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

Maximiliano Klemen\textsuperscript{1}, Miguel Ángel Carreira-Perpiñán\textsuperscript{2} and Pedro Lopez-Garcia\textsuperscript{1,3}

\textsuperscript{1}IMDEA Software Institute, Spain
\textsuperscript{2}University of California, Merced, USA
\textsuperscript{3}Spanish Council for Scientific Research (CSIC)

10th Workshop on Horn Clauses for Verification and Synthesis (HCVS)
April 23, 2023, Paris, France (co-located with ETAPS)
Motivating application: automatic static cost analysis/verification of Horn-clause programs → e.g., the CiaoPP system.
  + Allows analysis of other languages/IRs via transformation into Horn Clauses.
  + (Ciao) Prolog → direct translation,
  + but also C, Java (source/bytecode), ISA, LLVM IR, ...

Resources: non-func. numerical properties about the execution of a program.
  • Examples: resolution steps, execution time, energy consumption, # of calls to a predicate, # of network accesses, # of transactions, ...

Goal of static analysis:
estimating the resource usage of the execution of a program without running it with concrete data, as function of input data sizes and possibly other parameters.

Typical size metrics → actual value of a number, the length of a list, the number of constant and function symbols of a term, etc.

Resource analysis is very useful:
  • Automatic program optimization.
  • Verification of resource-related specifications.
  • Detection of performance bugs, help guiding software design, ...
  Example: developing energy-efficient software.
These techniques strongly depend on solving (or safely approximating) recurrence relations → bottleneck.

Using Computer Algebra Systems (CAS) or specialized solvers poses several difficulties and limitations for some recurrences:

- Contain complex expressions or recursive structures.
- Don’t have the form required by such solvers
  → e.g., an input data size variable does not decrease, but increases.

As a result, ad-hoc techniques need to be developed for such cases.
Our Proposal: Guess and Check Approach

- **Guess**: machine-learning sparse regression techniques.
- **Check**: Combination of an SMT-solver and a CAS.

Novel, general method for solving arbitrary, constrained recurrence relations:
Consider following Horn-clause program, in Prolog syntax:

\[
\begin{align*}
p(X, 0) & : - X = 0. \\
p(X, Y) & : - X > 0, \ X1 \ \text{is} \ \ X - 1, \ p(X1, Y1), \ p(Y1, Y2), \ Y \ \text{is} \ Y2 + 1.
\end{align*}
\]
Consider following Horn-clause program, in Prolog syntax:

\[
\begin{align*}
p(X, 0) & : - X = 0. \\
p(X, Y) & : - X > 0, X1 \text{ is } X - 1, p(X1, Y1), p(Y1, Y2), Y \text{ is } Y2 + 1.
\end{align*}
\]

CiaoPP first infers size relations for the different arguments of predicates.
Consider following Horn-clause program, in Prolog syntax:

\[
\begin{align*}
p(X, 0) & : - X = 0. \\
p(X, Y) & : - X > 0, X1 \textbf{ is } X - 1, p(X1, Y1), p(Y1, Y2), Y \textbf{ is } Y2 + 1.
\end{align*}
\]

CiaoPP first infers size relations for the different arguments of predicates.
Assume a calling mode where first argument is input and second one output.
Consider following Horn-clause program, in Prolog syntax:

\[
\begin{align*}
p(X, 0) & : - X = 0. \\
p(X, Y) & : - X > 0, \text{ X1 is X - 1, } p(X1, Y1), \text{ p(Y1, Y2), } \text{ Y is Y2 + 1.}
\end{align*}
\]

CiaoPP first infers size relations for the different arguments of predicates.
Assume a calling mode where first argument is input and second one output.
It will try to infer the size of the output argument as a function of the size of
the input argument: \( S_p(x) \).
Consider following Horn-clause program, in Prolog syntax:

```
p(X, 0) :- X = 0.
p(X, Y) :- X > 0, X1 is X - 1, p(X1, Y1), p(Y1, Y2), Y is Y2 + 1.
```

CiaoPP first infers size relations for the different arguments of predicates. Assume a calling mode where first argument is input and second one output. It will try to infer the size of the output argument as a function of the size of the input argument: \( S_p(x) \).

Using \( x = \text{size}(X) = X \) (actual value of \( X \)), size relations are set up:

\[
\begin{align*}
S_p(x) &= 0 \quad \text{if } x = 0 \\
S_p(x) &= S_p(S_p(x - 1)) + 1 \quad \text{if } x > 0
\end{align*}
\]
Consider following Horn-clause program, in Prolog syntax:

\[
\begin{align*}
p(X, 0) & : - X = 0. \\
p(X, Y) & : - X > 0, X1 \text{ is } X - 1, p(X1, Y1), p(Y1, Y2), Y \text{ is } Y2 + 1.
\end{align*}
\]

CiaoPP first infers size relations for the different arguments of predicates. Assume a calling mode where first argument is input and second one output. It will try to infer the size of the output argument as a function of the size of the input argument: \( S_p(x) \).

Using \( x = size(X) = X \) (actual value of \( X \)), size relations are set up:

\[
\begin{align*}
S_p(x) &= 0 \quad \text{if } x = 0 \\
S_p(x) &= S_p(S_p(x - 1)) + 1 \quad \text{if } x > 0
\end{align*}
\]

CiaoPP’s modular solver fails to find a closed-form function for it.
Consider following Horn-clause program, in Prolog syntax:

\[
\begin{align*}
p(X, 0) & : - X = 0. \\
p(X, Y) & : - X > 0, \ X1 \text{ is } X - 1, \ p(X1, Y1), \ p(Y1, Y2), \ Y \text{ is } Y2 + 1.
\end{align*}
\]

CiaoPP first infers size relations for the different arguments of predicates. Assume a calling mode where first argument is input and second one output. It will try to infer the size of the output argument as a function of the size of the input argument: \( S_p(x) \).

Using \( x = \text{size}(X) = X \) (actual value of \( X \)), size relations are set up:

\[
\begin{align*}
S_p(x) &= 0 \quad \text{if } x = 0 \\
S_p(x) &= S_p(S_p(x - 1)) + 1 \quad \text{if } x > 0
\end{align*}
\]

CiaoPP’s modular solver fails to find a closed-form function for it. It is a nested recurrence that cannot be solved by most state-of-the-art solvers.
Consider following Horn-clause program, in Prolog syntax:

\[
\begin{align*}
p(X, 0) & : - X = 0. \\
p(X, Y) & : - X > 0, X1 \text{ is } X - 1, p(X1, Y1), p(Y1, Y2), Y \text{ is } Y2 + 1.
\end{align*}
\]

CiaoPP first infers size relations for the different arguments of predicates.

Assume a calling mode where first argument is input and second one output.

It will try to infer the size of the output argument as a function of the size of the input argument: \( S_p(x) \).

Using \( x = size(X) = X \) (actual value of \( X \)), size relations are set up:

\[
\begin{align*}
S_p(x) & = 0 \quad \text{if } x = 0 \\
S_p(x) & = S_p(S_p(x - 1)) + 1 \quad \text{if } x > 0
\end{align*}
\]

CiaoPP’s modular solver fails to find a closed-form function for it.

It is a nested recurrence that cannot be solved by most state-of-the-art solvers.

Our proposed approach obtains \( S_p(x) = x \) (exact solution).
Consider following Horn-clause program, in Prolog syntax:

\[
\begin{align*}
p(X, 0) & \leftarrow X = 0. \\
p(X, Y) & \leftarrow X > 0, \ X1 \ is \ X - 1, \ p(X1, Y1), \ p(Y1, Y2), \ Y \ is \ Y2 + 1.
\end{align*}
\]
Consider following Horn-clause program, in Prolog syntax:

\[
\begin{align*}
p(X, 0) & : - X = 0. \\
p(X, Y) & : - X > 0, X1 \text{ is } X - 1, p(X1, Y1), p(Y1, Y2), Y \text{ is } Y2 + 1.
\end{align*}
\]

CiaoPP uses the size relations to infer the computational cost of a call to \( p/2 \), denoted \( C_p(x) \)
Consider following Horn-clause program, in Prolog syntax:

\[
\begin{align*}
p(X, 0) & : - X = 0. \\
p(X, Y) & : - X > 0, \text{ } X1 \text{ } \text{is} \text{ } X - 1, \text{ } p(X1, Y1), \text{ } p(Y1, Y2), \text{ } Y \text{ } \text{is} \text{ } Y2 + 1.
\end{align*}
\]

CiaoPP uses the size relations to infer the computational cost of a call to \( p/2 \), denoted \( C_p(x) \)

\[
\rightarrow \text{(in the example, number of resolution steps, and}
\]

without our approach CiaoPP would infer \( S_p(x) = \infty \) and \( C_p(x) = \infty \).

Not being able to solve a “simple” recurrence can cause arbitrarily large losses of precision in size/cost analysis.
The Context: Static Cost Analysis (CiaoPP)

Consider following Horn-clause program, in Prolog syntax:

\[
\begin{align*}
p(X, 0) & : - X = 0. \\
p(X, Y) & : - X > 0, X1 \text{ is } X - 1, p(X1, Y1), p(Y1, Y2), Y \text{ is } Y2 + 1.
\end{align*}
\]

CiaoPP uses the size relations to infer the computational cost of a call to \( p/2 \), denoted \( C_p(x) \)

\[
\begin{align*}
C_p(x) &= 1 \text{ if } x = 0 \\
C_p(x) &= C_p(x-1) + C_p(S_p(x-1)) + 1 \text{ if } x > 0
\end{align*}
\]

Plug the closed form \( S_p(x) = x \) inferred by our approach, CiaoPP obtains

\[ C_p(x) = 2x + 1 \]

Without our approach CiaoPP would infer \( S_p(x) = \infty \) and \( C_p(x) = \infty \).

Not being able to solve a "simple" recurrence can cause arbitrarily large losses of precision in size/cost analysis.
Consider following Horn-clause program, in Prolog syntax:

\[
\begin{align*}
p(X, 0) & : - X = 0. \\
p(X, Y) & : - X > 0, X1 \text{ is } X - 1, p(X1, Y1), p(Y1, Y2), Y \text{ is } Y2 + 1.
\end{align*}
\]

CiaoPP uses the size relations to infer the computational cost of a call to \( p/2 \), denoted \( C_p(x) \)

\[ \rightarrow \text{(in the example, number of resolution steps, and assuming the builtins >/2 and is/2 have zero cost)} \]

It sets up the following recurrence:

\[
\begin{align*}
C_p(x) &= 1 & \text{if } x = 0 \\
C_p(x) &= C_p(x - 1) + C_p(S_p(x - 1)) + 1 & \text{if } x > 0
\end{align*}
\]
Consider following Horn-clause program, in Prolog syntax:

\[
\begin{align*}
\text{p}(X, 0) & : - X = 0. \\
\text{p}(X, Y) & : - X > 0, \ X1 \text{ is } X - 1, \ \text{p}(X1, Y1), \ \text{p}(Y1, Y2), \ Y \text{ is } Y2 + 1.
\end{align*}
\]

CiaoPP uses the size relations to infer the computational cost of a call to \text{p}/2, denoted \( C_p(X) \)

\[ C_p(x) = \begin{cases} 1 & \text{if } x = 0 \\ C_p(x - 1) + C_p(S_p(x - 1)) + 1 & \text{if } x > 0 \end{cases} \]

Plugin the closed form \( S_p(x) = x \) inferred by our approach,
Consider following Horn-clause program, in Prolog syntax:

\[
\begin{align*}
p(X, 0) & : \quad X = 0. \\
p(X, Y) & : \quad X > 0, \ X1 \text{ is } X - 1, \ p(X1, Y1), \ p(Y1, Y2), \ Y \text{ is } Y2 + 1.
\end{align*}
\]

CiaoPP uses the size relations to infer the computational cost of a call to \( p/2 \), denoted \( C_p(x) \)

\[
\begin{align*}
&C_p(x) = 1 \quad \text{if } x = 0 \\
&C_p(x) = C_p(x - 1) + C_p(S_p(x - 1)) + 1 \quad \text{if } x > 0
\end{align*}
\]

It sets up the following recurrence:

Plugin the closed form \( S_p(x) = x \) inferred by our approach,
Consider following Horn-clause program, in Prolog syntax:

\[
\begin{align*}
p(X, 0) & : - X = 0. \\
p(X, Y) & : - X > 0, X1 \text{ is } X - 1, p(X1, Y1), p(Y1, Y2), Y \text{ is } Y2 + 1.
\end{align*}
\]

CiaoPP uses the size relations to infer the computational cost of a call to \( p/2 \), denoted \( C_p(x) \)

\[
\rightarrow \text{ (in the example, number of resolution steps, and assuming the builtins } >/2 \text{ and } is/2 \text{ have zero cost)}
\]

It sets up the following recurrence:

\[
\begin{align*}
C_p(x) &= 1 \quad \text{if } x = 0 \\
C_p(x) &= C_p(x - 1) + C_p(x - 1) + 1 \quad \text{if } x > 0
\end{align*}
\]

Plugin the closed form \( s_p(x) = x \) inferred by our approach,
Consider following Horn-clause program, in Prolog syntax:

\[
\begin{align*}
\text{p}(X, 0) & : - X = 0. \\
\text{p}(X, Y) & : - X > 0, X1 \text{ is } X - 1, \text{ p}(X1, Y1), \text{ p}(Y1, Y2), Y \text{ is } Y2 + 1.
\end{align*}
\]

CiaoPP uses the size relations to infer the computational cost of a call to \( \text{p}/2 \), denoted \( \mathcal{C}_p(x) \)

\( \rightarrow \) (in the example, number of resolution steps, and assuming the builtins \( >/2 \) and \( \text{is}/2 \) have zero cost)

It sets up the following recurrence:

\[
\begin{align*}
\mathcal{C}_p(x) &= 1 \quad \text{if } x = 0 \\
\mathcal{C}_p(x) &= 2 \mathcal{C}_p(x - 1) + 1 \quad \text{if } x > 0
\end{align*}
\]

Plugin the closed form \( \mathcal{S}_p(x) = x \) inferred by our approach,
Consider following Horn-clause program, in Prolog syntax:

\[
\begin{align*}
p(X, 0) & : - X = 0. \\
p(X, Y) & : - X > 0, X1 \text{ is } X - 1, p(X1, Y1), p(Y1, Y2), Y \text{ is } Y2 + 1.
\end{align*}
\]

CiaoPP uses the size relations to infer the computational cost of a call to \( p/2 \), denoted \( C_p(x) \):

\[C_p(x) = \begin{cases} 
1 & \text{if } x = 0 \\
2 C_p(x - 1) + 1 & \text{if } x > 0
\end{cases}\]

It sets up the following recurrence:

\[
\begin{align*}
C_p(x) &= 1 \quad \text{if } x = 0 \\
C_p(x) &= 2 C_p(x - 1) + 1 \quad \text{if } x > 0
\end{align*}
\]

Plugin the closed form \( S_p(x) = x \) inferred by our approach, CiaoPP obtains \( C_p(x) = 2^{x+1} - 1 \).
Consider following Horn-clause program, in Prolog syntax:

```prolog
p(X, 0) :- X = 0.
p(X, Y) :- X > 0, X1 is X - 1, p(X1, Y1), p(Y1, Y2), Y is Y2 + 1.
```

CiaoPP uses the size relations to infer the computational cost of a call to \( p/2 \), denoted \( C_p(x) \)

\[ C_p(x) = \begin{cases} 
1 & \text{if } x = 0 \\
2 C_p(x - 1) + 1 & \text{if } x > 0 
\end{cases} \]

It sets up the following recurrence:

- Plug the closed form \( S_p(x) = x \) inferred by our approach,
  CiaoPP obtains \( C_p(x) = 2^{x+1} - 1 \).
- Without our approach CiaoPP would infer \( S_p(x) = \infty \) and \( C_p(x) = \infty \).
Consider following Horn-clause program, in Prolog syntax:

\[
\begin{align*}
p(X, 0) & : - X = 0. \\
p(X, Y) & : - X > 0, X1 \text{ is } X - 1, p(X1, Y1), p(Y1, Y2), Y \text{ is } Y2 + 1.
\end{align*}
\]

CiaoPP uses the size relations to infer the computational cost of a call to \( p \/ 2 \), denoted \( C_p(x) \)

\[
\begin{align*}
C_p(x) & = 1 \quad \text{if } x = 0 \\
C_p(x) & = 2 C_p(x - 1) + 1 \quad \text{if } x > 0
\end{align*}
\]

It sets up the following recurrence:

Plugin the closed form \( S_p(x) = x \) inferred by our approach, CiaoPP obtains \( C_p(x) = 2^{x+1} - 1 \).

Without our approach CiaoPP would infer \( S_p(x) = \infty \) and \( C_p(x) = \infty \).

Not being able to solve a “simple” recurrence can cause arbitrarily large losses of precision in size/cost analysis.
Guess: First Stage of our Recurrence Solving Method

- Given the previous recurrence, with \( s_p(x) \equiv f(x) \):
  \[
  f(x) = \begin{cases} 
  0 & \text{if } x = 0 \\
  f(f(x - 1)) + 1 & \text{if } x > 0
  \end{cases}
  \]
- We use sparse linear regression to “guess” a candidate solution \( \hat{f}(\bar{x}) \) for it.
- We use a set of “base functions” \( T \), e.g.:
  \[T = \{ \lambda x.x, \lambda x.x^2, \lambda x.x^3, \lambda x.\lceil\log_2(x)\rceil, \lambda x.2^x, \lambda x.x \cdot \lceil\log_2(x)\rceil \}\]
  - Currently, \( T \) is fixed \( \rightarrow \) base functions that are representative of the common complexity orders.
  - We'll comment later about plans to obtain it.
- Model obtained: linear combination of terms \( t_i \) in \( T \):
  \[
  \hat{f}(\bar{x}) = \beta_0 + \beta_1 t_1(\bar{x}) + \beta_2 t_2(\bar{x}) + \cdots + \beta_n t_n(\bar{x})
  \]
  - \( \beta_i \)'s: coefficients (real numbers) estimated by regression
  - Goal: only a few coefficients are nonzero.
1. Generate a training set $S$.
   - Randomly generate input values to the recurrence $\rightarrow X_{\text{train}} = \{ \bar{x}_1, \ldots, \bar{x}_k \}$.
   - For each input value $\bar{x} \in X_{\text{train}}$, generate a training case $s$:
     
     $$s = \langle b, c_1, \ldots, c_n \rangle$$

     $c_i$: result (a scalar) of evaluating the base function $t_i \in T$ for input value $\bar{x}$
     $\rightarrow c_i = \llbracket t_i \rrbracket_{\bar{x}}$ for $1 \leq i \leq n$

     $b$ (dependent value): result (a scalar) of evaluating the recurrence for $\bar{x}$
     $\rightarrow b = f(\bar{x})$

   - Example: if $\bar{x} = \langle 5 \rangle$, then
     
     $$s = \langle f(5), \llbracket x \rrbracket_5, \llbracket x^2 \rrbracket_5, \llbracket x^3 \rrbracket_5, \llbracket \lceil \log_2(x) \rceil \rrbracket_5, \ldots \rangle$$
     $$= \langle 5, 5, 25, 125, 3, \ldots \rangle$$
2. Perform sparse linear regression using $S$:
   - Result: (column) vector $\tilde{\beta}$ of coefficients and an independent coefficient $\beta_0$.
   - Lasso regularization on the coefficients $\beta_i$.
   - $\ell_1$: penalty to encourage coefficients whose associated base functions have a small correlation with the dependent value to be exactly zero.
   - The level of penalization is controlled by a hyperparameter $\lambda \geq 0$.
     $\rightarrow$ found via cross-validation on a separate validation set (generated similarly as the training set $X_{\text{train}}$).

3. Obtain a measure $R^2$ of the accuracy of the estimation:
   $\rightarrow$ Using a test set $X_{\text{test}}$ of input values to the recurrence (generated similarly to $X_{\text{train}}$).

4. Round to zero the coefficient less than a given threshold $\epsilon$.
   $\rightarrow$ to discard the corresponding base functions.
   $\rightarrow$ We call it the “$\epsilon$-rounding”: $rm_\epsilon(\tilde{\beta}^T)$

5. The resulting closed-form is
   \[
   \hat{f}(\bar{x}) = rm_\epsilon(\tilde{\beta}^T) \cdot E(T, \bar{x}) + \beta_0
   \]
   $\rightarrow$ $E(T, \bar{x})$: vector of the terms in $T$ with the arguments bound to $\bar{x}$.
   - Both the Lasso regularization and the zero $\epsilon$-rounding discard many terms from $T$ in the final closed-form function.
6. Perform standard linear regression (without Lasso regularization)
   - on the same training set \( S \), but
   - different \( T \): removing from \( T \) the base functions corresponding to the coefficients \( \beta_i \) made zero previously (by Lasso and \( \epsilon \)-rounding).
   - In our example, we obtain (with \( \epsilon = 0.001 \)):
     \[
     \hat{f}(x) = 1.0 \times \text{ and } R^2 = 1
     \]
   - Since \( R^2 = 1 \), then \( \hat{f}(x) = x \) is a candidate closed-form solution
     → exact prediction of the recurrence for the test set.
   - If it was \( R^2 < 1 \), then \( \hat{f}(x) \) would be an approximation.
     → Still, can be useful in some applications (e.g., granularity control in parallel/distributed computing).
Check: Second Stage of our Recurrence Solving Method

- Verify whether the guessed candidate function is actually a solution for the recurrence.

- Example: the recurrence
  \[
  f(x) = \begin{cases} 
  0 & \text{if } x = 0 \\
  f(f(x - 1)) + 1 & \text{if } x > 0 
  \end{cases}
  \]

  is encoded as a first order logic formula
  \[
  \forall x \ (( x = 0 \implies f(x) = 0 ) \land ( x > 0 \implies f(x) = f(f(x - 1)) + 1 ))
  \]

- References to the target \( f(x) \) are replaced by the candidate \( \hat{f}(x) = x \).
  \[
  \forall x \ (( x = 0 \implies f(x) = 0 ) \land ( x > 0 \implies f(x) = f(f(x - 1)) + 1 ))
  \]

- If the negation of such formula is unsatisfiable, then the candidate function is an exact solution.

- We use an SMT-solver to check satisfiability.

- It is unsatisfiable \( \rightarrow \hat{f}(x) = x \) is an exact solution for \( f(x) \).

- Sometimes, it is necessary to consider a precondition for the domain of the recurrence, which is also included in the encoding. E.g., \( \hat{f}(x) = x \) if \( x \geq 0 \).
Check: Second Stage of our Recurrence Solving Method

- Verify whether the guessed candidate function is actually a solution for the recurrence.

**Example:** the recurrence
\[
\begin{align*}
f(x) &= 0 \quad \text{if } x = 0 \\
f(x) &= f(f(x - 1)) + 1 \quad \text{if } x > 0
\end{align*}
\]
is encoded as a first order logic formula
\[
\forall x \ ((x = 0 \implies f(x) = 0) \land (x > 0 \implies f(x) = f(f(x - 1)) + 1))
\]
References to the target \( f(x) \) are replaced by the candidate \( \hat{f}(x) = x \).
\[
\forall x \ ((x = 0 \implies f(x) = 0) \land (x > 0 \implies f(x) = f(f(x - 1)) + 1))
\]
If the negation of such formula is unsatisfiable, then the candidate function is an exact solution.

We use an SMT-solver to check satisfiability.

It is unsatisfiable \( \rightarrow \hat{f}(x) = x \) is an exact solution for \( f(x) \).

Sometimes, it is necessary to consider a precondition for the domain of the recurrence, which is also included in the encoding. E.g., \( \hat{f}(x) = x \) if \( x \geq 0 \).
Verify whether the guessed candidate function is actually a solution for the recurrence.

Example: the recurrence

\[
\begin{align*}
f(x) &= 0 & \text{if } x = 0 \\
f(x) &= f(f(x - 1)) + 1 & \text{if } x > 0
\end{align*}
\]

is encoded as a first order logic formula

\[
\forall x (( x = 0 \implies f(x) = 0 ) \land ( x > 0 \implies f(x) = f(f(x - 1)) + 1 ))
\]

References to the target \( f(x) \) are replaced by the candidate \( \hat{f}(x) = x \).

\[
\forall x (( x = 0 \implies x = 0 ) \land ( x > 0 \implies f(x) = f(f(x - 1)) + 1 ))
\]

If the negation of such formula is unsatisfiable, then the candidate function is an exact solution.

We use an SMT-solver to check satisfiability.

It is unsatisfiable \( \implies \hat{f}(x) = x \) is an exact solution for \( f(x) \).

Sometimes, it is necessary to consider a precondition for the domain of the recurrence, which is also included in the encoding. E.g., \( \hat{f}(x) = x \) if \( x \geq 0 \).
Check: Second Stage of our Recurrence Solving Method

- Verify whether the guessed candidate function is actually a solution for the recurrence.

- Example: the recurrence
  \[ f(x) = 0 \quad \text{if } x = 0 \]
  \[ f(x) = f(f(x - 1)) + 1 \quad \text{if } x > 0 \]
  
  is encoded as a first order logic formula
  \[ \forall x ((x = 0 \implies f(x) = 0) \land (x > 0 \implies f(x) = f(f(x - 1)) + 1)) \]

- References to the target \( f(x) \) are replaced by the candidate \( \hat{f}(x) = x \).
  \[ \forall x ((x = 0 \implies x = 0) \land (x > 0 \implies f(x) = f(f(x - 1)) + 1)) \]

- If the negation of such formula is unsatisfiable, then the candidate function is an exact solution.

- We use an SMT-solver to check satisfiability.

- It is unsatisfiable \( \rightarrow \hat{f}(x) = x \) is an exact solution for \( f(x) \).

- Sometimes, it is necessary to consider a precondition for the domain of the recurrence, which is also included in the encoding. E.g., \( \hat{f}(x) = x \) if \( x \geq 0 \).
Check: Second Stage of our Recurrence Solving Method

- Verify whether the guessed candidate function is actually a solution for the recurrence.
- Example: the recurrence
  \[ f(x) = \begin{cases} 0 & \text{if } x = 0 \\ f(x) = f(f(x - 1)) + 1 & \text{if } x > 0 \end{cases} \]
is encoded as a first order logic formula
  \[ \forall x \left( (x = 0 \implies f(x) = 0) \land (x > 0 \implies f(x) = f(f(x - 1)) + 1) \right) \]
- References to the target \( f(x) \) are replaced by the candidate \( \hat{f}(x) = x \).
  \[ \forall x \left( (x = 0 \implies x = 0) \land (x > 0 \implies x = f(f(x - 1)) + 1) \right) \]
- If the negation of such formula is unsatisfiable, then the candidate function is an exact solution.
- We use an SMT-solver to check satisfiability.
- It is unsatisfiable \( \implies \hat{f}(x) = x \) is an exact solution for \( f(x) \).
- Sometimes, it is necessary to consider a precondition for the domain of the recurrence, which is also included in the encoding. E.g., \( \hat{f}(x) = x \) if \( x \geq 0 \).
Verify whether the guessed candidate function is actually a solution for the recurrence.

Example: the recurrence

\[ f(x) = \begin{cases} 
  0 & \text{if } x = 0 \\
  f(f(x - 1)) + 1 & \text{if } x > 0 
\end{cases} \]

is encoded as a first order logic formula

\[ \forall x \ (( x = 0 \implies f(x) = 0 ) \land ( x > 0 \implies f(x) = f(f(x - 1)) + 1 )) \]

References to the target \( f(x) \) are replaced by the candidate \( \hat{f}(x) = x \).

\[ \forall x \ (( x = 0 \implies x = 0 ) \land ( x > 0 \implies x = f(f(x - 1)) + 1 )) \]

If the negation of such formula is unsatisfiable, then the candidate function is an exact solution.

We use an SMT-solver to check satisfiability.

It is unsatisfiable \( \rightarrow \hat{f}(x) = x \) is an exact solution for \( f(x) \).

Sometimes, it is necessary to consider a precondition for the domain of the recurrence, which is also included in the encoding. E.g., \( \hat{f}(x) = x \) if \( x \geq 0 \).
Check: Second Stage of our Recurrence Solving Method

- Verify whether the guessed candidate function is actually a solution for the recurrence.

Example: the recurrence
\[ f(x) = \begin{cases} 0 & \text{if } x = 0 \\ f(x) = f(f(x - 1)) + 1 & \text{if } x > 0 \end{cases} \]

is encoded as a first order logic formula
\[ \forall x \ ((x = 0 \implies f(x) = 0) \land (x > 0 \implies f(x) = f(f(x - 1)) + 1)) \]

References to the target \( f(x) \) are replaced by the candidate \( \hat{f}(x) = x \).
\[ \forall x \ ((x = 0 \implies x = 0) \land (x > 0 \implies x = f(x - 1) + 1)) \]

If the negation of such formula is unsatisfiable, then the candidate function is an exact solution.

We use an SMT-solver to check satisfiability.

It is unsatisfiable \( \rightarrow \hat{f}(x) = x \) is an exact solution for \( f(x) \).

Sometimes, it is necessary to consider a precondition for the domain of the recurrence, which is also included in the encoding. E.g., \( \hat{f}(x) = x \) if \( x \geq 0 \).
Check: Second Stage of our Recurrence Solving Method

- Verify whether the guessed candidate function is actually a solution for the recurrence.

Example: the recurrence

\[
\begin{align*}
    f(x) &= 0 & \text{if } x = 0 \\
    f(x) &= f(f(x - 1)) + 1 & \text{if } x > 0
\end{align*}
\]

is encoded as a first order logic formula

\[
\forall x \left( ( x = 0 \implies f(x) = 0 ) \land ( x > 0 \implies f(x) = f(f(x - 1)) + 1 ) \right)
\]

- References to the target \( f(x) \) are replaced by the candidate \( \hat{f}(x) = x \).

\[
\forall x \left( ( x = 0 \implies x = 0 ) \land ( x > 0 \implies x = f(x - 1) + 1 ) \right)
\]

- If the negation of such formula is unsatisfiable, then the candidate function is an exact solution.

- We use an SMT-solver to check satisfiability.

- It is unsatisfiable \( \rightarrow \hat{f}(x) = x \) is an exact solution for \( f(x) \).

- Sometimes, it is necessary to consider a precondition for the domain of the recurrence, which is also included in the encoding. E.g., \( \hat{f}(x) = x \) if \( x \geq 0 \).
Check: Second Stage of our Recurrence Solving Method

- Verify whether the guessed candidate function is actually a solution for the recurrence.

Example: the recurrence
\[
\begin{align*}
f(x) &= 0 & \text{if } x &= 0 \\
f(x) &= f(f(x - 1)) + 1 & \text{if } x &> 0
\end{align*}
\]
is encoded as a first order logic formula
\[
\forall x \left( (x = 0 \implies f(x) = 0) \land (x > 0 \implies f(x) = f(f(x - 1)) + 1) \right)
\]
References to the target \(f(x)\) are replaced by the candidate \(\hat{f}(x) = x\).
\[
\forall x \left( (x = 0 \implies x = 0) \land (x > 0 \implies x = x - 1 + 1) \right)
\]
If the negation of such formula is unsatisfiable, then the candidate function is an exact solution.

We use an SMT-solver to check satisfiability.

It is unsatisfiable \(\rightarrow \hat{f}(x) = x\) is an exact solution for \(f(x)\).

Sometimes, it is necessary to consider a precondition for the domain of the recurrence, which is also included in the encoding. E.g., \(\hat{f}(x) = x\) if \(x \geq 0\).
Check: Second Stage of our Recurrence Solving Method

- Verify whether the guessed candidate function is actually a solution for the recurrence.

- Example: the recurrence
  \[
  \begin{align*}
  f(x) &= 0 & \text{if } x = 0 \\
  f(x) &= f(f(x - 1)) + 1 & \text{if } x > 0
  \end{align*}
  \]
  is encoded as a first order logic formula
  \[
  \forall x \ (( x = 0 \implies f(x) = 0 ) \land ( x > 0 \implies f(x) = f(f(x - 1)) + 1 ))
  \]
- References to the target \( f(x) \) are replaced by the candidate \( \hat{f}(x) = x \).
  \[
  \forall x \ (( x = 0 \implies x = 0 ) \land ( x > 0 \implies x = x ))
  \]
- If the negation of such formula is unsatisfiable, then the candidate function is an exact solution.

- We use an SMT-solver to check satisfiability.

- It is unsatisfiable \( \implies \hat{f}(x) = x \) is an exact solution for \( f(x) \).

- Sometimes, it is necessary to consider a precondition for the domain of the recurrence, which is also included in the encoding. E.g., \( \hat{f}(x) = x \) if \( x \geq 0 \).
Check: Second Stage of our Recurrence Solving Method

- Verify whether the guessed candidate function is actually a solution for the recurrence.

- Example: the recurrence
  \[
  f(x) = \begin{cases} 
  0 & \text{if } x = 0 \\ 
  f(f(x - 1)) + 1 & \text{if } x > 0 
  \end{cases}
  \]
  is encoded as a first order logic formula
  \[
  \forall x ((x = 0 \implies f(x) = 0) \land (x > 0 \implies f(x) = f(f(x - 1)) + 1))
  \]
  References to the target \( f(x) \) are replaced by the candidate \( \hat{f}(x) = x \).
  \[
  \forall x ((x = 0 \implies x = 0) \land (x > 0 \implies x = x))
  \]
  If the negation of such formula is unsatisfiable, then the candidate function is an exact solution.

- We use an SMT-solver to check satisfiability.
- It is unsatisfiable \( \implies \hat{f}(x) = x \) is an exact solution for \( f(x) \).
- Sometimes, it is necessary to consider a precondition for the domain of the recurrence, which is also included in the encoding. E.g., \( \hat{f}(x) = x \) if \( x \geq 0 \).
Check: Second Stage of our Recurrence Solving Method

- Verify whether the guessed candidate function is actually a solution for the recurrence.

- Example: the recurrence
  \[ f(x) = \begin{cases} 
  0 & \text{if } x = 0 \\
  f(x) = f(f(x - 1)) + 1 & \text{if } x > 0 
  \end{cases} \]

  is encoded as a first order logic formula
  \[ \forall x \ ( ( x = 0 \implies f(x) = 0 ) \land ( x > 0 \implies f(x) = f(f(x - 1)) + 1 ) ) \]

- References to the target \( f(x) \) are replaced by the candidate \( \hat{f}(x) = x \).
  \[ \neg \forall x \ ( ( x = 0 \implies x = 0 ) \land ( x > 0 \implies x = x ) ) \]

- If the negation of such formula is unsatisfiable, then the candidate function is an exact solution.

- We use an SMT-solver to check satisfiability.

- It is unsatisfiable \( \implies \hat{f}(x) = x \) is an exact solution for \( f(x) \).

- Sometimes, it is necessary to consider a precondition for the domain of the recurrence, which is also included in the encoding. E.g., \( \hat{f}(x) = x \) if \( x \geq 0 \).
Check: Second Stage of our Recurrence Solving Method

- Verify whether the guessed candidate function is actually a solution for the recurrence.
- Example: the recurrence
  \[
  f(x) = \begin{cases} 
  0 & \text{if } x = 0 \\
  f(f(x - 1)) + 1 & \text{if } x > 0 
  \end{cases}
  \]
  is encoded as a first order logic formula
  \[
  \forall x \ ( ( x = 0 \implies f(x) = 0 ) \land ( x > 0 \implies f(x) = f(f(x - 1)) + 1 ) )
  \]
  References to the target \( f(x) \) are replaced by the candidate \( \hat{f}(x) = x \).
  \[
  \exists x \neg \left( ( x = 0 \implies x = 0 ) \land ( x > 0 \implies x = x ) \right)
  \]
  If the negation of such formula is unsatisfiable, then the candidate function is an exact solution.
- We use an SMT-solver to check satisfiability.
- It is unsatisfiable \( \implies \hat{f}(x) = x \) is an exact solution for \( f(x) \).
- Sometimes, it is necessary to consider a precondition for the domain of the recurrence, which is also included in the encoding. E.g., \( \hat{f}(x) = x \) if \( x \geq 0 \).
Implemented a prototype and evaluated it with recurrences that are generated by CiaoPP’s cost analysis

- our approach can find exact, verified, closed-form solutions, in a reasonable time for recurrences that cannot be solved by CiaoPP.
- Potentially, arbitrarily large gains in static cost analysis accuracy.

Our approach solves recurrences that current state-of-the-art CASs cannot (e.g., Wolfram Mathematica, Sympy).

Our prototype always returns a closed form and either:

- indicates if such closed form is an exact solution of the recurrence (i.e., if it has been formally verified), or
- otherwise, gives the accuracy of the estimation (*score*) obtained in the guess (ML) phase.
## Experimental Results: Times (seconds)

<table>
<thead>
<tr>
<th>Bench</th>
<th>Recurrence</th>
<th>CF</th>
<th>CFNew</th>
<th>T (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>merge-sz</td>
<td>( f(x, y) = \begin{cases} \max(f(x - 1, y), f(x, y - 1)) + 1 &amp; \text{if } x &gt; 0 \land y &gt; 0 \ x &amp; \text{if } x &gt; 0 \land y \leq 0 \ y &amp; \text{if } x \leq 0 \land y &gt; 0 \end{cases} )</td>
<td></td>
<td>( x + y )</td>
<td>0.92</td>
</tr>
<tr>
<td>merge</td>
<td>( f(x, y) = \begin{cases} \max(f(x - 1, y), f(x, y - 1)) + 1 &amp; \text{if } x &gt; 0 \land y &gt; 0 \ 0 &amp; \text{otherwise} \end{cases} )</td>
<td></td>
<td>( \max(0, x + y - 1) )</td>
<td>0.71</td>
</tr>
<tr>
<td>nested</td>
<td>( f(x) = \begin{cases} f(f(x - 1)) + 1 &amp; \text{if } x &gt; 0 \ 0 &amp; \text{otherwise} \end{cases} )</td>
<td></td>
<td>( x )</td>
<td>0.13</td>
</tr>
<tr>
<td>open-zip</td>
<td>( f(x, y) = \begin{cases} f(x - 1, y - 1) + 1 &amp; \text{if } x &gt; 0 \land y &gt; 0 \ f(x, y - 1) + 1 &amp; \text{if } x \leq 0 \land y &gt; 0 \ f(x - 1, y) + 1 &amp; \text{if } y \leq 0 \land x &gt; 0 \ 0 &amp; \text{otherwise} \end{cases} )</td>
<td></td>
<td>( \max(x, y) )</td>
<td>0.12</td>
</tr>
<tr>
<td>div</td>
<td>( f(x, y) = \begin{cases} f(x - y, y) + 1 &amp; \text{if } x \geq y \ 0 &amp; \text{otherwise} \end{cases} )</td>
<td></td>
<td>( \lceil \frac{x}{y} \rceil )</td>
<td>0.13</td>
</tr>
<tr>
<td>div-ceil</td>
<td>( f(x, y) = \begin{cases} f(x - y, y) + 1 &amp; \text{if } x \geq y \ 1 &amp; \text{if } x &lt; y \land x &gt; 0 \ 0 &amp; \text{otherwise} \end{cases} )</td>
<td></td>
<td>( \lceil \frac{x}{y} \rceil )</td>
<td>0.12</td>
</tr>
<tr>
<td>s-max</td>
<td>( f(x, y) = \begin{cases} \max(y, f(x - 1, y)) + 1 &amp; \text{if } x &gt; 0 \ y &amp; \text{otherwise} \end{cases} )</td>
<td>( x + y )</td>
<td>( x + y )</td>
<td>0.12</td>
</tr>
<tr>
<td>s-max-1</td>
<td>( f(x, y) = \begin{cases} \max(y, f(x - 1, y + 1)) + 1 &amp; \text{if } x &gt; 0 \ y &amp; \text{otherwise} \end{cases} )</td>
<td></td>
<td>( 2x + y )</td>
<td>0.14</td>
</tr>
<tr>
<td>sum-osc</td>
<td>( f(x, y) = \begin{cases} f(x - 1, y) + 1 &amp; \text{if } x &gt; 0 \land y &gt; 0 \ f(x + 1, y - 1) + y &amp; \text{if } x \leq 0 \land y &gt; 0 \ 1 &amp; \text{otherwise} \end{cases} )</td>
<td></td>
<td>( x + \frac{x^2}{2} + \frac{3y}{2} )</td>
<td>0.13</td>
</tr>
</tbody>
</table>
Conclusions

- Novel approach for solving or approximating arbitrary, constrained recurrence relations.
  - *guess* a candidate closed-form solution
    → sparse linear regression via Lasso regularization and cross-validation.
  - *check* that such candidate is actually a solution
    → SMT-solver and CAS combination.

- However, the guess stage doesn’t guarantee that an exact solution can be found (for the training set).

- Even if an exact solution is found, it is not always possible to verify it in the check stage.

- Nevertheless, approximated solutions can be useful in some applications (e.g., granularity control in parallel/distributed computing)
  → Our approach always produces an accuracy measure

- The experimental results with our prototype are quite promising.
Future Work

- Fully integrate our novel solver into the CiaoPP system, combining it with its current set of back-end solvers
  → more extensive experimentation
- Refine and improve our algorithms in several directions.
  - Automatically infer the set $T$ of base functions by using different heuristics.
  - 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.
Thank you for your attention!