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

Maximiliano Klemen[1], Miguel Ángel Carreira-Perpiñán[2] and
$\boxed{\text{Pedro Lopez-Garcia}^{1,3}}$

[1]IMDEA Software Institute, Spain
[2]University of California, Merced, USA
[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)

# Introduction and Motivation

- 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.

# Introduction and Motivation

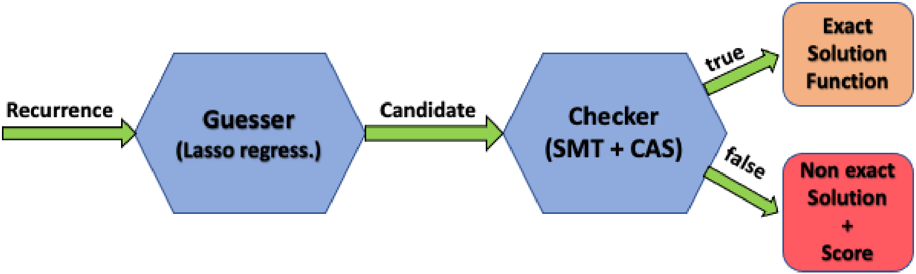
- These techniques strongly depend on solving (or safely approximating) recurrence relations $\rightarrow$ bottleneck.
- Using Computer Algebra Systems (CAS) or specialized solvers poses several difficulties and limitations for some recurrences:
  - Contain complex expressions or recursive structures.
  - Don't have the form required by such solvers

    $\rightarrow$ 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

Novel, general method for solving arbitrary, constrained recurrence relations:

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



# The Context: Static Cost Analysis (CiaoPP)

- 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.
```

# The Context: Static Cost Analysis (CiaoPP)

- 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.

# The Context: Static Cost Analysis (CiaoPP)

- 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.

# The Context: Static Cost Analysis (CiaoPP)

- 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: $\mathrm{S_p}(x)$.

# The Context: Static Cost Analysis (CiaoPP)

- 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: $\mathrm{S_p}(x)$.
- Using $x = size(X) = X$ (actual value of $X$), size relations are set up:

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

# The Context: Static Cost Analysis (CiaoPP)

- 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: $\mathrm{S_p}(x)$.
- Using $x = size(X) = X$ (actual value of $X$), size relations are set up:

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

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

# The Context: Static Cost Analysis (CiaoPP)

- 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: $\mathtt{S_p}(x)$.
- Using $x = size(X) = X$ (actual value of $X$), size relations are set up:

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

- 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.

# The Context: Static Cost Analysis (CiaoPP)

- 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: $\mathrm{S_p}(x)$.
- Using $x = size(X) = X$ (actual value of $X$), size relations are set up:
  $$\begin{array}{lll} \mathrm{S_p}(x) = & 0 & \text{if } x = 0 \\ \mathrm{S_p}(x) = & \mathrm{S_p}(\mathrm{S_p}(x-1)) + 1 & \text{if } x > 0 \end{array}$$
- 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 $\mathrm{S_p}(x) = x$ (exact solution).

# The Context: Static Cost Analysis (CiaoPP)

- 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.
```

# The Context: Static Cost Analysis (CiaoPP)

- 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 uses the size relations to infer the computational cost of a call to `p/2`, denoted $\mathrm{C_p}(x)$

# The Context: Static Cost Analysis (CiaoPP)

- 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 uses the size relations to infer the computational cost of a call to `p/2`, denoted $\mathtt{C_p}(x)$
  - → (in the example, number of resolution steps, and

# The Context: Static Cost Analysis (CiaoPP)

- 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 uses the size relations to infer the computational cost of a call to `p/2`, denoted $\mathrm{C_p}(x)$
  - $\rightarrow$ (in the example, number of resolution steps, and assuming the builtins $>$ /2 and *is*/2 have zero cost)

# The Context: Static Cost Analysis (CiaoPP)

- 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 uses the size relations to infer the computational cost of a call to `p/2`, denoted $\mathrm{C_p}(x)$
  - → (in the example, number of resolution steps, and assuming the builtins $>/2$ and *is*/2 have zero cost)
- It sets up the following recurrence:

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

# The Context: Static Cost Analysis (CiaoPP)

- 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 uses the size relations to infer the computational cost of a call to `p/2`, denoted $\mathrm{C_p}(x)$
  - → (in the example, number of resolution steps, and assuming the builtins $>/2$ and *is*/2 have zero cost)
- It sets up the following recurrence:

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

- Plugin the closed form $\mathrm{S_p}(x) = x$ inferred by our approach,

# The Context: Static Cost Analysis (CiaoPP)

- 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 uses the size relations to infer the computational cost of a call to `p/2`, denoted $\mathtt{C_p}(x)$
  - → (in the example, number of resolution steps, and assuming the builtins $>/2$ and *is*/2 have zero cost)
- It sets up the following recurrence:

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

- Plugin the closed form $\mathtt{S_p}(x) = x$ inferred by our approach,

# The Context: Static Cost Analysis (CiaoPP)

- 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 uses the size relations to infer the computational cost of a call to `p/2`, denoted $\mathtt{C_p}(x)$
  - → (in the example, number of resolution steps, and assuming the builtins $>/2$ and *is*/2 have zero cost)
- It sets up the following recurrence:

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

- Plugin the closed form $\mathtt{S_p}(x) = x$ inferred by our approach,

# The Context: Static Cost Analysis (CiaoPP)

- 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 uses the size relations to infer the computational cost of a call to `p/2`, denoted $\mathtt{C}_{\mathtt{p}}(x)$
  - → (in the example, number of resolution steps, and assuming the builtins $>/2$ and *is*/2 have zero cost)
- It sets up the following recurrence:

$$\begin{array}{lll} \mathtt{C}_{\mathtt{p}}(x) = & 1 & \text{if } x = 0 \\ \mathtt{C}_{\mathtt{p}}(x) = & 2\ \mathtt{C}_{\mathtt{p}}(x-1)+1 & \text{if } x > 0 \end{array}$$

- Plugin the closed form $\mathtt{S}_{\mathtt{p}}(x) = x$ inferred by our approach,

# The Context: Static Cost Analysis (CiaoPP)

- 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 uses the size relations to infer the computational cost of a call to p/2, denoted $\mathtt{C}_{\mathtt{p}}(x)$
  - → (in the example, number of resolution steps, and assuming the builtins $>/2$ and $is/2$ have zero cost)
- It sets up the following recurrence:

$$\begin{array}{lll} \mathtt{C}_{\mathtt{p}}(x) = & 1 & \text{if } x = 0 \\ \mathtt{C}_{\mathtt{p}}(x) = & 2\ \mathtt{C}_{\mathtt{p}}(x-1) + 1 & \text{if } x > 0 \end{array}$$

- Plugin the closed form $\mathtt{S}_{\mathtt{p}}(x) = x$ inferred by our approach, CiaoPP obtains $\mathtt{C}_{\mathtt{p}}(x) = 2^{x+1} - 1$.

# The Context: Static Cost Analysis (CiaoPP)

- 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 uses the size relations to infer the computational cost of a call to p/2, denoted $\mathrm{C_p}(x)$
  - → (in the example, number of resolution steps, and assuming the builtins $>$/2 and *is*/2 have zero cost)
- It sets up the following recurrence:

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

- Plugin the closed form $\mathrm{S_p}(x) = x$ inferred by our approach, CiaoPP obtains $\mathrm{C_p}(x) = 2^{x+1} - 1$.
- Without our approach CiaoPP would infer $\mathrm{S_p}(x) = \infty$ and $\mathrm{C_p}(x) = \infty$.

# The Context: Static Cost Analysis (CiaoPP)

- 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 uses the size relations to infer the computational cost of a call to `p/2`, denoted $\mathtt{C_p}(x)$
  → (in the example, number of resolution steps, and assuming the builtins $>/2$ and *is*/2 have zero cost)
- It sets up the following recurrence:

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

- Plugin the closed form $\mathtt{S_p}(x) = x$ inferred by our approach, CiaoPP obtains $\mathtt{C_p}(x) = 2^{x+1} - 1$.
- Without our approach CiaoPP would infer $\mathtt{S_p}(x) = \infty$ and $\mathtt{C_p}(x) = \infty$.
- Not being able to solve a "simple" recurrence can cause arbitrarily large losses of precission in size/cost analysis.

# Guess: First Stage of our Recurrence Solving Method

- Given the previous recurrence, with $\mathrm{S_p}(x) \equiv f(x)$:
  $$\begin{array}{lll} f(x) = & 0 & \text{if } x = 0 \\ f(x) = & f(f(x-1)) + 1 & \text{if } x > 0 \end{array}$$
- 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.

# Guess Stage: Example

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 $\bar{x}$
   $\rightarrow\ c_i = [\![t_i]\!]_{\bar{x}}$ 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

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

# Guess Stage: Example (contd.)

2. Perform sparse linear regression using $S$:
   - Result: (column) vector $\bar{\beta}$ 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": $rm_\epsilon(\bar{\beta}^T)$
5. The resulting closed-form is

$$\hat{f}(\bar{x}) = rm_\epsilon(\bar{\beta}^T) \cdot E(T, \bar{x}) + \beta_0$$

   - $\rightarrow$ $E(T, \bar{x})$: 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.

# Guess Stage: Example (contd.)

6. Perfom 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 $\epsilon = 0.001$):
  $\hat{f}(x) = 1.0\ x$ and $R^2 = 1$
  - Since $R^2 = 1$, then $\hat{f}(x) = x$ is a candidate closed-form solution
    $\rightarrow$ exact prediction of the recurrence for the test set.
  - If it was $R^2 < 1$, then $\hat{f}(x)$ would be an approximation.
    $\rightarrow$ 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

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

- is encoded as a first order logic formula

$$\forall x \; ((\, x = 0 \implies f(x) = 0 \,) \wedge (\, x > 0 \implies f(x) = f(f(x-1)) + 1 \,))$$

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

$$\forall x \; ((\, x = 0 \implies f(x) = 0 \,) \wedge (\, x > 0 \implies f(x) = f(f(x-1)) + 1 \,))$$

- 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
  $$\begin{array}{lll} f(x) = & 0 & \text{if } x = 0 \\ f(x) = & f(f(x-1)) + 1 & \text{if } x > 0 \end{array}$$
- is encoded as a first order logic formula
  $$\forall x \; ((\, x = 0 \implies f(x) = 0\,) \wedge (\, x > 0 \implies f(x) = f(f(x-1)) + 1\,))$$
- References to the target $f(x)$ are replaced by the candidate $\hat{f}(x) = x$.
  $$\forall x \; ((\, x = 0 \implies f(x) = 0\,) \wedge (\, x > 0 \implies f(x) = f(f(x-1)) + 1\,))$$
- 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
$$\begin{array}{lll} f(x) = & 0 & \text{if } x = 0 \\ f(x) = & f(f(x-1)) + 1 & \text{if } x > 0 \end{array}$$
- is encoded as a first order logic formula
$$\forall x \ ((\ x = 0 \implies f(x) = 0\ ) \wedge (\ x > 0 \implies f(x) = f(f(x-1)) + 1\ ))$$
- References to the target $f(x)$ are replaced by the candidate $\hat{f}(x) = x$.
$$\forall x \ ((\ x = 0 \implies x = 0\ ) \wedge (\ x > 0 \implies f(x) = f(f(x-1)) + 1\ ))$$
- 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
  $$\begin{array}{lll} f(x) = & 0 & \text{if } x = 0 \\ f(x) = & f(f(x-1)) + 1 & \text{if } x > 0 \end{array}$$
- is encoded as a first order logic formula
  $$\forall x \; ((\, x = 0 \implies f(x) = 0 \,) \wedge (\, x > 0 \implies f(x) = f(f(x-1)) + 1 \,))$$
- References to the target $f(x)$ are replaced by the candidate $\hat{f}(x) = x$.
  $$\forall x \; ((\, x = 0 \implies x = 0 \,) \wedge (\, x > 0 \implies f(x) = f(f(x-1)) + 1 \,))$$
- 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

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

- 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
  $$\begin{array}{lll} f(x) = & 0 & \text{if } x = 0 \\ f(x) = & f(f(x-1)) + 1 & \text{if } x > 0 \end{array}$$
- is encoded as a first order logic formula
  $$\forall x \; ((\, x = 0 \implies f(x) = 0\,) \wedge (\, x > 0 \implies f(x) = f(f(x-1)) + 1\,))$$
- References to the target $f(x)$ are replaced by the candidate $\hat{f}(x) = x$.
  $$\forall x \; ((\, x = 0 \implies x = 0\,) \wedge (\, x > 0 \implies x = f(f(x-1)) + 1\,))$$
- 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
  $$\begin{array}{ll} f(x) = \; 0 & \text{if } x = 0 \\ f(x) = \; f(f(x-1)) + 1 & \text{if } x > 0 \end{array}$$
- is encoded as a first order logic formula
  $$\forall x \; ((\, x = 0 \implies f(x) = 0 \,) \wedge (\, x > 0 \implies f(x) = f(f(x-1)) + 1 \,))$$
- References to the target $f(x)$ are replaced by the candidate $\hat{f}(x) = x$.
  $$\forall x \; ((\, x = 0 \implies x = 0 \,) \wedge (\, x > 0 \implies x = f(x-1) + 1 \,))$$
- 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
  $$\begin{array}{lll} f(x) = & 0 & \text{if } x = 0 \\ f(x) = & f(f(x-1)) + 1 & \text{if } x > 0 \end{array}$$
- is encoded as a first order logic formula
  $$\forall x \; ((\, x = 0 \implies f(x) = 0 \,) \wedge (\, x > 0 \implies f(x) = f(f(x-1)) + 1 \,))$$
- References to the target $f(x)$ are replaced by the candidate $\hat{f}(x) = x$.
  $$\forall x \; ((\, x = 0 \implies x = 0 \,) \wedge (\, x > 0 \implies x = f(x-1) + 1 \,))$$
- 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
  $$\begin{array}{lll} f(x) = & 0 & \text{if } x = 0 \\ f(x) = & f(f(x-1)) + 1 & \text{if } x > 0 \end{array}$$
- is encoded as a first order logic formula
  $$\forall x \; ((\, x = 0 \implies f(x) = 0\,) \wedge (\, x > 0 \implies f(x) = f(f(x-1)) + 1\,))$$
- References to the target $f(x)$ are replaced by the candidate $\hat{f}(x) = x$.
  $$\forall x \; ((\, x = 0 \implies x = 0\,) \wedge (\, x > 0 \implies x = x - 1 + 1\,))$$
- 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

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

- 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 $\rightarrow$ $\hat{f}(x) = x$ 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
  $$\begin{array}{lll} f(x) = & 0 & \text{if } x = 0 \\ f(x) = & f(f(x-1)) + 1 & \text{if } x > 0 \end{array}$$
- is encoded as a first order logic formula
  $$\forall x\ ((\ x = 0 \implies f(x) = 0\ ) \wedge (\ x > 0 \implies f(x) = f(f(x-1)) + 1\ ))$$
- References to the target $f(x)$ are replaced by the candidate $\hat{f}(x) = x$.
  $$\forall x\ ((\ x = 0 \implies x = 0\ ) \wedge (\ x > 0 \implies x = x\ ))$$
- 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 $\rightarrow$ $\hat{f}(x) = x$ 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

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

- is encoded as a first order logic formula

$$\forall x \; ((\, x = 0 \implies f(x) = 0 \,) \wedge (\, x > 0 \implies f(x) = f(f(x-1)) + 1 \,))$$

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

$$\neg \, \forall x \; ((\, x = 0 \implies x = 0 \,) \wedge (\, x > 0 \implies x = x \,))$$

- 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 $\rightarrow$ $\hat{f}(x) = x$ 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

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

- is encoded as a first order logic formula

$$\forall x \; ((\, x = 0 \implies f(x) = 0 \,) \wedge (\, x > 0 \implies f(x) = f(f(x-1)) + 1 \,))$$

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

$$\exists x \, \neg \; ((\, x = 0 \implies x = 0 \,) \wedge (\, x > 0 \implies x = x \,))$$

- 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 $\rightarrow$ $\hat{f}(x) = x$ 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 $x \geq 0$.

# Implementation and Evaluation

- 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)

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

# Conclusions

- Novel approach for solving or approximating arbitrary, constrained recurrence relations.
  - *guess* a candidate closed-form solution
    $\rightarrow$ sparse linear regression via Lasso regularization and cross-validation.
  - *check* that such candidate is actually a solution
    $\rightarrow$ SMT-solver and CAS combination.
- However, the guess stage doesn't guarante 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)
  $\rightarrow$ 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
  - $\rightarrow$ 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!