \min(y, 1) \times (x + y - 1)$, which is unreachable by (sparse) linear regression for any reasonable set of candidates, and of challenging depth for symbolic regression.

To implement domain splitting, there are two challenges to face: (1) partition the domain into appropriate subdomains that make regression more accurate and (2) generate random input points inside each subdomain.

In our implementation (Section 6), we test a very simple version of this idea, were generation is handled by a trivial rejection strategy, and where the domain is partitioned using the conditions $\varphi_i(\vec{x}) \wedge \bigwedge_{j=1}^{i-1} \neg\varphi_j(\vec{x})$ that define each clause of the input equation.

In other words, our splitting strategy is *purely syntactic*. More advanced strategies could learn better subdomains, e.g., by using a generalization of model trees, and are left for future work. However, as we will see in Section 6, our naive strategy already provides good improvements compared to Section 4.1. Intuitively, this seems to indicate that a large part of the "disjunctive behavior" of solutions originates from the "input disjunctivity" in the equation (which, of course, can be found in other places than the φ_i, but this is a reasonable first approximation).

Finally, for the verification step, we can simply construct an SMT formula corresponding to the equation as in Section 4.1, using an expression defined by cases for $\hat{f}$, e.g., with the `ite` construction in SMT-LIB.

5 Our Approach in the Context of Static Cost Analysis of (Logic) Programs

In this section, we describe how our approach could be used in the context of the motivating application, Static Cost Analysis. Although it is general, and could be integrated into any cost analysis system based on recurrence solving, we illustrate its use in the context of the CiaoPP system. Using a logic program, we first illustrate how CiaoPP sets up recurrence relations representing the sizes of output arguments of predicates and the cost of such predicates. Then, we show how our novel approach is used to solve a recurrence relation that cannot be solved by CiaoPP.

Example 1

Consider predicate `q/2` in Figure 2, and calls to it where the first argument is bound to a non-negative integer and the second one is a free variable.[1] Upon success of these calls, the second argument is bound to an non-negative integer too. Such calling mode, where the first argument is input and the second one is output, is automatically inferred by CiaoPP (see Hermenegildo et al. (2005) and its references).

The CiaoPP system first infers size relations for the different arguments of predicates, using a rich set of size metrics (see Navas et al. (2007); Serrano et al. (2014) for details). Assume that the size metric used in this example, for the numeric argument `X` is the *actual value* of it (denoted `int(X)`). The system will try to infer a function $\mathtt{S}_\mathtt{q}(x)$ that gives the size of the output argument of `q/2` (the second one), as a function of the size

[1] This set of calls is represented by the "entry" assertion at the beginning of the code, where property `nnegint` stands for the set of non-negative integers.

```
:- entry q/2: nnegint*var.
q(X,0):-
   X=0.
q(X,Y):-
   X>0,
   X1 is X - 1,
   q(X1,Y1),
   q(Y1,Y2),
   Y is Y2 + 1.
```

Fig. 2: A program with a nested recursion.

(x) of the input argument (the first one). For this purpose, the following size relations for $\mathtt{S}_\mathtt{q}(x)$ are automatically set up (the same as the recurrence in Equation 1 used in Section 2 as example).

$$\begin{array}{lcll} \mathtt{S}_\mathtt{q}(x) & = & 0 & \text{if } x = 0 \\ \mathtt{S}_\mathtt{q}(x) & = & \mathtt{S}_\mathtt{q}(\mathtt{S}_\mathtt{q}(x-1)) + 1 & \text{if } x > 0 \end{array} \tag{14}$$

The first and second recurrence correspond to the first and second clauses respectively (i.e., base and recursive cases). Once recurrence relations (either representing the size of terms, as the ones above, or the computational cost of predicates, as the ones that we will see later) have been set up, a solving process is started.

Nested recurrences, as the one that arise in this example, cannot be handled by most state-of-the-art recurrence solvers. In particular, the modular solver used by CiaoPP fails to find a closed-form function for the recurrence relation above. In contrast, the novel approach that we propose obtains the closed form $\hat{\mathtt{S}}_\mathtt{q}(x) = x$, which is an exact solution of such recurrence (as shown in Section 2).

Once the size relations have been inferred, CiaoPP uses them to infer the computational cost of a call to `q/2`. For simplicity, assume that in this example, such cost is given in terms of the number of *resolution steps*, as a function of the size of the input argument, but note that CiaoPP's cost analysis is parametric with respect to resources, which can be defined by the user by means of a rich assertion language, so that it can infer a wide range of resources, besides resolution steps. Also for simplicity, we assume that all builtin predicates, such as arithmetic/comparison operators have zero cost (in practice there is a "trust" assertion for each builtin that specifies its cost as if it had been inferred by the analysis).

In order to infer the cost of a call to `q/2`, represented as $\mathtt{C}_\mathtt{q}(x)$, CiaoPP sets up the following cost relations, by using the size relations inferred previously.

$$\begin{array}{lcll} \mathtt{C}_\mathtt{q}(x) & = & 1 & \text{if } x = 0 \\ \mathtt{C}_\mathtt{q}(x) & = & \mathtt{C}_\mathtt{q}(x-1) + \mathtt{C}_\mathtt{q}(\mathtt{S}_\mathtt{q}(x-1)) + 1 & \text{if } x > 0 \end{array} \tag{15}$$

We can see that the cost of the second recursive call to predicate `p/2` depends on the size of the output argument of the first recursive call to such predicate, which is given by function $\mathtt{S}_\mathtt{q}(x)$, whose closed form $\mathtt{S}_\mathtt{q}(x) = x$ is computed by our approach, as already explained. Plugging such closed form into the recurrence relation above, it can now be solved by CiaoPP, obtaining $\mathtt{C}_\mathtt{q}(x) = 2^{x+1} - 1$.

6 Implementation and Experimental Evaluation

We have implemented a prototype of our novel approach and performed an experimental evaluation in the context of the CiaoPP system, by solving recurrences similar to those generated during static cost analysis, and comparing the results with the current CiaoPP solver as well as with state-of-the-art cost analyzers and recurrence solvers. Our experimental results are summarized in Table 2 and Figure 3, where our approach is evaluated on the benchmarks of Table 1, and where we compare its results with other tools, as described below and in Section 7. Beyond these summaries, additional details on chosen benchmarks are given in paragraph *Evaluation and Discussion*, and full outputs are included in Appendix B.

Prototype. As mentioned earlier, our approach is parameterized by the regression technique employed, and we have instantiated it with both (sparse) linear and symbolic regression. To assess the performance of each individually, as well as their combinations with other techniques, we have developed three versions of our prototype, which appear as the first three columns (resp. bars) of Table 2 (resp. Figure 3, starting from the top). The simplest version of our approach, as described in Section 4.1, is `mlsolve(L)`. It uses linear regression with lasso regularization and feature selection. `mlsolve(L)+domsplit` is the same tool, adding the domain splitting strategy described in Section 4.2. Finally, `mlsolve(S)+domsplit` replaces (sparse) linear regression with symbolic regression.

The overall architecture, input and outputs of our prototypes were already illustrated in Figure 1 and described in Sections 2 and 4. The implementation is written in `Python 3.6/3.7`, using `Sympy` (Meurer et al. 2017) as CAS, `Scikit-Learn` (Pedregosa et al. 2011) for regularized linear regression and `PySR` (Cranmer 2023) for symbolic regression. We use `Z3` (de Moura and Bjørner 2008) as SMT-solver, and `Z3Py` (Z3Py 2023) as interface.

Legend. Columns 3–19 of Table 2 are categorized, from left to right, as 1) the prototypes of this paper's approach, 2) CiaoPP builtin solver, 3) other program analysis tools, and 4) Computer Algebra Systems (CAS) with recurrence solvers. Such solvers also appear in the y-axis of Figure 3, and there is a horizontal bar (among `a-q`) for each of them.

From top to bottom, our benchmarks are organized by category reflecting some of their main features. Benchmarks are manually translated into the input language of each tool, following a best-effort approach. The symbol "`(*)`" corresponds to help provided for certain imperative-oriented tools. More comments will be given in *tools* and *benchmarks* paragraphs below.

For the sake of simplicity, and for space reasons, we report the accuracy of the results by categorizing them using only 4 symbols. "✓" corresponds to an exact solution, "Θ" to a "precise approximation" of the solution, "$-$" to another non-trivial output, and "$\cdot$" is used in all remaining cases (trivial infinite bound, no output, error, timeout, unsupported). Nevertheless, the concrete functions output are included in Appendix B. Figure 3 also shows the number of closed-form functions inferred broken down by these four accuracy categories, in the form of stacked horizontal bars. In addition, when a solution f has superpolynomial growth, we relax our requirements and may write "e^{θ}" whenever $\hat{f}$ is not a precise approximation of f, but $\log \hat{f}$ is a precise approximation of

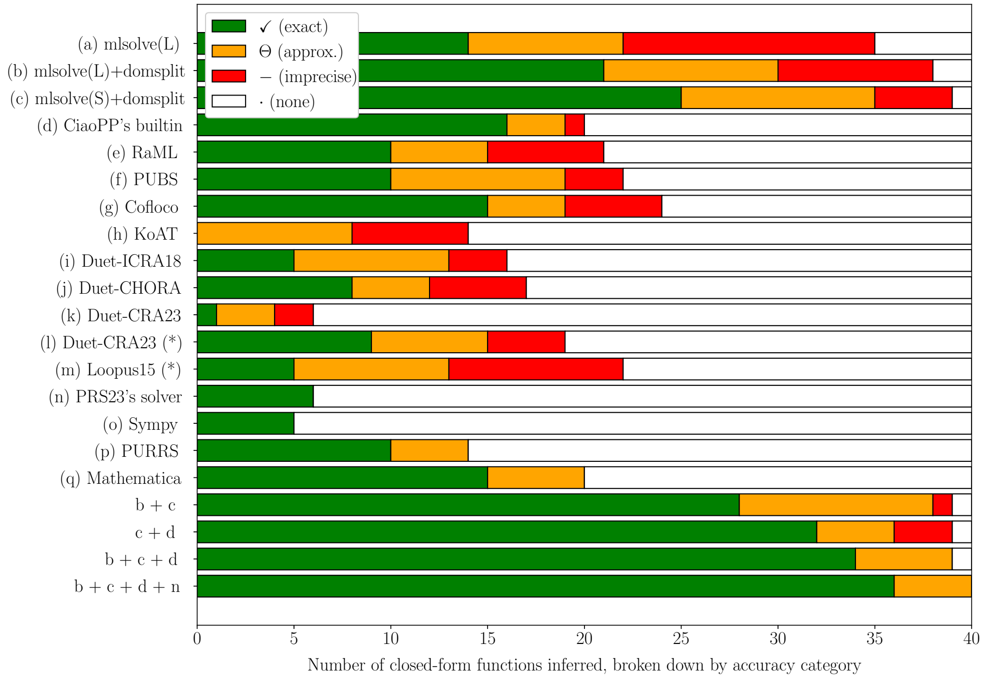

Fig. 3: Comparison of solver tools by accuracy of the result.

$\log f$.[2] We include "e^{θ}" in the "Θ" category in Figure 3. Given the common clash of definitions when using O notation on several variables, we make explicit the notion of approximation used here.

We say that a function $\hat{f}$ is a *precise approximation* of $f : \mathcal{D} \to \mathbb{R}$ with $\mathcal{D} \subset \mathbb{N}^m$ whenever

$$\forall u \in \mathcal{D}^{\mathbb{N}}, \|u_n\| \underset{n\to\infty}{\longrightarrow} \infty \implies \big(n \mapsto \hat{f}(u_n)\big) \in \Theta\big(n \mapsto f(u_n)\big),$$

where Θ is the usual uncontroversial one-variable big theta, and $\|\cdot\|$ is any norm on $\mathbb{R}^k$. In other words, a *precise approximation* is a function which is asymptotically within a constant factor of the approximated function, where "asymptotically" means that some of the input values tend to infinity, but not necessarily all of them. Note that $\max(x, y)$ and $x + y$ belong to the same class, but $(x, y) \mapsto x + y - 1$ is not a precise approximation of the solution of the `merge` benchmark (set $u_n = (n, 0)$): the approximations we consider give "correct asymptotic class in all directions".

Hence, bounds that end up in the "–" category may still be precise enough to be useful in the context of static analysis. We nevertheless resort to this simplification to avoid the presentation of an inevitably complex lattice of approximations. It may also be noted that different tools do not judge quality in the same way. CAS typically aim for *exact* solutions only, while static analysis tools often focus on provable *bounds* (exact or asymptotic), which the notion of approximation used in this table does not require. Once again, we use a single measure for the sake of simplicity.

[2] This accounts for the fact that approximation classes of functions with superpolynomial growth are too thin for our purposes: $x^{1.619}$ is not a precise approximation of $x^{1.618}$.

Table 1: Benchmarks

Category	**Bench**	**Equation**	**Solution**
`scale`	`highdim1`	$\vec{x} = (x_1, \ldots, x_{10})$, $f(\vec{x}) = \begin{cases} i + f(\vec{x}[x_i \leftarrow (x_i - 1)]) & \text{if } x_i > 0 \wedge \forall j < i,\, x_j = 0 \\ 0 & \text{if } \forall i,\, x_i = 0 \end{cases}$	$\sum_{k=1}^{10} kx_k$
	`poly2`	$f_9(x, y)$? (cf. `highdim2`)	xy
	`poly5`	$f_7(t, x, y, z)$? (cf. `highdim2`)	$txyz$
	`poly7`	$f_7(t,x,y,z) + f_9(t,x) + f_7(x,x,x,z) + f_{10}(y) + f_7(z,z,z,z)$? (cf. `highdim2`)	$txyz + tx + x^3z + y + z^4$
	`highdim2`	$\forall 1 \leq i \leq 10,\ f_i(x_i, \ldots, x_{10}) =$ $\begin{cases} f_{i+1}(x_{i+1}, \ldots, x_{10}) + f_i(x_i - 1, x_{i+1}, \ldots, x_{10}) & \text{if } x_i > 0 \\ 0 & \text{if } x_i = 0, \end{cases}$ $f_{11}() = 1$	$f_1(\vec{x}) = \prod_{k=1}^{10} x_k$

`amortized`	`loop_tarjan`	Appropriate encoding of total number of loop iterations of
		`void f(n) {`
		`  i = n; j = 0;`
		`  while(i>0) {`
		`    i--; j++; // push`
		`    while(j>0 && nondet())`
		`      j--; //pop`
		`  }`
		`}`
			$2n$ (worst-case)

	`enqdeq1`	Appropriate encoding of total number of ticks of `queue = ([], []); Do` n `enqueue; Do` n `dequeue` (cf. Figure 6)	$3n$
	`enqdeq2`	Appropriate encoding of total number of ticks of `queue = ([], Lo);` (with `Lo` of length k) `Do` n `enqueue; Do` m `dequeue` (cf. Figure 6)	$\begin{cases} n + m & \text{if } m \leq k \\ 2n + m & \text{if } m > k \end{cases}$

	`enqdeq3`	Appropriate encoding of total number of ticks of
		`queue = ([], []); Do` $2n$ `enqueue;`
		`Do` n `dequeue; Do` n `enqueue; Do` n `dequeue`
		(cf. Figure 6)
			7n

`max-heavy`	`merge-sz`	$f(x,y) = \begin{cases} \max(f(x-1,y), f(x,y-1)) + 1 & \text{if } x > 0 \wedge y > 0 \\ x & \text{if } x > 0 \wedge y = 0 \\ y & \text{if } x = 0 \wedge y > 0 \end{cases}$	$x + y$
	`merge`	$f(x,y) = \begin{cases} \max(f(x-1,y), f(x,y-1)) + 1 & \text{if } x > 0 \wedge y > 0 \\ 0 & \text{if } x = 0 \vee y = 0 \end{cases}$	$\begin{cases} x + y - 1 & \text{if } x > 0 \wedge y > 0 \\ 0 & \text{if } x = 0 \vee y = 0 \end{cases}$
	`open-zip`	$f(x,y) = \begin{cases} f(x-1,y-1) + 1 & \text{if } x > 0 \wedge y > 0 \\ f(x,y-1) + 1 & \text{if } x = 0 \wedge y > 0 \\ f(x-1,y) + 1 & \text{if } x > 0 \wedge y = 0 \\ 0 & \text{if } x = 0 \wedge y = 0 \end{cases}$	$\max(x, y)$
	`s-max`	$f(x,y) = \begin{cases} \max(y, f(x-1,y)) + 1 & \text{if } x > 0 \\ y & \text{if } x = 0 \end{cases}$	$x + y$
	`s-max-1`	$f(x,y) = \begin{cases} \max(y, f(x-1,y+1)) + 1 & \text{if } x > 0 \\ y & \text{if } x = 0 \end{cases}$	$2x + y$
`imp.`	`incr1`	$f(x) = \begin{cases} 1 + f(x+1) & \text{if } x < 10 \\ 1 & \text{if } x \geq 10 \end{cases}$	$\max(1, 11 - x)$
	`noisy_strt1`	$f(x) = \begin{cases} 0 & \text{if } x = 0 \vee x = 20 \\ 1 + f(x-1) & \text{if } x \neq 0 \wedge x \neq 20 \end{cases}$	$\begin{cases} x & \text{if } x < 20 \\ x - 20 & \text{if } x \geq 20 \end{cases}$
	`noisy_strt2`	$f(x) = \begin{cases} 0 & \text{if } x = 0 \vee x = 65536 \\ 1 + f(x-1) & \text{if } x \neq 0 \wedge x \neq 65536 \end{cases}$	$\begin{cases} x & \text{if } x < 65536 \\ x - 65536 & \text{if } x \geq 65536 \end{cases}$
	`multiphase1`	$f(i,n,r) = \begin{cases} 0 & \text{if } i \geq n \\ 1 + f(0, n, r-1) & \text{if } i < n \wedge r > 0 \\ 1 + f(i+1, n, r) & \text{if } i < n \wedge r = 0 \end{cases}$	$\begin{cases} 0 & \text{if } i \geq n \\ n + r & \text{if } i < n \wedge r > 0 \\ n - i & \text{if } i < n \wedge r = 0 \end{cases}$
	`lba_ex_viap`	$f(x,y,c) = \begin{cases} 1 + f(x+1, y+1, c) & \text{if } x + y < c \\ 0 & \text{if } x + y \geq c \end{cases}$	$\max\left(0, \left\lceil \frac{c-(x+y)}{2} \right\rceil\right)$

Table 1: Benchmarks (continued)

Category	**Bench**	**Equation**	**Solution**
`nested`	`nested`	$f(x) = \begin{cases} f(f(x-1))+1 & \text{if } x > 0 \\ 0 & \text{if } x = 0 \end{cases}$	x
	`nested_case`	$f(x,b) = \begin{cases} f(f(x-1,b),b)+1 & \text{if } x > 0 \land b = 0 \\ x + f(x-1,b) & \text{if } x > 0 \land b > 0 \\ 0 & \text{if } x = 0 \end{cases}$	$\begin{cases} x & \text{if } b = 0 \\ \frac{1}{2}x^2 + \frac{1}{2}x & \text{if } b > 0 \end{cases}$
	`nested_div`	$f(x) = \begin{cases} f\left(f\left(\left\lfloor \frac{x}{2} \right\rfloor\right)\right)+1 & \text{if } x > 0 \\ 0 & \text{if } x = 0 \end{cases}$	$\begin{cases} 3 & \text{if } x \geq 4 \\ 2 & \text{if } x = 3 \\ x & \text{if } x \leq 2 \end{cases}$
	`mccarthy91`	$f(x) = \begin{cases} f(f(x+11)) & \text{if } x \leq 100 \\ x - 10 & \text{if } x \geq 101 \end{cases}$	$\begin{cases} 91 & \text{if } x \leq 100 \\ x - 10 & \text{if } x \geq 101 \end{cases}$
	`golomb`	$f(x) = \begin{cases} f(x - f(x-1))+1 & \text{if } x > 0 \\ 1 & \text{if } x = 0 \end{cases}$	$\left\lfloor \frac{1+\sqrt{8x}}{2} \right\rfloor$
`misc.`	`div`	$f(x,y) = \begin{cases} f(x-y,y)+1 & \text{if } x \geq y \land y > 0 \\ 0 & \text{if } x < y \land y > 0 \end{cases}$	$\left\lfloor \frac{x}{y} \right\rfloor$
	`sum-osc`	$f(x,y) = \begin{cases} f(x-1,y)+1 & \text{if } x > 0 \land y > 0 \\ f(x+1,y-1)+y & \text{if } x = 0 \land y > 0 \\ 1 & \text{if } y = 0 \end{cases}$	$\begin{cases} 1 & \text{if } y = 0 \\ \frac{1}{2}y^2 + \frac{3}{2}y + x & \text{if } y > 0 \end{cases}$
	`bin_search`	$f(x) = \begin{cases} f\left(\left\lfloor \frac{x}{2} \right\rfloor\right)+1 & \text{if } x > 0 \\ 0 & \text{if } x = 0 \end{cases}$	$\begin{cases} 0 & \text{if } x = 0 \\ 1 + \lfloor \log_2(x) \rfloor & \text{if } x > 0 \end{cases}$
	`qsort_best`	$f(x) = \begin{cases} f\left(\left\lfloor \frac{x-1}{2} \right\rfloor\right) + f\left(\left\lceil \frac{x-1}{2} \right\rceil\right) + x & \text{if } x > 0 \\ 1 & \text{if } x = 0 \end{cases}$	? $\sim x\log_2(x)$
	`prs23_1`	$f(0) = 1$ $f(n+1) = \begin{cases} 2f(n) & \text{if } f(n) < 500 \\ f(n) & \text{if } f(n) \geq 500 \end{cases}$ $g(0) = 1$ $g(n+1) = \begin{cases} g(n) & \text{if } f(n) < 500 \\ g(n)+3 & \text{if } f(n) \geq 500 \land f(n) > g(n) \\ g(n)-1 & \text{if } f(n) \geq 500 \land f(n) \leq g(n) \end{cases}$	$f(n) = \begin{cases} 2^n & \text{if } n < 9 \\ 512 & \text{if } n \geq 9 \end{cases}$ $g(n) = \begin{cases} 1 & \text{if } n < 9 \\ 3n - 26 & \text{if } 9 \geq n < 180 \\ 514 - (n\%4) & \text{if } n \geq 180 \end{cases}$
`CAS-style`	`exp1`	$f(n) = \begin{cases} 2f(n-1) & \text{if } n > 0 \\ 3 & \text{if } n = 0 \end{cases}$	3×2^n
	`exp2`	$f(n) = \begin{cases} 2f(n-1)+1 & \text{if } n > 0 \\ 3 & \text{if } n = 0 \end{cases}$	$4 \times 2^n - 1$
	`exp3`	$f(x,y) = g(h(x,y))$ $g(z) = \begin{cases} 2g(z-1) & \text{if } z > 0 \\ 1 & \text{if } z = 0 \end{cases}$ $h(x,y) = \begin{cases} h(x-1,y)+1 & \text{if } x > 0 \\ y & \text{if } x = 0 \end{cases}$	$f(x,y) = 2^{x+y}$
	`fib`	$f(n) = \begin{cases} f(n-1)+f(n-2) & \text{if } n > 1 \\ 1 & \text{if } n = 1 \\ 0 & \text{if } n = 0 \end{cases}$	$\frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right)$
	`harmonic`	$f(n) = \begin{cases} f(n-1)+\frac{1}{n} & \text{if } n > 0 \\ 0 & \text{if } n = 0 \end{cases}$	$\sum_{k=1}^{n} \frac{1}{k}$ $\sim \ln(n)$
	`fact`	$f(n) = \begin{cases} nf(n-1) & \text{if } n > 0 \\ 1 & \text{if } n = 0 \end{cases}$	$n!$ $= e^{\Theta(n \log n)}$
	`cas_st1`	$f(n) = \begin{cases} 2nf(n-1) & \text{if } n > 0 \\ 1 & \text{if } n = 0 \end{cases}$	$2^n n!$ $= e^{\Theta(n \log n)}$
	`cas_st2`	$f(n) = \begin{cases} n^2 f(n-1) & \text{if } n > 0 \\ 1 & \text{if } n = 0 \end{cases}$	$(n!)^2$ $= e^{\Theta(n \log n)}$
	`cas_st3`	$f(n) = \begin{cases} nf(n-1)+1 & \text{if } n > 0 \\ 1 & \text{if } n = 0 \end{cases}$	$n! \sum_{k=0}^{n} \frac{1}{k!}$ $\sim en!$
	`cas_st4`	$f(n) = \begin{cases} 2f(n-1)+\frac{1}{n} & \text{if } n > 0 \\ 0 & \text{if } n = 0 \end{cases}$	$2^n \sum_{k=1}^{n} \frac{1}{2^k k}$ $\sim \ln(2) \times 2^n$
	`cas_st5`	$f(n) = \begin{cases} 2nf(n-1)+\frac{1}{n} & \text{if } n > 0 \\ 0 & \text{if } n = 0 \end{cases}$	$2^n n! \sum_{k=1}^{n} \frac{1}{k 2^k k!}$ $\sim (\mathrm{Ei}(1/2) + \ln(2) - \gamma) \cdot 2^n n!$

Table 2: Experimental evaluation and comparison

Category	Bench	mlsolve(L)	mlsolve(L)+domsplit	mlsolve(S)+domsplit	Ciao's builtin	RaML	PUBS	Cofloco	KoAT	Duet-ICRA18	Duet-CHORA	Duet-CRA23	Duet-CRA23 (*)	Loopus15 (*)	PRS23's solver	Sympy	PURRS	Mathematica
scale	highdim1	✓	✓	—	·	✓	Θ	✓	Θ	·	·	·	·	✓	·	·	·	·
	poly2	✓	✓	✓	✓	✓	✓	✓	·	·	·	·	✓	✓	·	·	·	✓
	poly5	✓	✓	✓	✓	✓	✓	✓	·	·	·	·	·	✓	·	·	·	✓
	poly7	—	—	✓	✓	✓	✓	✓	·	·	·	·	·	✓	·	·	·	✓
	highdim2	·	·	—	✓	·	✓	✓	·	·	·	·	·	—	·	·	·	✓
amortized	loop_tarjan	✓	✓	✓	·	✓	✓	✓	Θ	✓	✓	✓	✓	Θ	·	·	·	·
	enqdeq1	✓	✓	✓	·	✓	·	Θ	·	·	·	·	✓	—	·	·	·	·
	enqdeq2	—	—	Θ	·	Θ	·	Θ	·	·	·	·	Θ	—	·	·	·	·
	enqdeq3	✓	✓	✓	·	Θ	·	Θ	·	·	·	·	Θ	—	·	·	·	·
max-heavy	merge-sz	✓	✓	✓	✓	✓	Θ	✓	Θ	✓	✓	·	✓	Θ	·	·	·	·
	merge	—	✓	✓	—	—	✓	✓	—	—	—	·	—	—	·	·	·	·
	open-zip	✓	✓	✓	·	Θ	Θ	✓	Θ	Θ	Θ	·	·	Θ	·	·	·	·
	s-max	✓	✓	✓	Θ	✓	Θ	✓	Θ	Θ	✓	·	✓	Θ	·	·	·	·
	s-max-1	✓	✓	✓	·	✓	✓	✓	Θ	Θ	✓	·	Θ	Θ	·	·	·	·
imp.	incr1	—	✓	✓	·	·	✓	✓	—	Θ	✓	·	✓	✓	·	·	·	·
	noisy_strt1	·	✓	✓	Θ	Θ	Θ	✓	Θ	Θ	Θ	Θ	Θ	Θ	✓	·	Θ	Θ
	noisy_strt2	Θ	Θ	Θ	Θ	Θ	Θ	✓	Θ	Θ	·	Θ	Θ	Θ	✓	·	Θ	Θ
	multiphase1	—	✓	✓	·	·	·	✓	—	—	·	—	—	—	·	·	·	·
	lba_ex_viap	·	Θ	Θ	·	·	Θ	Θ	—	Θ	Θ	Θ	Θ	Θ	·	·	·	·
nested	nested	✓	✓	✓	·	✓	·	·	·	Θ	·	·	·	·	·	·	·	·
	nested_case	—	✓	✓	·	·	·	·	·	·	·	·	·	·	·	·	·	·
	nested_div	—	—	✓	·	—	·	·	·	·	·	·	·	·	·	·	·	·
	mccarthy91	·	—	✓	·	·	·	—	·	·	·	·	·	·	·	·	·	·
	golomb	Θ	Θ	—	·	—	·	·	·	·	·	·	·	·	·	·	·	·
misc.	div	✓	✓	✓	·	·	—	—	—	—	—	·	·	—	·	·	·	Θ
	sum-osc	—	✓	Θ	·	—	—	—	—	·	·	·	·	·	·	·	·	·
	bin_search	Θ	✓	✓	·	—	Θ	·	·	·	—	·	—	—	·	·	Θ	Θ
	qsort_best	—	Θ	—	·	—	—	·	·	·	—	·	·	·	·	·	Θ	Θ
	prs23_1	·	·	·	·	·	·	·	·	·	—	—	—	—	✓	·	·	·
CAS-style	exp1	✓	✓	✓	✓	·	✓	·	·	✓	✓	·	✓	·	✓	✓	✓	✓
	exp2	✓	✓	Θ	✓	·	✓	·	·	✓	✓	·	✓	·	✓	✓	✓	✓
	exp3	—	—	✓	✓	·	·	·	·	✓	✓	·	✓	·	·	·	·	✓
	fib	—	—	e^θ	✓	·	e^θ	·	·	·	e^θ	·	·	·	✓	✓	✓	✓
	harmonic	—	—	Θ	✓	·	·	·	·	·	·	·	·	·	·	·	✓	✓
	fact	Θ	Θ	✓	✓	·	·	—	·	·	·	·	·	·	·	✓	✓	✓
	cas_st1	—	—	✓	✓	·	·	—	·	·	·	·	·	·	·	✓	✓	✓
	cas_st2	e^θ	e^θ	✓	✓	·	·	·	·	·	·	·	·	·	·	·	✓	✓
	cas_st3	Θ	Θ	Θ	✓	·	·	·	·	·	·	·	·	·	·	·	✓	✓
	cas_st4	Θ	Θ	Θ	✓	·	·	·	·	·	·	·	·	·	·	·	✓	✓
	cas_st5	e^θ	e^θ	Θ	✓	·	·	·	·	·	·	·	·	·	·	·	✓	✓
Total number of ✓ (/40)		14	21	25	16	10	10	15	0	5	8	1	9	5	6	5	10	15

Experimental setup. Chosen hyperparameters are given for the sake of completeness.

For regularized linear regression (Algorithm 1), we set $\epsilon = 0.05$ for feature selection, $k = 2$ for cross-validation, which will be performed to choose the value of λ in a set Λ of 100 values taken from the interval $[0.001, 1]$. (It is also possible to construct the entire regularization path for the lasso, which is piecewise linear, and obtain the solution for every value of $\lambda \in \mathbb{R}$ (Hastie et al. 2015), but here we opted to provide an explicit set of possible values Λ.) If we use domain splitting, the choice of regression domain is as described in Section 4.2 (fit separately on the subdomains corresponding to each clause), but if we do not, we choose $\mathcal{D}$ to be (a power of) $\mathbb{N}_{>0}$ (this avoids the pathological behavior in 0 of some candidates). Verification is still performed on the whole domain of the equation, i.e., $\varphi_{pre} = \vee_i \varphi_i$. For each run of Algorithm 1, $n = 100$ input values in $\mathcal{D}$ are generated, with each variable in $[0, b]$, where $b \in \{20, 10, 5, 3\}$ is chosen so that generation ends within a timeout of 2 seconds.[3]

Candidate functions are selected as follows. For each number m of input variables we choose sets $\mathcal{F}_{\text{small}}$, $\mathcal{F}_{\text{medium}}$, $\mathcal{F}_{\text{large}}$ representative of common complexities encountered in cost analysis. They are chosen explicitly for $m = 1$ and 2, and built as combinations of base functions for $m \geq 3$. Regression is performed for each set, with a timeout of 10 seconds,[4] and the best solution is chosen. Chosen sets are indicated in Figures 4 and 5.

m	S_{small}	S_{medium}	S_{large}
1	$\{x, x^2\}$	$\{\lceil \log_2(x) \rceil, \lfloor \sqrt{x} \rfloor, x\lceil \log_2(x) \rceil\}$	$\{\lfloor \log_2(x) \rfloor, x\lfloor \log_2(x) \rfloor, 2^x, 5^x, x \cdot 2^x, x!\}$
2	$\{x, y\}$	$\{x^2, xy, y^2, x^2y, xy^2, x^2y^2\}$	$\left\{ \begin{array}{l} \lfloor \frac{x}{y} \rfloor, \lfloor \frac{y}{x} \rfloor, \lceil \frac{x}{y} \rceil, \lceil \frac{y}{x} \rceil, 2^x, 2^y, \max(x, y), \\ \lceil \log_2(x) \rceil, \lfloor \log_2(x) \rfloor, x\lceil \log_2(x) \rceil, x\lfloor \log_2(x) \rfloor, \\ \lceil \log_2(y) \rceil, \lfloor \log_2(y) \rfloor, y\lceil \log_2(y) \rceil, y\lfloor \log_2(y) \rfloor, \end{array} \right\}$

Fig. 4: Candidate functions for linear regression in dimension $\leq$ 2. $\mathcal{F}_{\text{small}} = S_{\text{small}}$, $\mathcal{F}_{\text{medium}} = \mathcal{F}_{\text{small}} \cup S_{\text{medium}}$, $\mathcal{F}_{\text{large}} = \mathcal{F}_{\text{medium}} \cup S_{\text{large}}$. Lambdas omitted for conciseness.

$\tilde{\mathcal{F}}_{\text{small}} = \{x\}$, $\tilde{\mathcal{F}}_{\text{medium}} = \{x, x^2\}$, $\tilde{\mathcal{F}}_{\text{large}} = \{x, x^2, x^3\}$. For $s \in \{\text{small}, \text{medium}, \text{large}\}$,

$$\mathcal{F}_s = \bigcup_{\substack{k \in \mathbb{N} \\ k \leq m + |\tilde{\mathcal{F}}_s|}} \left\{ \prod_{i=1}^{k} f_i(x_{\psi(i)}) \,\middle|\, \forall i\, f_i \in \tilde{\mathcal{F}}_s,\ \psi : [1, k] \to [1, m],\ \forall i, j \left(i \neq j \implies (f_i, \psi(i)) \neq (f_j, \psi(j))\right) \right\}$$

Fig. 5: Candidate functions for linear regression in dimension ≥ 3. $\mathcal{F}_s$ is defined by combination of simpler base functions, as bounded products of applications of base functions to individual input variables. For example, with $m = 3$, $\mathcal{F}_{\text{small}} = \{x, y, z, xy, xz, yx, xyz\}$.

In the case of symbolic regression, instead of candidates, we need to choose operators allowed in expression trees. We choose binary operators $\{+, -, \max, \times, \div, (\cdot)^{(\cdot)}\}$ and unary operators $\{\lfloor \cdot \rfloor, \lceil \cdot \rceil, (\cdot)^2, (\cdot)^3, \log_2(\cdot), 2^{(\cdot)}, (\cdot)!\}$. Nodes, including leaves, have default cost 1. Ceil and floor have cost 2 and binary exponentiation has cost 3. We have 45 populations of 33 individuals, and run `PySR` for 40 iterations and otherwise default options.

In all cases, to account for randomness in the algorithms, experiments are run twice and the best solution is kept. Variability is very low when exact solutions can be found.

[3] This `Python` prototype does not use tabling, and evaluation can be long for recurrences with non-linear recursion. Much higher values of b could be chosen with a less naive evaluation strategy.

[4] For each subdomain in the case of `domsplit`.

Experiments are run on a small Linux laptop with `1.1GHz Intel Celeron N4500` CPU, `4 GB`, `2933 MHz DDR4` memory. Lasso regression takes 1.4s to 8.4s with mean 2.5s (on each candidate set, each benchmark, excluding equation evaluation and regression timeouts[5]), showing that our approach is reasonably efficient. Symbolic regression (on each subdomain, no timeout, without early exit conditions) takes 56s to 133s with mean 66s. It may be noted that these results correspond to initial prototypes, and that experiments focus on testing the accuracy of our approach: many avenues for easy optimizations are left open, e.g., stricter (genetic) resource management in symbolic regression.

Benchmarks. We present 40 benchmarks, i.e., (systems of) recurrence equations, extracted from our experiments and organized in 7 categories that test different features. We believe that such benchmarks, evenly distributed among the categories considered, are reasonably representative and capture the essence of a much larger set of benchmarks, including those that arise in static cost analysis, and that their size and diversity provide experimental insight on the scalability of our approach.

Category `scale` tests how well solvers can handle increasing dimensions of the input space, for simple linear or polynomial solutions. The intuition for this test is that (global and numerical) machine learning methods may suffer more from the curse of dimensionality than (compositional and symbolic) classical approaches. We also want to evaluate when the curse will start to become an issue. Here, functions are multivariate, and most benchmarks are systems of equations, in order to easily represent nested loops giving rise to polynomial cost.

Category `amortized` contains examples inspired by the amortized resource analysis line of work illustrated by `RaML`. Such problems may be seen as situations in which the worst-case cost of a function is not immediately given by the sum of worst-case costs of called functions, requiring some information on intermediate state of data structures, which is often tracked using a notion of *potential* (Tarjan 1985). For the sake of comparison, we also track those intermediate states as *sizes*, and encode the corresponding benchmarks in an equivalent way as systems of equations on *cost* and *size*, following an approach based[6] on (Debray and Lin 1993). `loop_tarjan` comes from (Sinn et al. 2017), whereas `enqdeq1-3` are adapted from (Hoffmann 2011). In each example, the cost function of the entry predicate admits a simple representation, whereas the exact solutions for intermediate functions are more complicated.

Category `max-heavy` contains examples where either the equation or (the most natural expression of) the solution contain max operators. Such equations may typically arise in the worst-case analysis of non-deterministic programs (or complex deterministic programs simplified to simple non-deterministic programs during analysis), but may also appear naturally in size equations of programs with tests.

Category `imp.` tests features crucial to the analysis of imperative programs, that tend to be of lesser criticality in other paradigms. Those features can be seen as features

[5] Timeouts occur for all base function sets on only 1 of 40 benchmarks, `highdim2`. They also occur on some functions sets for 2 additional benchmarks, `highdim1` ("medium" and "large" sets) and `multiphase1` ("large" base function set).

[6] In Debray and Lin (1993), the authors set up the equations at the same time as they reduce them using intermediate approximate solutions. Here, we are referring to the full system of size/cost equations representing the program, without simplifications/reductions.

```
enq((Li, Lo), X, ([X|Li], Lo)) :-
    tick(1).

deq((Li, [X|Lo]), X, (Li, Lo)) :-
    tick(1).
deq(([], []),    _X, ([], [])) :-
    tick(1).
deq((Li, []),     X, ([], Lo)) :-
    length(Li, N), tick(N),
    reverse(Li, [X|Lo]),
    tick(1).
```

```
menq(Q, 0, Q).
menq(Q, N, QR) :- N>0,
    N1 is N-1,
    enq(Q, _, Q1),
    menq(Q1, N1, QR).

mdeq(Q, 0, Q).
mdeq(Q, N, QR) :- N>0,
    N1 is N-1,
    deq(Q, _, Q1),
    mdeq(Q1, N1, QR).
```

```
enqdeq1(N) :-
    menq(([],[]), N, Q), mdeq(Q, N, _).
enqdeq2(N, M, K) :- length(Lo, K),
    menq(([],Lo), N, Q), mdeq(Q, M, _).
enqdeq3(N) :- N2 is 2*N,
    menq(([],[]), N2, Q1),
    mdeq(Q1, N, Q2),
    menq(Q2, N, Q3),
    mdeq(Q3, N, _).
```

Fig. 6: Prolog encoding of the `enqdeq` benchmarks, inspired from Section 2.1 of Hoffmann (2011) introducing amortized analysis, where a queue datastructure is implemented as two lists that act as stacks. We encode the `enqdeq` problems for each tool following a best-effort approach. Recurrence equations are set up in terms of compositions of cost and size functions.

of loops (described as challenges in recent work including Flores-Montoya (2017) and Sinn et al. (2017)), such as increasing variables (corresponds to recursive call with increasing arguments), noisy/exceptional behavior (e.g., early exit), resets and multiphase loops.

Category `nested` contains examples with nested recursion, a class of equations known for its difficulty (Tanny 2013), that can even be used to prove undecidability of existence of solutions for recurrence equations (Celaya and Ruskey 2012). Such equations are extremely challenging for "reasoning-based" approaches that focus on the structure of the equation itself. However, focusing on the solution rather than the problem (to put it another way, on semantics rather than syntax), we may notice that several of these equations admit simple solutions, that may hence be guessed via regression or template methods. We include the famous McCarthy 91 function (Manna and McCarthy 1970).

Category `misc.` features examples with arithmetic (euclidean division), sublinear (logarithmic) growth, deterministic divide-and-conquer, and alternating activation of equations' clauses. We are not aware of simple closed-forms for the solution of `qsort_best`, although its asymptotic behavior is known. Hence, we do not expect tools to be able to infer exact solutions, but rather to aim for good approximations and/or bounds. Example `prs23_1` is given as a motivating example in Wang and Lin (2023) for studying eventually periodic conditional linear recurrences, and displays a particular difficulty: cases are defined by conditions on results of recursive calls rather than simply on input values.

Finally, category `CAS-style` contains several examples from CAS's problem domain, i.e., that can be reduced to unconditional single variable equations with a single recursive case of shape $f(n) = a(n) \times f(n-1) + b(n)$, with a and b potentially complicated, e.g., hypergeometric. This is quite different from the application case of program cost analysis, where we have multiple variables, conditionals are relevant, but recursive cases are closer to (combinations of) $f(n) = a \times f(\phi(n)) + b(n)$, with a constant but $\phi(n)$ non-trivial.

In the case of a system of equations, we only evaluate the solution found for the topmost predicate (the entry function). It may be noted that all of our equations admit a single solution $f : \mathcal{D} \to \mathbb{R}$ where $\mathcal{D} \subset \mathbb{N}^m$ is defined by $\varphi_{pre} = \vee_i \varphi_i$, and that the evaluation strategy (defined in Section 3) terminates everywhere on $\mathcal{D}$. Precise handling and evaluation of partial solutions and non-deterministic equations is left for future work.

Tools and benchmark translation. Obtaining meaningful comparisons of static analysis tools is a challenge. Different tools analyze programs written in different languages, discrepancies from algorithms to implementations are difficult to avoid, and tools are particularly sensitive to representation choices: multiple programs, resp., equations, may have the same (abstract) semantics, resp., solutions, but be far from equally easy to analyze. In order to compare tools anyway, a common choice is to use off-the-shelf translators to target each input language, despite inevitable and unequal accuracy losses caused by those automated transformations. In this paper, we adopt a different approach: each benchmark is written separately in the input format of each tool, aiming for representations adapted to each of them, following a best-effort approach. This is motivated by the lack of available translation tools, by the loss of accuracy observed for those we could find, and by the desire to nonetheless offer a broad comparison with related work.

We cannot list all choices made in the translation process, and only present the most important ones. As a guiding principle, we preserve as much as possible the underlying numerical constraints and control-flow defined by each benchmark, whether the problem is being encoded as a program or a recurrence equation. In principle, it should be straightforward to translate between these representations: a system of recurrence equations may directly be written as a numerical program with recursive functions. However, there is an obstruction, caused by missing features in the tools we evaluated. When it is sufficiently easy to rewrite a benchmark in order to avoid the missing feature, we do so, but otherwise we have to deem the problem unsupported.

Main missing features include support for systems of equations, multiprocedural programs, multivariate equations, loop structures beyond a single incremented index variable, non-determinism, complex numerical operations (polynomials, divisions, rounding functions, but also maximum and minimum), and finally support for general recursion, including nested calls and non-linear recursion.

For tools focusing on loops, with limited support for multiprocedural programs, we perform the mechanical translation of linear tail-recursion to loops when possible, even if this might be seen as a non-trivial help. Such cases are indicated by "`(*)`" in our table. This includes `Loopus`, which analyzes C and LLVM programs without recursive functions, and `Duet-CRA23`, i.e., the 2023 version of `duet`'s `cra` analysis.[7] Since `Duet-CRA23` is still able to perform some interprocedural analysis, we report its results on recursive-style benchmarks (column without "`(*)`"), but it gives better results when this imperative translation is applied.[8] We sometimes attempt less mechanical transformations to loops,

[7] Main branch, tested as `https://github.com/zkincaid/duet/tree/7a5bb0fad9`, May 25, 2023.

[8] `Duet`'s README indicates that the discontinued feature ICRA for analysis of general recursive programs may still be found in the `Newton-ark2` branch. Unfortunately, we were neither able to get it (`008d278f52`, May 2020) to run, nor the most recent artifact with this feature (Kincaid et al. 2019), so we resort to the previous one (Kincaid et al. 2018), in Column `Duet-ICRA18`. Recursive functions are also supported by a more template-based approach (Breck et al. 2020), tested in column `Duet-CHORA`.

e.g., for `fib`. The transformation discussed in this paragraph could also have been applied to `KoAT`,[9,10] but we do not report full results for conciseness.

When support for explicit max operators is limited or absent, we transform deterministic programs with max into non-deterministic programs, in a way that preserves upper bounds, but in general not lower bounds.[11] This transformation is not needed for `mlsolve`, which accepts any nested expression with max and min, but is required for `Ciao`,[12] `PUBS`, `Cofloco`, `Loopus` and all versions of `duet`. It is applied for `RaML` too, using a trick, as we did not find a native construct for non-determinism. Finally, this cannot be applied or does not help to obtain results for `PRS23`, `Sympy`, `PURRS` and `Mathematica`.

Evaluation and Discussion. Overall, the experimental results are quite promising, showing that our approach is accurate and efficient, as well as scalable. We first discuss the stacked horizontal bar chart in Figure 3, which offers a graphical and comprehensive overview of the accuracy of our approach in comparison to state-of-the-art tools. We then provide comments on Table 2, which presents more detailed information.

As mentioned earlier, bars `a-c` (the first three from the top) of Figure 3 represent the three implemented versions of our approach. We can see that in general, for the benchmark set considered, which we believe is reasonably representative, each of these versions individually outperforms any of the other state-of-the-art solvers (included in either cost analyzers or CASs), which are represented by bars `d-q`. Bar `b` + `c` represents the full version of our approach, which combines both regression methods, lasso (`b`) and symbolic (`c`), and the domain splitting strategy. The three bars at the bottom of the chart are included to assess how the combination of our machine learning-based solver with other solvers would potentially improve the accuracy of the closed-forms obtained. The idea is to use it as a complement of other back-end solvers in a higher-level combined solver in order to obtain further accuracy improvements. We can see that the combination (bar `b` + `c` + `d`) of CiaoPP's builtin solver with the full version of our approach mentioned above would obtain exact solutions for 85% of the benchmarks, and accurate approximations for the rest of them, except for one benchmark. As shown in Table 2, this benchmark is `prs23_1`, which can only be solved by `PRS23`'s solver. For our set of benchmarks, the best overall accuracy would be achieved by the combination represented by the bottom bar, which adds `PRS23`'s solver to the combination above.

We now comment on the experimental results of Table 2, category by category.

For category `scale`, we notice that `mlsolve` can go up to dimension 4 without much trouble (actually up to 7, i.e., querying for f_4 in `highdim2`, and failing after that). The impact of the curse of dimensionality on our machine learning approach thus appears tolerable, and comparable to that on the template-based `RaML`, which timeouts reaching dimension 7, i.e., for f_4 in `highdim2`, allowing potentials to be polynomials of degree 7.

[9] We additionally would have needed to delay cost evaluation to the end, working on size equations instead. It does improve the results in the case where `KoAT` is limited by its lack of support for recursivity (discontinued `Com_2` construct), e.g., for `exp1` or `fib`.

[10] KoAT version tested: `https://github.com/aprove-developers/KoAT2-Releases/tree/c1c5290109`, May 10, 2023. ITS inputs, default options `koat2 analyse -i <filename>`.

[11] For example, a recursive call `return` $\max(f(x-1,y), f(x,y-1))$ may be replaced by `if(nondet()) return` $f(x-1,y)$ `else return` $f(x,y-1)$.

[12] Several size and cost analyses are available in Ciao. Here, we report on the one implemented in `infercost`, available via `analyze(steps_ualb)`, run on numerical programs encoding the equations.

PUBS and Cofloco perform well and quickly in this category, showing the value of compositional and symbolic methods. Surprisingly, Loopus performs well only until dimension 10 where an approximation is performed, and duet gives nearly no non-trivial answer. The benchmarks being naturally presented in a multivariate recursive or nested way, they are difficult to encode in PRS23, Sympy and PURRS. The polynomial (locally monovariate) benchmarks are solved by Mathematica, if we provide the order in which subequations must be solved. The benchmarks are similarly difficult to encode in (cost-based) KoAT, and helping it with manual control-flow linearization did not suffice to get "Θ" results.

For category amortized, RaML performs well, but perhaps not as well as could have been imagined. For enqdeq1,2,3, it outputs bounds $[2n, 3n]$, $[n+m, 2n+m]$ and $[5n, 8n]$, respectively. The simplicity of the solutions allows mlsolve to perform well (using the full size and cost recurrence equation encoding discussed above, cf. Debray and Lin (1993), Footnote 6 and Figure 6). This performance can be attributed to the fact that mlsolve does not need to find and use the complicated cost expressions of the intermediate predicates: only full execution traces starting from the entry point are considered. Note that Cofloco also obtains reasonable results, making use of its disjunctive analysis features (visible with the -conditional_ubs option), although it loses accuracy in several places, and outputs linear bounds with incorrect constants. After simplification of its output, Duet-CRA23 obtains $3n$, $2n+m$ and $8n$ on enqdeq, but it may be noted that writing such benchmarks in an imperative fashion may be seen as unreasonable help to the control-flow analysis. Loopus obtains only quadratic bounds on the enqdeq, and no other tool obtains non-trivial bounds. loop_tarjan is handled by nearly all code analysis tools, although surprisingly Loopus loses a minor amount accuracy and outputs $2n+1$ instead of $2n$. It may be noted that no tool gets the exact solution for enqdeq2. For mlsolve, this hints at the fact that improved domain splitting strategies may be beneficial, as they may allow it to discover the $m \leq k$ constraint, which is not directly visible in the equations.

For category max-heavy, we simply observe that support for max operators is rare, although some disjunctive analysis and specific reasoning can be enough to get exact results after the max → nondet transformation, as is the case for Cofloco. Many tools can infer linear bounds for these benchmarks, without being able to get the correct coefficients. More challenging benchmarks could be created using nested min-max expressions and non-linear solutions to separate more tools.

In category imp., where examples are inspired from imperative problems, we notice that the domsplit extension is crucial to allow mlsolve to deal with even basic for loops (benchmark incr1, which also encodes the classical difficulty of recurrence-based methods with increasing arguments). Cofloco performs best in this category thanks to its control-flow refinement features. For noisy_strt, several tools notice the two regimes but struggle to combine them correctly and precisely. PRS23 gets the exact solution, a testimony to its precise (but monovariate) phase separation. As expected, mlsolve fails on noisy_strt2 even when it succeeds on noisy_strt1, misled by the late change of behavior, invisible to naive input generation, although counter-examples to the x solution are produced. Such counter-examples may be used in future work to guide the regression process and learn better subdomains. Given the known limitations of RaML on integers, some inequalities that are natural in an imperative setting are difficult to encode without accuracy loss. For tools that solve `lba_ex_viap`,[13] small rounding errors appear.

Category `nested` is composed of particularly challenging problems for which no general solving method is known, although low (model) complexity solutions exist for particular cases, giving a strong advantage to the `mlsolve` approach. It may be noted that `RaML` also gets the exact solution for the `nested` benchmark and that `Duet-ICRA18` obtains the nearly exact $\max(1, x)$ solution. `mlsolve(S)+domsplit` performs best. It sometimes uses non-obvious expressions to fit the solution, such as ceil(2.75–2.7/x) for the $x > 0$ subdomain on `nested_div`. Although no tool solves the `golomb` recurrence in our experimental setup, it may be noted that `mlsolve(S)` finds the exact solution when $\sqrt{\cdot}$ is added to the list of allowed operators, showing the flexibility of the approach.

In category `misc.`, division, logarithm and complex control-flow, which are typically difficult to support, give an advantage to the `mlsolve` approach, as control-flow has low impact and operators can easily be included. It must be noted that our definition of "precise approximation" makes linear bounds insufficient for the division and logarithm benchmarks, and forces the $f(x, 0) = 1$ line to be considered in `sum-osc`. For `bin_search`, both `PUBS` and `PURRS` are able to obtain $1+\log_2(x)$ bounds on the $x > 0$ subdomain, close to the exact solution $1+\lfloor\log_2(x)\rfloor$. Closed-form solutions are not available for `qsort_best`, and the $x \log_2(x)$ asymptotic behavior is only recovered by `mlsolve(L)+domsplit`,[14] as well as `PURRS` and `Mathematica` if helped by preprocessing, although other tools do infer quadratic bounds. Interestingly, `prs23_1` is only solved by `PRS23` (Wang and Lin 2023), because of the particular difficulty of using recursive calls in update conditions, even though this naturally encodes imperative loops with conditional updates.

Finally, as could be expected, `CAS-style` is the only category in which computer algebra systems obtain good results. All monovariate benchmarks are solved by `PURRS` and `Mathematica`, and only `Mathematica` solves `exp3` (with hints on resolution order). Surprisingly, `Sympy` fails for `harmonic` and `cas_st2`, but still generally obtains good results. It may be noted that for the last three benchmarks, `Mathematica` represents solutions as special functions while `PURRS` and `Ciao`'s builtin solver choose summations. These representations are equivalent and none can be obviously preferred to the other. The category is challenging for code analysis tools, but several deal or partly deal with the case of exponential bounds and of `fib`. In this last case, the exponential part of the bound obtained is 2^x for `PUBS` and `Duet-CHORA` and 1.6169^x for `mlsolve(S)`, closer to the golden ratio ($\varphi \approx 1.6180...$) expected at infinity. Interestingly, while the exact solutions of the last `cas_st*` are too complex to be expressed by `mlsolve(S)` in our experimental setting, it is still able to obtain very good approximations of the asymptotic behavior, returning $x! \times 2.71828$, $2^x \times 0.69315$ and $2^x \times x! \times 0.5701515$ for `cas_st`3,4,5 respectively, with coefficients close to the constants $e \approx 2.71828...$, $\ln(2) \approx 0.693147...$ and $\mathrm{Ei}(1/2) + \ln(2) - \gamma \approx 0.57015142...$, showing its ability to get good approximations, even in the case of complex equations with complex solutions.

[13] Inspired from an example in the `VIAP` repository, `https://github.com/VerifierIntegerAssignment/`.

[14] It may be noted that our chosen hyperparameters, with a bound $b = 20$ on inputs, make it hard for `mlcost` to distinguish base functions on such one-dimensional benchmark. Results improve further if larger inputs are included in the training set, as shown in Appendix B.

The results presented in mlsolve columns correspond to the accuracy of proposed candidates. We now say a few words on their verification. It must be noted that while the mlsolve version presented in Section 4 easily handles regression for systems of equations, verification has only been implemented for single equations. The verification strategy can be extended to systems of equations, but the naive approach has the disadvantage of requiring candidates for all subfunctions. More subtle solutions are left for future work.

Among the 40 benchmarks, mlsolve(L) finds the exact solution for 14, 9 of which come from single function problems. It is able to prove correctness of all of those 9 candidates, using the translation described in Equation 13.

As mentioned at the end of Section 4.2, verification can be performed in a similar way in the domsplit cases. This translation has not yet been fully integrated in our prototype, but has been tested manually on a few examples, showing that Z3 can provide valuable insights for those piecewise candidates (verification of the noisy_start1 candidate, counter-examples, etc.), but is not strong enough to directly work with complex expressions involving factorials, logarithms, or difficult exponents.

Additional experiment: comparison with symbolic (linear) equation solving. To further justify our choice of methods, we conducted an additional experiment comparing numerical linear regression (used in the case of mlsolve(L)) with symbolic linear equation solving in terms of efficiency.

As mentioned in the introduction, efficiency is not the primary reason we choose search and optimization algorithms typical of machine learning methods over more classical exact symbolic methods. Instead, we prioritize expressivity and flexibility. While symbolic linear equation solving is limited to situations where an exact solution can be expressed as an affine combination of base functions, our approach enables the discovery of approximate solutions (particularly valuable in certain cost analysis applications) when closed-form solutions are too complex to find. Moreover, our modular approach allows the exploration of model spaces more complex than affine ones, which cannot be handled by complete methods (e.g., compare mlsolve(S) with mlsolve(L)).

Nevertheless, in situations where an exact solution can be expressed as an affine combination of base functions, we may choose to use *exact, symbolic* linear equation solvers instead of (float-based, iterative) linear regression solvers to infer template coefficients, as we have done in this paper. While this alternative approach is limited to cases where an exact solution exists within the model space, it has the advantage of providing exact, symbolic coefficients rather than imprecise floating-point numbers. However, aside from the limitation to exact solutions, such infinite-precision symbolic methods do not scale as well to large problems as the approximate, iterative methods used in linear regression. To evaluate this scalability issue, we conducted experiments comparing these methods within the Mathematica and Sympy CAS, using the Fibonacci (fib) benchmark, 22 base functions (including powers of the golden ratio φ and its conjugate $\overline{\varphi}$ to enable an exact solution), and $n \in [10, 100]$ data points. It is worth noting that this example is intended solely to provide orders of scale of computation time: it is small enough to allow the use of exact symbolic methods and, for the same reason, employs handpicked base functions that may not be available in practice.

For Mathematica, we use the LinearSolve function on symbolic inputs, which returns one of the possible solutions when the system is underdetermined (other functions can be

Tool	n				Output Solution (if applicable)
	10	20	34	100	
`mlsolve(L)`	`< 0.2s`	`< 0.2s`	`< 0.2s`	`< 0.2s`	$\texttt{0.45} \cdot (\varphi^n - \overline{\varphi}^n)$

`Mathematica`
(symbolic)
	`28.8s`	`O.M.`
		`(> 30min)`
			`T.O./O.M.`	`T.O./O.M.`	$\frac{1}{\sqrt{5}}(\varphi^n - \overline{\varphi}^n)$,
					+ NullSpace

`Sympy`	`und.`	`und.`	`2.0s`	`4.3s`	$\frac{\sqrt{5}}{5}\varphi^n - \frac{\sqrt{5}}{5}\overline{\varphi}^n$

Table 3: Comparison of linear regression and symbolic linear equation solvers for coefficient search on the `fib` benchmark, with 22 base functions and n training points. Legend: underdetermined systems (`und.`), timeouts (`T.O.`), and out-of-memory errors (`O.M.`).

used to compute the full set of solutions). For $n = 10$ input points, we observe that the solution it proposes (possibly chosen for its parsimony) corresponds to the exact closed form for the `fib` benchmark, although it obtains it at a `>100`× overhead compared to `mlsolve(L)`. In fact, this CAS obtains `O.M./T.O.` errors for $n \geq 20$, and is not able to reach the stage where the systems become fully determined. Investigating this issue shows that it is caused by an inappropriate symbolic simplification strategy, where `Mathematica` conserves large polynomial coefficients in $\mathbb{Q}[\sqrt{5}]$ during computation, without simplifying them to 0 early enough when possible, leading to wasted computational resources (note that this only happens for symbolic and not for `float`-based linear equation solving).

The CAS `Sympy` does not display this performance issue, and is able to obtain the exact symbolic solution as soon as the system is fully determined, although at a `>10`× overhead compared to numerical iterative methods.

`mlsolve(L)` obtains a numerical approximation (with the correct features) of the expected solution, spending less than $0.2s$ in regression for all chosen values of n, suggesting that this time is mostly spent in bookkeeping tasks rather than the core of regression itself. Note that `mlsolve(L)`'s focus on sparse candidates allows to obtain the correct solution even for underdetermined benchmarks.

Nonetheless, symbolic linear equation solving can be utilized in certain cases to achieve exact symbolic regression, albeit at the expense of time. As a direction for future work, it could be integrated with our method and applied selectively when a solution is deemed feasible, focusing on a subset of base functions identified by Lasso and a subset of data points for efficiency.

7 Related Work

Exact Recurrence Solvers. Centuries of work on recurrence equations have created a large body of knowledge, whose full account is a matter of mathematical history (Dickson 1919), with classical results such as closed forms of *C-recursive sequences*, i.e., of solutions of linear recurrence equations with constant coefficients (Kauers and Paule 2010; Petkovšek and Zakrajšek 2013).

Despite important decision problems remaining open (Ouaknine and Worrell 2012), the field of symbolic computation now offers multiple algorithms to obtain solutions and/or qualitative insights on various classes of recurrence equations. For (monovariate) linear equations with polynomial coefficients, whose solutions are named *P-recursive sequences*, computing all their polynomial, rational and hypergeometric solutions, whenever they exist, is a closed problem (Petkovšek 1992; Petkovšek et al. 1997; Abramov 1995).

Several of the algorithms available in the literature have been implemented in popular Computer Algebra Systems such as `Sympy` (Meurer et al. 2017), `Mathematica` (Mathematica 2023), `Maple` (Heck and Koepf 1993) and `Matlab` (Higham and Higham 2016). These algorithms are built on insights coming from mathematical frameworks including

- *Difference Algebra* (Karr 1981; Bronstein 2000; Abramov et al. 2021; Levin 2008), which considers wide classes of sequences via towers of $\Pi\Sigma^*$-extensions, and creates analogies between recurrence equations and differential equations,
- *Finite Calculus* (Gleich 2005), used in parts of Ciao's builtin solver, partly explaining its good results in the CAS category of our benchmarks,
- *Operational Calculus* (Berg 1967; Kincaid et al. 2018), and
- *Generating Functions* (Wilf 1994; Flajolet and Sedgewick 2009),

which may be mixed with simple *template-based methods*, e.g., finding polynomial solutions of degree d by plugging such polynomial in the equation and solving for coefficients.

Importantly, all of these techniques have in common the central role given to the relation between a *sequence* $f(n)$ and the *shifted sequence* $f(n-1)$, via, e.g., a shift operator σ, or multiplication by the formal variable of a generating function. As discussed before, it turns out that this simple observation highlights an important obstacle to the application of CAS to program analysis via generalized recurrence equations: these techniques tend to focus on *monovariate* recurrences built on *finite differences*, i.e., on recursive calls of shape $f(n-k)$ with k constant, instead of more general recursive calls $f(\phi(n))$. Generalizations of these approaches exist, to handle multivariate "partial difference equations", or "q-difference equations" with recursive calls $f(q \cdot n)$ with $q \in \mathbb{C}$, $|q| < 1$, but this is insufficient to deal with recurrences arising in static cost analysis, which may contain inequations, difficult recursive calls that cannot be discarded via change of variables, piecewise definitions, or non-determinism.

CAS hence tend to focus on *exact resolution* of a class of recurrence equations that is quite different from those that arise in static cost analysis, and do not provide bounds or approximations for recurrences they are unable to solve.

In addition to classical CAS, we have tested `PURRS` (Bagnara et al. 2005), which shares similarities with these solvers. `PURRS` is however more aimed at automated complexity analysis, which is why it provides some support for approximate resolution, as well as handling of some non-linear, multivariate, and divide-and-conquer equations.

Recurrence Solving for Invariant Synthesis and Verification. Another important line of work uses recurrence solving as a key ingredient in generation and verification of program invariants, with further applications such as loop summarization and termination analysis. Much effort in this area has been put on improving *complete* techniques for restricted classes of programs, usually without recursion and built on idealized numerical loops with abstracted conditionals and loop conditions. These techniques can nevertheless be applied to more general programs, yielding *incomplete* approaches, via approximations and abstractions.

This line is well-illustrated by the work initiated in (Kovács 2007; Kovács 2008), where *all* polynomial invariants on program variables are generated for a subclass of loops introduced in (Rodríguez-Carbonell and Kapur 2004), using recurrence-solving methods in addition to the ideal-theoretic approach of (Rodríguez-Carbonell and Kapur 2007). Recent work on the topic enabled inference of polynomial invariants on combinations of program variables for a wider class of programs with polynomial arithmetic (Humenberger et al. 2018; Amrollahi et al. 2022). A recommended overview can be found in the first few sections of (Amrollahi et al. 2023).

This approach is also key to the compositional recurrence analysis line of work (Farzan and Kincaid 2015; Kincaid et al. 2017; 2018; Breck et al. 2020; Kincaid et al. 2023), implemented in various versions of `duet`, with a stronger focus on abstraction-based techniques (e.g., the *wedge* abstract domain), ability to discover some non-polynomial invariants, and some (discontinued) support for non-linear recursive programs, although the approach is still affected by limited accuracy in disjunctive reasoning.

Idealized numerical loops with precise conditionals are tackled by (Wang and Lin 2023), tested in this paper under the name `PRS23`, which builds upon the work developed in `VIAP` and its recurrence solver (Rajkhowa and Lin 2017; 2019). `PRS23` focuses on restricted classes of loops, with ultimately periodic case application. Hence, the problem of precise invariant generation, with fully-considered branching and loop conditions, for extended classes of loops and recursive programs, is still left largely open.

In addition, it may be noted that *size* analysis, although typically encountered in the context of static cost analysis, can sometimes be seen as a form of numerical invariant synthesis, as illustrated by (Lommen and Giesl 2023), which exploits closed form computations on *triangular weakly non-linear loops*, presented, e.g., in Frohn et al. (2020).

Cost Analysis via (Generalized) Recurrence Equations. Since the seminal work of Wegbreit (1975), implemented in Metric, multiple authors have tackled the problem of cost analysis of programs (either logic, functional, or imperative) by automatically setting up recurrence equations, before solving them, using either generic CAS or specialized solvers, whose necessity were quickly recognized. Beyond Metric, applied to cost analysis of `Lisp` programs, other important early work include ACE (Le Metayer 1988), and Rosendahl (1989) in an abstract interpretation setting.

We refer the reader to previous publications in the Ciao line of work for further context and details (Debray et al. 1990; Debray and Lin 1993; Debray et al. 1997; Navas et al. 2007; Serrano et al. 2014; Lopez-Garcia et al. 2016; 2018).

We also include tools such as `PUBS`[15] (Albert et al. 2011; 2013) and `Cofloco`[16] (Flores-Montoya 2017) in this category. These works emphasize the shortcomings of using too simple recurrence relations, chosen as to fit the limitations of available CAS solvers, and the necessity to consider non-deterministic (in)equations, piecewise definitions, multiple variables, possibly increasing variables, and to study non-monotonic behavior and control-flow of the equations. They do so by introducing the vocabulary of *cost relations*, which may be seen as systems of non-deterministic (in)equations, and proposing new coarse approximate resolution methods.

It may be noted that `duet`, mentioned above, also proposes an option for resource bound analysis, as a specialization of its numerical analysis. Additionally, recent work

[15] `https://costa.fdi.ucm.es/~costa/pubs/pubs.php`

[16] `https://github.com/aeflores/CoFloCo`

in the cost analyzer KoAT has given a greater importance to recurrence solving, e.g., in Lommen and Giesl (2023).

Other Approaches to Static Cost Analysis. Automatic static cost analysis of programs is an active field of research, and many approaches have been proposed, using different abstractions than recurrence relations.

For functional programs, type systems approaches have been studied. This includes the concept of *sized types* (Vasconcelos and Hammond 2003; Vasconcelos 2008), but also the potential method implemented in RaML (Hoffmann 2011; Hoffmann et al. 2012).

The version of RaML (1.5) tested in this paper is limited to potentials (and hence size and cost bounds) defined by multivariate polynomials of bounded degrees. One of the powerful insights of RaML is to represent polynomials, not in the classical basis of monomials x^k, but in the binomial basis $\binom{x}{k}$, leading to much simpler transformations of potentials when type constructors are applied. Thanks to this idea, the problem of inference of non-linear bounds can be reduced to a linear programming problem. A promising extension of RaML to *exponential* potentials has recently been presented (Kahn and Hoffmann 2020), using Stirling numbers of the second kind instead of binomial coefficients, but, at the time of writing this paper, no implementation is available to the best of the authors' knowledge.

Other important approaches include those implemented in Loopus, via size-change graphs (Zuleger et al. 2011) and difference constraints (Sinn et al. 2017), as well as those implemented in AProve (Giesl et al. 2017), KoAT (Giesl et al. 2022) and LoAT (Frohn and Giesl 2022), which translate programs to (extensions of) Integer Transition Systems (ITS) and use, among other techniques, automated inference of ranking functions, summarization, and alternate cost and size inference (Frohn and Giesl 2017).

Dynamic Inference of Invariants/Recurrences. Our approach is related to the line of work on dynamic invariant analysis, which proposes to identify likely properties over variables from observed program traces. Pioneer work on this topic is exemplified by the tool Daikon (Ernst 2000; Ernst et al. 2001), which is able to infer some linear relationships among a small number of explicit program variables, as well as some template "derived variables", although it is limited in expressivity and scalability. Further work on dynamic invariant analysis made it possible to discover invariants among the program variables that are polynomial (Nguyen et al. 2012) and tropical-linear (Nguyen et al. 2014), i.e., a subclass of piecewise affine functions. More recently, (Nguyen et al. 2022) added symbolic checking to check/remove spurious candidate invariants and obtain counterexamples to help inference, in a dynamic, iterative guess and check method, where the checking is performed on the program code. Finally, making use of these techniques, an approach aimed at learning asymptotic complexities, by dynamically inferring linear and divide-and-conquer recurrences before extracting complexity orders from them, is presented in (Ishimwe et al. 2021).

While these works are directly related to ours, as they take advantage of sample traces to infer invariants, there are several, important differences from our work. A key difference is that, instead of working directly on the program code, our method processes recurrence relations, which may be seen as abstractions of the program obtained by previous static analyses, and applies regression to training sets obtained by evaluating the recurrences on input "sizes" instead of concrete data.[17] Hence, we apply dynamic inference techniques on already abstracted programs: this allows complex invariants to be represented by simple expressions, which are thus easier to discover dynamically.

Moreover, our approach differs from these works in the kind of invariants that can be inferred, and in the regression techniques being used. The approach presented in (Nguyen et al. 2012) discovers polynomials relations (of bounded degree) between program variables. It uses an equation solving method to infer equality invariants – which correspond to exact solutions in our context – although their work recovers some inequalities by other means. Similarly to one instantiation of our guess method (but not both), they generate templates that are affine combinations of a predefined set of terms, which we call base functions. However, unlike (Nguyen et al. 2012) we obtain solutions that go beyond polynomials using both of our guessing methods, as well as approximations. These approximations can be highly beneficial in certain applications, as discussed in Section 1 and further elaborated on in the following section.

The Dynaplex (Ishimwe et al. 2021) approach to dynamic complexity inference has different goals than ours: it aims at asymptotic complexity instead of concrete cost functions, and, in contrast to our approach, does not provide any soundness guarantees. Dynaplex uses linear regression to find recurrences, but applies the Master theorem and pattern matching to obtain closed-form expressions representing asymptotic complexity bounds, unlike our approach, which uses a different solving method and obtains concrete cost functions (either exact or approximated), instead of complexity orders.

Nevertheless, it must be noted that all of these works are complementary to ours, in the sense that they can be used as part of our general framework. Indeed, we could represent recurrence relations as programs (with input arguments representing input data sizes and one output argument representing the cost) and apply, e.g., (Nguyen et al. 2012) to find polynomial closed-forms of the recurrences. Similarly, we could apply the approach in (Nguyen et al. 2014), which is able to infer piecewise affine invariants using tropical polyhedra, in order to extend/improve our domain splitting technique.

8 Conclusions and Future Work

We have developed a novel approach for solving or approximating arbitrary, constrained recurrence relations. It consists of a *guess* stage that uses machine learning techniques to infer a candidate closed-form solution, and a *check* stage that combines an SMT-solver and a CAS to verify that such candidate is actually a solution. We have implemented a prototype and evaluated it within the context of CiaoPP, a system for the analysis of logic programs (and other languages via Horn clause tranformations). The *guesser* component of our approach is parametric w.r.t. the machine learning technique used, and we have instantiated it with both *sparse (lasso) linear regression* and *symbolic regression* in our evaluation.

The experimental results are quite promising, showing that in general, for the considered benchmark set, our approach outperforms state-of-the-art cost analyzers and recur-

[17] Note that such recurrence relations can be seen as invariants on the cost of predicates, and are useful in themselves for a number of applications, although for some other applications we are interested in representing them as closed-form functions.

rence solvers. It also can find closed-form solutions for recurrences that cannot be solved by them. The results also show that our approach is reasonably efficient and scalable.

Another interesting conclusion we can draw from the experiments is that our machine learning-based solver could be used as a complement of other (back-end) solvers in a combined higher-level solver, in order to obtain further significant accuracy improvements. For example, its combination with CiaoPP's builtin solver will potentially result in a much more powerful solver, obtaining exact solutions for 88% of the benchmarks and accurate approximations for the rest of them, except for one benchmark (which can only be solved by PRS23's solver).

Regarding the impact of this work on logic programming, note that an improvement in a recurrence solver can potentially result in arbitrarily large accuracy gains in cost analysis of (logic) programs. Not being able to solve a recurrence can cause huge accuracy losses, for instance, if such a recurrence corresponds to a predicate that is deep in the control flow graph of the program, and such accuracy loss is propagated up to the main predicate, inferring no useful information at all.

The use of regression techniques (with a randomly generated training set by evaluating the recurrence to obtain the dependent value) does not guarantee that a solution can always be found. Even if an exact solution is found in the *guess* stage, it is not always possible to prove its correctness in the *check* stage. Therefore, in this sense, our approach is *not complete*. However, note that in the case where our approach does not obtain an exact solution, the closed-form candidate inferred by the *guess* stage, together with its *accuracy score*, can still be very useful in some applications (e.g., granularity control in parallel/distributed computing), where good approximations work well, even though they are not upper/lower bounds of the exact solutions.

As a proof of concept, we have considered a particular deterministic evaluation for constrained recurrence relations, and the verification of the candidate solution is consistent with this evaluation. However, it is possible to implement different evaluation semantics for the recurrences, to support, e.g., non-deterministic or probabilistic programs, adapting the verification stage accordingly. Note that we need to require termination of the recurrence evaluation as a precondition in verification of the obtained results. This is partly due to the particular evaluation strategy of recurrences that we are considering. In practice, non-terminating recurrences can be discarded in the *guess* stage, by setting a timeout. Our approach can also be combined with a termination prover in order to guarantee such a precondition.

As a future work, we plan to fully integrate our novel solver into the CiaoPP system, combining it with its current set of back-end solvers (referred to as CiaoPP's builtin solver in this paper) in order to improve the static cost analysis. As commented above, the experimental results encourage us to consider such a potentially powerful combination, which, as an additional benefit, would allow CiaoPP to avoid the use of external commercial solvers.

We also plan to further refine and improve our algorithms in several directions. As already explained, the set $\mathcal{F}$ of base functions is currently fixed, user-provided. We plan to automatically infer it by using different heuristics. We may perform an automatic analysis of the recurrence we are solving, to extract some features that allow us to select the terms that most likely are part of the solution. For example, if the system of recurrences involves a subproblem corresponding to a program with doubly nested loops, we can select a quadratic term, and so on. Additionally, machine learning techniques may be applied to learn a suitable set of base functions from selected recurrence features (or the programs from which they originate). Another interesting line for future work is to extend our solver to deal (directly) with non-deterministic recurrences.

Finally, we plan to use the counterexamples found by the *checker* component to provide feedback (to the *guesser*) and help refine the search for better candidate solutions, such as by splitting the recurrence domains. Although our current domain splitting strategy already provides good improvements, it is purely syntactic. We also plan to develop more advanced strategies, e.g., by using a generalization of model trees.

References

Abramov, S. 1995. Rational solutions of linear difference and q-difference equations with polynomial coefficients. *USSR Computational Mathematics and Mathematical Physics*, *29*, 7–12.

Abramov, S. A., Bronstein, M., Petkovšek, M., and Schneider, C. 2021. On rational and hypergeometric solutions of linear ordinary difference equations in $\Pi\Sigma^*$-field extensions. *Journal of Symbolic Computation*, *107*, 23–66.

Albert, E., Arenas, P., Genaim, S., and Puebla, G. 2011. Closed-Form Upper Bounds in Static Cost Analysis. *Journal of Automated Reasoning*, *46*, 2, 161–203.

Albert, E., Genaim, S., and Masud, A. N. 2013. On the inference of resource usage upper and lower bounds. *ACM Trans. Comput. Log.*, *14*, 3, 22:1–22:35.

Amrollahi, D., Bartocci, E., Kenison, G., Kovács, L., Moosbrugger, M., and Stankovič, M. Solving Invariant Generation for Unsolvable Loops. In *Static Analysis* 2022, pp. 19–43. Springer.

Amrollahi, D., Bartocci, E., Kenison, G., Kovács, L., Moosbrugger, M., and Stankovič, M. 2023. (Un)Solvable Loop Analysis. *arXiv*, *2306.01597*.

Bagnara, R., Pescetti, A., Zaccagnini, A., and Zaffanella, E. 2005. PURRS: Towards Computer Algebra Support for Fully Automatic Worst-Case Complexity Analysis. Technical report. `arXiv:cs/0512056`.

Berg, L. 1967. *Introduction to the Operational Calculus.* North-Holland Publishing Co. Translated from Einführung in die Operatorenrechnung, 1962.

Breck, J., Cyphert, J., Kincaid, Z., and Reps, T. W. Templates and recurrences: better together. In *PLDI* 2020, pp. 688–702. ACM.

Bronstein, M. 2000. On solutions of linear ordinary difference equations in their coefficient field. *Journal of Symbolic Computation*, *29*, 6, 841–877.

Celaya, M. and Ruskey, F. 2012. An Undecidable Nested Recurrence Relation. *arXiv*, *1203.0586.*

Cranmer, M. 2023. Interpretable Machine Learning for Science with PySR and SymbolicRegression.jl. *arXiv*, *2305.01582*.

de Moura, L. M. and Bjørner, N. Z3: An Efficient SMT Solver. In Ramakrishnan, C. R. and Rehof, J., editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008* 2008, volume 4963 of *Lecture Notes in Computer Science*, pp. 337–340. Springer.

Debray, S. K. and Lin, N.-W. 1993. Cost Analysis of Logic Programs. *ACM TOPLAS*, *15*, 5, 826–875.

Debray, S. K., Lin, N.-W., and Hermenegildo, M. V. Task Granularity Analysis in Logic Programs. In *Proc. PLDI'90* 1990, pp. 174–188. ACM.

Debray, S. K., Lopez-Garcia, P., Hermenegildo, M. V., and Lin, N.-W. Lower Bound Cost Estimation for Logic Programs. In *ILPS'97* 1997, pp. 291–305. MIT Press.

Dickson, L. E. 1919. *History of the theory of numbers*, volume I: Divisibility and Primality, chapter XVII, Recurring Series: Lucas' u_n, v_n, pp. 392–410. Carnegie Institution of Washington. Available at `https://archive.org/details/historyoftheoryo01dick/page/392/mode/2up`.

Ernst, M., Cockrell, J., Griswold, W., and Notkin, D. 2001. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, *27*, 2, 99–123.

Ernst, M. D. 2000. *Dynamically Discovering Likely Program Invariants*. PhD thesis, University of Washington, USA. Advisor: Notkin, David.

Farzan, A. and Kincaid, Z. Compositional Recurrence Analysis. In *Formal Methods in Computer-Aided Design, FMCAD* 2015, pp. 57–64. IEEE.

Flajolet, P. and Sedgewick, R. 2009. *Analytic Combinatorics*. Cambridge University Press. Also available from `https://algo.inria.fr/flajolet/Publications/book.pdf`.

Flores-Montoya, A. 2017. *Cost Analysis of Programs Based on the Refinement of Cost Relations*. PhD thesis, Technische Universität, Darmstadt. Advisor: Reiner Hähnle.

Frohn, F. and Giesl, J. Analyzing runtime complexity via innermost runtime complexity. In *LPAR-21* 2017, volume 46, pp. 249–268. Extended version at `https://verify.rwth-aachen.de/giesl/papers/LPAR17Report.pdf`.

Frohn, F. and Giesl, J. Proving Non-Termination and Lower Runtime Bounds with LoAT (system description). In *Automated Reasoning* 2022, pp. 712–722. Springer.

Frohn, F., Hark, M., and Giesl, J. Termination of polynomial loops. In *Static Analysis - 27th International Symposium, SAS* 2020, volume 12389 of *Lecture Notes in Computer Science*, pp. 89–112. Springer.

Giesl, J., Aschermann, C., Brockschmidt, M., Emmes, F., Frohn, F., Fuhs, C., Hensel, J., Otto, C., Plücker, M., Schneider-Kamp, P., Ströder, T., Swiderski, S., and Thiemann, R. 2017. Analyzing Program Termination and Complexity Automatically with AProVE. *Journal of Automated Reasoning*, *58*, 1, 3–31.

Giesl, J., Lommen, N., Hark, M., and Meyer, F. Improving Automatic Complexity Analysis of Integer Programs. In *The Logic of Software. A Tasting Menu of Formal Methods* 2022, volume 13360 of *LNCS*, pp. 193–228. Springer.

Gleich, D. F. 2005. Finite Calculus: A Tutorial for Solving Nasty Sums. Combinatorics, Stanford University.

Hastie, T., Tibshirani, R., and Friedman, J. 2009. *The Elements of Statistical Learning: Data Mining, Inference and Prediction*. Springer New York, NY, second edition.

Hastie, T., Tibshirani, R., and Wainwright, M. 2015. *Statistical Learning with Sparsity: The Lasso and Generalizations*. Chapman & Hall/CRC.

Heck, A. and Koepf, W. 1993. *Introduction to MAPLE*, volume 1993. Springer.

Hermenegildo, M., Puebla, G., Bueno, F., and Garcia, P. L. 2005. Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and The Ciao System Preprocessor). *Science of Computer Programming*, *58*, 1–2, 115–140.

Higham, D. J. and Higham, N. J. 2016. *MATLAB guide*. SIAM.

Hoffmann, J. 2011. *Types with Potential: Polynomial Resource Bounds via Automatic Amortized Analysis*. PhD thesis, Fakultät für Mathematik, Informatik und Statistik der Ludwig-Maximilians-Universität München. Advisor: Martin Hofmann.

Hoffmann, J., Aehlig, K., and Hofmann, M. 2012. Multivariate Amortized Resource Analysis. *ACM Transactions on Programming Languages and Systems*, *34*, 3, 14:1–14:62.

Humenberger, A., Jaroschek, M., and Kovács, L. Invariant generation for multi-path loops with polynomial assignments. In *Verification, Model Checking, and Abstract Interpretation - 19th International Conference, VMCAI* 2018, volume 10747 of *Lecture Notes in Computer Science*, pp. 226–246. Springer.

Ishimwe, D., Nguyen, K., and Nguyen, T. 2021. Dynaplex: Analyzing Program Complexity using Dynamically Inferred Recurrence Relations. *Proc. ACM Program. Lang.*, *5*, OOPSLA, 1–23.

Kahn, D. M. and Hoffmann, J. Exponential automatic amortized resource analysis. In *FOSSACS* 2020, volume 12077 of *LNCS*, pp. 359–380. Springer.

Karr, M. 1981. Summation in finite terms. *J. ACM*, *28*, 2, 305–350.

Kauers, M. and Paule, P. 2010. *The Concrete Tetrahedron: Symbolic Sums, Recurrence Equations, Generating Functions, Asymptotic Estimates.* Texts and monographs in symbolic computation. Springer.

Kincaid, Z., Breck, J., Boroujeni, A. F., and Reps, T. 2017. Compositional Recurrence Analysis Revisited. *SIGPLAN Not.*, *52*, 6, 248–262.

Kincaid, Z., Breck, J., Cyphert, J., and Reps, T. W. 2019. Closed forms for numerical loops. *Proc. ACM Program. Lang.*, *3*, POPL, 55:1–55:29.

Kincaid, Z., Cyphert, J., Breck, J., and Reps, T. W. 2018. Non-linear reasoning for invariant synthesis. *Proc. ACM Program. Lang.*, *2*, POPL, 54:1–54:33.

Kincaid, Z., Koh, N., and Zhu, S. 2023. When less is more: Consequence-finding in a weak theory of arithmetic. *Proc. ACM Program. Lang.*, *7*, POPL, 1275–1307.

Klemen, M., Carreira-Perpiñán, M., and Lopez-Garcia, P. Solving Recurrence Relations using Machine Learning, with Application to Cost Analysis. In Pontelli, E., Costantini, S., Dodaro, C., Gaggl, S., Calegari, R., Garcez, A. D., Fabiano, F., Mileo, A., Russo, A., and Toni, F., editors, *Technical Communications of the 39th International Conference on Logic Programming (ICLP 2023)* 2023, volume 385 of *Electronic Proceedings in Theoretical Computer Science (EPTCS)*, pp. 155–168. Open Publishing Association (OPA).

Kovács, L. 2007. *Automated Invariant Generation by Algebraic Techniques for Imperative Program Verification in Theorema.* PhD thesis, RISC, Johannes Kepler University Linz. Advisors: Tudor Jebelean and Andrei Voronkov.

Kovács, L. Reasoning Algebraically About P-Solvable Loops. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS* 2008, volume 4963 of *Lecture Notes in Computer Science*, pp. 249–264. Springer.

Le Metayer, D. 1988. ACE: An Automatic Complexity Evaluator. *ACM TOPLAS*, *10*, 2.

Levin, A. 2008. *Difference Algebra.* Algebra and Applications. Springer, NY.

Lommen, N. and Giesl, J. Targeting completeness: Using closed forms for size bounds of integer programs. In *Frontiers of Combining Systems (FroCos)* 2023, pp. 3–22. Springer.

Lopez-Garcia, P., Darmawan, L., Klemen, M., Liqat, U., Bueno, F., and Hermenegildo, M. V. 2018. Interval-based Resource Usage Verification by Translation into Horn Clauses and an Application to Energy Consumption. *TPLP*, *18*, 2, 167–223.

Lopez-Garcia, P., Klemen, M., Liqat, U., and Hermenegildo, M. V. 2016. A General Framework for Static Profiling of Parametric Resource Usage. *TPLP (ICLP'16 Special Issue)*, *16*, 5-6, 849–865.

Manna, Z. and McCarthy, J. Properties of programs and partial function logic. In *Machine Intelligence 5, Proceedings of the Fifth Annual Machine Intelligence Workshop* 1970, pp. 27–38. Edinburgh University Press, 1969.

Mathematica 2023. Wolfram Mathematica (v13.2): the World's Definitive System for Modern Technical Computing. `https://www.wolfram.com/mathematica`. Accessed: May 25, 2023.

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., Štěpán Roučka, Saboo, A., Fernando, I., Kulal, S., Cimrman, R., and Scopatz, A. 2017. SymPy: symbolic computing in Python. *PeerJ Computer Science*, *3*, e103.

Navas, J., Mera, E., Lopez-Garcia, P., and Hermenegildo, M. User-Definable Resource Bounds Analysis for Logic Programs. In *Proc. of ICLP'07* 2007, volume 4670 of *LNCS*, pp. 348–363. Springer.

Nguyen, T., Kapur, D., Weimer, W., and Forrest, S. Using Dynamic Analysis to Discover Polynomial and Array Invariants. In Glinz, M., Murphy, G. C., and Pezzè, M., editors, *34th International Conference on Software Engineering (ICSE)* 2012, pp. 683–693. IEEE Computer Society.

Nguyen, T., Kapur, D., Weimer, W., and Forrest, S. Using Dynamic Analysis to Generate Disjunctive Invariants. In Jalote, P., Briand, L. C., and van der Hoek, A., editors, *36th International Conference on Software Engineering (ICSE)* 2014, pp. 608–619. ACM.

Nguyen, T., Nguyen, K., and Dwyer, M. B. 2022. Using Symbolic States to Infer Numerical Invariants. *IEEE Transactions on Software Engineering*, *48*, 10, 3877–3899.

Ouaknine, J. and Worrell, J. Decision problems for linear recurrence sequences. In Finkel, A., Leroux, J., and Potapov, I., editors, *Reachability Problems* 2012, pp. 21–28. Springer.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., VanderPlas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., and Duchesnay, E. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*, *12*, 2825–2830.

Petkovšek, M. 1992. Hypergeometric solutions of linear recurrences with polynomial coefficients. *Journal of Symbolic Computation*, *14*, 2, 243–264.

Petkovšek, M., Wilf, H. S., and Zeilberger, D. 1997. $A = B$. A K Peters, Ltd. Available online at `https://www2.math.upenn.edu/~wilf/AeqB.html`.

Petkovšek, M. and Zakrajšek, H. Solving linear recurrence equations with polynomial coefficients. In *Computer Algebra in Quantum Field Theory: Integration, Summation and Special Functions* 2013, pp. 259–284. Springer. Preprint at `https://users.fmf.uni-lj.si/petkovsek/PetZakPreprint.pdf`.

Podelski, A. and Rybalchenko, A. A Complete Method for the Synthesis of Linear Ranking Functions. In *VMCAI'04* 2004, volume 2937 of *LNCS*, pp. 239–251. Springer.

Rajkhowa, P. and Lin, F. VIAP - Automated System for Verifying Integer Assignment Programs with Loops. In *19th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, SYNASC* 2017, pp. 137–144. IEEE Computer Society.

Rajkhowa, P. and Lin, F. 2019. A Recurrence Solver And Its Uses in Program Verification. Unpublished. `https://github.com/VerifierIntegerAssignment/rec_paper/blob/master/Detail_proof/ful` Accessed: September 27, 2023.

Rodríguez-Carbonell, E. and Kapur, D. Automatic generation of polynomial loop invariants: Algebraic foundations 2004, ISSAC '04, 266–273. ACM.

Rodríguez-Carbonell, E. and Kapur, D. 2007. Generating all polynomial invariants in simple loops. *Journal of Symbolic Computation*, *42*, 4, 443–476.

Rosendahl, M. Automatic Complexity Analysis. In *4th ACM Conference on Functional Programming Languages and Computer Architecture (FPCA'89)* 1989, pp. 144–156. ACM Press.

Serrano, A., Lopez-Garcia, P., and Hermenegildo, M. V. 2014. Resource Usage Analysis of Logic Programs via Abstract Interpretation Using Sized Types. *TPLP, ICLP'14 Special Issue*, *14*, 4-5, 739–754.

Sinn, M., Zuleger, F., and Veith, H. 2017. Complexity and resource bound analysis of imperative programs using difference constraints. *J. Autom. Reason.*, *59*, 1, 3–45.

Tanny, S. 2013. An invitation to nested recurrence relations. Talk given at Canadian Discrete and Algorithmic Mathematics (CanaDAM) 2013. Slides at https://canadam.math.ca/2013/program/slides/Tanny.Steve.pdf.

Tarjan, R. E. 1985. Amortized computational complexity. *SIAM Journal on Algebraic Discrete Methods*, *6*, 2, 306–318.

Vasconcelos, P. and Hammond, K. Inferring Cost Equations for Recursive, Polymorphic and Higher-Order Functional Programs. In *IFL'03* 2003, volume 3145 of *LNCS*, pp. 86–101. Springer.

Vasconcelos, P. B. 2008. *Space Cost Analysis Using Sized Types*. PhD thesis, University of St Andrews. Advisor: Kevin Hammond.

Wang, C. and Lin, F. 2023. Solving Conditional Linear Recurrences for Program Verification: The Periodic Case. *Proc. ACM Program. Lang.*, *7*, OOPSLA1, 28–55.

Wegbreit, B. 1975. Mechanical Program Analysis. *Communications of the ACM*, *18*, 9, 528–539.

Wilf, H. S. 1994. *generatingfunctionology*. Academic Press, 2nd edition. At `https://www2.math.upenn.edu/~wilf/gfology2.pdf`.

Z3Py 2023. Z3 API in Python. `https://ericpony.github.io/z3py-tutorial`. Accessed: May 25, 2023.

Zuleger, F., Gulwani, S., Sinn, M., and Veith, H. Bound analysis of imperative programs with the size-change abstraction. In *Static Analysis Symposium* 2011, pp. 280–297. Springer.

Appendix A New Contributions w.r.t. our Previous Work

This work is an extended and revised version of our previous work presented at ICLP 2023 as a Technical Communication (Klemen et al. 2023). The main additions and improvements include:

- We report on a much more extensive experimental evaluation using a larger, representative set of increasingly complex benchmarks, and compare our approach with recurrence solving capabilities of state-of-the-art cost analyzers and CASs. In particular, we compare it with RaML, PUBS, Cofloco, KoAT, Duet, Loopus, PRS23, Sympy, PURRS and Mathematica.
- Since our *guess-and-check* approach is parametric in the regression technique used, in (Klemen et al. 2023) we instantiate it to (sparse) linear regression, but here we also instantiate it to symbolic regression, and compare the results obtained with both regression methods.
- In Section 4.2 we introduce a technique that processes multiple subdomains of the original recurrence separately, which we call *domain splitting*. This strategy is orthogonal to the regression method used, and improves the results obtained, as our experimental evaluation shows.
- In Section 7 we include a more extensive discussion on related work.
- In general, we have made some technical improvements, including a better formalization of recurrence relation solving.

Appendix B Extended Experimental Results – Full Outputs

In Table 2, to provide a more comprehensive overview within the constraints of available space, only a few symbols are used to categorize the outputs of the tools. For the sake of completeness, we include here full outputs of the tools in cases where approximate solutions were obtained (symbols "Θ" and "–"). We do not include exact outputs (symbol "✓"), since they are simply equivalent to those in Table 1. We neither include details and distinctions in case where we obtained errors, trivial $+\infty$ bounds, timeouts, or were deemed unsupported, as they are better suited for direct discussions with tool authors.

highdim1

- `mlsolve(S)+domsplit`.
  $x_1 + 6.40410461112967 \cdot x_3 + 8.31920033464576 \cdot x_4 + 6.40410461112967 \cdot x_5 + 8.31920033464576 \cdot x_6 + 8.31920033464576 \cdot x_7 + 8.31920033464576 \cdot x_8 + 8.31920033464576 \cdot x_9$ (case $\forall i, x_i > 0$)
- `PUBS`.
  $11 + 10 \cdot \max(0, x_1 + x_2 + x_3 + x_4 + x_5 + x_6 + x_7 + x_8 + x_9 + x_{10})$
- `KoAT`.
  $x_1 + 2x_2 + 3x_3 + 4x_4 + 5x_5 + 6x_6 + 7x_7 + 8x_8 + 9x_9 + 10x_{10} + 66$

poly7

- `mlsolve(L)`.
  $txyz + 0.02 \cdot txy + 0.88 \cdot tx + x^3z + z^4 - 0.01 \cdot z^3 + 6.16$
- `mlsolve(L)+domsplit`.
  $txyz + 0.95 \cdot tx + x^3z + z^4 - 0.01 \cdot z^3 + 5.76$

highdim2

- `mlsolve(S)+domsplit`.
  $x_1^2 x_5 x_6 \cdot x_3! \cdot \max(x_5, x_9!)$ (case $x_1 > 0$)
- `Loopus15 (*)`.
  $x_1 \max(x_1, x_2) \max(x_1, x_2, x_3) x_4 x_5 x_6 x_7 x_8 x_9 x_{10}$

loop_tarjan

- `KoAT` and `Loopus15 (*)`.
  $2n + 1$

enqdeq1

- `Cofloco`.
  $4n$
- `Loopus15 (*)`.
  $n^2 + 2n$

enqdeq2

- `mlsolve(L)`.
  $-0.53 \cdot k + 0.06 \cdot mn + 0.96 \cdot m + 0.72 \cdot n + 6.64$
- `mlsolve(L)+domsplit`.
  $-0.46 \cdot k + 0.06 \cdot mn + 0.9 \cdot m + 0.95 \cdot n + 5.21$
- `mlsolve(S)+domsplit`.
  $\max(n{+}m,\ 1.7164959 \cdot (n{+}m) - \max(8.868849717670328,\ k))$
- `RaML`.
  $2n + m$
- `Duet-CRA23 (*)`.
  Raw `max:302(max:302(max:302((param1:37 + (2 * param0:35)), param1:37), param0:35), 0)`.
  Simplifies to $2n + m$
- `Cofloco`.
  Without disjunctive bounds, $\max(\max(2m, 2\max(0,k) + \max(0, m-k)), \max(n+2m, \max(0, -n+m-k) + \max(0, 2n+k) + \max(0, n+k)) + n)$
  With disjunctive bounds, 7 cases, simplifies to
  $$\begin{cases} n & \text{if } m = 0 \\ 2n + 2m & \text{if } m \geq 1, m \leq n + k \\ 3n + m + k & \text{if } m \geq 1, m \geq n + k + 1 \end{cases}$$
- `Loopus15 (*)`.
  $mn + m + n$

enqdeq3

- `RaML` and `Duet-CRA23 (*)`.
  $8n$
- `Cofloco`.
  $9n + \max(0, 4n - 1)$, simplified to $13n - 1$ when $n > 0$.
- `Loopus15 (*)`.
  $5n + 5n^2$

merge-sz

- `PUBS`.
  $\max(0, x + y - 1) + \max(\max(0, y), \max(0, x))$, simplifies to $x + y + \max(x, y) - 1$.
- `KoAT`.
  $3x + 3y$
- `Loopus15 (*)`.
  $2x + 2y$

merge

- `mlsolve(L)`.
  $x + y - 1$
- `Ciao's builtin`, `RaML` and `Loopus15 (*)`.
  $x + y$
- `KoAT`.
  $3x + 3y$
- `Duet-ICRA18`.
  Raw `max:796(max:796(max:796(max:796((-1 + param0:57 + param1:60), 1), (-1 + param0:57 + param1:60)), 0), (-1 + param0:57 + param1:60))`.
  Simplifies to $\max(1, x + y - 1)$
- `Duet-CHORA` and `Duet-CRA23 (*)`.
  $\max(0, x + y - 1)$

open-zip

- `RaML`, `PUBS`, `Duet-CHORA` and `Loopus15 (*)`.
  $x + y$
- `KoAT`.
  $x + 2y$
- `Duet-ICRA18`.
  Raw `max:1322(max:1322(max:1322(max:1322(param0:63, 0), 1), param1:66), (-1 + param1:66 + param0:63))`.
  Simplifies to $\max(1, x, y, x + y - 1)$

s-max

- `Ciao's builtin` and `PUBS`.
  $x + y + 1$
- `KoAT`.
  $x + 4y + 1$

- `Duet-ICRA18`.
  Raw `max:662(max:662(max:662((param1:52 + param0:49), (2 + param1:52)), (1 + param1:52)), param1:52)`.
  Simplifies to $y + \max(2, x)$
- `Loopus15 (*)`.
  $x + 2y$

s-max-1

- `KoAT`.
  $3x + 4y + 1$
- `Duet-ICRA18`.
  Raw `max:662(max:662(max:662(max:662(max:662((-1 + param1:52 + (2 * param0:49)), (param1:52 + (2 * param0:49))), (3 + param1:52)), (2 + param1:52)), param1:52), (1 + param1:52))`.
  Simplifies to $y + \max(2x, 3)$
- `Duet-CRA23 (*)`.
  Raw `max:136(max:136(max:136(param1:27, (1 + param1:27)), (-1 + param1:27 + (2 * param0:25))), (param1:27 + (2 * param0:25)))`.
  Simplifies to $y + \max(1, 2x)$
- `Loopus15 (*)`.
  $3x + 2y$

incr1

- `mlsolve(L)`.
  $8.1 - 0.46 \cdot x$
- `KoAT`.
  $x + 10$
- `Duet-ICRA18`.
  Raw `max:399(max:399(2,(11+(-1*param0:30))),1)`.
  Simplifies to $\max(2, 11 - x)$

noisy_strt1

- `Ciao's builtin`, `RaML`, `PUBS`, `Duet-CRA23` and `Duet-CRA23 (*)`.
  x
- `Duet-CHORA`.
  Raw $\max(\max((-20 + x), 0), x)$.
  Simplifies to x
- `Duet-ICRA18`.
  Raw `max:561(max:561(max:561(max:561(1, param0:30), min:560(19, param0:30)), 0), (-20 + param0:30))`.
  Simplifies to $\max(1, x)$
- `KoAT`.
  $2x$
- `Loopus15 (*)`.
  "`max(0,(x + -20)) assuming {(>= x1 0)}`"
- `PURRS`.
  $x - 20$ "assuming $x \geq 20$"
- `Mathematica`.
  0 if $x = 0$ or $x = 20$, $x + cst$ otherwise

noisy_strt2

- `mlsolve(L)`, `Ciao's builtin`, `RaML`, `PUBS`, `Duet-CRA23` and `Duet-CRA23 (*)`.
  x
- `mlsolve(L)+domsplit` and `mlsolve(S)+domsplit`.
  0 if $x = 0$ or $x = 65536$, x otherwise
- `KoAT`.
  $2x$
- `Duet-ICRA18`.
  Raw `max:561(max:561(max:561(max:561(1, param0:30), min:560(65535, param0:30)), 0), (-65536 + param0:30))`.
  Simplifies to $\max(1, x)$
- `Loopus15 (*)`.
  "`max(0, (x + -65536) assuming {(>= x1 0)}`"
- `PURRS`.
  $x - 65536$ "assuming $x \geq 65536$"
- `Mathematica`.
  0 if $x = 0$ or $x = 65536$, $x + \text{cst}$ otherwise

multiphase1

- `mlsolve(L)`.
  $-0.02 \cdot in - 0.6 \cdot i + 1.83 \cdot n + 0.59 \cdot r - 6.49$
- `KoAT`.
  $nr + i + n + r$
- `Duet-ICRA18`.
  Raw `max:372(max:372(max:372(0, param2:58), (param1:55 + param2:58)), (param1:55 + (-1 * param0:52)))`.
  Simplifies to $\max(n - i, n + r)$
- `Duet-CRA23` and `Duet-CRA23 (*)`.
  Raw `max:201(max:201(max:201((param1:37 + -param0:35), min:200(param2:39, -param0:35)), (param2:39 + param1:37)), 0)`.
  Simplifies to $\max(n - i, n + r)$
- `Loopus15 (*)`.
  $r + nr + \max(0, n - i)$

lba_ex_viap

- `mlsolve(L)+domsplit`.
  $.49 \cdot (c - x - y) + .33$ if $x + y < c$, 0 if $x + y \geq c$.
- `mlsolve(S)+domsplit`.
  $\lceil (c - (x + y)) \cdot 0.473.. \rceil$ if $x + y < c$, 0 if $x + y \geq c$.
- `PUBS`, `Duet-ICRA18`, `Duet-CHORA`, `Duet-CRA23` and `Duet-CRA23 (*)`.
  $\max(0, c/2 - x/2 - y/2 + 1/2)$
- `Cofloco`.
  $-x/2 - y/2 + c/2$ if $x + y < c$, 0 if $x + y \geq c$.
- `KoAT`.
  $x + y + c$
- `Loopus15 (*)`.
  $\max(0, c - x - y)$

nested

- `Duet-ICRA18`.
  $\max(1, x)$

nested_case

- `mlsolve(L)`.
  $0.5 \cdot x \cdot (x + 1)$

nested_div

- `mlsolve(L)`.
  $0.14 \cdot \lceil \log_2(n) \rceil + 1.62$
- `mlsolve(L)+domsplit`.
  $0.24 \cdot \lceil \log_2(n) \rceil + 0.12 \cdot \lfloor \log_2(n) \rfloor + 1.7$ if $x > 0$, 0 if $x = 0$.
- `RaML`.
  x

mccarthy91

- `mlsolve(L)+domsplit`.
  $x - 10$ if $x \geq 101$, `ERROR` if $x \leq 100$.
- `Cofloco`.
  $x - 10$ if $x \geq 101$, inf if $x \leq 100$.

golomb

- `mlsolve(L)`.
  $1.38\sqrt{n} + 0.06 \cdot \lceil \log_2(n) \rceil - 0.16$
- `mlsolve(L)+domsplit`.
  $1.44\sqrt{n} - 0.15$ if $x > 0$, 1 if $x = 0$

- `mlsolve(S)+domsplit`.
  $\lceil x^{0.5857451} \rceil$
- `RaML`.
  $1 + 1.5x + .5x^2$

div

- `PUBS` and `Duet-CHORA`.
  x
- `Cofloco` and `Loopus15 (*)`.
  $\max(0, x - y + 1)$
- `KoAT`.
  $x + y + 1$
- `Duet-ICRA18`.
  $\max(1, 2 + x - 2y)$
- `Mathematica`.
  x/y (without floor, if hinted that y is a constant)

sum-osc

- `mlsolve(L)`.
  $.5 \cdot y^2 + 1.5 \cdot y + x$
- `mlsolve(S)+domsplit`.
  $$\begin{cases} \lfloor x - 0.9866641 + \frac{(y+0.9866641)^3}{2y} \rfloor \\ \qquad \text{if } x > 0, y > 0 \\ \lfloor 0.886569412557 \cdot (0.756346845291 \cdot y + 1)^2 \rfloor \\ \qquad \text{if } x = 0, y > 0 \\ 1 \\ \qquad \text{if } y = 0 \end{cases}$$
- `RaML`.
  $1 + .5 \cdot y^2 + 1.5 \cdot y + x$
- `PUBS`.
  $1 + \max(0, x + 2y - 1) \cdot \max(1, \max(0, y))$
  Simplifies to $1 + xy + 2y^2 - y$
- `Cofloco`.
  $$\begin{cases} \max(x + 2y - 1, (x + 2y - 1) * y) + 1 & \text{if } y > 0 \\ 1 & \text{if } y = 0 \end{cases}$$
  Simplifies to 1 if $y = 0$, $1 + xy + 2y^2 - y$ if $y > 0$.
- `KoAT`.
  $1 + x + y + 2y^2$

bin_search

- `mlsolve(L)`.
  $1 + \lfloor \log_2(x) \rfloor$ (for all $x \geq 0$)
- `RaML`.
  $2x$
- `PUBS`.
  $\log_2(1 + \max(0, 2x - 1))$
  Simplifies to 1 if $x = 0$, $1 + \log_2(x)$ if $x > 0$
- `Duet-CHORA`, `Duet-CRA23 (*)` and `Loopus15 (*)`.
  x
- `PURRS`.
  $1 + \log_2(x)$ (ub), after overapproximating floor.
- `Mathematica`.
  $\log_2(x) + \text{cst}$, after overapproximating floor and some rewriting.

qsort_best

- `mlsolve(L)`.
  $0.05 \cdot x^2 + 0.1 \cdot x\lceil \log_2(x) \rceil + 3.43 \cdot x - 0.96 \cdot \lceil \log_2(x) \rceil + 0.27 \cdot \lfloor \sqrt{x} \rfloor - 0.77$
- `mlsolve(L)+domsplit`.
  $0.31 \cdot x\lceil \log_2(x) \rceil + 3.8 \cdot x - 2.64 \cdot \lceil \log_2(x) \rceil + 0.33$ if $x > 0$, 1 if $x = 0$.
- `mlsolve(S)+domsplit`.
  $\lceil 1.9746461095749 \cdot (x + 0.179665875875)^{1.286233} \rceil$ if $x > 0$, 1 if $x = 0$.
- `mlsolve(L)` (with $b = 100$).
  $0.33 \cdot x\lfloor \log_2(x) \rfloor + 5.52 \cdot x - 15.35 \cdot \lfloor \log_2(x) \rfloor + 15.07$
- `mlsolve(L)+domsplit` (with $b = 100$).
  $0.34 \cdot x\lceil \log_2(x) \rceil + 4.99 \cdot x - 11.58 \cdot \lceil \log_2(x) \rceil + 13.14$ if $x > 0$, 1 if $x = 0$.
- `mlsolve(S)+domsplit` (with $b = 100$).
  $1.0148813 \cdot x \log_2(x + 4.7831106)$ if $x > 0$, 1 if $x = 0$.
- `RaML`.
  $1 + 4x + 1/3x^2$
- `PUBS`.
  $(\max(0, 2x - 1) + 1 - 1) \cdot \max(0, x)$
  Simplifies to $2x^2 - x$
- `Duet-CHORA`.
  $1 + x \cdot 2^x$
- `PURRS`.
  $x \cdot \log_2(x) + 3x$ (ub), after overapproximating floor/ceil and some rewriting.
- `Mathematica`.
  $x \cdot \log_2(x) + \text{cst} \cdot x$, after overapproximating floor/-ceil and some rewriting.

prs23_1

- `Duet-CHORA`.
  $f(n) \leq \max(1, 1 + 499n)$, $g(n) \leq \max(1, 3n)$
- `Duet-CRA23` and `Duet-CRA23 (*)` .
  Raw $f(n) \leq$
```
max:212(min:211(min:211((1 + (499
* param0:15)), 998), pow:25(2, param0:15)),
min:211(min:211(min:211(min:211(min:211(min:211(
min:211(min:211( (-498 + ( 499 * param0:15)),
(1 + ( 499 * log:26( 2, 998)))), (1 + (
499 * log:26( 2, ( 1 + ( 499 * log:26( 2,
998))))))), (1 + ( 499 * log:26( 2, ( -498 +
( 499 * param0:15)))))), (1 + (499 * log:26(
2, (1 + ( 499 * log:26( 2, ( -498 + ( 499 *
param0:15))))))))), (-498 + (499 * log:26(2,
pow:25( 2, param0:15))))), 998), (1/2 *
pow:25(2, param0:15))), (1/2 * pow:25(2, (1/2 *
pow:25(2, param0:15)))))),
```
  $g(n) \leq$ `max:212(1, min:211((-2+(3 * param0:15)), (1+(3 * param0:15) + (-3 * log:26(2, 500))))).`
  f simplifies to 2^n if $2 \leq n \leq 9$, 998 if $n \geq 10$.
  g simplifies to $\max(1, 1 + 3n - 3\log_2(500)) \approx \max(1, 3n - 25.90...)$.
- `Loopus15 (*)`.
  $f(n) \leq 998$ and $g(n) \leq 1 + 3n$

exp2

- `mlsolve(S)+domsplit`.
  $4 \cdot 2^x$ (first candidate), $\lfloor 4 \cdot 2^x - .027... \cdot x \rfloor$ (second candidate)

exp3

- `mlsolve(L)`.
  $24879911.68 \cdot x^2y^2 - 382407544.6 \cdot x^2y + 910687706.81 \cdot x^2 - 426104965.68 \cdot xy^2 + 6615353066.27 \cdot xy - 16347484322.02 \cdot x + 1437670074.84 \cdot y^2 - 22905346457.85 \cdot y + 64530269113.55$
- `mlsolve(L)+domsplit`.
  $7007804.6 \cdot x^2y^2 - 77376230.79 \cdot x^2y + 49836049.42 \cdot x^2 - 87385060.39 \cdot xy^2 + 867536772.6 \cdot xy + 70890458.63 \cdot x + 178082073.72 \cdot y^2 - 1539360337.12 \cdot y - 1728336336.08$

fib

- `mlsolve(L)`.
  $69.13 \cdot x^2 + 10.73 \cdot x \cdot \lceil \log_2(x) \rceil - 133.88 \cdot x \cdot \lfloor \log_2(x) - 1079.01 \cdot x + 553.65 \cdot \lceil \log_2(x) \rceil - 135.4 \cdot \lfloor \sqrt{x} \rfloor + 1549.33 \cdot \lfloor \log_2(x) + 729.74$

- `mlsolve(L)+domsplit`.
  0 if $x = 0$, 1 if $x = 1$, $107.11 \cdot x^2 - 106.74 \cdot x \cdot \lceil \log_2(x) \rceil - 239.41 \cdot x \cdot \lfloor \log_2(x) - 1133.71 \cdot x + 1896.18 \cdot \lceil \log_2(x) \rceil - 706.19 \cdot \lfloor \sqrt{x} \rfloor + 2708.6 \cdot \lfloor \log_2(x) \rfloor - 1859.08$ if $x > 1$.
- `mlsolve(S)+domsplit`.
  0 if $x = 0$, 1 if $x = 1$, $0.453497199508044 \cdot 1.61690120042049^x - 0.239336049422447$ if $x > 1$.
- `PUBS`.
  $2^{\max(0,x-1)}$
- `Duet-CHORA`.
  2^x

harmonic

- `mlsolve(L)`.
  $0.68 \cdot \sqrt{x} + 0.21$
- `mlsolve(L)+domsplit`.
  0 if $x = 0$, $0.82 \cdot \sqrt{x} - 0.15 \cdot \lceil \log_2(x) \rceil + 0.15$ if $x > 0$
- `mlsolve(S)+domsplit`.
  0 if $x = 0$, $\lfloor \log_2(0.58528787 \cdot x + 1.9952444) \rfloor$ if $x > 0$

fact

- `mlsolve(L)`.
  $x! + 16.0$
- `mlsolve(L)+domsplit`.
  $0.02 \cdot 2^x + x! + 32.0$
- `Cofloco`.
  1

cas_st1

- `mlsolve(L)`.
  $176380.52 \cdot 2^x \cdot x - 1246486.37 \cdot 2^x - 346.58 \cdot 5^x + 140930.04 \cdot x^2 - 22021.02 \cdot x \cdot \lceil \log_2(x) \rceil + 1019395.76 \cdot x \cdot \lfloor \log_2(x) \rfloor + 150189.16 \cdot x + 405206.28 \cdot \lceil \log_2(x) \rceil - 1598623.07 \cdot \lfloor \sqrt{x} \rfloor - 1533107.27 \cdot \lfloor \log_2(x) \rfloor + 1799.42 \cdot x! + 3447651.05$
- `mlsolve(L)+domsplit`.
  1 if $x = 0$, $176380.52 \cdot 2^x \cdot x - 1246486.37 \cdot 2^x - 346.58 \cdot 5^x + 140930.04 \cdot x^2 - 22021.02 \cdot x \cdot \lceil \log_2(x) \rceil + 1019395.76 \cdot x \cdot \lfloor \log_2(x) \rfloor + 150189.16 \cdot x + 405206.28 \cdot \lceil \log_2(x) \rceil - 1598623.07 \cdot \lfloor \sqrt{x} \rfloor - 1533107.27 \cdot \lfloor \log_2(x) \rfloor + 1799.42 \cdot x! + 3447651.05$ if $x > 0$
- `Cofloco`.
  1 if $x = 0$, 2 if $x > 0$

cas_st2

- `mlsolve(L)`.
  $1532499438.63 \cdot 2^x \cdot x - 10961243871.35 \cdot 2^x - 2533831.22 \cdot 5^x + 1109062299.38 \cdot x^2 - 409557670.88 \cdot x \cdot \lceil \log_2(x) \rceil + 10970271676.25 \cdot x \cdot \lfloor \log_2(x) \rfloor + 774384179.03 \cdot x + 3890181725.0 \cdot \lceil \log_2(x) \rceil - 18733069879.29 \cdot \lfloor \sqrt{x} \rfloor - 16344125381.56 \cdot \lfloor \log_2(x) \rfloor + 9112306.73 \cdot x! + 35710669116.92$
- `mlsolve(L)+domsplit`.
  1 if $x = 0$, $1532499438.25 \cdot 2^x \cdot x - 10961243867.81 \cdot 2^x - 2533831.22 \cdot 5^x + 1109062296.0 \cdot x^2 - 409557670.74 \cdot x \cdot \lceil \log_2(x) \rceil + 10970271677.52 \cdot x \cdot \lfloor \log_2(x) \rfloor + 774384185.44 \cdot x + 3890181726.66 \cdot \lceil \log_2(x) \rceil - 18733069876.69 \cdot \lfloor \sqrt{x} \rfloor - 16344125386.41 \cdot \lfloor \log_2(x) \rfloor + 9112306.73 \cdot x! + 35710669102.86$ if $x > 0$.

cas_st3

- `mlsolve(L)`.
  $-0.04 \cdot 2^x + 2.72 \cdot x! + 320.0$
- `mlsolve(L)+domsplit`.
  1 if $x = 0$, $0.01 \cdot 2^x \cdot x - 0.13 \cdot 2^x + 2.72 \cdot x! + 320.0$ if $x > 0$.
- `mlsolve(S)+domsplit`.
  $x! \cdot 2.71828$

cas_st4

- `mlsolve(L)`.
  $0.69 \cdot 2^x - 0.02 \cdot x^2 - 0.04 \cdot x \cdot \lceil \log_2(x) \rceil + 0.13 \cdot x \cdot \lfloor \log_2(x) \rfloor + 0.04 \cdot x + 0.51 \cdot \lceil \log_2(x) \rceil - 0.14 \cdot \lfloor \sqrt{x} \rfloor - 1.01 \cdot \lfloor \log_2(x) \rfloor - 0.35$
- `mlsolve(L)+domsplit`.
  0 if $x = 0$, $0.69 \cdot 2^x - 0.02 \cdot x^2 - 0.03 \cdot x \cdot \lceil \log_2(x) \rceil + 0.13 \cdot x \cdot \lfloor \log_2(x) \rfloor + 0.08 \cdot x + 0.39 \cdot \lceil \log_2(x) \rceil - 0.17 \cdot \lfloor \sqrt{x} \rfloor - 0.99 \cdot \lfloor \log_2(x) \rfloor - 0.31$ if $x > 0$.
- `mlsolve(S)+domsplit`.
  $2^x \cdot 0.69315$

cas_st5

- `mlsolve(L)`.
  $100563.63 \cdot 2^x \cdot x - 710686.18 \cdot 2^x - 197.6 \cdot 5^x + 80351.45 \cdot x^2 - 12555.04 \cdot x \cdot \lceil \log_2(x) \rceil + 581209.77 \cdot x \cdot \lfloor \log_2(x) \rfloor + 85630.47 \cdot x + 231028.75 \cdot \lceil \log_2(x) \rceil - 911456.69 \cdot \lfloor \sqrt{x} \rfloor - 874103.34 \cdot \lfloor \log_2(x) \rfloor + 1025.94 \cdot x! + 1965682.95$
- `mlsolve(L)+domsplit`.
  0 if $x = 0$, $100563.63 \cdot 2^x \cdot x - 710686.18 \cdot 2^x - 197.6 \cdot 5^x + 80351.45 \cdot x^2 - 12555.04 \cdot x \cdot \lceil \log_2(x) \rceil + 581209.77 \cdot x \cdot \lfloor \log_2(x) \rfloor + 85630.47 \cdot x + 231028.75 \cdot \lceil \log_2(x) \rceil - 911456.69 \cdot \lfloor \sqrt{x} \rfloor - 874103.34 \cdot \lfloor \log_2(x) \rfloor + 1025.94 \cdot x! + 1965682.95$ if $x > 0$.
- `mlsolve(S)+domsplit`.
  $2^x \cdot x! \cdot 0.5701515$