Weakest Precondition Inference for Non-Deterministic Linear Array Programs

Sumanth Prabhu S[1,2(✉)], Deepak D'Souza[2(✉)], Supratik Chakraborty[3(✉)], R Venkatesh[1(✉)], and Grigory Fedyukovich[4(✉)]

[1] Tata Consultancy Services Research, Pune, India
r.venky@tcs.com
[2] Indian Institute of Science, Bengaluru, India
sumanth.prabhu@tcs.com, deepakd@iisc.ac.in
[3] Indian Institute of Technology, Bombay, India
supratik@cse.iitb.ac.in
[4] Florida State University, Tallahassee, USA
grigory@cs.fsu.edu

Abstract. *Precondition inference* is an important problem with many applications. Existing precondition inference techniques for programs with arrays have limited ability to find and prove the weakest preconditions, especially when programs have non-determinism. In this paper, we propose an approach to overcome the limitation. As the problem is uncomputable in general, our approach targets a special class of programs called linear array programs that are commonly encountered in practical applications and have been studied before. We also focus on a class of quantified formulas for pre- and postconditions that suffice to specify program properties in many applications. Our approach uses two novel techniques called *Structural Array Abduction* (SAA) and *Specialized Maximality Checking* (SMC). SAA is an abduction-based technique used to infer quantified preconditions and necessary inductive invariants. SMC proves that an inferred precondition is the weakest by finding an under-approximated program and solving the complement verification problem on it using SAA. When inconclusive, it attempts to weaken the precondition. Our approach can infer (and also prove) the weakest preconditions for a range of benchmarks relatively quickly, and outperforms competing techniques.

1 Introduction

Precondition inference is concerned with finding a set of initial states from which all terminating executions of a given program reach states satisfying a given postcondition. The *weakest* precondition refers to the largest such set of initial states. The weakest precondition can be used as a contract on a library function's input, for run-time argument value checks, as a summary in compositional verification, and in many more applications [2, 11, 12, 24, 46, 47, 52, 53].

Finding the weakest precondition, especially in the presence of unbounded loops and data structures like arrays, is challenging and uncomputable in general. To show that a precondition is valid requires reasoning about all possible

B. Finkbeiner and L. Kovács (Eds.): TACAS 2024, LNCS 14571, pp. 175–195, 2024.
https://doi.org/10.1007/978-3-031-57249-4_9

executions of loops. Almost always this necessitates the inference of adequate *inductive invariants*. However, automatic invariant inference is an equally difficult problem, and in the case of array programs, the required invariants are often quantified formulas, adding to the difficulty of reasoning about them. Moreover, existing invariant inference techniques [22, 28, 30, 32, 41] rely on a precondition being provided by the user. This makes it difficult to use such techniques directly in our problem setting, where preconditions are not available to begin with.

Even if we are able to find a precondition for a given program and postcondition, proving that the precondition is the weakest presents significant technical challenges. Specifically, we need to prove that adding any new state to the set of initial states represented by the precondition results in an execution that terminates in a state violating the postcondition. To find such a proof, existing quantified precondition inference techniques assume the program to be deterministic, i.e., from every initial state, there is a unique program execution [49, 53]. However, it often becomes necessary to use non-deterministic features when modeling programs, thereby admitting multiple possible executions starting from the same initial state. Such non-deterministic features may be needed to model user input, non-deterministic functions, external functions, or when programs are abstracted. Hence, assuming that all programs are deterministic significantly restricts the applicability of existing techniques for finding weakest preconditions.

We propose a novel technique for inferring weakest preconditions for a class of terminating non-deterministic programs that manipulate arrays, with respect to postconditions expressed in a rich language of formulas. Specifically, we target the class of *linear array programs*, defined formally in Section 3. This includes programs used in many practical applications, and the literature describes several verification techniques for this class of programs [7, 8, 40]. However, existing techniques for weakest precondition inference either apply to deterministic linear array programs, or deal with non-determinism in simpler classes of programs. Our work fills this gap, making it possible to infer weakest preconditions for linear array programs with non-determinism.

The proposed technique works in the *infer-check-weaken* framework [1,27,49, 50,54]. It first infers a precondition along with adequate inductive invariants. A maximality check follows to see whether the precondition is weakest. If the check yields a negative answer, the precondition is weakened. This loop continues until the weakest precondition is found. In this framework, our core contributions are Structural Array Abduction (SAA) for inferring preconditions and associated invariants, and Specialized Maximality Checking (SMC) for proving that the inferred precondition is maximal (or weakest).

At a high level, SAA "guesses" candidate preconditions and inductive invariants as (quantified) formulas, and checks their correctness using an SMT solver. Since quantified formulas over arrays are challenging to reason about even with state-of-the-art SMT solvers, the guessing has to be done carefully. SAA uses abductive inference for this purpose. First, it constructs an abduction query to find what property of array elements at the start of a loop iteration will result in a desired property after the iteration. The array property thus inferred is

then combined with a *range formula* [22], which is a predicate representing the boundary between indices of the array that are processed and those that are yet to be processed. A set of rules guide the construction of appropriate abduction queries and range formulas.

Though SAA is effective in finding weak preconditions, it is not guaranteed to find the weakest precondition. SMC is used to check whether a precondition is indeed the weakest. This amounts to determining whether for every initial state that violates the precondition, there is a terminating execution that results in a state violating the postcondition. To accomplish this, SMC uses the insight that every execution of a non-deterministic program is also an execution of an *under-approximation* of the original program obtained by suitably restricting the non-determinism in control flows (i.e., `if` statements). Specifically, the existence of inductive invariants for *the complement verification problem*, i.e., under-approximated program with complemented pre-and postconditions, proves that the inferred precondition is indeed the weakest for the given (terminating) program and postcondition. SMC uses SAA to find an under-approximated program and its inductive invariants. When SAA fails, SMC weakens the precondition from a set of candidates obtained in a syntax-guided way, like in [22].

Our technique is implemented in a tool called MaxPrANQ. It takes constrained Horn clauses (CHCs) as input, which is a convenient way to model and reason about programs symbolically (details in Sec 3.2). On a challenging set of 66 precondition inference tasks, our tool inferred the weakest precondition for all 66 and automatically proved 59 of them to be the weakest. In comparison, the state-of-the-art tool PreQSyn [49] could only solve 2/66 benchmarks, and P-Gen [53] did not find a precondition for any of them. To further gauge the difficulty level of reasoning about our benchmarks, we tried using two state-of-the-art inductive invariant inference tools, FreqHorn [22] and Spacer [30], to simply prove the correctness of the preconditions inferred by MaxPrANQ. Neither FreqHorn nor Spacer could however complete the task for the entire set of 66 benchmarks in the given time. This shows that even proving the correctness of the weakest preconditions was difficult for our benchmarks, let alone inferring the preconditions automatically.

The primary contributions of our paper are:

1. SAA: a method for finding preconditions, inductive invariants, and stronger guard conditions for non-deterministic linear array programs.
2. SMC: a method for checking if a precondition is the weakest and, when inconclusive, weakening it.
3. MaxPrANQ: a tool for finding the weakest preconditions, with witnesses of validity (inductive invariants) and maximality.

The rest of the paper has following sections: Sect 2 has a running example, Sect 3 provides necessary background, Sect 4 gives an overview of our algorithm, SAA and SMC descriptions are in Sect 5 and Sect 6, resp., Sect 7 gives evaluation details, Sect 8 has related work, and limitations and future work are in Sect 9.

```
int N = nondet(); // N ≥ 0
int A[N], B[N], C[N];
// pre(N, A, B, C);
for (int i = 0; i < N; i++)
  if (nondet()) C[i] = i;
  else A[i] = C[i];
assert(∀j. 0≤j<N ⟹ A[j]=B[j]);
```

(a) Program

```
int N = nondet();
int A[N], B[N], C[N];
// ¬pre
assume(∃j. 0≤j<N ∧ (A[j]≠B[j] ∨ B[j]≠C[j]));
for (int i = 0; i < N; i++)
  if (A[i]≠B[i]) C[i] = i; //new guard
  else A[i] = C[i];
assert(∃j. 0≤j<N ∧ A[j]≠B[j]);//¬post
```

(b) Maximality Proof

Fig. 1: A non-deterministic array program and its maximality proof.

2 A Running Example

Fig. 1a shows a non-deterministic program with a postcondition that requires a universally quantified weakest precondition. The program has three arrays: A, B, and C, each of parametric size N. For each array index i, the program chooses non-deterministically whether to write i to the i-th element of C or copy the i-th element of C into the corresponding index of A. The postcondition, as stated in the `assert`, requires that the arrays A and B have the same content. Our goal is to infer the weakest precondition (denoted by `pre`) over A, B, C, and N under which the program satisfies the postcondition.

Existing weakest precondition inference techniques [49, 53] diverge for non-deterministic programs like the one in Fig. 1a. For instance, P-Gen [53] fails to find a precondition, and PreQSyn [49] fails to prove that the precondition it finds is the weakest in 200 seconds. This is because they either fail to generalise a set of initial states to a quantified precondition or, when they do, they cannot prove it to be the weakest for non-deterministic programs. In contrast, SAA finds the precondition: $\forall j.\, 0 \leq j < N \implies (A[j] = B[j] \wedge B[j] = C[j])$ (details in Sect 5.3), and SMC proves this to be the weakest precondition, all within a few seconds.

To prove maximality, SMC finds an under-approximated program, as shown in Fig. 1b. In this program, the non-determinism in the `if` statement is restricted by a new guard: $A[i] \neq B[i]$. Furthermore, the `assume` condition is the complement of the precondition inferred by SAA earlier, and the condition in the `assert` is also complemented. The existence of an adequate inductive invariant for this program (which in turn can be found by SAA) proves that all its executions from every initial state violating the inferred precondition result in states violating the given postcondition as the program is terminating. In other words, the inferred precondition is indeed the weakest for the program and postcondition in Fig. 1a.

3 Background

3.1 Linear Array Programs

Fig 2 shows a grammar for linear array programs over a set of integer and array variables, $\mathcal{V}$ and $\mathcal{A}$, respectively. In the figure, $v \neq i \in \mathcal{V}$, $a \in \mathcal{A}$, $i \in \mathcal{V}$ is a fixed

$$
\begin{aligned}
\mathit{program} &\rightarrow \mathit{stmts} \\
\mathit{stmts} &\rightarrow \mathit{assign} \mid \mathit{forloop} \mid \mathit{stmts};\ \mathit{stmts} \\
\mathit{forloop} &\rightarrow \texttt{for}(i = 0;\ i < u;\ i = i + 1)\ \{\mathit{assign}\}
\end{aligned}
$$

$$
\begin{aligned}
\mathit{assign} \rightarrow\ & v = t(\mathcal{V}, \mathcal{A}) \mid a[i] = t(\mathcal{V}, \mathcal{A}) \\
& \mid \texttt{if}(:: g_1(\mathcal{V}, \mathcal{A}) \rightarrow \{\mathit{assign}\} \\
& \quad \cdots \\
& \quad :: g_n(\mathcal{V}, \mathcal{A}) \rightarrow \{\mathit{assign}\}) \\
& \mid \mathit{assign};\ \mathit{assign}
\end{aligned}
$$

Fig. 2: Linear Array Programs

loop counter, $u \in \mathcal{V} \cup \mathbb{Z}$, t is a linear arithmetic expression, and each g_i (or guard) is a boolean combination of linear expressions over $\mathcal{V}$ and $\mathcal{A}$, with $\bigvee_{i=1}^{n} g_i = \top$. The `if` statement is a set of guarded assignments. When such an `if` statement is executed, exactly one guard that evaluates to true in the current program state is non-deterministically chosen and the corresponding assignment statement is executed[5]. A program is *non-deterministic* if there are program states in which more than one guard of an `if` could evaluate to true.

Let P be a linear array program over $\mathcal{V}$ and $\mathcal{A}$. A pre/postcondition for P is a formula of the form $\forall x.\, R(x, \mathcal{V}) \implies Q(x, \mathcal{V}, \mathcal{A})$ or $\exists x.\, R(x, \mathcal{V}) \wedge Q(x, \mathcal{V}, \mathcal{A})$, where $x \notin \mathcal{V}$ is an integer variable, R is a linear predicate over x and $\mathcal{V}$ that represents a range of indices of array(s), and Q is a linear predicate over $\mathcal{V}$ and elements of array(s) in $\mathcal{A}$, the latter being accessed only through linear index expressions in x. As an example, $\forall x.\, (0 \leq x \leq N) \implies (C[x] \leq B[x])$ qualifies for a pre/postcondition, where $N \in \mathcal{V}$ and $C, B \in \mathcal{A}$. Following standard Floyd-Hoare logic, we say a pair of conditions (ψ, ρ) is a valid pre- and postcondition pair for P, if every execution of P that begins in a state satisfying ψ ends in a state satisfying ρ.

The *weakest precondition inference* problem we consider is: given a linear array program P and a postcondition ρ, find the weakest precondition ψ such that (ψ, ρ) forms a valid pre- and postcondition pair for P.

Weakest precondition inference for linear array programs is undecidable in general [49]. Therefore, we cannot hope for an algorithm that infers weakest preconditions in all cases. Nevertheless, many practical and useful programs can be modeled as linear array programs (see for example [7, 8, 40]). This motivates us to design techniques for finding weakest preconditions that work well for a large subclass of linear array programs.

3.2 Modeling Linear Array Programs as CHCs

In recent years, it is becoming popular to represent a program and its pre- and postcondition as a system of first-order logic (FOL) formulas with uninterpreted relations, called *constrained Horn clauses* (CHCs) [10, 19, 29, 33, 35–37, 43, 45]. In CHCs, the uninterpreted relations represent invariants and the goal is to find interpretations for them. We will consider the task of precondition inference as a CHC-solving task, with the missing precondition represented by a relation.

[5] The usual `if-then-else` statement is easily represented as two guarded assignments.

$$\boldsymbol{pre}(N, A, B, C) \wedge i = 0 \implies \boldsymbol{inv}_1(i,N,A,B,C) \quad (C1)$$
$$\boldsymbol{inv}_1(i,N,A,B,C) \wedge i < N \wedge C' = store(C, i, i) \wedge i' = i+1 \implies \boldsymbol{inv}_1(i',N,A,B,C') \quad (C2)$$
$$\boldsymbol{inv}_1(i,N,A,B,C) \wedge i < N \wedge A' = store(A, i, C[i]) \wedge i' = i+1 \implies \boldsymbol{inv}_1(i',N,A',B,C) \quad (C3)$$
$$\boldsymbol{inv}_1(i,N,A,B,C) \wedge \neg(i < N) \wedge \neg(\forall j.\, 0 \leq j < N \implies A[j] = B[j]) \implies \bot \quad (C4)$$

Fig. 3: CHC system for the program from Fig. 1a.

Definition 1. *A CHC is a formula in a FOL $\mathcal{L}$ (linear integer arithmetic with arrays in this paper) over a set of relations $\mathcal{R}$ with one of the following forms:*

$$\varphi(\vec{x}_0) \implies \boldsymbol{r}_0(\vec{x}_0) \quad (1)$$
$$\bigwedge_{0 \leq i \leq k} \boldsymbol{r}_i(\vec{x}_i) \wedge \varphi(\vec{x}_0, \ldots, \vec{x}_{k+1}) \implies \boldsymbol{r}_{k+1}(\vec{x}_{k+1}) \quad (2)$$
$$\bigwedge_{0 \leq i \leq k} \boldsymbol{r}_i(\vec{x}_i) \wedge \varphi(\vec{x}_0, \ldots, \vec{x}_k) \implies \bot \quad (3)$$

where, for every i, $\boldsymbol{r}_i \in \mathcal{R}$[6], and $\vec{x}_i$ represents the vector of variables $(x_1, \ldots, x_{a_{\boldsymbol{r}_i}})$, where $a_{\boldsymbol{r}_i}$ is the arity of $\boldsymbol{r}_i$. φ, called a *constraint*, is an $\mathcal{L}$-formula in conjunctive normal form without uninterpreted relations. CHCs of type (1) are called *facts*[7], of type (2) *inductive*, and of type (3) *queries*. Note that each CHC has a leading quantification over $\vec{x}$ (e.g. $\forall \vec{x}_0 \ldots \vec{x}_{k+1}$ for type (2)) that is implicit in the paper.

For a CHC C, we use the following notations: $body(C)$ (resp. $head(C)$) denotes the left (resp. right) side of the implication in C, $rels()$ denotes the relations from $\mathcal{R}$ that appear in $body(C)$, or $head(C)$, and $args()$ denotes the variables in $body(C)$, or $head(C)$. We assume the constraint φ of a CHC C can be partitioned into two formulas: $assign(C)$ and $guard(C)$, denoting the assignment statement and control-flow guard conditions (if any). A *system* of CHCs S is a finite set of CHCs. For any system S, if there is a CHC C with $|rels((body(C)))| \geq 1$, then S is *non-linear*, otherwise *linear*.

We assume the input CHC system is induced by a linear array program with $n \geq 0$ sequential loops. In particular, it is a linear CHC system over $\mathcal{R} = \{\boldsymbol{pre}, \boldsymbol{inv}_1, \ldots, \boldsymbol{inv}_n\}$, where $\boldsymbol{pre}$ denotes the precondition, and each $\boldsymbol{inv}_i$ denotes an inductive invariant for the i-th sequential loop.

Example 1. A linear system of CHCs induced by the program from Fig 1a is shown in Fig 3. In the system, the precondition is represented by the relation $\boldsymbol{pre}$ and the inductive invariant by $\boldsymbol{inv}_1$. $C1$ is the initialization CHC with $\boldsymbol{pre}$. The two CHCs $C2$ and $C3$ correspond to non-deterministic writes in the loop, while $C4$ is the query CHC, which has the `assert` condition. It is worth noting that interpretations for $\boldsymbol{pre}$ and $\boldsymbol{inv}_1$ that make each CHC valid gives a

[6] $\boldsymbol{r}_i$'s in each form are not necessarily distinct.
[7] The input CHC system will not have facts but they manifest in Algorithm 4.

precondition and an adequate inductive invariant. For example,

$$\begin{aligned}\boldsymbol{pre} &\mapsto \lambda N, A, B, C.\, \forall j.\, 0 \leq j < N \implies (A[j] = B[j] \wedge B[j] = C[j]) \\ \boldsymbol{inv}_1 &\mapsto \lambda N, A, B, C, i.\, \forall j.\, 0 \leq j < i \implies A[j] = B[j] \wedge \\ &\qquad\qquad \forall j.\, i \leq j < N \implies (A[j] = B[j] \wedge B[j] = C[j])\end{aligned}$$

A *map of interpretations* $\mathcal{M}$ for $\mathcal{R}$ assigns to each relation symbol $\boldsymbol{r} \in \mathcal{R}$ an interpretation of the form $\lambda x_1 \cdots \lambda x_{a_{\boldsymbol{r}}}.\, \psi(x_1, \ldots, x_{a_{\boldsymbol{r}}})$, where ψ is a $\mathcal{L}$-formula. We use the notation $\mathcal{M}[\boldsymbol{r}]$ to denote the interpretation for $\boldsymbol{r}$ by $\mathcal{M}$. For a formula α and a map $\mathcal{M}$ for $\mathcal{R}$, we write $\alpha[\mathcal{M}/\mathcal{R}]$ to denote the formula obtained by replacing each atomic formula of the form $\boldsymbol{r}(t_1, \ldots, t_{a_{\boldsymbol{r}}})$ in α by $\mathcal{M}[\boldsymbol{r}](t_1, \ldots, t_{a_{\boldsymbol{r}}})$.

Solution to CHCs A *solution* to a CHC C is a map $\mathcal{M}$ for $\mathcal{R}$ such that the formula $(\mathit{body}(C) \implies \mathit{head}(C))[\mathcal{M}/\mathcal{R}]$ is valid; in this case, we say C is satisfiable. $\mathcal{M}$ is a solution to a system S if it satisfies all the CHCs in S; in this case, we say S is *satisfiable*.

Let S be a system of CHCs induced by a program P and a postcondition ρ. If $\mathcal{M}$ is a solution to S, then $(\mathcal{M}[\boldsymbol{pre}], \rho)$ forms a valid pre/postcondition for P.

3.3 Abductive Inference

The core method used in SAA for inference is *abduction*. Given a formula $(\boldsymbol{r}(\vec{x}) \wedge \alpha(\vec{y})) \implies \beta(\vec{y})$, where $\boldsymbol{r}$ represents a relation, α (hypothesis) and β (conclusion) are formulas without relations, and the variables in $\vec{x}$ are also present in $\vec{y}$, the problem of abduction is to find an interpretation $\lambda x_1 \cdots \lambda x_{a_{\boldsymbol{r}}}.\, \psi$ to $\boldsymbol{r}$ such that:

$$\psi(\vec{x}) \wedge \alpha(\vec{y}) \not\Longrightarrow \bot \quad \text{and} \quad \psi(\vec{x}) \wedge \alpha(\vec{y}) \implies \beta(\vec{y})$$

Example 2. Consider the abduction problem $(\boldsymbol{r}(x) \wedge y = 42) \implies (x - y > 0)$. The maximal solution for the problem is $\boldsymbol{r} \mapsto \lambda x.\, x > 42$.

A given abduction problem can have multiple solutions. SAA seeks the maximal solution. There are techniques, like quantifier elimination, to find maximal solutions [16], but they are limited to non-array theories. To overcome this, range abduction [49] proposes a suitable array-to-integer abstraction, which SAA also uses.

Non-linear CHCs have more than one relation in *body*, requiring an extension of the abduction problem called *multi-abduction*. In multi-abduction, interpretations to multiple relations need to be inferred. SAA encounters non-linear CHCs while searching for maximality proofs, which involve the guard and inductive invariant relations. To solve the multi-abduction problem, SAA uses the technique from [1] after performing the array-to-integer abstraction from [49].

Example 3. The following is a multi-abduction problem: $(\boldsymbol{r}_1(A, i) \wedge \boldsymbol{r}_2(B, i) \wedge C[i] = 42) \implies (A[i] + B[i] > C[i])$. A maximal solution is $\boldsymbol{r}_1 \mapsto \lambda A, i.\, A[i] > 42$ and $\boldsymbol{r}_2 \mapsto \lambda B, i.\, B[i] \geq 0$.

Algorithm 1: WeakestPre(S)

Input: S – a system of non-deterministic CHCs over $\mathcal{R} = \{\boldsymbol{pre}, \boldsymbol{inv}_1, \ldots, \boldsymbol{inv}_n\}$
Output: $\langle \{weakest, unknown\}, \mathcal{M}[\boldsymbol{pre}] \rangle$

1 $\langle res, \mathcal{M} \rangle \leftarrow \text{SAA}(S, \varnothing)$;
2 **while** *res* **do**
3 $\quad \langle G, \Gamma \rangle \leftarrow \text{GetSplCHCs}(S, \mathcal{M})$;
4 $\quad \langle max, \_ \rangle \leftarrow \text{SAA}(G, \Gamma)$;
5 $\quad$ **if** *max* **then return** $\langle weakest, \mathcal{M}[\boldsymbol{pre}] \rangle$;
6 $\quad \langle res, \mathcal{M} \rangle \leftarrow \text{Weaken}(S, \mathcal{M})$;
7 **return** $\langle unknown, \mathcal{M}[\boldsymbol{pre}] \rangle$;

4 Inferring Weakest Preconditions

An overview of our weakest precondition inference algorithm is in Algorithm 1. The algorithm takes as input a CHC system S over $\{\boldsymbol{pre}, \boldsymbol{inv}_1, \ldots \boldsymbol{inv}_n\}$. It first computes a solution $\mathcal{M}$ to S using SAA (line 1). Though this solution gives a precondition (as the solution will have interpretations to $\boldsymbol{pre}, \boldsymbol{inv}_1, \ldots \boldsymbol{inv}_n$), it is not guaranteed to be the weakest. Hence, in a loop, the algorithm performs maximality checking and weakening (lines 2 to 6). When the maximality check succeeds, the solution is guaranteed to be the weakest (Theorem 1); hence the algorithm returns the current precondition $\mathcal{M}[\boldsymbol{pre}]$ (line 5). Otherwise, the algorithm assumes the maximality check is inconclusive and tries to find a weakening (line 6). The algorithm progresses and continues the same loop if a weakening is found. When the weakening is inconclusive, the loop terminates, and the current precondition is returned without a maximality guarantee (line 7).

SAA takes a CHC system, which is either a precondition inference task (S), or the encoding of maximality check (G) with additional non-CHC constraints (Γ). It finds a solution $\mathcal{M}$ to the CHC system that also satisfies the additional constraints.

Algorithm 1 proves the maximality of a precondition by encoding a CHC system G and non-CHC constraints Γ, together called *specialized CHCs* (line 3). G has the same set of CHCs as the input CHC system S except the following: 1) the relation $\boldsymbol{pre}$ is replaced by the formula $\neg\mathcal{M}[\boldsymbol{pre}]$, 2) the postcondition is the negation of the postcondition in S, and 3) new *guard relations*: $\boldsymbol{g}_{Ci}, \ldots, \boldsymbol{g}_{Cj}$ are added to *body* of CHCs: $Ci, \ldots Cj$ corresponding to non-deterministic `if` conditions. Thus, G is a CHC system over the invariant relations of S and new guard relations. A solution to G gives stronger `if` conditions and inductive invariants for the complement pre- and postcondition. The non-CHC constraints in Γ make sure the disjunction of $\boldsymbol{g}_{Ci}, \ldots, \boldsymbol{g}_{Cj}$ is $\top$; thus ensuring the interpretations for them are not too strong.

When SAA fails to find a solution to G, Algorithm 1 calls Weaken. At a high level, Weaken enumerates candidate preconditions obtained in a syntax-guided way like [22] and then tries to find inductive invariants using SAA again.

$(\exists j.\, 0 \leq j < N \wedge (A[j] \neq B[j] \ \vee \ B[j] \neq C[j])) \ \wedge \ i = 0 \implies \boldsymbol{inv}_1(i, N, A, B, C)$

$\boldsymbol{inv}_1(i,N,A,B,C) \wedge \boldsymbol{g}_{C2}(i,N,A,B,C) \wedge i < N \wedge C' = store(C,i,i) \wedge i' = i+1 \implies \boldsymbol{inv}_1(i',N,A,B,C')$

$\boldsymbol{inv}_1(i,N,A,B,C) \wedge \boldsymbol{g}_{C3}(i,N,A,B,C) \wedge i < N \wedge A' = store(A,i,C[i]) \wedge i' = i+1 \implies \boldsymbol{inv}_1(i',N,A',B,C)$

$\boldsymbol{inv}_1(i,N,A,B,C) \wedge \neg(i < N) \wedge \neg(\exists j.\, 0 \leq j < N \wedge A[j] \neq B[j]) \implies \bot$

$\top \implies \boldsymbol{g}_{C2}(i,N,A,B,C) \vee \boldsymbol{g}_{C3}(i,N,A,B,C)$

Fig. 4: Specialized CHCs for the CHCs from Example 1.

Example 4. Fig. 4 shows the specialized CHCs for the CHC system from Fig. 3 and the precondition from Example 1 (viz. $\forall j.\, 0 \leq j < N \implies (A[j] = B[j] \wedge B[j] = C[j])$). This system has following changes: 1) In the first CHC, the relation $\boldsymbol{pre}$ is replaced by the complement of the precondition, 2) The next two CHCs have $\boldsymbol{g}_{C2}, \boldsymbol{g}_{C3}$ in *body*, 3) In the fourth CHC, the postcondition is complemented, and 4) The last constraint is a non-CHC constraint that makes sure $\boldsymbol{g}_{C2} \vee \boldsymbol{g}_{C3}$ is $\top$.

Theorem 1 (Soundness of Algorithm 1[8]). *For a system S, if Algorithm 1 terminates with "weakest" then $\mathcal{M}[\boldsymbol{pre}]$ is the weakest precondition for S.*

5 Structural Array Abduction

Structural Array Abduction (SAA) solves CHCs. In Algorithm 1, SAA solves program-induced CHCs to identify preconditions, specialized CHCs for maximality proofs, and CHCs with candidate weakened preconditions to find invariants.

5.1 Algorithm Description

SAA aims to find interpretations to $\boldsymbol{pre}$ and $\boldsymbol{inv}$ of the following form:

$$\bigwedge \big(\forall x.\, R(x, \mathcal{V}) \implies Q(x, \mathcal{V}, \mathcal{A})\big) \quad \text{or} \quad \bigvee \big(\exists x.\, R(x, \mathcal{V}) \wedge Q(x, \mathcal{V}, \mathcal{A})\big) \tag{4}$$

Similar to the postcondition, here, R is a linear predicate over x and $\mathcal{V}$ that represents a range of indices of array(s), and Q is a linear predicate over $\mathcal{V}$ and elements of array(s) in $\mathcal{A}$, the latter being accessed only through linear index expressions in x. Such a form is sufficient to represent inductive invariants for a large class of array programs, as observed in existing works [22, 28, 30, 31, 38].

A relatively complete guessing algorithm involves enumerating all candidate solutions in the form of 4 and then checking them using an SMT solver. However, given the large number of candidate solutions and the inherent challenge that quantified formulas with arrays pose for SMT solvers, SAA brings a novel improvement. It narrows down the search by guessing likely candidate solutions using a logical method, as presented in Algorithm 2.

[8] Proofs are in [48].

Algorithm 2: SAA (S, Γ)

Input: S – set of CHCs over $\mathcal{R} \cup \mathcal{R}_{\boldsymbol{g}}$, Γ – non-CHC constraints over $\mathcal{R}_{\boldsymbol{g}}$
Output: $(\{\top, \bot\}$ – result, $\mathcal{M}$ – solution to S that also satisfies $\Gamma)$

```
1 while ∃C ∈ S. CheckSAT(¬(body(C) ⟹ head(C))[M/R]) do
2     if C is not fact then
3         M ← ArrayAbduce(S, C, M);
4     else
5         M ← WeakenFact(S, M);
6     if M is unchanged then M ← NextCandidate() ;
7 res ← CheckSAT(Γ);
8 return(res, M);
```

Algorithm 2 begins with an initial candidate solution, e.g., $\forall \boldsymbol{r} \in \mathcal{R}.\, \mathcal{M}[\boldsymbol{r}] = \top$ in our implementation, and checks whether it is a solution to all CHCs. If not, the algorithm attempts to make the candidate a solution to the failed CHC mainly through abduction-based strengthening (line 3), or heuristics-based weakening if the CHC is a *fact* (line 5). If neither strengthening nor weakening results in a change to the candidate, the algorithm proceeds to the next candidate in the fixed form. When a candidate is found to be a solution, it is checked for additional constraints in Γ.

The abduction-based strengthening method is presented in Algorithm 3. It seeks new interpretations for the relations in *body* of a CHC, which can be $\boldsymbol{pre}$, $\boldsymbol{inv}$, or $\boldsymbol{g}_C$, that imply the interpretation for the relation in *head* of the CHC. This constitutes the abduction problem, as defined in Sec 3.3. However, existing abduction solvers cannot be used directly as they do not support quantified formulas with arrays. Hence, in Algorithm 3, Q and R from the fixed form (4) are determined separately and then combined into a quantified formula.

To find Q, the algorithm constructs an abduction query based on the rules provided in Table 1. In the abduction query, the hypothesis (α) is the assignment formula present in the constraint of the CHC (line 1), and the conclusion (β) is derived from the table based on the type of the CHC (line 2). Since the query contains array terms, which are not supported by existing abduction solvers, they are replaced by integer terms in a manner similar to the approach presented in [49] (e.g., $A[i]$ is replaced by a new integer variable a_i). Subsequently, the query is solved using an integer abduction solver to obtain a maximal solution (line 4). When the CHC has a guard relation $\boldsymbol{g}_C$ in its *body*, an additional abduction query is constructed to find interpretations for the other guard relations $\boldsymbol{g}_{C'}$ (line 6). Finally, integer terms in the solutions of the abduction queries are mapped back to corresponding array terms (line 7).

SAA uses the concept of range formulas as described in [22] to determine R. In the context of linear array programs, these range formulas can take the form of $0 \leq j < u$, $0 \leq j < i$, and $i \leq j < u$[9], where j is a free variable and u

[9] In [22], these are referred as *Range*, *progressRange*, and *regressRange*, respectively.

Algorithm 3: ArrayAbduce$(S, C, \mathcal{M})$

Input: S – set of CHCs over $\mathcal{R} \cup \mathcal{R}_{\boldsymbol{g}}$, C – CHC in S where $rels(body(C))$ is $\{\boldsymbol{r}\}$, or $\{\boldsymbol{r}, \boldsymbol{g}_C\}$, where $\boldsymbol{r}$ is either $\boldsymbol{pre}$ or some $\boldsymbol{inv}$ from $\mathcal{R}$, and $\boldsymbol{g}_C \dots \boldsymbol{g}_{C'}$ are from $\mathcal{R}_{\boldsymbol{g}}$ for the same control-flow condition, $\mathcal{M}$ – mapping from $\mathcal{R} \cup \mathcal{R}_{\boldsymbol{g}}$ to predicates

Output: $\mathcal{M}'$ – updated $\mathcal{M}$ with new interpretations to $\boldsymbol{r}$ and $\boldsymbol{g}_C \dots \boldsymbol{g}_{C'}$

1 $\alpha \leftarrow assign(C) \wedge \mathcal{M}[\boldsymbol{g}_C]$;
2 $\beta \leftarrow$ Get$\beta(C, \mathcal{M})$ `// cf. Tab 1;`
3 Transform α and β to integer formulas;
4 $\langle Q, Q_{\boldsymbol{g}_C} \rangle \leftarrow$ AbdSolver$(\boldsymbol{r}(\vec{x}_{\boldsymbol{r}}) \wedge \boldsymbol{g}_C(\vec{x}_{\boldsymbol{g}_C}) \wedge \alpha \implies \beta)$;
5 **if** $Q_{\boldsymbol{g}_C} \neq \bot$ **then**
6 $\quad \langle Q_{\boldsymbol{g}_{C' \neq C}} \rangle \leftarrow$ AbdSolver$(\bigvee_{C' \neq C} \boldsymbol{g}_{C'}(\vec{x}_{\boldsymbol{g}_{C'}}) \implies \neg Q_{\boldsymbol{g}_C})$;
7 Transform Q, $Q_{\boldsymbol{g}_C} \dots Q_{\boldsymbol{g}_{C'}}$ to array formulas;
8 $\langle R, j \rangle \leftarrow$ Get$R(S, C)$ `// cf. Tab 1;`
9 **if** universally quantified **then** $\mathcal{M}[r] \leftarrow \mathcal{M}[r] \wedge \forall j.\, R \implies Q$;
10 **else** $\mathcal{M}[r] \leftarrow \mathcal{M}[r] \vee \exists j.\, R \wedge Q$;
11 $\mathcal{M}[\boldsymbol{g}_C] \leftarrow Q_{\boldsymbol{g}_C} \dots \mathcal{M}[\boldsymbol{g}_{C'}] \leftarrow Q_{\boldsymbol{g}_{C'}}$;
12 **return** $\mathcal{M}$;

is the upper bound of the loop (cf. Fig. 2). From these formulas, a suitable R is selected based on the type of the CHC in Table 1.

The resulting R and Q are appropriately combined into a quantified formula and conjoined (or disjoined in the case of existential quantification) with the existing interpretation (lines 9 and 10). The guard relations are updated by the non-quantified formulas $Q_{\boldsymbol{g}_C}, \dots, Q_{\boldsymbol{g}_{C'}}$.

Table 1 provides a set of rules for determining R and β for all types of CHCs that Algorithm 3 may encounter. These CHCs include: 1) *precond*: the initialization CHC with $\boldsymbol{pre}$ in *body*, 2) *query*: the CHC with postcondition, 3) *intra*: CHCs representing potentially non-deterministic updates within a loop, and 4) *inter*: CHCs occurring between two loops. For example, when a CHC C falls into the *precond* category, β is the Q present in $\mathcal{M}[rels(head(C))]$ corresponding to the range $i \leq j < u$, and formula R is $0 \leq j < u$. We give an intuition to these rules while illustrating our technique in the following section.

When a *fact* CHC is unsatisfiable, SAA uses heuristic-based weakening for the *head* relation (Algorithm 2, line 5). This method generates a candidate set of Q formulas using the syntax of *body* of the CHC and combines it with $i \leq j < u$ to get a quantified formula.

Theorem 2. *If the input CHC system S has a solution in the form of 4, then Algorithm 2 will find it provided all the SMT checks return a result.*

5.2 Distinguishing SAA With Closely-Related Techniques

SAA uses the concept of range formulas from [22] and array to integer abduction technique from range abduction [49]. Nevertheless, there are notable differences:

$$\frac{\boldsymbol{pre} \in rels(body(C))}{\beta \leftarrow Q(i \le j < u, \mathcal{M}[rels(head(C))]) \quad R \leftarrow 0 \le j < u}\ precond \qquad \frac{rels(head(C)) = \varnothing}{\beta \leftarrow Q(0 \le j < u, \rho) \quad R \leftarrow 0 \le j < i}\ query$$

$$\frac{rels(head(C)) \subseteq rels(body(C))}{\beta \leftarrow Q(0 \le j < i, \mathcal{M}[rels(head(C))]) \quad R \leftarrow i \le j < u}\ intra$$

$$\frac{rels(head(C)) \cap rels(body(C)) = \varnothing \text{ and } rels(head(C)) \neq \varnothing}{\beta \leftarrow Q(i \le j < u, \mathcal{M}[rels(head(C))]) \quad R \leftarrow 0 \le j < i}\ inter$$

Table 1: Rules for all CHC types to construct formulas R and Q in Algorithm 3.

1) While [22] relies on preconditions to infer invariants, SAA is capable of inferring invariants even in the absence of preconditions. 2) Both [22] and range abduction can't handle nonlinear CHCs resulting from guarded relations, which SAA support by using multi-abduction. 3) In our experiments, we observed that range abduction tends to generate stronger preconditions compared to SAA. 4) Range abduction performs two abduction queries for each CHC, whereas SAA requires only one. 5) Range abduction uses the Houdini algorithm [23] for weakening, which is not necessary for SAA.

5.3 Illustration

Consider the CHCs from Fig. 3. For these CHCs, the range formulas are: $0 \le j < N$, $0 \le j < i$, and $i \le j < N$, as the upper bound u of the loop is N.

The algorithm begins with $\mathcal{M}[\boldsymbol{pre}] = \mathcal{M}[\boldsymbol{inv}_1] = \top$. But, the *query* CHC (C4) is unsatisfiable as $\mathcal{M}[\boldsymbol{inv}_1]$ is too weak. So, SAA tries to find a strengthening for $\boldsymbol{inv}_1$ using abduction. Recall that the postcondition (ρ) is $\forall j.\, 0 \le j < N \implies A[j] = B[j]$. While ρ itself can make this CHC satisfiable, it might be too strong for other CHCs with $\boldsymbol{inv}_1$. Therefore, the rule for *query* in the table suggests to consider R as $0 \le j < i$, and β as $A[j] = B[j]$ from ρ, corresponding to the range $0 \le j < N$. The abduction query $(\boldsymbol{inv}_1(A, B, C, i, N) \wedge \top) \implies A[j] = B[j]$ yields Q as $A[j] = B[j]$. Combining R and Q results in:

$$\mathcal{M}[\boldsymbol{inv}_1] \overset{\text{cand}}{\mapsto} \forall j.\, (0 \le j < i) \implies A[j] = B[j]$$

Next, an *intra* CHC $C2$ is unsatisfiable. This is due to the absence of restrictions on the values of A and B in the range $i \le j < N$ within $\boldsymbol{inv}_1$. One way to fix this is to find a Q in the range $i \le j < N$ that implies $A[j] = B[j]$. This approach aligns with the rule for *intra* CHC, where β is $A[j] = B[j]$ corresponding to the range $0 \le j < i$ of $\mathcal{M}[\boldsymbol{inv}_1]$, and R is $i \le j < N$. Further, $assign(C2)$ is $C'[j] = j$ (primed variables denote updated variables), resulting in the following abduction query: $(\boldsymbol{inv}_1(A, B, C, i, N) \wedge C'[j] = j) \implies A[j] = B[j]$. This query yields $A[j] = B[j]$ as Q. Combining R and Q into a quantified formula, and conjoining it with $\mathcal{M}[\boldsymbol{inv}_1]$ gives:

$$\mathcal{M}[\boldsymbol{inv}_1] \overset{\text{cand}}{\mapsto} \forall j.\, (0 \le j < i) \implies A[j] = B[j] \wedge \forall j.\, (i \le j < N) \implies A[j] = B[j]$$

In the third iteration, another *intra* CHC $C3$ fails the check. Here, $assign(C3)$ is $A'[j] = B[j]$ and β is $A'[j] = B[j]$, resulting in the following abduction query: $(\boldsymbol{inv}_1(A, B, C, i, N) \wedge A'[j] = C[j]) \implies A'[j] = B[j]$. This query yields $B[j] = C[j]$ as Q. Combining this with R, which is $i \leq j < N$, results in:

$$\mathcal{M}[\boldsymbol{inv}_1] \overset{\text{cand}}{\mapsto} \forall j.\, (0 \leq j < i) \implies A[j] = B[j] \wedge \\ \forall j.\, (i \leq j < N) \implies (A[j] = B[j] \wedge B[j] = C[j])$$

Subsequently, a *precond* CHC, $C1$, fails the check. These CHCs have an initialization of the counter variable (i.e., $i = 0$), rendering the formula within the range $0 \leq j < i$ trivially $\top$. Therefore, the rule for *precond* CHC selects β from the other range, i.e., β is $A[j] = B[j] \wedge B[j] = C[j]$ from the range $i \leq j < N$ of $\mathcal{M}[\boldsymbol{inv}_1]$. This leads to the following abduction query: $(\boldsymbol{pre}(A, B, C, i, N) \wedge \top) \implies (A[j] = B[j] \wedge B[j] = C[j])$, which yields Q as $A[j] = B[j] \wedge B[j] = C[j]$. Further, R is $0 \leq j < N$, resulting in:

$$\mathcal{M}[\boldsymbol{pre}] \overset{\text{cand}}{\mapsto} \forall j.\, (0 \leq j < N) \implies (A[j] = B[j] \wedge B[j] = C[j]).$$

The algorithm terminates as the candidate $\mathcal{M}$ is a solution.

6 Specialized Maximality Checking

While SAA effectively infers precondition, it may not always be the weakest. To check for maximality, a specialized CHC system (G and Γ) is generated in Algorithm 1 using the method GetSplCHCs, which is described in this section. This section also covers the method to weaken a precondition.

6.1 GetSplCHCs method

The GetSplCHCs method constructs a new CHC system G by iterating over all the CHCs in the input system S while performing the following:

1. Replacing $\boldsymbol{pre}$ with $\neg\mathcal{M}[\boldsymbol{pre}]$ and the postcondition ρ with $\neg\rho$, and
2. For each relation $\boldsymbol{inv}_i$, if there exist two *intra* CHCs C and C' with $guard(C) \wedge guard(C') \not\Rightarrow \bot$, then for each *intra* CHC C of $\boldsymbol{inv}_i$, a new relation $\boldsymbol{g}_C(args(body(C))$ is added to $body(C)$.

Example 5. The CHC system from Fig. 3, has *intra* CHCs $C2$ and $C3$ with $guard(C2) = guard(C3) = i < N$, so $guard(C) \wedge guard(C') \not\Rightarrow \bot$. As a result, two new relations $\boldsymbol{g}_{C2}$ and $\boldsymbol{g}_{C3}$ are introduced into $body(C2)$ and $body(C3)$, leading to the CHC system shown in Fig. 4. For this system, SAA finds: $\boldsymbol{g}_{C2} \mapsto A[i] \neq B[i]$ and $\boldsymbol{g}_{C3} \mapsto A[i] = B[i]$, along with an invariant for $\boldsymbol{inv}_1$.

A solution to G can result in interpretations for guard relations that block all executions (e.g., $\bot$). To prevent this, the following non-CHC constraint (Γ) will be introduced:

Algorithm 4: WEAKEN$(S, \mathcal{M})$

Input: S – set of CHCs over $\mathcal{R}$, $\mathcal{M}$ – mapping for $\mathcal{R}$
Output: $\langle \{\top, \bot\}, \mathcal{M} \rangle$ – $\top$ indicates $\mathcal{M}$ has a weakened $\boldsymbol{pre}$ and $\bot$ indicates failure

```
1 Δ ← CandidatePrecond(S, M);
2 for δ ∈ Δ do
3     S' ← replace pre by δ in S;
4     (res, M') ← SAA(S', ∅);
5     if res then return (⊤, M') where M'[pre] = δ ;
6 return ⟨⊥, M⟩;
```

$$\top \implies \bigvee_{1 \leq j \leq m} \left(\boldsymbol{g}_{C_j}(args(body(C_j)) \wedge guard(C_j) \right)$$

Theorem 3. *If a CHC system S induced by a program P has a solution $\mathcal{M}$, and its specialized CHCs (G and Γ) are satisfied, then $\mathcal{M}[\boldsymbol{pre}]$ is the weakest precondition of P.*

6.2 Weakening Procedure

When SAA is inconclusive on the specialized CHCs, the precondition $\mathcal{M}[\boldsymbol{pre}]$ is weakened, as shown Algorithm 4.The algorithm begins by computing a set of potential candidate preconditions Δ. We assume that this set is computed in a syntax-guided fashion like in [22] (can also be provided by the user). Only candidate preconditions that are strictly weaker than $\mathcal{M}[\boldsymbol{pre}]$ are taken into consideration. For each such candidate $\delta \in \Delta$, SAA is invoked to infer inductive invariants by passing the CHC system S with the relation $\boldsymbol{pre}$ replaced by δ. This process continues till success or all the candidates have been exhausted. Whenever the method succeeds, the precondition is weaker by construction.

7 Evaluation

Implementation Our algorithm is implemented in a tool called MAXPRANQ on top of the HORNSPEC framework [50]. The tool takes as input a set of CHCs with preconditions and invariants represented as uninterpreted relations. It returns the weakest precondition, along with proof of validity (viz. inductive invariants) and maximality (viz. specialized CHCs and its solution). It uses Z3 [14] to solve SMT queries. Quantifier elimination is done by model-based projection [3, 20].

Research Questions We evaluate MAXPRANQ on the following questions:

RQ1 Can MAXPRANQ find weakest preconditions for a range of benchmarks?
RQ2 How well does MAXPRANQ perform in comparison with state-of-the-art techniques?
RQ3 How challenging is it for existing techniques to infer invariants for our benchmarks even with the preconditions being *given*?

Benchmarks and Configuration We use 66 precondition inference tasks with universal quantified postconditions. While we initially intended to use the precondition inference benchmarks from [53], none of them had quantified postconditions. Hence, we derived our benchmarks from existing *verification tasks* of [22] that had quantified postconditions. Specifically, we considered multiple loop benchmarks from [22], where the first loop is an initialization loop, and the other loops perform various array update operations. We then removed the first loop so that a non-trivial quantified precondition would need to be synthesized. Overall, we consider 26 multiple loop benchmarks from [22]. Since a majority of the 26 benchmarks were deterministic, we added non-deterministic guards to the update operations and introduced a similar update operation in the other branch . To further test our tool, we adapted these benchmarks to 40 more benchmarks by using common array update operations and postconditions. We performed the experiments on a Ubuntu machine with 2.5 GHz processor and 16 GB memory. A timeout of 200 seconds was given to all the tools.

Tools for comparison We compare our tool against PreQSyn [49], an abduction-based precondition inference tool, and P-Gen [53], a predicate abstraction based tool. Additionally, we compare against the CHC solvers that can generate quantified inductive invariants: FreqHorn [22], a SyGuS based tool, and Spacer [30] (Z3 v4.8.10), an extension of PDR for quantified formulas.

RQ1 MaxPrANQ found and automatically proved 59/66 weakest preconditions. For the remaining 7, it found the weakest precondition, but couldn't prove it automatically due to failure in finding a solution to the specialized CHCs. Overall, it solved 125 CHC systems – 66 universal and 59 existential quantification. The time taken was less than 30 seconds on all except one benchmark. Details are in Fig 5 and [48].

RQ2 PreQSyn found and automatically proved 2/66 weakest preconditions. On the remaining 64, it found preconditions for 56 but could not prove; for 8, it did not find a precondition. To compare with our maximality checking, we provided the 56 preconditions generated by PreQSyn to our SMC module. Out of 56, SMC proved 52 to be the weakest, where 7 were weakened before proving. We observe that PreQSyn's maximality checking is unsuitable for non-deterministic programs, and its preconditions are not always the weakest.

P-Gen did not find a precondition for any of the 66 benchmarks. Its output was not a precondition on 41, and on the rest it was stuck in the refinement loop. Our experiments conclude that P-Gen's inference engine is unable to generalize and find quantified preconditions when postconditions are quantified. Hence, our technique complements P-Gen's capability of finding preconditions for quantifier-free postconditions.

RQ3 The reader may wonder whether the benchmarks themselves are easier to solve, given the limited availability of weakest precondition tools for non-deterministic programs. We experimentally demonstrate that this is not the case

by passing the CHCs *with* preconditions generated by MaxPrANQ to state-of-the-art CHC solvers for arrays: FreqHorn and Spacer. Out of 66 benchmarks, FreqHorn found inductive invariants for 56 and Spacer found 34. In comparison, MaxPrANQ found preconditions and invariants for all the 66 benchmarks.

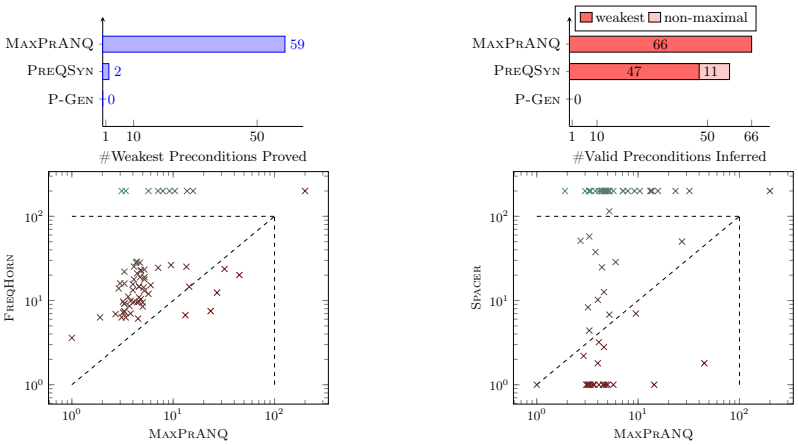

Fig. 5: The bar graphs show the # weakest proven and valid preconditions inferred by the tools. The scatter plots show the time taken by the tools for invariant inference.

8 Related Work

The problem of precondition inference has received considerable attention [2,11, 12,24,46,47,52,53]. In particular, for programs with arrays, closely related works include [12,49,53]. The work in [12] infers preconditions by abstract interpretation, [53] by CEGAR based predicate abstraction, and [49] by range abduction. Compared to [12], we don't need a predefined abstract domain. We work in a framework similar to [53] and [49], but they target deterministic programs. Their maximality check assumes that from a precondition only one execution reaches the postcondition, which is not the case for non-deterministic programs. The novelty of our SAA algorithm, compared to range abduction [49], is in how it constructs abduction queries using a set of rules based on the structure of the CHCs, and support for non-linear CHCs. Range abduction, on the other hand, creates two abduction queries and employs the Houdini technique [23], which can generate stronger preconditions, as observed in our experiments.

Precondition inference is closely related to the problem of invariant inference. For inferring universally quantified invariants, several techniques have been proposed. The main methods include predicate abstraction [32,41], abstract interpretation [28], PDR [30], and syntax guided synthesis [22]. These techniques are

crucially dependent on given preconditions, which are missing in our setting. Without preconditions, they generate trivial solution like $\bot$.

The validity of a precondition can be established by techniques that do not explicitly generate inductive invariants. Such techniques include array smashing [5], converting to array-free nonlinear CHCs [44], over-approximating unknown bound of loops to a smaller known bound [40], accelerating entire transition relations [6], using CHC transformation [4, 34], induction based techniques [7–9] and trace logic based techniques [25]. These techniques are useful for assertion checking and not directly for precondition inference, however.

CHCs are widely used to symbolically encode different synthesis tasks [18, 21,50,51,55]. However, none of these works handle CHCs with arrays. SAA uses abduction that has been used for programs without arrays to infer invariants [16, 17], preconditions [15, 26], and specifications [1, 50, 54].

The concept of SMC resembles angelic verification [13, 42], but differs in how it is solved. Angelic verification neither guarantees maximality nor computes inductive invariants, and uses user supplied specifications. A recent work [27] proposes a reduction of maximality checking to finding termination proofs for CHC systems with integers. In comparison, SMC reduces to finding inductive invariants and guards by exploiting the fact that the programs are terminating.

9 Limitations and Future Work

Usage of Theorem Prover: The preconditions and invariants guessed by SAA in our evaluation are in a fragment of the theory of one-dimensional arrays and linear integer arithmetic that state-of-the-art SMT solvers support reasonably well. However, in a general case, SAA might generate a challenging precondition/invariant for our SMT solver. In such instances, MaxPrANQ logs a failure and switches to another precondition/invariant. To handle such cases, we plan to complement our SMT solver by an automated theorem prover like Vampire [39]. This can also help us to handle preconditions with alternating quantification.

Non-linear CHC Support: The multi-abduction done in SAA can help in handling non-linear CHCs, which can encode programs with recursive functions. For this, the range analysis in SAA has to be tweaked to determine a loop counter-like variable for recursive functions, which we target for future work.

Termination and Compositional Verification: The assumption of terminating programs helps in proving maximality by inferring invariants and stronger guards. Relaxing this assumption would require a more complex maximality checking. Finally, an immediate future work is to integrate this technique in an existing verifier to scale it compositionally.

Acknowledgement

We would like to thank Kumar Madhukar for providing feedback on our paper.

References

1. A. Albarghouthi, I. Dillig, and A. Gurfinkel. Maximal specification synthesis. In *POPL*, pages 789–801. ACM, 2016.
2. A. Astorga, P. Madhusudan, S. Saha, S. Wang, and T. Xie. Learning stateful preconditions modulo a test generator. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 775–787, 2019.
3. N. Bjørner and M. Janota. Playing with quantified satisfaction. In *LPAR (short papers)*, volume 35 of *EPiC Series in Computing*, pages 15–27. EasyChair, 2015.
4. N. Bjørner, K. McMillan, and A. Rybalchenko. On solving universally quantified Horn clauses. In *International Static Analysis Symposium*, pages 105–125. Springer, 2013.
5. B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software. In *The essence of computation*, pages 85–108. Springer, 2002.
6. M. Bozga, P. Habermehl, R. Iosif, F. Konečný, and T. Vojnar. Automatic verification of integer array programs. In *Computer Aided Verification*, pages 157–172. Springer Berlin Heidelberg, 2009.
7. S. Chakraborty, A. Gupta, and D. Unadkat. Verifying array manipulating programs by tiling. In *International Static Analysis Symposium*, pages 428–449. Springer, 2017.
8. S. Chakraborty, A. Gupta, and D. Unadkat. Verifying array manipulating programs with full-program induction. In *TACAS (1)*, volume 12078 of *Lecture Notes in Computer Science*, pages 22–39. Springer, 2020.
9. S. Chakraborty, A. Gupta, and D. Unadkat. Diffy: Inductive reasoning of array programs using difference invariants. In *CAV (2)*, volume 12760 of *Lecture Notes in Computer Science*, pages 911–935. Springer, 2021.
10. A. Champion, T. Chiba, N. Kobayashi, and R. Sato. Ice-based refinement type discovery for higher-order functional programs. In *TACAS, Part I*, volume 10805 of *LNCS*, pages 365–384. Springer, 2018.
11. P. Cousot, R. Cousot, M. Fähndrich, and F. Logozzo. Automatic inference of necessary preconditions. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 128–148. Springer, 2013.
12. P. Cousot, R. Cousot, and F. Logozzo. Precondition inference from intermittent assertions and application to contracts on collections. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 150–168. Springer, 2011.
13. A. Das, S. K. Lahiri, A. Lal, and Y. Li. Angelic verification: Precise verification modulo unknowns. In *CAV, Part I*, volume 9206 of *LNCS*, pages 324–342. Springer, 2015.
14. L. M. de Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *TACAS*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.
15. I. Dillig and T. Dillig. Explain: a tool for performing abductive inference. In *International Conference on Computer Aided Verification*, pages 684–689. Springer, 2013.
16. I. Dillig, T. Dillig, B. Li, and K. L. McMillan. Inductive invariant generation via abductive inference. In *OOPSLA*, pages 443–456. ACM, 2013.

17. M. Echenim, N. Peltier, and Y. Sellami. Ilinva: Using abduction to generate loop invariants. In *FroCoS*, volume 11715 of *LNCS*, pages 77–93. Springer, 2019.
18. P. Ezudheen, D. Neider, D. D'Souza, P. Garg, and P. Madhusudan. Horn-ICE learning for synthesizing invariants and contracts. *PACMPL*, 2(OOPSLA):131:1–131:25, 2018.
19. G. Fedyukovich, M. B. S. Ahmad, and R. Bodík. Gradual Synthesis for Static Parallelization of Single-Pass Array-Processing Programs. In *PLDI*, pages 572–585. ACM, 2017.
20. G. Fedyukovich, A. Gurfinkel, and A. Gupta. Lazy but Effective Functional Synthesis. In *VMCAI*, volume 11388 of *LNCS*, pages 92–113. Springer, 2019.
21. G. Fedyukovich, S. Prabhu, K. Madhukar, and A. Gupta. Solving Constrained Horn Clauses Using Syntax and Data. In *FMCAD*, pages 170–178. IEEE, 2018.
22. G. Fedyukovich, S. Prabhu, K. Madhukar, and A. Gupta. Quantified Invariants via Syntax-Guided Synthesis. In *CAV, Part I*, volume 11561 of *LNCS*, pages 259–277. Springer, 2019.
23. C. Flanagan and K. R. M. Leino. Houdini: an Annotation Assistant for ESC/Java. In *FME*, volume 2021 of *LNCS*, pages 500–517. Springer, 2001.
24. T. Gehr, D. Dimitrov, and M. Vechev. Learning commutativity specifications. In *International Conference on Computer Aided Verification*, pages 307–323. Springer, 2015.
25. P. Georgiou, B. Gleiss, and L. Kovács. Trace logic for inductive loop reasoning. In *FMCAD*, pages 255–263. IEEE, 2020.
26. R. Giacobazzi. Abductive analysis of modular logic programs. In *ILPS*, volume 94, pages 377–391, 1994.
27. Y. Gu, T. Tsukada, and H. Unno. Optimal chc solving via termination proofs. *Proceedings of the ACM on Programming Languages*, 7(POPL):604–631, 2023.
28. S. Gulwani, B. McCloskey, and A. Tiwari. Lifting abstract interpreters to quantified logical domains. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 235–246, 2008.
29. A. Gurfinkel, T. Kahsai, A. Komuravelli, and J. A. Navas. The SeaHorn Verification Framework. In *CAV*, volume 9206 of *LNCS*, pages 343–361. Springer, 2015.
30. A. Gurfinkel, S. Shoham, and Y. Vizel. Quantifiers on demand. In *ATVA*, volume 11138 of *LNCS*, pages 248–266, 2018.
31. T. A. Henzinger, T. Hottelier, L. Kovács, and A. Rybalchenko. Aligators for arrays (tool paper). In *Logic for Programming, Artificial Intelligence, and Reasoning: 17th International Conference, LPAR-17, Yogyakarta, Indonesia, October 10-15, 2010. Proceedings 17*, pages 348–356. Springer, 2010.
32. T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan. Abstractions from proofs. *ACM SIGPLAN Notices*, 39(1):232–244, 2004.
33. H. Hojjat, F. Konecný, F. Garnier, R. Iosif, V. Kuncak, and P. Rümmer. A verification toolkit for numerical transition systems - tool paper. In *FM*, volume 7436 of *LNCS*, pages 247–251. Springer, 2012.
34. H. Hojjat and P. Rümmer. The ELDARICA Horn Solver. In *FMCAD*, pages 158–164. IEEE, 2018.
35. T. Kahsai, R. Kersten, P. Rümmer, and M. Schäf. Quantified heap invariants for object-oriented programs. In *LPAR*, volume 46 of *EPiC Series in Computing*, pages 368–384. EasyChair, 2017.
36. T. Kahsai, P. Rümmer, H. Sanchez, and M. Schäf. Jayhorn: A framework for verifying Java programs. In *CAV, Part I*, volume 9779 of *LNCS*, pages 352–358. Springer, 2016.

37. N. Kobayashi, R. Sato, and H. Unno. Predicate abstraction and CEGAR for higher-order model checking. In *ACM*, pages 222–233. ACM, 2011.
38. L. Kovács and A. Voronkov. Finding loop invariants for programs over arrays using a theorem prover. In *International Conference on Fundamental Approaches to Software Engineering*, pages 470–485. Springer, 2009.
39. L. Kovács and A. Voronkov. First-order theorem proving and vampire. In *International Conference on Computer Aided Verification*, pages 1–35. Springer, 2013.
40. S. Kumar, A. Sanyal, R. Venkatesh, and P. Shah. Property checking array programs using loop shrinking. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 213–231, Cham, 2018. Springer International Publishing.
41. S. K. Lahiri and R. E. Bryant. Constructing quantified invariants via predicate abstraction. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 267–281. Springer, 2004.
42. S. K. Lahiri, A. Lal, S. Gopinath, A. Nutz, V. Levin, R. Kumar, N. Deisinger, J. Lichtenberg, and C. Bansal. Angelic checking within static driver verifier: Towards high-precision defects without (modeling) cost. In *FMCAD*, pages 169–178. IEEE, 2020.
43. Y. Matsushita, T. Tsukada, and N. Kobayashi. RustHorn: CHC-Based Verification for Rust Programs. In *ESOP*, volume 12075 of *LNCS*, pages 484–514. Springer, 2020.
44. D. Monniaux and L. Gonnord. Cell morphing: From array programs to array-free horn clauses. In *Static Analysis*, pages 361–382. Springer Berlin Heidelberg, 2016.
45. D. Mordvinov and G. Fedyukovich. Verifying Safety of Functional Programs with Rosette/Unbound. *CoRR*, abs/1704.04558, 2017. `https://github.com/dvvrd/rosette`.
46. Y. Moy. Sufficient preconditions for modular assertion checking. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 188–202. Springer, 2008.
47. S. Padhi, R. Sharma, and T. Millstein. Data-driven precondition inference with learned features. *ACM SIGPLAN Notices*, 51(6):42–56, 2016.
48. S. Prabhu, D. D'Souza, S. Chakraborty, R. Venkatesh, and G. Fedyukovich. Weakest precondition inference for non-deterministic linear array programs (extended version). 2024. `https://doi.org/10.6084/m9.figshare.25050077`.
49. S. Prabhu, G. Fedyukovich, and D. D'Souza. Maximal Quantified Precondition Synthesis for Linear Array Loops. In *ESOP*, volume TBA of *LNCS*, page TBA. Springer, 2024. to appear.
50. S. Prabhu, G. Fedyukovich, K. Madhukar, and D. D'Souza. Specification Synthesis with Constrained Horn Clauses. In *PLDI*, pages 1203–1217. ACM, 2021.
51. S. Prabhu, K. Madhukar, and R. Venkatesh. Efficiently learning safety proofs from appearance as well as behaviours. In *SAS*, volume 11002 of *LNCS*, pages 326–343. Springer, 2018.
52. S. Sankaranarayanan, S. Chaudhuri, F. Ivančić, and A. Gupta. Dynamic inference of likely data preconditions over predicates by tree learning. In *Proceedings of the 2008 international symposium on Software testing and analysis*, pages 295–306, 2008.
53. M. N. Seghir and D. Kroening. Counterexample-guided precondition inference. In *European Symposium on Programming*, pages 451–471. Springer, 2013.
54. Z. Zhou, R. Dickerson, B. Delaware, and S. Jagannathan. Data-driven abductive inference of library specifications. *Proceedings of the ACM on Programming Languages*, 5(OOPSLA):1–29, 2021.

55. H. Zhu, S. Magill, and S. Jagannathan. A data-driven CHC solver. In *PLDI*, pages 707–721. ACM, 2018.