Solving Recurrence Relations using Machine Learning, with Application to Cost Analysis

[1,2]Maximiliano Klemen, [3]Miguel Á. Carreira-Perpiñán, and [4,2]Pedro Lopez-Garcia

[1]Universidad Politécnica de Madrid (UPM) Madrid, Spain

[2]IMDEA Software Institute, Madrid, Spain

[3]University of California, Merced, USA

[4]Spanish Council for Scientific Research, Madrid, Spain

{maximiliano.klemen,pedro.lopez}@imdea.org, mcarreira-perpinan@ucmerced.edu

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 other languages) are based on setting up recurrence relations representing (bounds on) the computational cost of predicates, and solving them to find closed-form functions that are equivalent to (or a bound on) them. Such recurrence solving is a bottleneck in current tools: many of the recurrences that arise during the analysis cannot be solved with current solvers, such as 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 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. We have implemented a prototype and evaluated it with recurrences generated by a cost analysis system (the one in CiaoPP). The experimental results are quite promising, showing that our approach can find closed-form solutions, in a reasonable time, for classes of recurrences that cannot be solved by such a system, nor by current CASs.

1 Introduction and Motivation

The motivation of the work presented in this paper stems from automatic static cost analysis and verification of logic programs [3, 2, 4, 13, 17, 9, 8]. 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 channels 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 [18, 16, 3, 2, 4, 13, 1, 17, 9], 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).

S. Costantini, E. Pontelli, A. Russo, F. Toni,
R. Calegari, A. D'Avila Garcez, C. Dodaro, F. Fabiano,
S. Gaggl, A. Mileo, (Eds.): ICLP 2023
EPTCS 385, 2023, pp. 155–168, doi:10.4204/EPTCS.385.16

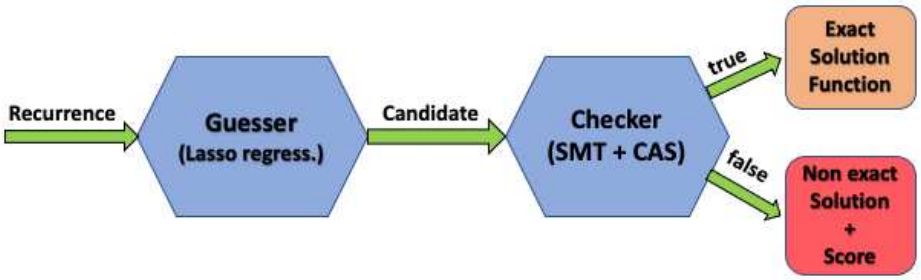

Figure 1: Control flow diagram of our novel solver based on machine learning.

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 to find a closed form. 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 input data sizes (a ranking function), which could be inferred by applying established techniques used in termination analysis [15]. 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, summation, or logarithmic.

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. Section 6 comments on our prototype implementation and its experimental evaluation. Finally, Section 7 summarizes some conclusions and lines for future work.

2 Overview of our Approach

We now give an overview of the two stages of our approach already mentioned: *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 function $\hat{f}(\vec{x})$ that has the same domain as $f(\vec{x})$, and for all $\vec{x}$ in such domain, $\hat{f}(\vec{x}) = f(\vec{x})$. By a closed-form function $\hat{f}$ we mean an expression that is built by using only elementary arithmetic functions, e.g., constants, addition, subtraction, multiplication, division, exponential, or even factorial functions. In particular, this means that $\hat{f}$ does not contain any subexpressions built by using the same function $\hat{f}$ (i.e., $\hat{f}$ is not

recursively defined). We will use the following recurrence as an example to illustrate our approach:

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

2.1 The "guess" stage (sparse linear regression via Lasso)

We use a sparse linear regression mechanism (see Section 3 for more details), so that any possible model we can obtain (which constitutes a candidate solution) must be a linear combination of a predefined set of terms, but using a usually small subset of terms. That is, a function $\hat{f}(\vec{x})$ of the form:

$$\hat{f}(\vec{x}) = \beta_0 + \beta_1\, t_1(\vec{x}) + \beta_2\, t_2(\vec{x}) + \cdots + \beta_n\, t_n(\vec{x})$$

where the t_i's are arbitrary functions on $\vec{x}$ from a set T of candidate terms that we call *base functions*, and the β_i's are the coefficients (real numbers) that are estimated by regression, but so that only a few coefficients are nonzero. Currently, the set T is fixed, and contains the base functions that are representative of the common complexity orders (in Section 7 we comment on future plans to obtain it). For illustration purposes, assume that we use the following set T of base functions:

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

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

1. Generate a training set S. First, a set $X_{\text{train}} = \{\vec{x}_1, \ldots, \vec{x}_k\}$ of input values to the recurrence function is randomly generated. Then, starting with an initial $S = \emptyset$, for each input value $\vec{x}_i \in X_{\text{train}}$, a training case s_i is generated and added to S. For any input value $\vec{x} \in X_{\text{train}}$ the corresponding training case s is a tuple of the form:

   $$s = \langle b, c_1, \ldots, c_n \rangle$$

   where $c_i = [\![t_i]\!]_{\vec{x}}$ for $1 \leq i \leq n$, and $[\![t_i]\!]_{\vec{x}}$ represents the result (a scalar) of evaluating the base function $t_i \in T$ for input value $\vec{x}$, where T is a set of n base functions, as already explained. The (dependent) value b (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 $\vec{x} \in X_{\text{train}}$ such that $\vec{x} = \langle 5 \rangle$, its corresponding training case s in our example will be:

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

2. Perform the sparse regression in two steps using the training set S created above. In the first step, we use linear regression with Lasso (ℓ_1) regularization [6] on the coefficients. This is a penalty term that encourages coefficients whose associated base functions have a small correlation with the dependent value to be exactly zero. This way, typically most of the base functions in T will be discarded, 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 [6], 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 step is a (column) vector $\vec{\beta}$ of coefficients, and an independent coefficient β_0. Finally, we generate a test set X_{test} (again, randomly in the same way as the training set) of

input values to the recurrence function to obtain a measure R^2 of the accuracy of the estimation. Additionally, we discard those terms whose corresponding coefficient is less than a given threshold ε. The resulting closed-form expression that estimates the target function is

$$\hat{f}(\vec{x}) = \mathsf{rm}_\varepsilon(\vec{\beta}^T) \cdot E(T,\vec{x}) + \beta_0$$

where $E(T,\vec{x})$ is a vector of the terms in T with the arguments bound to $\vec{x}$, and rm_ε takes a vector of coefficients and returns another vector where the coefficients less than ε are rounded to zero. Both the Lasso regularization and the pruning function discard many terms from T in the final function.

3. Finally, our method performs again a standard linear regression (without Lasso regularization) on the training set S, but without using those base functions corresponding to the terms discarded previously by Lasso and the ε-pruning. In our example, with $\varepsilon = 0.05$, we obtain:

$$\hat{f}(x) = 1.0\, x$$

with a value $R^2 = 1$, which means that the estimation obtained predicts exactly the values for the test set, and thus, it is a candidate solution for the recurrence in Equation 1. If R^2 were less than 1, it would mean that the function obtained is not a candidate (exact) solution, but a (possibly unsafe) approximation, as there are values in the test set that cannot be exactly predicted.

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 shows the recurrence relation we target to solve, followed by the candidate solution obtained previously using linear regression:

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

Now, Expression (3) below shows the encoding of the recurrence as a first order logic formula.

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

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.

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

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 solving more problems than only using one of these systems in isolation.

3 Preliminaries

Recurrence relations. A recurrence relation of order k, $k > 0$, for a function f, is a set of equations that give k initial values for f, and an equation that recursively defines any other value of f as a function g that takes k previous values of f as parameters. For example, the following recurrence relation of second order ($k = 2$), with g being the arithmetic addition $+$, defines the Fibonacci function:

$$f(n) = \begin{cases} 1 & \text{if n = 0 or n = 1} \\ f(n-1) + f(n-2) & \text{if n} \geq 2 \end{cases} \tag{5}$$

A challenging class of recurrences that we can solve with our approach are "nested" recurrences, e.g., recurrences of the form $f(n) = g(f(f(n-1)))$.

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 linear regression. In all cases, the symbols can be subscribed. We use $\vec{x}$ to denote a finite sequence $\langle x_1, x_2, \ldots, x_n \rangle$, for some $n > 0$. Given a sequence S and an element x, $\langle x|S \rangle$ is a new sequence with first element x and tail S.

Given a piecewise function:

$$f(\vec{x}) = \begin{cases} e_1(\vec{x}) & \text{if } \varphi_1(\vec{x}) \\ e_2(\vec{x}) & \text{if } \varphi_2(\vec{x}) \\ \vdots & \vdots \\ e_k(\vec{x}) & \text{if } \varphi_k(\vec{x}) \end{cases} \tag{6}$$

where $f \in \mathscr{D} \to \mathbb{R}^+$, with $\mathscr{D} = \{\vec{x} | \vec{x} \in \mathbb{Z}^m \wedge \varphi_{\text{pre}}(\vec{x})\}$ for some boolean constraint φ_{pre}, and $e_i(\vec{x}), \varphi_i(\vec{x})$ are arbitrary expressions and constraints over $\vec{x}$ respectively. We say that φ_{pre} is the *precondition* of f, and that f is a *constrained recurrence relation* if and only if:

- $\exists i \in [1,k]$ such that e_i contains a call to f.
- $\exists i \in [1,k]$ such that e_i does not contain any call to f (i.e., it is in *closed form*).
- $\varphi_{\text{pre}} \models \bigvee_{1 \leq i \leq k} \varphi_i$.

Given a concrete input $\vec{d} \in \mathscr{D}$, we evaluate $f(\vec{d})$ deterministically, assuming the evaluation of f as a nested *if-then-else* control structure as follows:

if $\varphi_1(\vec{d})$ **then**
  return $e_1(\vec{d})$
else
  if $\varphi_2(\vec{d})$ **then**
    return $e_2(\vec{d})$
  else
    ...
  end if
end if

More formally, let $\mathsf{def}(f)$ denote the definition of a (piecewise) constrained recurrence relation f represented as the sequence $\langle(e_1(\vec{x}),\varphi_1(\vec{x})),\ldots,(e_k(\vec{x}),\varphi_k(\vec{x}))\rangle$, where each element of the sequence is a pair representing a case. The order of such sequence determines the evaluation strategy. Then, the evaluation of f for a concrete value $\vec{d}$, denoted $\mathsf{EvalFun}(f(\vec{d}))$, is defined as follows:

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

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

Our goal is to find a function $\hat{f} \in \mathscr{D} \to \mathbb{R}^+$ such that for all $\vec{d} \in \mathscr{D}$:

- 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.

In particular, we look for a definition of the form:

$$\hat{f}(\vec{x}) = \beta_0 + \beta_1\, t_1(\vec{x}) + \beta_2\, t_2(\vec{x}) + \cdots + \beta_n\, t_n(\vec{x}) \tag{7}$$

where $\beta_i \in \mathbb{R}$, and t_i are expressions over $\vec{x}$, not including recursive references to $\hat{f}$. If the above conditions are met, we say that $\hat{f}$ is a *closed form* for f.

To illustrate the need of introducing an evaluation strategy for the recurrence that is consistent with the termination of the program, consider the following Prolog program which does not terminate for a call `p(X)` where `X` is bound to an integer:

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

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

$$\begin{array}{lll} \mathtt{C_p}(x) = & 1 & \text{if } x = 0 \\ \mathtt{C_p}(x) = & 1 + \mathtt{C_p}(x+1) & \text{if } x > 0 \end{array} \tag{8}$$

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

Linear Regression. Linear regression [5] is a statistical technique used to approximate the linear relationship between a number of independent variables and a dependent (output) variable. Given a vector of independent (input) variables $X = (X_1, \ldots, X_p)^T \in \mathbb{R}^p$, we predict the output variable Y using the formula

$$Y = \beta_0 + \sum_{i=1}^{p} \beta_i X_i \tag{9}$$

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

$$\beta = \underset{\beta \in \mathbb{R}^p}{\arg\min} \sum_{i=1}^{n} \left(y_i - \beta_0 - \sum_{j=1}^{p} x_{ij} \beta_j \right)^2 \tag{10}$$

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 [6], which adds an ℓ_1 penalty on β (i.e., the sum of absolute values of each coefficient) to Expression 10:

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

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.

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. Algorithms 1 and 2 correspond to the *guesser* and *checker* components, respectively, which are shown in Figure 1.

Algorithm 1 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* S that reflects the accuracy of the approximation, in the range $[0,1]$. If $S \sim 1$, the approximation can be considered a candidate solution. Otherwise, $\hat{F}$ is a (possibly unsafe) approximation. In line 1 we start by generating a set $\mathscr{I}$ of random inputs for F. Each input $\vec{x_i}$ is a m-tuple verifying precondition φ_{pre}, where m is the number of arguments of F. In line 2 we produce the training set $\mathscr{X}$. The independent inputs are generated by evaluating the base functions in $T = \langle t_1, t_2, \ldots, t_p \rangle$ with each tuple $\vec{x} \in \mathscr{I}$. 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 first element in the vectors of the training set. In line 3 we generate a first linear model by applying function CVLassoRegression to the generated training set. CVLassoRegression performs a linear regression

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

Input : $F \in \mathscr{D} \to \mathbb{R}^+$: target recurrence relation.
φ_{pre}: precondition defining $\mathscr{D}$.
$T \subseteq \mathscr{D} \to \mathbb{R}^+$: set of base functions.
Λ: 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, $k \geq 2$.
$\varepsilon \in \mathbb{R}^+$: threshold for term ($t_i \in T$) selection.

Output: $\hat{F} \in \mathsf{Exp}$: a candidate solution (or an approximation) for F.
$S \in [0,1]$: score, indicates the accuracy of the estimation (R^2).

1 $\mathscr{I} \leftarrow \{\vec{x}_i | \vec{x}_i \in \mathbb{Z}^m \wedge \varphi_{\text{pre}}(\vec{x}_i)\}_{i=1}^N$; // N Random inputs for F
2 $\mathscr{X} \leftarrow \{\langle F(\vec{x}) | E(T,\vec{x})\rangle | \vec{x} \in \mathscr{I}\}$; // Training set
3 $(\vec{\beta}', \beta_0') \leftarrow \mathsf{CVLassoRegression}(\mathscr{X}, \Lambda, k)$;
4 $(T', \mathscr{X}') \leftarrow \mathsf{RemoveTerms}(T, \mathscr{X}, \vec{\beta}', \beta_0', \varepsilon)$;
5 $(\vec{\beta}, \beta_0, S) \leftarrow \mathsf{LinearRegression}(\mathscr{X}')$;
6 $\hat{F} \leftarrow \lambda\vec{x} \cdot \vec{\beta}^T \times E(T', \vec{x}) + \beta_0$;
7 **return** $(\hat{F}, S)$;

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. The parameter k indicates performing $k-$fold cross-validation, 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 T together with their corresponding input values from the training set $\mathscr{X}$, returning the new set of base functions T' and its corresponding training set $\mathscr{X}'$. In line 5, standard linear regression (without regularization nor cross-validation) is applied, obtaining the final coefficients $\vec{\beta}$ and β_0. Additionally, from this step we also obtain the score 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 mainly relies on an SMT-solver and a CAS. Concretely, given the constrained recurrence relation $F \in \mathscr{D} \to \mathbb{R}^+$ defined as

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

Algorithm 2: Solution Checking (*Checker*).

Input : $F \in \mathscr{D} \to \mathbb{R}^+$: target recurrence relation.
φ_{pre}: precondition defining $\mathscr{D}$.
$\hat{F} \in \mathsf{Exp}$: a candidate solution for F.
Output: true if $\hat{F}$ is a solution for F, false otherwise.

```
1  φ_previous ← true ;
2  Formula ← true ;
3  foreach (e, φ) ∈ def(F) do
4  |  Eq ← replaceCalls("F(x⃗) − e = 0", F(x⃗), F̂, φ_pre, φ);
5  |  if ¬containsCalls(Eq, F) then
6  |  |  Eq ← simplifyCAS(inlineCalls(Eq, F̂, def(F̂)));
7  |  |  if supportedSMT(Eq) then
8  |  |  |  Formula ← "Formula ∧ (φ_pre ∧ φ_previous ∧ φ ⟹ Eq)";
9  |  |  |  φ_previous ← "φ_previous ∧ ¬φ" ;
10 |  |  else
11 |  |  |  return false;
12 |  |  end
13 |  else
14 |  |  return false;
15 |  end
16 end
17 return (⊭_SMT ⟦¬Formula⟧_SMT);
```

our algorithm constructs the logic formula:

$$\left[\!\!\left[\bigwedge_{i=1}^{k} \left(\left(\bigwedge_{j=1}^{i-1} \neg \varphi_j(\vec{x}) \right) \wedge \varphi_i(\vec{x}) \wedge \varphi_{\text{pre}}(\vec{x}) \implies \mathsf{Eq}_i \right) \right]\!\!\right]_{\text{SMT}} \tag{12}$$

where Eq_i is the result of replacing in $F(\vec{x}) = e_i(\vec{x})$ each occurrence of F, if possible, by the definition of the candidate solution $\hat{F}$ (by using replaceCalls in line 4), and performing a simplification by the CAS (by using simplifyCAS in line 6). A goal of such simplification is to obtain (sub)expressions supported by the SMT-solver. The function $\mathsf{replaceCalls}(\mathsf{expr}, F(\vec{x}'), \hat{F}, \varphi_{\text{pre}}, \varphi)$ replaces every subexpression in expr of the form $F(\vec{x}')$ by $\hat{F}(\vec{x}')$, if $\varphi_{\text{pre}}(\vec{x}') \wedge \varphi \implies \varphi_{\text{pre}}(\vec{x}')$. The operation $[\![e]\!]_{\text{SMT}}$ is the translation of any expression e to an SMT-LIB expression. Although all variables appearing in Formula 12 are declared as integers, we omit these details in Algorithm 2 and in Formula 12 for the sake of brevity. Note that this encoding is consistent with the evaluation (EvalFun) described in Section 3. 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 $\vec{x} \in \mathscr{D}$ such that $\mathsf{EvalFun}(F(\vec{x}))$ terminates. Otherwise, it returns false. Note that, if it is not possible to replace all occurrences of F by $\hat{F}$, or if after performing the simplification by simplifyCAS there are subexpressions not supported by the SMT-solver, then the algorithm finishes returning false.

5 Our Approach in the Context of Static Cost Analysis

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* `p/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. 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 [7] and its references).*

```
1 :- entry p/2: nnegint*var.
2 p(X,0):-
3   X=0.
4 p(X,Y):-
5   X>0,
6   X1 is X - 1,
7   p(X1,Y1),
8   p(Y1,Y2),
9   Y is Y2 + 1.
```

Figure 2: A program with a nested recursion.

The CiaoPP system first infers size relations for the different arguments of predicates, using a rich set of size metrics (see [13, 17] 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 $\mathtt{S_p}(x)$ that gives the size of the output argument of `p/2` (the second one), as a function of the size (x) of the input argument (the first one). For this purpose, the following size relations for $\mathtt{S_p}(x)$ are automatically set up (the same as the recurrence in Equation 1 used in Section 2 as example):

$$\begin{array}{lll} \mathtt{S_p}(x) = & 0 & \text{if } x = 0 \\ \mathtt{S_p}(x) = & \mathtt{S_p}(\mathtt{S_p}(x-1))+1 & \text{if } x > 0 \end{array} \tag{13}$$

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 latter) 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, sketched in next section, obtains the closed form $\hat{\mathtt{S}}_\mathtt{p}(x) = x$, 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 `p/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 `p/2`, represented as $\mathtt{C_p}(x)$, CiaoPP sets up the following cost relations, by using the size relations inferred previously:

$$\begin{array}{lll} \mathtt{C_p}(x) = & 1 & \text{if } x = 0 \\ \mathtt{C_p}(x) = & \mathtt{C_p}(x-1) + \mathtt{C_p}(\mathtt{S_p}(x-1)) + 1 & \text{if } x > 0 \end{array} \tag{14}$$

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 $\mathtt{S_p}(x)$, whose closed form $\mathtt{S_p}(x) = x$ is computed by our approach, as already explained. Plugin such closed form into the recurrence relation above, it can be solved now by CiaoPP, obtaining $\mathtt{C_p}(x) = 2^{x+1} - 1$.

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 generated during static cost analysis. Our prototype takes a recurrence and returns a closed form obtained together with two measures: 1) the accuracy of the estimation (*score*) of the candidate closed-form solution generated by the machine learning phase, and 2) an indication of whether such closed form is an exact solution of the recurrence (i.e., if it has been formally verified). It is implemented in *Python 3*, using *Sympy* [11] as CAS, and *Scikit-Learn* [14] for the regression with Lasso regularization. We use *Z3* [12] as SMT-solver, and *Z3Py* [19] as interface.

Our experimental results are shown in Table 1. Column **Bench** shows the name that we have assigned to each recurrence that we have chosen (which is inspired in the logic program such recurrence originated from during cost/size analysis), and Column **Recurrence** shows their definitions, where we use the same function symbol, f, for all of them. Such recurrences are challenging for CiaoPP, either because they cannot be solved by any of the back-end solvers, or because they are necessarily over-estimated in the solving process. Some recurrences, like **nested**, are problematic even for most of the current state-of-the-art solvers. For each recurrence and for each argument x of it, there is an implicit constraint that x is an integer, $x \geq 0$, which we do not include for brevity. Also, the disjunction of all the constraints defining the cases of the recurrence is a constraint φ_{pre} that defines the domain of the corresponding function. Column **CF** shows the closed forms obtained by our previous recurrence solver, and Column **CFNew** shows the closed forms obtained by our approach, applying Algorithms 1 and 2. All of them have been verified as exact solutions to the recurrences by Algorithm 2. As already said in Section 3, the closed form solution of any recurrence gives the same results as the recurrence for the inputs for which the evaluation of the recurrence terminates. The base cases of the recurrences have been considered apart from the others, and included in the final solution. Finally, Column **T(s)** shows the total time, in seconds (executing on a MacBook Pro machine, 2.4GHz Intel Core i7 CPU, 8 GB 1333 MHz DDR3 memory), needed to obtain the closed forms and verify them. For all the experiments, we have set $k = 2$, in order to perform $2-$fold cross-validation. We have also set the range for λ to 100 values taken from the interval $[0.001, 1]$, and $\varepsilon = 0.05$. Regarding the set T of base functions, for recurrences with one or two arguments, we provide a predefined set of representative functions of the most common complexity orders, as well as some compositions of them. For recurrences with three or more arguments, we provide an initial set of simple functions, that are combined automatically to generate the base functions t_i for the set T.

Table 1: Closed forms obtained with the previous (**CF**) and new solver (**CFNew**).

Bench	Recurrence	CF	CFNew	T (s)
merge-sz	$f(x,y)=\begin{cases} max(f(x-1,y), & \\ f(x,y-1))+1 & \text{if } x>0\wedge y>0 \\ x & \text{if } x>0\wedge y=0 \\ y & \text{if } x=0\wedge y>0 \end{cases}$	–	$x+y$	0.92
merge	$f(x,y)=\begin{cases} max(f(x-1,y), & \\ f(x,y-1))+1 & \text{if } x>0\wedge y>0 \\ 0 & \text{if } x=0\vee y=0 \end{cases}$	–	$\begin{cases} x+y-1 & \text{if } x>0\wedge y>0 \\ 0 & \text{if } x=0\vee y=0 \end{cases}$	0.71
nested	$f(x)=\begin{cases} f(f(x-1))+1 & \text{if } x>0 \\ 0 & \text{if } x=0 \end{cases}$	–	x	0.13
open-zip	$f(x,y)=\begin{cases} f(x-1,y-1)+1 & \text{if } x>0\wedge y>0 \\ f(x,y-1)+1 & \text{if } x=0\wedge y>0 \\ f(x-1,y)+1 & \text{if } x>0\wedge y=0 \\ 0 & \text{if } x=0\wedge y=0 \end{cases}$	–	$\max(x,y)$	0.12
div	$f(x,y)=\begin{cases} f(x-y,y)+1 & \text{if } x>=y\wedge y>0 \\ 0 & \text{if } x<y\wedge y>0 \end{cases}$	–	$\left\lfloor \frac{x}{y} \right\rfloor$	0.13
div-ceil	$f(x,y)=\begin{cases} f(x-y,y)+1 & \text{if } x>=y\wedge y>0 \\ 1 & \text{if } x<y\wedge x>0 \\ 0 & \text{if } x=0\wedge y>0 \end{cases}$	–	$\left\lceil \frac{x}{y} \right\rceil$	0.12
s-max	$f(x,y)=\begin{cases} max(y,f(x-1,y))+1 & \text{if } x>0 \\ y & \text{if } x=0 \end{cases}$	$x+y$	$x+y$	0.12
s-max-1	$f(x,y)=\begin{cases} max(y,f(x-1,y+1))+1 & \text{if } x>0 \\ y & \text{if } x=0 \end{cases}$	–	$2x+y$	0.14
sum-osc	$f(x,y)=\begin{cases} f(x-1,y)+1 & \text{if } x>0\wedge y>0 \\ f(x+1,y-1)+y & \text{if } x=0\wedge y>0 \\ 1 & \text{if } y=0 \end{cases}$	–	$\begin{cases} x+\frac{y^2}{2}+\frac{3y}{2} & \text{if } y>0 \\ 1 & \text{if } y=0 \end{cases}$	0.13

As we can see, none of the recurrences are solvable by the current CiaoPP solver, except `s-max`. The specialized solver for such recurrence has been developed relatively recently. Also, none of the recurrences are solvable by the CASs *Mathematica* [10] and *Sympy* [11], which we can arguably consider state-of-the-art CASs. In contrast, our new solver is able to infer exact closed-forms functions for all the recurrences in a reasonable time.

7 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 a sparse linear regression via Lasso regularization and cross-validation 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 with recurrences generated by the cost analysis module of CiaoPP, and are not solvable by it nor by the (arguably state-of-the-art) CASs *Mathematica* and *Sympy*. The experimental results are quite

promising, showing that our approach can find exact, verified, closed-form solutions, in a reasonable time, for such recurrences, which imply potentially, 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 to the main predicate, inferring not useful information at all.

Since our technique uses linear regression with a randomly generated training set (by evaluating the recurrence to obtain the dependent value), it is not guaranteed that a solution can be found. Even if an exact solution is found in the first stage, it is not always possible to prove its correctness in the second stage. Therefore, in this sense, this approach is *not complete*. However, it is able to find some solutions that current state-of-the-art solvers are unable to find. 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, adapting the verification stage accordingly. Note that we need to require the termination of the recurrence evaluation as a precondition for the conclusions obtained. This is also due to the particular evaluation strategy of recurrences that we are considering. In practice, non-terminating recurrences can be discarded in the first stage, by setting a timeout. Our approach can also be combined with a termination prover in order to guarantee such a precondition. Finally, note that an alternative use of our tool is to omit the verification stage, using only the closed-form function inferred by the first stage, together with an error measure. This can be useful in some applications (e.g., granularity control in parallel/distributed computing) where it is enough to have good although unsafe approximations.

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 in order to improve the static cost analysis. We also plan to further refine and improve our algorithms in several directions. As already explained, currently the set T of base functions is fixed, user-provided. We plan to automatically infer it by using different heuristics. We can perform an automatic analysis of the recurrence we are solving, to extract some features that allow selection of the terms that most likely are part of the solution. For example, if the recurrence has a nested, double recursion, then we can select a quadratic term, etc. Also, machine learning techniques may be applied to learn a good set of base functions from some features of the programs.

Acknowledgments This work has been 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 Louis Rustenholz, John Gallagher, Manuel Hermenegildo, José F. Morales and the anonymous reviewers for very useful feedback. Louis Rustenholz also recreated the experimental results and double-checked them.

References

[1] E. Albert, P. Arenas, S. Genaim & G. Puebla (2011): *Closed-Form Upper Bounds in Static Cost Analysis*. *Journal of Automated Reasoning* 46(2), pp. 161–203, doi:10.1007/s10817-010-9174-1.

[2] S. K. Debray & N. W. Lin (1993): *Cost Analysis of Logic Programs*. *ACM TOPLAS* 15(5), pp. 826–875, doi:10.1145/161468.161472.

[3] S. K. Debray, N.-W. Lin & M. V. Hermenegildo (1990): *Task Granularity Analysis in Logic Programs*. In: *Proc. PLDI'90*, ACM, pp. 174–188, doi:10.1145/93542.93564.

[4] S. K. Debray, P. Lopez-Garcia, M. V. Hermenegildo & N.-W. Lin (1997): *Lower Bound Cost Estimation for Logic Programs*. In: *ILPS'97*, MIT Press, pp. 291–305, doi:10.7551/mitpress/4283.001.0001.

[5] Trevor Hastie, Robert Tibshirani & Jerome Friedman (2009): *The Elements of Statistical Learning: Data Mining, Inference and Prediction*, second edition. Springer New York, NY, doi:10.1007/978-0-387-84858-7.

[6] Trevor Hastie, Robert Tibshirani & Martin Wainwright (2015): *Statistical Learning with Sparsity: The Lasso and Generalizations*. Chapman & Hall/CRC, doi:10.1201/b18401.

[7] M. Hermenegildo, G. Puebla, F. Bueno & P. Lopez Garcia (2005): *Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and The Ciao System Preprocessor)*. *Science of Computer Programming* 58(1–2), pp. 115–140, doi:10.1016/j.scico.2005.02.006.

[8] P. Lopez-Garcia, L. Darmawan, M. Klemen, U. Liqat, F. Bueno & M. V. Hermenegildo (2018): *Interval-based Resource Usage Verification by Translation into Horn Clauses and an Application to Energy Consumption*. *TPLP* 18(2), pp. 167–223, doi:10.1017/S1471068418000042.

[9] P. Lopez-Garcia, M. Klemen, U. Liqat & M. V. Hermenegildo (2016): *A General Framework for Static Profiling of Parametric Resource Usage*. *TPLP (ICLP'16 Special Issue)* 16(5-6), pp. 849–865, doi:10.1017/S1471068416000442.

[10] (2023): *Wolfram Mathematica (v13.2): the World's Definitive System for Modern Technical Computing*. `https://www.wolfram.com/mathematica`. Accessed: May 25, 2023.

[11] Aaron Meurer, Christopher P. Smith, Mateusz Paprocki, Ondřej Čertík, Sergey B. Kirpichev, Matthew Rocklin, AMiT Kumar, Sergiu Ivanov, Jason Keith Moore, Sartaj Singh, Thilina Rathnayake, Sean Vig, Brian E. Granger, Richard P. Muller, Francesco Bonazzi, Harsh Gupta, Shivam Vats, Fredrik Johansson, Fabian Pedregosa, Matthew J. Curry, Andy R. Terrel, Štěpán Roučka, Ashutosh Saboo, Isuru Fernando, Sumith Kulal, Robert Cimrman & Anthony Scopatz (2017): *SymPy: symbolic computing in Python*. *PeerJ Computer Science* 3, p. e103, doi:10.7717/peerj-cs.103.

[12] Leonardo Mendonça de Moura & Nikolaj Bjørner (2008): *Z3: An Efficient SMT Solver*. In C. R. Ramakrishnan & Jakob Rehof, editors: *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008*, *Lecture Notes in Computer Science* 4963, Springer, pp. 337–340, doi:10.1007/978-3-540-78800-3_24.

[13] J. Navas, E. Mera, P. Lopez-Garcia & M. Hermenegildo (2007): *User-Definable Resource Bounds Analysis for Logic Programs*. In: *Proc. of ICLP'07*, *LNCS* 4670, Springer, pp. 348–363, doi:10.1007/978-3-540-74610-2_24.

[14] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake VanderPlas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot & Edouard Duchesnay (2011): *Scikit-learn: Machine Learning in Python*. *Journal of Machine Learning Research* 12, pp. 2825–2830, doi:10.5555/1953048.2078195. Available at `https://dl.acm.org/doi/10.5555/1953048.2078195`.

[15] A. Podelski & A. Rybalchenko (2004): *A Complete Method for the Synthesis of Linear Ranking Functions*. In: *VMCAI'04*, *LNCS* 2937, Springer, pp. 239–251, doi:10.1007/978-3-540-24622-0_20.

[16] M. Rosendahl (1989): *Automatic Complexity Analysis*. In: *4th ACM Conference on Functional Programming Languages and Computer Architecture (FPCA'89)*, ACM Press, pp. 144–156, doi:10.1145/99370.99381.

[17] A. Serrano, P. Lopez-Garcia & M. V. Hermenegildo (2014): *Resource Usage Analysis of Logic Programs via Abstract Interpretation Using Sized Types*. *TPLP, ICLP'14 Special Issue* 14(4-5), pp. 739–754, doi:10.1017/S147106841400057X.

[18] B. Wegbreit (1975): *Mechanical Program Analysis*. *Communications of the ACM* 18(9), pp. 528–539, doi:10.1145/361002.361016.

[19] (2023): *Z3 API in Python*. `https://ericpony.github.io/z3py-tutorial`. Accessed: May 25, 2023.