Optimal CHC Solving via Termination Proofs

YU GU, University of Tsukuba, Japan
TAKESHI TSUKADA, Chiba University, Japan
HIROSHI UNNO, University of Tsukuba, Japan and RIKEN AIP, Japan

Motivated by applications to open program reasoning such as maximal specification inference, this paper studies *optimal CHC solving*, a problem to compute maximal and/or minimal solutions of constrained Horn clauses (CHCs). This problem and its subproblems have been studied in the literature, and a major approach is to iteratively improve a solution of CHCs until it becomes optimal. So a key ingredient of optimization methods is the optimality checking of a given solution.

We propose a novel optimality checking method, as well as an optimization method using the proposed optimality checker, based on a computational theoretical analysis of the optimality checking problem. The key observation is that the optimality checking problem is closely related to the termination analysis of programs, and this observation is useful both theoretically and practically. From a theoretical perspective, it clarifies a limitation of an existing method and incorrectness of another method in the literature. From a practical perspective, it allows us to apply techniques of termination analysis to the optimality checking of a solution of CHCs. We present an optimality checking method based on constraint-based synthesis of termination arguments, implemented our method, evaluated it on CHCs that encode maximal specification synthesis problems, and obtained promising results.

CCS Concepts: • **Theory of computation → Logic and verification**; **Automated reasoning**; **Program verification**.

Additional Key Words and Phrases: specification synthesis, termination analysis, constrained Horn clause

ACM Reference Format:
Yu Gu, Takeshi Tsukada, and Hiroshi Unno. 2023. Optimal CHC Solving via Termination Proofs. *Proc. ACM Program. Lang.* 7, POPL, Article 21 (January 2023), 28 pages. https://doi.org/10.1145/3571214

1 INTRODUCTION

Specification synthesis is an enabling technique for modular verification of *open programs* which interact with other libraries and user programs; Open library functions can be invoked arbitrarily by unknown user programs, and conversely open user programs invoke library functions as black boxes. Depending on the assumptions on the type of interactions between the target program and its environment, specification synthesis takes various forms such as sufficient precondition inference [Padhi et al. 2016; Sankaranarayanan et al. 2008; Seghir and Kroening 2013; Srivastava and Gulwani 2009], angelic verification [Blackshear and Lahiri 2013; Das et al. 2015; Lahiri et al. 2020], and maximal specification synthesis [Albarghouthi et al. 2016; Prabhu et al. 2021; Zhou et al. 2021].

This paper focuses on a variant of maximal specification synthesis, namely computation of maximal solutions of a given set of constrained Horn clauses (CHCs). Since our idea to compute a

Authors' addresses: Yu Gu, University of Tsukuba, Japan, kou@logic.cs.tsukuba.ac.jp; Takeshi Tsukada, Chiba University, Japan, tsukada@math.s.chiba-u.ac.jp; Hiroshi Unno, University of Tsukuba, Japan and RIKEN AIP, Japan, uhiro@cs.tsukuba.ac.jp.

2475-1421/2023/1-ART21
https://doi.org/10.1145/3571214

maximal solution is applicable to synthesizing a minimal solution, we shall study the problem to calculate maximal and/or minimal solution of a given set of CHCs. We call this class of problems *Constrained Horn Clause optimization* (or *CHC optimization*). This subsumes problems such as multi-abduction [Albarghouthi et al. 2016] and maximal CHC solving [Prabhu et al. 2021].

The key ingredient of our optimal solution synthesis method is a novel optimality checker that *guarantees the optimality* via constraint-based synthesis of termination arguments; Our optimality checking method is derived based on a computational theoretical analysis of the problem of deciding the optimality of a given solution for (subclasses of) CHC optimization problems: we show that optimality checking can be reduced to pfwCSP [Unno et al. 2021] that extends CHCs with non-Horn clauses, well-foundedness, and functionality constraints. The analysis also reveals a limitation of existing methods [Albarghouthi et al. 2016; Hashimoto and Unno 2015; Srivastava and Gulwani 2009], i.e. there exists a problem which cannot be solved by existing methods. It also proves the unsoundness of an optimality checking method [Prabhu et al. 2021].[1]

Our CHC optimization method iteratively refines the current solution of the given CHCs C until an optimal one is found. To this end, we solve a series $C_0, C_1, C_2, \dots$ of constraint sets where $C_0 = C$ and each $C_{i+1} (i \geq 0)$ is obtained from C by adding a constraint that specifies that any solution of C_{i+1} *strictly improves* the solution found for C_i. To express $C_{i+1} (i \geq 0)$, we introduce and use a new class of predicate constraint satisfaction problems that extends CHCs with *non-emptiness* constraints on predicate variables. Our theoretical analysis shows that the new class is computationally easier than that of existentially quantified CHCs [Beyene et al. 2013] that has been used for the same purpose by the previous predicate constraint optimization method [Hashimoto and Unno 2015].

This paper further presents a data-driven approach to automating the constraint-based optimality checking and specification refinement. Specifically, we present a method based on CounterExample Guided Inductive Synthesis (CEGIS) [Solar-Lezama et al. 2006] and stratified families of templates [Unno et al. 2021] for solving a new class pfwnCSP of predicate constraint satisfaction problems that extends pfwCSP with non-emptiness constraints.

We implemented our specification synthesis method, evaluated it on a diversity of specification synthesis problems, and obtained promising results: despite the computational hardness of optimality checking, our method synthesized optimal specifications for non-trivial programs. Thus our novel approach to optimality checking via termination proofs, along with the computational theoretical analysis, paves the way for future directions of optimal specification synthesis.

Organization of This Paper. The rest of the paper is organized as follows. Section 2 presents a brief overview of our approach to optimal specification synthesis. Section 3 defines the class of CHC optimization problems and its subclasses. Section 4 gives the theoretical analysis of the (subclasses of) CHC optimization. Section 5 derives our optimality checking method based on termination proofs from the observations in the theoretical analysis. Section 6 formalizes our CHC optimization procedure by introducing the class pfwnCSP of predicate constraint satisfaction problems, which is used in the three sub-procedures: constraint-based optimality checking, constraint-based solution refinement (or non-optimality checking), and pfwnCSP solving. We report on our implementation and evaluation of the presented method in Section 7 and compare it with related work in Section 8. We conclude the paper with some remarks on future work in Section 9.

[1]It is worth noting and emphasizing here that the main technical contents of Prabhu et al. [2021] are heuristic methods that aim to construct a "practically useful" solution, which are independent of the flaw in their optimality checker. It also discusses a way to weaken a given solution.

```
s = 0; i = 0;
while * do
  if * then
    s = s + f(i);
  else
    i = i + 1;
assert(s >= 0);
```

(a) Example of an input program.

```
def f(x):
  y = rand();
  if S(x,y) then
    return y
  else diverge

s = 0; i = 0;
while * do
  if * then
    s = s + f(i);
  else
    i = i + 1;
assert(s >= 0);
```

(b) Reduction of the sufficiency of F to the safety verification.

```
a = rand$_1^\forall$(); b = rand$_2^\forall$();
assert(not S(a,b))

def f(x):
  y = rand$_3^\exists$();
  if S(x,y) or (x,y)=(a,b) then
    return y
  else diverge

s = 0; i = 0;
while rand$_4^\exists$() do
  if rand$_5^\exists$() then
    s = s + f(i);
  else
    i = i + 1;
assert(s >= 0);
```

(c) Reduction of the maximality of F to the angelic-demonic reachability analysis.

Fig. 1. Example of a maximal specification inference problem and reduction of the sufficiency and maximality checking to program analysis.

2 OVERVIEW

This section provides the idea of the method of this paper. The main contribution of this paper is an application of the termination analysis of programs to the CHC optimization problem. We explain why the termination analysis is relevant, as well as differences between our approach and existing methods [Albarghouthi et al. 2016; Zhou et al. 2021], using examples of the *maximal specification inference* problem [Albarghouthi et al. 2016], which is closely related to the CHC optimization problem via the well-known relationship between programs and CHCs.

Maximal Specification Inference

The *maximal specification inference* problem asks, given a program with assertions and an external function f, to give a maximal (i.e. weakest) specification for f among those ensuring the safety of the program. Following [Albarghouthi et al. 2016], we assume that a specification of a function $\mathtt{f}\colon \mathcal{D} \longrightarrow \mathcal{D}'$ is a predicate F over $\mathcal{D} \times \mathcal{D}'$. A function f satisfies this specification if and only if $F(x, \mathtt{f}(x))$ holds for every $x \in \mathcal{D}$. A specification S is *sufficient* if the program is safe whenever the implementation of f satisfies S. Otherwise it is *insufficient*. The goal of the maximal specification inference is to find a maximal specification (with respect to the standard subset ordering) among sufficient specifications.

An example of the input program is shown in Figure 1a. Specifications $S_1(x, y) :\Leftrightarrow (x \geq 0 \Rightarrow y \geq 0)$ and $S_2(x, y) :\Leftrightarrow (y = 0)$ are sufficient, and $S_3(x, y) :\Leftrightarrow (x \geq 10 \Rightarrow y \geq 0)$ is insufficient. A maximal specification is S_1, which is actually the unique maximal specification for the program in Figure 1a. There may be more than one maximal specification in general.

We construct a maximal specification by iteratively improving sufficient specifications. Let S_0 be a sufficient specification. We check if S_0 is maximal, and if it is not, we construct another sufficient specification S_1 that is strictly larger than S_0. We iterate this process until a maximal specification is found. The key step is the maximality check.

Sufficiency Checking as Program Analysis

The sufficiency of a given specification S is reducible to the safety verification of a program. The idea is to give an implementation of the "external" function f that is the "worst" among those satisfying F. The "worst" implementation is

```
def f(x):
  y = rand();
  if S(x,y) then return y else diverge
```

which nondeterministically returns a value following the specification S. This is the "worst" in the sense that, if the input program with this implementation is safe, then so is the program with arbitrary implementation of f that satisfies S. Hence S is sufficient if and only if the program in Figure 1b is safe.

Note that this idea of the reduction is applicable to check whether S is insufficient: S is insufficient if and only if the program in Figure 1b is unsafe. So the insufficiency of a specification can be reduced to the *reachability problem*,[2] which is the dual of the safety verification problem.

Maximality Checking as Program Analysis

The maximality checking is also reducible to program analysis, although the result of the reduction is a more difficult problem. Let S be a specification, which we would like to prove to be maximal. If S is maximal, for every pair (a, b) such that $\neg S(a, b)$, the extension $S_{a,b}(x, y) :\Leftrightarrow (S(x, y) \lor (x, y) = (a, b))$ is insufficient:

$$S \text{ is maximal} \qquad \Longleftrightarrow \qquad \forall (a, b) \in \mathcal{D} \times \mathcal{D}'. \quad \neg S(a, b) \text{ implies } S_{a,b} \text{ is insufficient.}$$

For a given (a, b), the insufficiency checking of $S_{a,b}$ is reducible to the reachability analysis of a certain program. Describing the universal quantifier $\forall (a, b) \in \mathcal{D} \times \mathcal{D}'$ as a program results in a single program of which the reachability problem is equivalent to the maximality of S.

Figure 1c is the resulting program. It first chooses the pair $(a, b) \in \mathcal{D} \times \mathcal{D}'$, nondeterministically, such that $\neg S(a, b)$. Then it executes the input program with the "worst" implementation of f with respect to $S_{a,b}$. The resulting program (Figure 1c) is nondeterministic. It is worth noting here that the program of Figure 1c has two kinds of nondeterministic branches:[3]

- The nondetermenistic choices of a and b are *demonic*, that means, we should prove that the program reaches the assertion failure whatever the choices of a and b are.
- All other nondeterministic choices are *angelic*, that means, we should find appropriate choices so that the program reaches the assertion failure.

By this way, the maximality checking is reduced to the reachability analysis of a program with angelic and demonic nondeterministic branches.

Once the maximality problem is reduced to the reachability analysis of a program, a variety of reachability analysis methods [Beyene et al. 2014; Cook et al. 2006; Fedyukovich et al. 2018; Giesl et al. 2017; Gonnord et al. 2015; Heizmann et al. 2014; Kura et al. 2021; Kuwahara et al. 2014; Lee et al. 2001; Unno et al. 2021; Urban et al. 2016] become applicable to solve the problem. A standard

[2]We do not distinguish the reachability problem and the termination problem, which asks reachability to the end of the program.

[3]The meaning of angelic and demonic choices might be counter-intuitive. In the terminology here, reaching an assertion failure is regarded as a good event because the maximality of a given specification is equivalent to the assertion failure for every proper extension of the specification.

```
a = rand_1^∀(); b = rand_2^∀();
assert(a >= 0 and b < 0)

def f(x):
  y = b;
  if (x < 0 or y >= 0) or (x,y)=(a,b)
  then return y
  else diverge
```

```
s = 0; i = 0;
while i <= a do
  if a == i and s >= 0 then
     s = s + f(i);
   else
     i = i + 1;
assert(s >= 0);
```

Fig. 2. Program in Figure 1c with particular specification $S = S_1$ and particular implementations of $\texttt{rand}^{\exists}()$.

technique is to reduce the reachability analysis to the safety verification problem by finding an appropriate *ranking function* [Alias et al. 2010; Ben-Amram and Genaim 2014, 2017; Bradley et al. 2005; Leike and Heizmann 2014; Podelski and Rybalchenko 2004; Urban 2013; Urban and Miné 2014], which gives an upper bound of the number of steps needed to reach the desired program point. A slight difference from a common setting is that the program in Figure 1c has not only demonic branches but also angelic nondeterministic choices $\texttt{rand}_i^{\exists}()$. We remove $\texttt{rand}_i^{\exists}()$ by replacing it with an appropriate concrete function $\texttt{choice}_i(\vec{x})$, which may depend on the arguments $\vec{x}$ representing the current state of the program execution. Given a ranking function and choice functions, their appropriateness can be ensured by solving the safety verification of a program.

We give an example of appropriate choice functions and a ranking function for the program of Figure 1c with specification $S_1(x, y) = (x \geq 0 \Rightarrow y \geq 0)$. Representing a state of the program as a tuple of current values of a, b, s and i, an example of appropriate choice functions is given by

$$\texttt{choice}_3(a, b, s, i) = b \qquad \texttt{choice}_5(a, b, s, i) = \begin{cases} \texttt{true} & \text{if } a = i \wedge s \geq 0 \\ \texttt{false} & \text{if } a \neq i \vee s < 0. \end{cases}$$

$$\texttt{choice}_4(a, b, s, i) = \begin{cases} \texttt{true} & \text{if } a \leq i \\ \texttt{false} & \text{if } a > i \end{cases}$$

The program obtained by replacing $\texttt{rand}_i^{\exists}()$ with $\texttt{choice}_i\texttt{(a,b,s,i)}$ is shown in Figure 2. Under the choice above, the program first executes `i = i + 1` until `i` becomes `a`, then it executes `s = s + f(i)` where `f(i)` returns `b`, and then exits the loop. Then `s = b < 0` (provided that the assertion `assert(a >= 0 and b < 0)` is passed) and hence the assertion `assert(s >= 0)` fails as expected. A ranking function is a witness of the fact that the chosen execution trace eventually breaks the loop: in this example, the number of the loop iteration is bounded by `a+1`.

Comparison with Other Methods

Albarghouthi et al. [2016] proposed a method for the maximal specification inference problem. Their method uses a solver of the *multi-abduction* problem developed in the same paper.

The *multi-abduction* problem coincides with the maximal specification inference problem for *loop-free* programs. An important feature of the multi-abduction problem is that the maximality checking of a candidate solution is reducible to SMT of quantified formulas. This fact can be explained from the view point of program analysis as follows: a loop-free program has a simple control flow, so its execution can be directly described by a first-order formula, where $x = \texttt{rand}^{\forall}()$ and $y = \texttt{rand}^{\exists}()$ are replaced with $\forall x$ and $\exists y$, respectively. Ranking function is needless because the termination of a loop-free program is trivial.

The method for the maximal specification inference in Albarghouthi et al. [2016] tries to prove the reachability of a complex program by finding an *under-approximation* that reaches the assertion failure. The class of under-approximations that they used consists of loop-free programs, which can be obtained by expanding a loop an appropriate number of times and pruning branches that seem unnecessary. Their method avoids the need of a ranking function by focusing on under-approximations that obviously terminate.

Unfortunately this approach has a limitation: there exists a program that the method of Albarghouthi et al. [2016] cannot find any maximal specification. Consider the maximal specification inference problem for the program in Figure 1a. This program has a unique maximal specification, namely $S(x, y) :\Leftrightarrow (x \geq 0 \Rightarrow y \geq 0)$. With this specification S, the program with angelic and demonic branches in Figure 1c reaches the assertion failure. However none of its loop-free under-approximation reaches the assertion failure, i.e. for every loop-free under-approximation of the program in Figure 1c, the daemon has appropriate choices for the values of $\texttt{rand}^{\forall}()$ so that the program will not reach the assertion failure. The point is that every loop-free program has a bound n such that the program terminates within n steps. Given an under-approximation of the program in Figure 1c, the daemon chooses a number greater than the bound n as the value for `a`. Then `f(a)` is not called in the under-approximation, and hence the under-approximation cannot distinguish between S and $S_{\mathsf{a},\mathsf{b}}$.

From the view-point of the termination analysis, the method of Albarghouthi et al. [2016] tries to find a ranking function independent of the daemon's choice `(a,b)`. However, all appropriate ranking functions for Figure 1c depend on `(a,b)`. So a more elaborated termination argument is inevitable in some cases.

Our method employs techniques of the termination analysis and actually find the maximal solution for the program in Figure 1a.

The termination analysis of a program is a quite hard problem, and one may wonder if the maximality checking problem is reducible to an easier problem such as SMT, which is decidable for some theories. Prabhu et al. [2021] proposed a reduction of the maximality checking (of the CHC optimization problem, which generalizes the maximal specification inference) to SMT, but we found that their algorithm is incorrect: The algorithm may erroneously judge a non-maximal solution as maximal. We show that the maximality checking is as hard as the termination analysis[4] of a Turing-complete programming language (Theorem 4.2), and hence it is not reducible to decidable SMT. Actually this is the starting point of our work and the reason why we thought that the termination analysis would be useful to the maximality checking.

3 THE PROBLEMS AND CLASSIFICATION

This section briefly explains the optimization problems studied in this paper and compare them with problems discussed in other work [Albarghouthi et al. 2016; Hashimoto and Unno 2015; Prabhu et al. 2021; Zhou et al. 2021]. We assume the basic knowledge of first-order logic (see, *e.g.*, Smullyan [1968]).

3.1 Preliminaries: Underlying Logics, Structures and Constraint Horn Clauses

The optimization problems are parameterized by *first-order signatures*. It consists of sets of *(basic) sorts*, *function symbols* and *predicate symbols*. Each sort basically corresponds to a data type of the target programs: for example, if the target program uses integers and integer lists, the signature

[4]Here the termination analysis means the problem to ask if a given program terminates on *all* inputs. This differs from (and is strictly harder than) the *halting problem* of Turing machines, which coincides with the problem to ask if a given program terminates on a *given* input.

has sorts `int` and `int_list` of integers and integer lists. Each symbol is sorted: a function symbol is associated to its sort $s_1 \times \cdots \times s_n \to s$, where $s_1, \ldots, s_n, s$ are basic sorts, and a predicate symbol to $s_1 \times \cdots \times s_n \to \mathit{Prop}$.

Example 3.1. The signature for *linear integer arithmetic* has unique sort `int` of integers, function symbols $+ : \mathtt{int} \times \mathtt{int} \to \mathtt{int}$ and $\mathtt{0}, \mathtt{1} : \mathtt{int}$, and a predicate symbol $< : \mathtt{int} \times \mathtt{int} \to \mathit{Prop}$. We sometimes consider the signature with a function symbol $((-) \bmod n) : \mathtt{int} \to \mathtt{int}$ for each non-zero natural number n. □

Given a signature, the set of formulas is defined as usual. A formula may have predicate variables and/or function variables (in addition to predicate and function symbols specified in the signature). In order to distinguish predicate/function variables from standard variables of first order logic, the latter are sometimes called *object variables*. Object, predicate and function variables are sorted and only well-sorted terms and formulas are considered in the sequel. A *sentence* is a closed formula, *i.e.* a formula with no free variable. A *theory* is a set of sentences. The top-level universal quantifier is often omitted: for example, $\forall x, y. \big(\varphi(x, y) \Rightarrow \exists z. \psi(x, y, z)\big)$ is written as $\varphi(x, y) \Rightarrow \exists z. \psi(x, y, z)$.

A *structure* $\mathcal{A}$ determines the interpretation of each symbol. For each (basic) sort s, $\mathcal{A}[\![s]\!]$ is a set; For each function symbols f of sort $s_1 \times \cdots \times s_n \to s$, $\mathcal{A}[\![f]\!]$ is a function from $\mathcal{A}[\![s_1]\!] \times \cdots \times \mathcal{A}[\![s_n]\!]$ to $\mathcal{A}[\![s]\!]$; For each predicate symbol P of sort $s_1 \times \cdots \times s_n \to \mathit{Prop}$, $\mathcal{A}[\![P]\!]$ is a function from $\mathcal{A}[\![s_1]\!] \times \cdots \times \mathcal{A}[\![s_n]\!]$ to the set of truth values $\{\bot, \top\}$. A structure determines the truth value of each sentence. The theory consisting of true sentences w.r.t. $\mathcal{A}$ is written as $\mathit{Th}(\mathcal{A})$.

Example 3.2. The structure $\mathbb{Z}$ interprets each symbol in the signature of linear integer arithmetic as expected: for example, `int` is interpreted as the set of integers $\mathbb{Z}$. The *theory of linear integer arithmetic* is $\mathit{Th}(\mathbb{Z})$. □

Let us fix a first-order signature and its structure.

A *constraint language* is just a set of formulas (with no function/predicate variable).

Example 3.3. LIA is a constraint language consisting of all formulas of the signature of linear integer arithmetic. QF-LIA consists of quantifier-free formulas of the signature of linear integer arithmetic. □

Let us fix a constraint language.

A *constrained Horn clause* (or *CHC*) is a formula of one of the following forms

$$\forall \vec{x}. \quad P_1(\vec{t}_1) \land \cdots \land P_n(\vec{t}_n) \land \varphi \Longrightarrow Q(\vec{u})$$
$$\forall \vec{x}. \quad P_1(\vec{t}_1) \land \cdots \land P_n(\vec{t}_n) \land \varphi \Longrightarrow \bot,$$

where $\vec{x}$ is the sequence of object variables appearing in the formula, $\vec{t}_1, \ldots, \vec{t}_n$ and $\vec{u}$ are sequences of terms, φ is a formula in the constraint language, and $P_1, \ldots, P_n$ and Q are predicate variables. We assume that a CHC has no function variable.[5] The position of Q is called the *head position*.

A *CHC system* is a finite set of CHCs, regarded as their conjunction. A formula that is logically equivalent to a conjunction of CHCs is sometimes called a CHC even if it is not a CHC in the proper sense. An example of such a formula is $\forall \vec{x}.\ P(\vec{t}) \land \varphi \Longrightarrow Q_1(\vec{u}_1) \land Q_2(\vec{u}_2)$, which is equivalent to the conjunction of $\forall \vec{x}.\ P(\vec{t}) \land \varphi \Longrightarrow Q_1(\vec{u}_1)$ and $\forall \vec{x}.\ P(\vec{t}) \land \varphi \Longrightarrow Q_2(\vec{u}_2)$.

A *solution* of a CHC system is an interpretation of predicate variables that makes all formulas in the system true. For a solution ξ, we write $\xi(P)$ for the interpretation of P under ξ. The solutions are naturally ordered: $\xi \leq \zeta$ if and only if $\forall \vec{x}. \xi(P)(\vec{x}) \Longrightarrow \zeta(P)(\vec{x})$ for every predicate variable P. The problem to decide if a given CHC system has a solution is called the *CHC satisfiability problem*.

[5] Formulas with function variables shall be used in Sections 5 and 6.

3.2 CHC Optimization

Now we define the CHC optimization problem.

Definition 3.4 (CHC optimization [Hashimoto and Unno 2015]). For interpretations ξ and ζ of predicate variables $PV = \{P_1, \ldots, P_n\}$ and a subset $\mathcal{X} \subseteq PV$, we write $\xi <_{\mathcal{X}} \zeta$ if $\forall \vec{x}. \xi(P)(\vec{x}) \Rightarrow \zeta(P)(\vec{x})$ for every $P \in \mathcal{X}$ and the implication is proper for some $P \in \mathcal{X}$. The *CHC optimization* is a problem of one of the following forms:

$$\textbf{maximize } \mathcal{X} \textbf{ s.t. } C \qquad \textbf{minimize } \mathcal{X} \textbf{ s.t. } C$$

where C is a CHC system and $\mathcal{X}$ is a subset of predicate variables in C. In the former case, the goal is to find a *maximal* solution ξ of C w.r.t. $<_{\mathcal{X}}$, *i.e.* a solution ξ such that there is no solution ζ of C satisfying $\xi <_{\mathcal{X}} \zeta$. In the latter case, the goal is to find a *minimal* solution w.r.t. $<_{\mathcal{X}}$. □

Remark 1. Note that the CHC optimization problem asks to find a solution that is maximal/minimal among all *semantic solutions*. As we shall see, our procedure tries to construct a *logical formula representing a maximal/minimal solution*. Hence our goal is to find a *syntactic* solution that is *semantically* maximal/minimal.

Other settings can be found in the literature. For example, one may want to find a syntactic solution that is maximal/minimal among those expressive in a certain class of formulas [Hashimoto and Unno 2015; Srivastava and Gulwani 2009; Zhou et al. 2021]. Examples of such classes of formulas include the set of formulas shorter than a certain bound [Hashimoto and Unno 2015; Zhou et al. 2021] and the set of conjunctions of a certain finite set of formulas [Srivastava and Gulwani 2009]. □

Remark 2. One may be interested in a more complicated optimization problem like

$$\textbf{maximize } \mathcal{X} \textbf{ minimize } \mathcal{Y} \textbf{ maximize } \mathcal{Z} \textbf{ s.t. } C,$$

where $\mathcal{X}$, $\mathcal{Y}$ and $\mathcal{Z}$ are disjoint sets of predicate variables in C. This requires to find a solution that is maximal in $\mathcal{X}$ and $\mathcal{Z}$ and minimal in $\mathcal{Y}$, where $\mathcal{X}$ is more significant (*i.e.*, if $\xi <_{\mathcal{X}} \zeta$ and $\xi <_{\mathcal{Y}} \zeta$, then ζ is more desirable). This problem can be solved by iteratively solving the CHC optimization problem. First solve **maximize** $\mathcal{X}$ **s.t.** C and let ξ be a maximal solution w.r.t. $<_{\mathcal{X}}$. Our procedure returns a formula $\xi(P)$ for each $P \in \mathcal{X}$. Let C' be the CHC system obtained by replacing $P \in \mathcal{X}$ in C with $\xi(P)$. Then it suffices to solve the subproblem **minimize** $\mathcal{Y}$ **maximize** $\mathcal{Z}$ **s.t.** C'. □

Some subproblems are studied in other papers [Albarghouthi et al. 2016; Prabhu et al. 2021; Zhou et al. 2021].

- **Linear multi-abduction** [Albarghouthi et al. 2016]: This is a problem to find a maximal solution of
$$\forall \vec{x}. \quad P_1(\vec{t}_1) \land \cdots \land P_n(\vec{t}_n) \land \varphi \Rightarrow \bot,$$
where $P_1, \ldots, P_n$ are *distinct* predicate variables.
- **(Non-linear) multi-abduction** [Albarghouthi et al. 2016]: This is a problem to find a maximal solution of
$$\forall \vec{x}. \quad P_1(\vec{t}_1) \land \cdots \land P_n(\vec{t}_n) \land \varphi \Rightarrow \bot.$$
Here a predicate variable can appear more than once.
- **open CHC optimization**: This is a problem of the form **maximize** $\mathcal{X}$ **s.t.** C, where C is a CHC system that is open w.r.t. $\mathcal{X}$. Here a CHC system is *open* w.r.t. $\mathcal{X}$ if, for every $P \in \mathcal{X}$, P has no head occurrence in C. This problem coincides with *maximal specification inference* [Albarghouthi et al. 2016].

Among them, linear multi-abduction is easiest and CHC optimization is hardest.

From the view point of specification inference, open CHC optimization corresponds to the inference of the maximal specification of an external function `f`: the caller of `f` together with assertions is given but the implementation of `f` is not. The CHC optimization problem coincides with a variant of the maximal specification inference problem, where both the caller of `f` and the implementation of `f` are given.

4 THEORETICAL ANALYSIS OF CHC OPTIMIZATION

This section analyzes the CHC optimization problem from a theoretical view point.

The theoretical analysis of this section is motivated by an incorrect reduction of the maximality checking of a given solution of CHCs to SMT solving proposed by Prabhu et al. [2021]. As shown by Albarghouthi et al. [2016], the reduction is correct for the multi-abduction problem but, as we shall see, it does not work for the CHC optimization. So it is natural to ask if it is possible to reduce the maximality checking of the CHC optimization problem to SMT.

We show that the maximality checking of the CHC optimization problem is Π^0_2-complete (Theorem 4.2). This result gives a negative answer to the above question: the maximality checking is undecidable and hence is not reducible to a decidable SMT. Theorem 4.2 also has a positive consequence: techniques to solve any Π^0_2-hard problem would be applicable to the maximality checking. Motivated by this observation, we shall discuss in the next section how one can apply techniques for the termination analysis of programs, which is a typical Π^0_2-complete problem, to the maximality checking.

This section also discusses the existence of a maximal solution.

4.1 Motivation: Maximality Checking and SMT Solving

The maximality problem for multi-abduction can be reduced to (quantified) SMT, as shown by Albarghouthi et al. [2016], and hence the maximality problem for multi-abduction over QF-LIA is decidable. It is natural to ask if the maximality checking for CHC optimization is also reducible to SMT solving, and actually an attempt can be found in the literature [Prabhu et al. 2021]. In this subsection, we briefly review the maximality-checking algorithm for multi-abduction [Albarghouthi et al. 2016] and a subtlety of CHC optimization.

The key property of multi-abduction problem is the downward-closedness of solutions: if ζ is a solution and $\xi \leq \zeta$, then ξ is also a solution. This allows us to reduce the maximality checking to SMT, as we shall see. Consider the following multi-abduction problem:

$$\forall x, y. \quad P(x) \wedge P(y) \wedge \varphi(x, y) \Rightarrow \bot$$

and let ξ be a non-maximal solution, *i.e.* $\xi < \zeta$ for some solution ζ. The shape of $\zeta(P)$ is unclear. A trick is to choose an element c in the difference between $\xi(P)$ and $\zeta(P)$ and to consider $\zeta'(P)(x) := (\xi(P)(x) \vee x = c)$. Then $\xi < \zeta' \leq \zeta$ and, by the downward-closedness of solutions, ζ' is also a solution. Hence, if ξ is not maximal, there exists a solution ζ' such that $\zeta'(P)(x) = (\xi(P)(x) \vee x = c)$ for some c with $\neg\xi(P)(c)$. The existence of such a solution can be reduced to (quantified) SMT, namely the validity of

$$\exists c. \neg\psi(c) \wedge \Big(\forall x, y. (\psi(x) \vee x = c) \wedge (\psi(y) \vee y = c) \wedge \varphi(x, y) \Rightarrow \bot\Big)$$

where $\psi(x) := \xi(P)(x)$.

This approach based on the one-point extension of the current solution does not work for CHC systems. For example, consider the following CHC system over LIA

$$\forall x, y. \quad P(x) \wedge (x \bmod 2 = y \bmod 2) \Rightarrow P(y)$$

and a solution $\xi(P)(x) = (x \bmod 2 = 0)$. The unique maximal solution is $\zeta(P)(x) = \top$ and thus ξ is not maximal. However there is no solution of the form $\zeta'(P)(x) = (\xi(P)(x) \lor x = c)$ for any constant c such that $\neg\xi(P)(c)$; to extend ξ, one needs to add more than one points.

The maximality problem for CHC optimization was studied in [Prabhu et al. 2021]. They proposed an algorithm for checking the maximality that reduces the maximality problem to SMT, based on the above-discussed idea of one-point extension [Prabhu et al. 2021, Algorithm 6 and Theorem 5.2]. We found that their algorithm is incorrect by the reason discussed above:[6] The algorithm may erroneously judge a non-maximal solution as maximal.

It is natural to ask if the maximality problem for CHCs is reducible to SMT, despite the essential difference between multi-abduction and CHC optimization. The theoretical analysis in the next subsection gives an answer.

4.2 Undecidability of Maximality Checking

Let us first formally define the maximality checking problem.

Definition 4.1. The *maximality checking problem* asks if, given a CHC system C over predicate variables $PV = \{P_1, \dots, P_n\}$, a subset $\mathcal{X} \subseteq PV$ of predicate variables and a syntactic representation of a solution ξ (i.e. a formula $\xi(P)$ in the constraint language for each predicate variable $P \in PV$), there exists a solution ζ such that $\xi <_{\mathcal{X}} \zeta$. Note that ξ must be definable in the constraint language whereas ζ is not necessarily so.[7] □

The next theorem shows the hardness of the maximality problem for CHCs over QF-LIA, a commonly used constraint language.

Theorem 4.2. *The maximality problem for (open) CHCs over QF-LIA is Π^0_2-complete.*

Proof. We prove (1) the maximality checking problem for open CHCs is Π^0_2-hard and (2) the maximality checking problem for CHCs is in Π^0_2. Since the maximality checking problem for open CHCs is a subproblem of the problem for CHCs, the theorem follows from these claims.

(1) We reduce the validity problem for Π^0_2 formulas, which is Π^0_2-complete, to the maximality problem. The validity problem for Π^0_2 formulas asks if $\forall x.\exists y.P(x, y)$ is true for a given computable predicate $P(x, y)$ over integers x, y. Given a computable predicate P, consider the following program with an external function `f`:

```
i = f();  j = 0;
while (not P(i,j)) and (not P(i,-j)) do
    j = j + 1;
assert(false);
```

(here `P(i,j)` is a procedure that calculates $P(i, j)$, which is computable by assumption). The candidate specification for `f` is $\perp$, which is trivially sound. The specification $\perp$ is maximal if and only if the program fails whatever `f` returns. The latter condition is equivalent to $\forall x.\exists y.P(x, y)$ as required.

By the standard translation of programs to CHCs, we obtain a CHC system C representing the above program. Note that the multiplication is definable by a program only using $+$, $\leq$, 0 and 1, the resulting CHCs do not need to contain the multiplication symbol. Let F be the predicate symbol corresponding to `f`. It suffices to find a solution ξ of C that is definable by the constraint

[6] We contacted the authors of the paper [Prabhu et al. 2021] and they confirmed that the proof of the correctness theorem [Prabhu et al. 2021, Theorem 5.2] of their maximality checking algorithm isMaximal [Prabhu et al. 2021, Algorithm 6] was wrong.

[7] A solution ξ obtained by a CHC solver usually satisfies this condition. The comparator ζ should be unrestricted since we are interested in maximality in the semantic domain.

language and satisfies $\xi(F) = \bot$. We consider a slight modification C' of C: for each Horn clause $(\forall\vec{x}.\Phi \Rightarrow \Psi) \in C$, C' has its variant $\forall\vec{x}y.\Phi \wedge F(y) \Rightarrow \Psi$ where y is a fresh variable not in $\vec{x}$. Then C' is essentially the same as C if the assignment for F is not empty, but all clauses in C' is "disabled" if F is empty. Then the assignment ξ that maps every predicate variable $Q \in PV$ to $\bot$ is a solution of C', and $\forall x.\exists y.P(x,y)$ is true if and only if the solution ξ is a maximal solution of C' with respect to $<_{\{F\}}$.

(2) Let C be a CHC, $\mathcal{X} = \{P_1, \dots, P_n\}$ be a chosen subset of predicate variables and ξ be a candidate solution for C. Let $\varphi_i := \xi(P_i)$. Then ξ is maximal w.r.t. $\mathcal{X}$ if and only if

$$C \cup \{\, \varphi(\vec{x}_i) \Rightarrow P_i(\vec{x}_i) \mid i = 1, \dots, n \,\} \cup \{\top \Rightarrow P_i(\vec{c}_i) \mid i = 1, \dots, n\}$$

is unsatisfiable for every $\vec{c} := (\vec{c}_i)_{i=1,\dots,n}$ such that $\models \neg\varphi_1(\vec{c}_1) \vee \cdots \vee \neg\varphi_n(\vec{c}_n)$. It is well-known that the unsatisfiability of CHCs over LIA is in Σ^0_1; hence there exists a computable predicate Ψ such that the above CHC system is unsatisfiable if and only if $\models \exists y.\Psi(\vec{c}, y)$. So ξ is maximal w.r.t. $<_{\mathcal{X}}$ if and only if $\models \forall\vec{c}.\exists y.(\varphi_1(\vec{c}_1) \wedge \cdots \wedge \varphi_n(\vec{c}_n)) \vee \Psi(\vec{c}, y)$. So the maximality problem belongs to Π^0_2. □

This result has both positive and negative consequences.

On the negative side, it shows that the maximality problem cannot be completely solved by any decision procedures, including SMT, no matter how cleverly it is reduced.

Corollary 4.3. *Both the maximality problem and the non-maximality problem for (open) CHCs over QF-LIA are undecidable.* □

On the positive side, it suggests that an approach to a Π^0_2-complete problem would be applicable to the maximality problem. Section 5 discusses an approach to the maximality/minimality checking inspired by Theorem 4.2.

4.3 Existence of an Optimal Solution

Let us first examine the existence of a maximal/minimal solution in the semantic domain. It is well-known that a satisfiable CHC system has the minimum solution. Although it does not necessarily have the maximum solution, it always has a maximal solution.

Theorem 4.4. *Every satisfiable CHC system has a maximal solution.*

Proof. Let C be a CHC system over predicate variables PV, $\mathcal{X} \subseteq PV$ and $\mathcal{S}$ be the set of all solution of C. The solutions are ordered by $\zeta \leq_{\mathcal{X}} \xi :\Leftrightarrow (\forall P \in \mathcal{X}.\zeta(P) \Rightarrow \xi(P))$.

First we prove the claim for the case $\mathcal{X} = PV$. We use Zorn's lemma. To apply Zorn's lemma, it suffices to show that, for every linearly ordered subset $\mathcal{L} \subseteq \mathcal{S}$, there exists a solution $\zeta \in \mathcal{S}$ such that $\forall\xi \in \mathcal{L}.\ \xi \leq_{PV} \zeta$. Given a linearly ordered subset $\mathcal{L} \subseteq \mathcal{S}$, we define ζ by $\zeta(P) := \bigvee_{\xi\in\mathcal{L}} \xi(P)$. Obviously $\forall\xi \in \mathcal{L}.\ \xi \leq_{PV} \zeta$, and it is not difficult to check that ξ is a solution of C. So we can apply Zorn's lemma, concluding that $\mathcal{S}$ has a maximal element w.r.t. $\leq_{PV}$.

Let us consider the general case. For a solution ξ, we write $\hat{\xi}$ for the solution that coincides with ξ on $\mathcal{X}$ and $\hat{\xi}(Q) = \min\{\zeta(Q) \mid \zeta \in \mathcal{S} \wedge \forall P \in \mathcal{X}.\zeta(P) = \xi(P)\}$ for $Q \notin \mathcal{X}$. The right-hand-side is well-defined since every satisfiable CHC system has the minimum solution. Then $\xi \leq_{\mathcal{X}} \zeta$ if and only if $\hat{\xi} \leq_{PV} \hat{\zeta}$. Given a linearly ordered subset $\mathcal{L} \subseteq \mathcal{S}$ w.r.t. $\leq_{\mathcal{X}}$, the set $\hat{\mathcal{L}} := \{\hat{\xi} \mid \xi \in \mathcal{L}\}$ is a linearly ordered subset w.r.t. $\leq_{PV}$. We can apply the above argument to $\hat{\mathcal{L}}$ to construct an upper bound of $\mathcal{L}$ w.r.t. $\leq_{\mathcal{X}}$. □

A more interesting question is whether there exists a maximal solution with a syntactic representation, since a solver of the CHC optimization problem is usually expected to output a maximal/minimal solution. Albarghouthi et al. [2016, Theorem 5] gave a negative result for the theory of linear real arithmetic (LRA).

Theorem 4.5 ([Albarghouthi et al. 2016, Theorem 5]). *The multi-abduction problem over LRA may not have a maximal solution in LRA.* □

We show that this is also the case for LIA.

Proposition 4.6. *The multi-abduction problem*

$$P(x, y) \land P(x + y, y) \Longrightarrow \bot$$

over LIA does not have a maximal solution in LIA (even with the mod operators).

Proof. (Sketch) Assume for contradiction that there is a maximal solution $\varphi(x, y)$ in LIA. Since LIA with the modulo operators $((-) \bmod n)$ ($n > 1$ is a constant) admits quantifier elimination, we can assume without loss of generality that $\varphi(x, y)$ is quantifier-free. Let n be a common multiplier of $\{m \in \mathbb{N} \mid (e \bmod m) \text{ appears in } \varphi\}$. Then φ can be transformed into the following from:

$$\varphi(x, y) \quad \Leftrightarrow \quad \bigvee_{i=1}^{N} \psi_i(x, y)$$

where $\psi_i(x, y)$ is

$$(x \bmod n = c'_i) \land (y \bmod n = c''_i) \land \bigwedge_{j=1}^{M_i} a_{i,j}x + b_{i,j}y + c_{i,j} \geq 0$$

for each i. An important property of ψ_i is that, if $\psi_i(m, n)$ and $\psi_i(m+kn, n)$ hold, then $\psi_i(m+k'n, n)$ holds for every $0 \leq k' \leq k$.

Since φ is a maximal solution, at least one of $\varphi(m, n), \varphi(m + n, n), \varphi(m + 2n, n)$ must be true for every m; otherwise $\varphi \lor (x = m + n \land y = n)$ is a solution which is strictly greater than φ. Hence at least $N + 1$ points from $(0, n), (n, n), (2n, n), \dots, ((3N + 3)n, n)$ must satisfy φ. By the pigeonhole principle, there exists i such that ψ_i is true at least two distinct points in the above list, say (mn, n) and $(m'n, n)$, $m < m'$. Then $\psi_i((m + 1)n, n)$ also holds because of the above discussed property of ψ_i. Hence both $\varphi(mn, n)$ and $\varphi((m + 1)n, n)$ is true and thus φ violates the constraint, a contradiction. □

5 MAXIMALITY AS TERMINATION

This section discusses our approach to the maximality problem using the termination analysis of programs. This approach is motivated by the theoretical analysis in the previous section: we have seen that the maximality problem is Π^0_2-complete and the termination analysis[8] is a well-known Π^0_2-complete problem.

Main results of this section are reductions of the (non)optimality[9] of a given solution to constraint satisfaction problems that extends the CHC satisfiability problem:

- The non-minimality and non-maximality problems are reduced to the satisfiability problem of CHCs with *non-emptiness constraints*.

[8]The termination analysis here asks if a given program terminates on *all* inputs. The termination analysis in this sense is strictly harder than the *halting problem*, which asks if a given program terminates on a *given* input.

[9]We shall discuss reductions of both the optimality checking and the non-optimality checking, giving a constraint set that is satisfiable when a given solution is optimal, as well as a constraint set that is satisfiable when the given solution is not optimal. One may find it redundant since the satisfiability of the latter is equivalent to the unsatisfiability of the former. It is, however, not redundant because of the asymmetry between satisfiability and unsatisfiability: although the satisfiability of a constraint set is witnessed by a solution, it is usually hard to ensure that a constraint set is unsatisfiable. In our implementation, to decide whether a given solution is optimal or not, we run two constraint solvers in parallel, one of which solves the constraint corresponding to the optimality checking and the other to the non-optimality.

- The minimality and maximality problems are reduced to the satisfiability problem of CHCs with *function variables* and *well-foundedness predicate.*

The latter is based on the idea of the termination analysis of programs, illustrated in Section 2. Here the well-foundedness predicate comes from the termination analysis.

For simplicity, this section studies the following CHC system with a single predicate variable P

$$\forall x.\quad \iota(x) \implies P(x) \qquad\qquad \forall x.\quad P(x) \implies \sigma(x)$$
$$\forall x, y, z.\quad P(x) \land P(y) \land \tau(x, y, z) \implies P(z). \tag{1}$$

We also assume a solution ϑ of the above system.

5.1 Non-Optimality and CHC with Non-Emptiness

This subsection discusses a way to find a better solution, under the assumption that the solution ϑ is not optimal. The non-optimality checking can be easily reduced to the satisfiability problem for CHCs with non-emptiness, *i.e.* a system of CHCs with *non-emptiness constraint* $P \neq \emptyset$ for some predicate variables P.

The candidate solution ϑ is non-maximal if and only if the following CHC system with non-emptiness is satisfiable:

$$Q \neq \emptyset \qquad \forall x.\, \vartheta(x) \land Q(x) \implies \bot \qquad \forall x.\, \vartheta(x) \implies P(x)$$
$$\forall x.\, Q(x) \implies P(x) \qquad \forall x.\, P(x) \implies \sigma(x)$$
$$\forall x, y, z.\, P(x) \land P(y) \land \tau(x, y, z) \implies P(z).$$

Suppose that ϑ is not maximal, *i.e.* there exists another solution ϑ' such that $\forall x.(\vartheta(x) \implies \vartheta'(x))$ and $\exists x.(\neg\vartheta(x) \land \vartheta'(x))$. Then the substitution given by $P \mapsto \vartheta'$ and $Q \mapsto \neg\vartheta \land \vartheta'$ is a solution of the above system. Conversely, if the above system is satisfiable, the assignment to P by a solution is strictly larger than ϑ, since Q is non-empty and disjoint with ϑ and P contains both Q and ϑ. Since $\iota(x) \implies \vartheta(x)$, the assignment to P by the solution satisfies the CHC system (1).

The non-minimality checking problem can be solved by a similar way. The candidate solution ϑ is non-minimal if and only if the following system is satisfiable:

$$Q \neq \emptyset \qquad \forall x.\, Q(x) \implies \vartheta(x) \qquad \forall x.\, P(x) \land Q(x) \implies \bot$$
$$\forall x.\, \iota(x) \implies P(x) \qquad \forall x.\, P(x) \implies \vartheta(x)$$
$$\forall x, y, z.\, P(x) \land P(y) \land \tau(x, y, z) \implies P(z).$$

If ϑ is not minimal, there exists another solution ϑ' of (1) such that $\forall x.(\vartheta'(x) \implies \vartheta(x))$ and $\exists x.(\neg\vartheta'(x) \land \vartheta(x))$. Then the assignment given by $P \mapsto \vartheta'$ and $Q \mapsto \vartheta \land \neg\vartheta'$ is a solution of the above constraints. Conversely, if the above constraints are satisfiable, then the value of P in the solution is a strictly smaller solution of (1). Actually $P(x) \implies \vartheta(x)$ requires P to be smaller than or equal to ϑ. Furthermore, since Q must contain an element satisfying ϑ, the constraint $P(x) \land Q(x) \implies \bot$ requires that P is strictly smaller than ϑ.

Example 5.1. Consider the following system of CHCs

$$P(1), \qquad \forall x, y.\, P(x) \land P(y) \implies P(x + y)$$

and its solution $\vartheta(x) = \top$, which is not minimum. The system of CHCs with non-emptiness for the non-minimality checking is given by

$$Q \neq \emptyset \quad \forall x.\, Q(x) \implies \top \quad \forall x.\, P(x) \land Q(x) \implies \bot$$
$$P(1) \quad \forall x.\, P(x) \implies \top.$$
$$\forall x, y, z.\, P(x) \land P(y) \implies P(x + y)$$

A solution is $Q(x) = (-3 \leq x \leq -1)$ and $P(x) = (x \geq 0)$. □

For a satisfiable system of CHCs with non-emptiness constraint, the construction of a solution of the system is not harder than the standard CHC solving, at least in theory. This would be surprising since the satisfiability checking for this kind of systems is quite hard (i.e. Π^0_2-complete). The procedure for generating a solution simply enumerates the elements c of the domain $\mathcal{D}$ and tries to construct a solution of the CHC system obtained by replacing $Q \neq \emptyset$ with $Q(c)$. Since the system is assumed to be satisfiable, the procedure eventually finds an appropriate element $\hat{c}$ and gives a solution. This gives an algorithm using the CHC solver as an oracle.

In Section 6.4, we shall discuss a solver of CHCs with non-emptiness that basically follows the above-mentioned idea. A key to effectively solve the problems is a heuristic to choose an appropriate element c that one expects to be contained in a non-empty predicate Q.

Remark 3. As shown by Albarghouthi et al. [2016], the checking of non-optimality of a multi-abduction can be reduced to SMT (of formulas with both existential and universal quantifiers). Theorem 4.2 shows that the maximality problem for open CHCs (which coincides with the maximal specification inference in [Albarghouthi et al. 2016]) cannot be reduced to SMT. Note that the SMT over LIA is decidable but the maximality problem for open CHCs is Π^0_2-complete, which is strictly harder than any decidable problems (and even than the halting problem). □

5.2 Minimality Checking as Termination

Let us consider the case that the candidate solution ϑ is minimal. We discuss a way to show that it is actually minimal.

It is well-known that every satisfiable CHC system has the minimum solution. More precisely, the subset of constraints

$$\begin{aligned} &\forall x. && \iota(x) \Longrightarrow P(x) \\ &\forall x, y, z. && P(x) \wedge P(y) \wedge \tau(x, y, z) \Longrightarrow P(z) \end{aligned}$$

has the minimum solution $\hat{\vartheta}$, and the CHC system is satisfiable if and only if $\models \hat{\vartheta}(x) \Longrightarrow \sigma(x)$; hence $\hat{\vartheta}$ is the minimum solution of the CHC system (1) if it is satisfiable.

There are several characterizations of the minimum solution $\hat{\vartheta}$; here we discuss a characterization by a proof system. Let $\mathcal{D}$ be the domain of x, y, z in the CHC system (1). A *judgement* is of the form $P(c)$, where $c \in \mathcal{D}$. The proof rules are:

$$\frac{}{P(c)}[\models \iota(c)] \qquad \frac{P(d) \quad P(d')}{P(c)}[\models \tau(d, d', c)]$$

where [] is the side condition needed to use the rule.

Proposition 5.2. *$P(c)$ is provable if and only if $\models \hat{\vartheta}(c)$.*

Proof. The left-to-right direction can be proved easily by induction on the structure of proofs. To prove the converse, it suffices to show that $\{c \in \mathcal{D} \mid P(c) \text{ is provable}\}$ is a solution, which is also easy. □

Therefore the candidate solution ϑ is minimum if and only if $P(c)$ is provable for every $c \in \mathcal{D}$ such that $\models \vartheta(c)$. The problem whether $P(c)$ is provable is a kind of termination analysis, as we shall see below.

The termination problem considered here is concerned with transition systems with *angelic* and *demonic* nondeterminism. The angelic branches are chosen so that the system satisfies a required property; the demonic branches are chosen so that the system violates the property. The *angelic-demonic termination problem* asks if, given a system with angelic and demonic branches, it is possible to choose angelic branches so that the system terminates whatever demonic branches are chosen.

We regard the goal-oriented proof search as the angelic-demonic termination problem.

- To construct a proof of $P(c)$, one needs to choose a proof rule. When one chooses the right rule, one also needs to choose premises $P(d)$ and $P(d')$. These nondeterminismtic branches are angelic, *i.e.* an appropriate rule and premises should be chosen.
- When the right-rule is chosen, one needs to proof the premises $P(d)$ and $P(d')$ as well. Instead of checking existence of proofs for all premises, the daemon chooses one premise which should be examined and we check the existence of a proof of the chosen premise. This choice is implemented by demonic nondeterministic branching.

This system terminates from c if and only if $P(c)$ is provable.[10] Hence ϑ is minimum if and only if the system terminates from every state c such that $\models \vartheta(c)$.

We reformulate the termination analysis as a constraint solving. Observe that it suffices to choose a proof rule for each c such that $\models \vartheta(c) \wedge \neg\iota(c)$, since $P(c)$ is trivially provable if $\iota(c)$ holds. Let c be an element such that $\models \vartheta(c) \wedge \neg\iota(c)$. There is only one proof rule that concludes $P(c)$, namely,

$$\frac{P(d_1) \qquad P(d_2)}{P(c)} \qquad \text{where } \models \tau(d_1, d_2, c).$$

If the choice is appropriate, the premises are provable, that means, d_1 and d_2 again belong to the minimum solution. As we have to choose d_1 and d_2 for each c, the choices induce two functions defined by $f_1(c) = d_1$ and $f_2(c) = d_2$ (for each c such that $\models \vartheta(c) \wedge \neg\iota(c)$). Since $\models \tau(d_1, d_2, c)$ must hold and d_1 and d_2 must belong to the minimum solution, we have the following constraint for function symbols f_1 and f_2:

$$(\vartheta(z) \wedge \neg\iota(z)) \Rightarrow (\vartheta(f_1(z)) \wedge \vartheta(f_2(z)) \wedge \tau(f_1(z), f_2(z), z)).$$

Furthermore $g_n(\dots g_2(g_1(c))\dots)$ eventually reaches a state satisfying ι for every choice of $g_i = f_1$ or f_2. This kind of constraint can be expressed by using the *well-foundedness predicate WF*: $WF(W)$ is true if and only if there is no infinite sequence $c_0, c_1, c_2, \dots,$ such that $W(c_i, c_{i+1})$ for every $i \in \mathbb{N}$. In this case, the above-mentioned requirement is the well-foundedness of the relation

$$W \quad := \quad \{(c, f_1(c)) \mid \ \models \vartheta(c) \wedge \neg\iota(c)\} \cup \{(c, f_2(c)) \mid \ \models \vartheta(c) \wedge \neg\iota(c)\}.$$

Summarizing the above argument results in the following constraint set, where $f_1, f_2 : \mathcal{D} \longrightarrow \mathcal{D}$ and $W : \mathcal{D} \times \mathcal{D} \longrightarrow Prop$:

$$\begin{array}{rcl} \forall z. \quad \vartheta(z) \wedge \neg\iota(z) & \Longrightarrow & \begin{array}{l}\vartheta(f_1(z)) \wedge \vartheta(f_2(z)) \wedge W(z, f_1(z)) \\ \quad \wedge W(z, f_2(z)) \wedge \tau(f_1(z), f_2(z), z)\end{array} \\ & WF(W). & \end{array} \tag{2}$$

Theorem 5.3. *ϑ is the minimum solution of* (1) *if and only if the constraint set* (2) *is satisfiable.*

Proof. Suppose that ϑ is minimum. Then $\models \vartheta(c)$ if and only if $P(c)$ is provable. For each c such that $\models \vartheta(c) \wedge \neg\iota(c)$, let $r(c)$ be the size of the smallest proof of $P(c)$. Choose a proof of $P(c)$ of size $r(c)$. If the proof concludes $P(c)$ from $P(d)$ and $P(d')$, then $f_1(c) := d$ and $f_2(c) := d'$. Let W be the predicate defined by: $W(c, c')$ if and only if both $P(c)$ and $P(c')$ are provable and $r(c) > r(c')$. Then f_1, f_2, W satisfies the above constraint.

[10]We omit the proof of this fact. Omitting the proof does not cause any logical problem, because this is used only to illustrate the intuition behind the constraint set defined below, the correctness of which will be proved independently of the argument here.

To prove the converse, suppose that the above constraints are satisfiable. Let (f_1, f_2, W) be a solution. We construct a proof of $P(c)$ for each c such that $\models \vartheta(c)$ by induction on W. If $\models \iota(c)$, then $P(c)$ is trivially provable. If $\models \neg\iota(c)$, then $f_1(c)$ and $f_2(c)$ satisfy ϑ and are strictly smaller than c with respect to W. Hence by the induction hypothesis, $P(f_1(c))$ and $P(f_2(c))$ are provable. Then $P(c)$ is provable since $\models \tau(f_1(c), f_2(c), c)$. □

Example 5.4. Consider the following system of CHCs:

$$P(1), \qquad \forall x, y.\ P(x) \land P(y) \Longrightarrow P(x + y).$$

Then the minimum solution is $P(x) = (x \geq 1)$. The constraints expressing the termination problem is: $WF(W)$ and

$$\forall z. \quad z > 1 \Longrightarrow \begin{array}{l} f_1(z) \geq 1 \land f_2(z) \geq 1 \land W(z, f_1(z)) \\ \quad \land W(z, f_2(z)) \land z = f_1(z) + f_2(z). \end{array}$$

A solution is $f_1(x) = 1$, $f_2(x) = (x - 1)$, and $W(x, y) = (x > y \geq 1)$. □

5.3 Maximality Checking as Termination

Suppose that the candidate solution ϑ of (1) is maximal. This subsection discusses how to prove the maximality of ϑ.

Since ϑ is maximal, there is no solution ϑ' which is strictly greater than ϑ. In particular, for every c such that $\models \neg\vartheta(c)$, there is no solution that subsumes $\vartheta(x) \lor (x = c)$. Hence the CHC system obtained by replacing $\iota(x)$ in (1) with $\vartheta_c(x) := (\vartheta(x) \lor x = c)$ is unsatisfiable, because ι in (1) is a lower bound of the solution. Therefore, for the minimum solution ϱ_c of $\{\, \vartheta_c(x) \Longrightarrow P(x),\ P(x) \land P(y) \land \tau(x, y, z) \Longrightarrow P(z) \,\}$, we have $\models \neg(\forall x. \varrho_c(x) \Longrightarrow \sigma(x))$, *i.e.* $\models \varrho_c(d_c) \land \neg\sigma(d_c)$ for some d_c.

The validity of $\varrho_c(d_c)$ can be reduced to the termination problem, as in the previous section, because ϱ_c is the minimum solution. The corresponding constraint set is

$$\begin{array}{ll} \forall z. & D_c(z) \land \neg\vartheta_c(z) \Longrightarrow D_c(f_{c,1}(z)) \land D_c(f_{c,2}(z)) \\ & \qquad \land W_c(z, f_{c,1}(z)) \land W_c(z, f_{c,2}(z)) \land \tau(f_{c,1}(z), f_{c,2}(z), z) \\ WF(W_c) & \\ D_c(d_c) & \end{array} \tag{3}$$

where $\vartheta_c(x) = (\vartheta(x) \lor x = c)$ and all predicate variables have subscript c expressing the dependency. The main difference with (2) is that the candidate minimal solution ϑ is replaced with the predicate variable D_c representing (an underapproximation of) the minimum solution.

Proposition 5.5. *The CHC system (1) has no solution containing ϑ_c if and only if*

$$(3) \cup \{\neg\sigma(d_c)\}$$

is satisfiable. Here d_c appearing in (3) and in $\neg\sigma(d_c)$ is regarded as a nullary function symbol, the value of which should be specified by a solution.

Proof. Similar to the proof of Theorem 5.3. □

The solution ϑ is maximal if and only if the condition in the above proposition is true for every c with $\models \neg\vartheta(c)$. Collecting solutions for all c, we define

$$g(c) := d_c \qquad f_1(c, x) := f_{c,1}(x) \qquad f_2(c, x) := f_{c,2}(x)$$
$$W((c, x), (c', y)) :\Leftrightarrow (c = c') \land W_c(x, y)$$
$$D(c, x) :\Leftrightarrow D_c(x)$$

where $d_c, f_{c,1}, f_{c,2}, W_c$ and D_c are solutions of the constraint set in Proposition 5.5 if $\models \neg\vartheta(c)$; otherwise $D_c = W_c = \bot$ and $d_c, f_{c,1}$ and $f_{c,2}$ are arbitrary. The constraints for these functions and relations are given by:

$$\begin{aligned}
&\forall x.\quad \neg\vartheta(x) \Longrightarrow D(x, g(x)) \wedge \neg\sigma(g(x))\\
&\forall x, z.\quad D(x,z) \wedge \neg(\vartheta(z) \vee z = x) \wedge \neg\vartheta(x)\\
&\quad \Longrightarrow D(x, f_1(x,z)) \wedge D(x, f_2(x,z)) \wedge W((x,z),(x,f_1(x,z)))\\
&\qquad \wedge W((x,z),(x,f_2(x,z))) \wedge \tau(f_1(x,z), f_2(x,z), z)\\
&\quad WF(W).
\end{aligned} \tag{4}$$

Theorem 5.6. *ϑ is a maximal solution of (1) if and only if the constraint set (4) is satisfiable.* □

Example 5.7. Consider the following system of CHCs:

$$P(1), \qquad \forall x, y.\, P(x) \wedge P(y) \Longrightarrow P(x+y),$$
$$\forall x.\, P(x) \Longrightarrow x \geq -5.$$

Then the unique maximal solution is $\vartheta(x) = (x \geq 0)$. The constraint set expressing the termination problem is:

$$\begin{aligned}
&\forall x.\quad x < 0 \Longrightarrow D(x, g(x)) \wedge g(x) < -5\\
&\forall x, z.\quad D(x,z) \wedge \neg(z \geq 0 \vee z = x) \wedge \neg(x \geq 0)\\
&\quad \Longrightarrow D(x, f_1(x,z)) \wedge D(x, f_2(x,z)) \wedge W((x,z),(x,f_1(x,z)))\\
&\qquad \wedge f_1(x,z) + f_2(x,z) = z \wedge W((x,z),(x,f_2(x,z)))\\
&WF(W).
\end{aligned}$$

A solution is $D(x,y) = (x < 0 \wedge (\exists n.(1 \leq n \leq 6) \wedge (y = nx))$, $W((x,y),(x,y')) = (y < y' < 0)$, $g(x) = 6x$, $f_1(x,y) = x$, and $f_2(x,y) = (y - x)$. □

6 CHC OPTIMIZATION METHOD

This section describes our CHC optimization procedure using the reductions of (non)optimality checking to constraint satisfaction problems given in Section 5. We first explain the overall flow of our procedure (Section 6.1) and define a class of satisfaction problems, pfwnCSP, to which the (non)maximality checking problem is reduced (Section 6.2). We then give reductions of (non)maximality checking to pfwnCSP (Section 6.3) and develop a data-driven solver for pfwnCSP (Section 6.4).

6.1 CHC Optimization via Iterated Satisfaction

Given a CHC optimization problem **maximize** $\{P_1, \ldots, P_m\}$ **s.t.** C, our procedure starts from an arbitrary solution θ of C and iteratively updates θ until it becomes optimal. In each step, our procedure checks if there exists a solution θ' of C such that

$$\theta(P_1) \subsetneq \theta'(P_1) \;\wedge\; \theta(P_2) \subseteq \theta'(P_2) \;\wedge\; \ldots \;\wedge\; \theta(P_n) \subseteq \theta'(P_n). \tag{5}$$

That means, θ' is not worse than θ for $P_2, \ldots, P_n$ and θ' strictly improves θ on P_1. Note that the existence (resp. absence) of such a solution θ' is reducible to a constraint satisfaction problem as discussed in Section 5.1 (resp. Section 5.3). If such a solution θ' is found, set θ to θ' and continue the procedure. If the absence of such a solution is confirmed, it suffices to solve the subproblem

$$\textbf{maximize}\ \{P_2, \ldots, P_m\}\ \textbf{s.t.}\ C \cup \{\forall \vec{x}_1.\varphi_1(\vec{x}_1) \Rightarrow P_1(\vec{x}_1), \ldots, \forall \vec{x}_n.\varphi_n(\vec{x}_n) \Rightarrow P_n(\vec{x}_n)\}$$

Algorithm 1: Optimize

```
Input: A CHC Optimization Problem $(mode, \Delta, C, [P_1; \dots; P_m])$
Output: $OptSat(\theta)$ if an optimal solution $\theta$ is found, $Unsat$ if $C$ is not satisfiable
procedure OPTIMIZE $(mode, \Delta, C, [P_1; \dots; P_m])$ =
    case SOLVE$(C)$ of
        | $Sat(\theta) \longrightarrow$ AUX$(mode, \Delta, C, [P_1; \dots; P_m], \theta, \emptyset)$;
        | $Unsat \longrightarrow$ return $Unsat$;
end
procedure AUX $(mode, \Delta, C, [P_1; \dots; P_m], \theta, \theta_0)$ =
    if $m = 0$ then return $OptSat(\theta \uplus \theta_0)$;
    result := parallel any
        ■ SOLVE(ISOPT$(mode, \Delta, C, P_1, [P_2; \dots; P_m], \theta)$) ;
        ■ SOLVE(ISNONOPT$(mode, \Delta, C, P_1, [P_2; \dots; P_m], \theta)$) ;
    case result of
        | $First(Sat(\_))$ | $Second(Unsat)$ $\longrightarrow$
            $\Delta' := \Delta \setminus \{P_1\}$;
            $C' := \theta\restriction_{P_1}(C_\Delta)$;
            $\theta'_0 := \theta_0 \cup \{P_1 \mapsto \theta(P_1)\}$;
            AUX$(mode, \Delta', C', [P_2; \dots; P_m], \theta, \theta'_0)$
        | $Second(Sat(\theta_{refined}))$ $\longrightarrow$
            AUX$(mode, \Delta, C, [P_1; \dots; P_m], \theta_{refined}, \theta_0)$
        | $First(Unsat)$ $\longrightarrow$
            $Sat(\theta_{refined})$ := SOLVE(ISNONOPT$(mode, \Delta, C, P_1, [P_2; \dots; P_m], \theta)$);
            AUX$(mode, \Delta, C, [P_1; \dots; P_m], \theta_{refined}, \theta_0)$
end
```

where $\varphi_i = \theta(P_i)$ for each i. In this subproblem, the predicate variable P_1 is no longer a target of optimization as it has already been optimized. Furthermore the subpreblem requires that a solution subsumes the current solution θ. A minimization problem **minimize** $\{P_1, \dots, P_m\}$ **s.t.** C can be solved similarly.

Algorithm 1 formally describes our procedure OPTIMIZE for solving a given CHC optimization problem. OPTIMIZE$(\uparrow, \Delta, C, [P_1; \dots; P_m])$ computes a solution of **maximize** $\{P_1, \dots, P_m\}$ **s.t.** C, where Δ is the sort assignment for predicate variables in CHCs C, and OPTIMIZE$(\downarrow, \Delta, C, [P_1; \dots; P_n])$ computes a solution of **minimize** $\{P_1, \dots, P_m\}$ **s.t.** C. OPTIMIZE first computes any solution of C and then iteratively updates the solution until it becomes optimal, using the auxiliary procedure AUX.

AUX$(mode, \Delta, C, [P_1; \dots; P_m], \theta, \theta_0)$ returns an optimal solution θ_{opt} that is better than θ (i.e., if $mode = \uparrow$, we have $\theta(P_i) \subseteq \theta_{\text{opt}}(P_i)$, and otherwise $\theta(P_i) \supseteq \theta_{\text{opt}}(P_i)$). The basic idea of AUX has already explained at the beginning of this subsection. It reduces the existence and absence of a better solution (i.e. a solution satisfying (5)) to constraint satisfaction problems, ISNONOPT$(mode, \Delta, C, P_1, [P_2; \dots; P_m], \theta)$ and ISOPT$(mode, \Delta, C, P_1, [P_2; \dots; P_m], \theta)$, respectively, and solves these problems by using a solver SOLVE. The two constraint satisfaction problems are solved in parallel.[11] At the case analysis on Line 11, the first case is when the absence of a better solution is confirmed, and the second and third cases are when its existence is ensured. In the third case, the return value *Unsat* for Solve(IsOpt(*mode*, $\Delta, C, P_1, [P_2; \dots; P_m], \theta$) only tells us existence of a better solution without providing any witness, so we need to compute a better solution by solving IsNonOpt(*mode*, $\Delta, C, P_1, [P_2; \dots; P_m], \theta$).

We explain the sub-procedures IsOpt, IsNonOpt, and Solve in more details in Sections 6.3 and 6.4. Note that the classes of constraints generated by IsOpt and IsNonOpt go beyond CHCs: IsNonOpt requires CHCs extended with non-emptiness constraints and IsOpt requires pfwCSP [Unno et al. 2021], which extends CHCs with non-Horn clauses, well-foundedness, and functionality constraints. We thus introduce the class pfwnCSP that incorporates all the required extensions first in the next section.

6.2 pfwnCSP: Extending CHCs

We introduce an extension of CHCs that are equivalent to those in Unno et al. [2021].[12]

A *constrained clause* is a formula of the form

$$\forall \vec{y}. \quad \phi \vee \left(\bigvee_{i=1}^{l} P_i(\vec{x_i}) \right) \vee \left(\bigvee_{i=l+1}^{m} \neg P_i(\vec{x_i}) \right),$$

or equivalently,

$$\forall \vec{y}. \quad (P_{\ell+1}(\vec{x_{\ell+1}}) \wedge \dots \wedge P_m(\vec{x_m}) \wedge \varphi) \Rightarrow (P_1(\vec{x_1}) \vee \dots \vee P_\ell(\vec{x_\ell})),$$

where φ is a formula in the constraint language, $P_1, \dots, P_m$ are predicate variables and $0 \le \ell \le m$. A *predicate Constraint Satisfaction Problem* (pCSP) C is the problem to find a solution of a finite set of clauses [Satake et al. 2020]. Here, as usual, a solution is an interpretation of predicate variables that satisfies all given clauses. The notion of constrained clause is a generalization of constrained Horn clause: a constrained Horn clause is a constrained clause with $\ell = 0, 1$. Hence pCSP is a generalization of the CHC Satisfaction Problem.

We introduce the new class pfwnCSP that further extends pCSP by allowing following constraints:

- *Non-emptiness constraint*: $P \neq \emptyset$ where P is a predicate variable. This constraint requires that the interpretation of P in a solution θ must be non-empty, *i.e.* $\theta(P) \neq \emptyset$.
- *Well-foundedness constraint*: $WF(P)$ where P is a binary relation on a certain sort. This constraint requires that the interpretation of P in a solution θ is well-founded, *i.e.* there is no infinite sequence $a_1, a_2, \dots$ such that $\theta(P)(a_i, a_{i+1})$ holds for every i.

A constraint may have function variables in addition to predicate variables. A pfwnCSP C is a finite set of constraints over predicate variables and function variables, consisting of constrained clauses, non-emptiness constraints, and well-foundedness constraints.

The formal definition of pfwnCSP used in this paper is as follows. Assume a signature Σ and a structure $\mathcal{A}$, fixed in the sequel.

The set of *sorts* is defined by the following grammar:

$$S \quad ::= \quad \alpha \mid S_1 \times S_2 \mid S_1 + S_2,$$

[11] In Algorithm 1, the **parallel any** statement (Line 8) runs the given two statements in parallel and stops when one is over. If the first statement (resp. the second statement) terminates and returns a value v, then the value of **parallel any** is *First*(v) (resp. *Second*(v)).

[12] An additional feature is non-emptiness constraint. Although this constraint is definable using a function variable (since $P \neq \emptyset$ is equivalent to $P(f())$ for a fresh nullary function variable f), non-emptiness constraint is preferable since it is easier than general function variables.

where α ranges over basic sorts specified by the signature Σ, $S_1 \times S_2$ is the product sort and $S_1 + S_2$ is the sum sort, of which the interpretation is the disjoint union $\mathcal{A}[\![S_1]\!] + \mathcal{A}[\![S_2]\!]$ of the interpretations of S_1 and S_2. A *term* of sort σ is defined as usual. In particular, if t_1 and t_2 are terms of sort S_1 and S_2, then (t_1, t_2) is a term of sort $S_1 \times S_2$ and $\mathrm{inj}_i(t_i)$ $(i = 1, 2)$ is a term of sort $S_1 + S_2$.

Remark 4. An important difference from the setting in Section 3 is the sum sort $S_1 + S_2$. The sum sort is useful to pack multiple predicates into a single predicate: if P_i is a predicate over S_i for $i = 1, \ldots, n$, the tuple $(P_1, \ldots, P_n)$ of n predicates can be represented by a single predicate Q over the sort $S_1 + \cdots + S_n$ defined by $Q(\mathrm{inj}_i(x)) :\Leftrightarrow P_i(x)$. □

A *sort environment* Δ is a finite set of sort declarations of the forms $P \colon S \longrightarrow \mathit{Prop}$ and $f : S \longrightarrow S'$. Given Δ, the set of formulas possibly using predicate variables and function variables in Δ is defined as usual. We shall consider only well-sorted formulas.

Definition 6.1. A pfwnCSP (Δ, C) is a pair of a sort environment and a finite set of constraints over predicate variables and function variables, consisting of constrained clauses, non-emptiness constraints, and well-foundedness constraints. We assume that $f \colon S \longrightarrow S' \in \Delta$ implies that S' does not contain the sort constructor $+$. The set of constraints must satisfy the following conditions:

- For each predicate variable P, the set C contains at most one of $P \neq \emptyset$ and $\mathit{WF}(P)$.
- If $\mathit{WF}(P) \in C$, then $(P \colon S \times S \longrightarrow \mathit{Prop}) \in \Delta$ for some S.

A *solution* θ of C is an assignment to a predicate variable $(P \colon S \longrightarrow \mathit{Prop}) \in \Delta$ a predicate $\theta(P) \subseteq \mathcal{A}[\![S]\!]$ and to a function variable $(f \colon S \longrightarrow S') \in \Delta$ a function $\theta(f) \colon \mathcal{A}[\![S]\!] \longrightarrow \mathcal{A}[\![S']\!]$ that satisfies all constraints in C. The semantics of additional constraints are given as follows.

- $\theta \models P \neq \emptyset$ if $\theta(P) \neq \emptyset$.
- $\theta \models \mathit{WF}(P)$ (where $(P \colon \sigma \times \sigma \longrightarrow \mathit{Prop}) \in \Delta$) if there is no infinite sequence $a_1, a_2, \ldots$ of elements in $\mathcal{A}[\![\sigma]\!]$ such that $\theta(P)(a_i, a_{i+1})$ holds for every i. □

6.3 Constraint Generation

This subsection describes the subprocedures IsOpt and IsNonOpt, which reduce the absence and existence of a better solution to pfwnCSP. The basic idea has already explained in Section 5.

The sub-procedure $\textsc{IsNonOpt}(\uparrow, \Delta, C, P, [P_1; \ldots; P_m], \theta)$ is rather straightforward. We would like to have a constraint set characterizing the subset of solutions θ' of C such that $\theta(P_i) \subseteq \theta'(P_i)$ for $i = 1, \ldots, m$ and $\theta(P) \subsetneq \theta'(P)$. We define[13]

$$\begin{aligned}
&\textsc{IsNonOpt}(\uparrow, \Delta, C, P, [P_1; \ldots; P_m], \theta)\\
&\quad := \quad C \cup \left\{ \forall \vec{x}_i.\, \theta(P_i)(\vec{x}_i) \Rightarrow P(\vec{x}_i) \mid 1 \leq i \leq m \right\}\\
&\quad \cup \quad \left\{ \forall \vec{x}.\, \theta(P)(\vec{x}) \vee N(\vec{x}) \Rightarrow P(\vec{x}), \quad \forall \vec{x}.\, \theta(P)(\vec{x}) \wedge N(\vec{x}) \Rightarrow \bot, \quad N \neq \emptyset \right\}.
\end{aligned}$$

Here the predicate sort environment Δ' for the generated constraint set is $\Delta \uplus \{N : S \longrightarrow \mathit{Prop}\}$, where N is a fresh predicate variable and $S \longrightarrow \mathit{Prop}$ is the sort of P. The first and second parts say that a solution of $\textsc{IsNonOpt}(\uparrow, \Delta, C, P, [P_1; \ldots; P_m], \theta)$ is not worse than θ on $\{P_1, \ldots, P_m\}$, and the third part says that a solution is strictly better than θ on P. The predicate N under-approximates the difference $\theta'(P) \setminus \theta(P)$ between the current solution θ and a solution θ' of the generated constraint set. The inclusion between $\theta'(P)$ and $\theta(P)$ must be strict since N must be non-empty. The definition

[13] Although $\forall \vec{x}.\theta(P)(\vec{x}) \vee N(\vec{x}) \Rightarrow P(\vec{x})$ is not a clause, it is equivalent to the conjunction of clauses $\forall \vec{x}.\theta(P)(\vec{x}) \Rightarrow P(\vec{x})$ and $\forall \vec{x}.N(\vec{x}) \Rightarrow P(\vec{x})$. We shall implicitly use this kind of equivalences in this subsection to simplify the presentation.

of $\textsc{IsNonOpt}(\downarrow, \Delta, C, P, [P_1; \dots; P_m], \theta)$ is similar:

$$\begin{aligned}&\textsc{IsNonOpt}(\downarrow, \Delta, C, P, [P_1; \dots; P_m], \theta)\\ &\quad := \quad C \cup \left\{ \forall \vec{x}_i.\, P(\vec{x}_i) \Rightarrow \theta(P_i)(\vec{x}_i) \mid 1 \le i \le m \right\}\\ &\quad \cup \quad \left\{ \forall \vec{x}.\, P(\vec{x}) \vee N(\vec{x}) \Rightarrow \theta(P)(\vec{x}), \quad \forall \vec{x}.\, P(\vec{x}) \wedge N(\vec{x}) \Rightarrow \bot, \quad N \neq \emptyset \right\}.\end{aligned}$$

Note that $\textsc{IsNonOpt}(\mathit{mode}, \Delta, C, P, [P_1; \dots; P_m], \theta)$ belongs to the class of CHCs extended with non-emptiness constraints.

The correctness of the generated constraint sets is straightforward.

Lemma 6.2. *Let θ be a solution of (Δ, C).*

- *For every solution θ' of $\textsc{IsNonOpt}(\mathit{mode}, \Delta, C, P, [P_1; \dots; P_m], \theta)$, we have $\theta(P_i) \le_{\mathit{mode}} \theta'(P_i)$ for $i = 1, \dots, m$ and $\theta(P) <_{\mathit{mode}} \theta'(P)$.*
- *If there exists a solution θ' of C such that $\theta(P_i) \le_{\mathit{mode}} \theta'(P_i)$ for $i = 1, \dots, m$ and $\theta(P) <_{\mathit{mode}} \theta'(P)$, then $\textsc{IsNonOpt}(\mathit{mode}, \Delta, C, P, [P_1; \dots; P_m], \theta)$ is satisfiable.* □

The subprocedure $\textsc{IsOpt}(\mathit{mode}, \Delta, C, P, [P_1; \dots; P_m], \theta)$ is more complicated. Let us first explain the main differences from the setting in Sections 5.2 and 5.3.

- The number of predicate variables in C may be greater than 1.
- Only a part of the predicate variables is the target of optimization.
- For each predicate variable Q, C may have more than 1 clauses of the form $\cdots \Rightarrow Q(\vec{x})$.

These differences are not essential and the ideas discussed in Sections 5.2 and 5.3 are applicable to general CHCs. Roughly speaking, these gaps can be filled by preprocessing of CHCs. After the preprocessing, the CHCs has a single "large" predicate obtained by combining all predicate variables in Δ. When $\Delta = \{P_i \colon S_i \longrightarrow \mathit{Prop} \mid 1 \le i \le n\}$, such a "large" predicate variable R has sort $(S_1 + S_2 + \cdots + S_n) \longrightarrow \mathit{Prop}$, which is abbreviated as $(\sum_{i=1}^{n} S_i) \longrightarrow \mathit{Prop}$, and satisfies $\forall x : S_i.R(\mathrm{inj}_i(x)) \Leftrightarrow P_i(x)$. The constraint generation for the optimality checking given below is basically obtained by this way.

We introduce some notations and conventions. For each $Q \in \mathrm{dom}(\Delta)$, we write S_Q for the sort of the argument of Q (i.e. $Q \colon S_Q \longrightarrow \mathit{Prop} \in \Delta$). For simplicity, we assume that the constraint language admits the quantifier elimination (or equivalently, the existential quantifier can be used freely in the constraint language). Then, under a mild assumption on the constraint language, each CHC can be written as

$$\forall y_1 : S_{L_1}. \dots \forall y_n : S_{L_n}. \forall z : S_Q. \varphi(y_1, \dots, y_n, z) \wedge L_1(y_1) \wedge \cdots \wedge L_n(y_n) \Rightarrow Q(z)$$

or its variant with $\bot$ at the head position. We assume that each CHC is of the above form. When $\mathcal{R}$ is the above CHC, $S_{\mathit{Head}(\mathcal{R})} = S_Q$ and $S_{\mathit{Body}(\mathcal{R})} = S_{L_1} \times \cdots \times S_{L_n}$. The subset $C[Q]$ (resp. $C[\bot]$) of C consists of those with Q (rsep. $\bot$) at the head position. Then $C = C[\bot] \cup \bigcup_{Q \in \mathrm{dom}(\Delta)} C[Q]$.

Figure 3 shows the constraint set for checking the minimality of a given solution. In every solution, the interpretation of D_Q is an under-approximation of the interpretation of Q in the minimum solution of C. Hence the constraint $\forall x : S_P.\theta(P)(x) \Rightarrow D_P(x)$ ensures that θ coincides with the minimum solution of C on P. The predicate variable W and function variables $f^{\mathcal{R}}$ are used to describe the minimality requirements for $(D_Q)_{Q \in \mathrm{dom}(\Delta)}$ and correspond to W and $f(z) = (f_1(z), f_2(z))$ in the constraint set (2) in Section 5.2. The sort $(\sum_{Q \in \mathrm{dom}(\Delta)} s_Q) \times (\sum_{Q \in \mathrm{dom}(\Delta)} s_Q) \longrightarrow \mathit{Prop}$ of W can be explained by the intuition that we are reasoning about a CHC system with a single "large" predicate variable R of sort $(\sum_{Q \in \mathrm{dom}(\Delta)} s_Q) \longrightarrow \mathit{Prop}$. The constraint $\forall x \colon S_Q.\, D_Q(x) \Rightarrow \bigvee_{\mathcal{R} \in C[Q]} \mathit{ProvableBy}_{\mathcal{R}}(x)$ requires that, if $D_Q(x)$ is true, this fact should be justified by a clause $\mathcal{R}$ in $C[Q]$. A clause $\mathcal{R} \equiv \forall x\vec{y}.\varphi(x, \vec{y}) \wedge \bigwedge_{i=1}^{k} L_i(y_i) \Rightarrow Q(x)$ justifies $D_Q(a)$ if there exist $\vec{b}$

$$\begin{aligned}
\Delta' \triangleq\ & \{D_Q\colon S_Q \to \mathit{Prop} \mid Q \in \mathrm{dom}(\Delta)\} \\
\uplus\ & \{W\colon (\textstyle\sum_{Q\in\mathrm{dom}(\Delta)} s_Q) \times (\sum_{Q\in\mathrm{dom}(\Delta)} s_Q) \to \mathit{Prop}\} \\
\uplus\ & \{f^{\mathcal{R}}\colon S_{\mathit{Head}(\mathcal{R})} \to S_{\mathit{Body}(\mathcal{R})} \mid \mathcal{R} \in C[Q] \text{ for some } Q \in \mathrm{dom}(\Delta)\} \\
C' \triangleq\ & \{\mathit{WF}(W),\quad \forall x\colon \sigma_P.\ \theta(P)(x) \Rightarrow D_P(x)\} \\
\cup\ & \{\forall z\colon s_Q.\ D_Q(z) \Rightarrow \textstyle\bigvee_{\mathcal{R}\in C[Q]} \mathit{ProvableBy}_{\mathcal{R}}(z) \mid Q \in \mathrm{dom}(\Delta)\}
\end{aligned}$$

$$\begin{aligned}
\mathit{ProvableBy}_{\mathcal{R}}(z) \triangleq\ & \varphi(f^{\mathcal{R}}(z), z) \wedge \bigwedge_{i=1}^{n} D_{L_i}\big(\pi_i(f^{\mathcal{R}}(z))\big) \wedge W\big(\mathrm{inj}_Q(z), \mathrm{inj}_{L_i}(\pi_i(f^{\mathcal{R}}(z)))\big) \\
& \Big[\text{if } \mathcal{R} \text{ is } \forall y_1 \dots y_n z.\varphi((y_1,\dots,y_n),z) \wedge L_1(y_1) \wedge \cdots \wedge L_n(y_n) \Rightarrow Q(z)\Big]
\end{aligned}$$

Fig. 3. pfwCSP Constraint Generation $\mathit{IsOpt}(\downarrow, \Delta, C, P, \mathcal{X}, \theta) \triangleq (\Delta', C')$ for CHC Minimality Checking

such that $\varphi(a, \vec{b})$ holds and $D_{L_i}(b_i)$ has a "smaller witness" than $D_Q(a)$. The auxiliary predicate $\mathit{ProvableBy}_{\mathcal{R}}(x)$ describes this property, assuming that $\vec{b}$ is given by $f^{\mathcal{R}}(x)$.

The constraint set $\mathit{IsOpt}(\uparrow, C_\Delta, P, \mathcal{X}, \theta)$ for the maximality checking is given in Figure 4. Following the discussion in Section 5.3, we would like to give a constraint set describing the following condition:

> For each a such that $\neg\theta(P)(a)$, let θ'_a be the least solution of $\bigcup_{Q\in\mathrm{dom}(\Delta)} C[Q]$ that additionally satisfies $\theta(P)(x) \vee (x = a) \Rightarrow \theta'_a(P)$ and $\theta(Q)(x) \Rightarrow \theta'_a(Q)(x)$ (for each $Q \in \mathcal{X}$). Then θ'_a violates some rule in $C[\bot]$.

Constraints for under-approximations of θ'_a are given by the same idea as the minimality checking. The predicate variables D_Q and W and function variables $f^{\mathcal{R}}$ are essentially the same as those in the minimal case, except that those variables for the maximality checking have additional parameters of sort S_P corresponding to a. The assignment $\hat{\theta}(Q)(a, -)$ is the lower bound for $\theta'_a(Q)$. The constraint $\forall x\colon S_P.\ \neg\theta(P)(x) \Rightarrow \bigvee_{\mathcal{R}\in\Delta[\bot]} \mathit{FailAt}_{\mathcal{R}}(x)$ requires the violation of θ'_x to some $\mathcal{R} \in C[\bot]$.

The correctness of $\mathit{IsOpt}(\mathit{mode}, \Delta, C, P, \mathcal{X}, \theta)$ can be proved by arguments similar to those in Sections 5.2 and 5.3. The generated constraints belong to the subclass pfwCSP [Unno et al. 2021] of pfwnCSP without non-empty predicates.

6.4 Data-Driven pfwnCSP Solving

The sub-procedure Solve is for solving a pfwnCSP (Δ, C). Our procedure Solve is an extension of an existing data-driven method for solving pfwCSP over LIA [Unno et al. 2021] with non-emptiness constraints. Solve is based on CounterExample Guided Inductive Synthesis (CEGIS).

Before entering the CEGIS process, Solve eliminates non-emptiness constraints. It transform $(P \neq \emptyset) \in C$ into a pair of a *singleton set constraint*, *singleton*(Q), and a clause $\forall x.Q(x) \Rightarrow P(x)$, where Q is a fresh predicate variable. The singleton set constraint *singleton*(Q) requires that a solution assigns a singleton set to Q, as expected.

Solve iteratively accumulates example instances $\mathcal{E}$ of the original constraints (Δ, C), which are instances of clauses obtained by substituting variables with concrete values. For example, if $\forall xyz.\varphi(x, y, z) \wedge P(x) \wedge Q(y) \Rightarrow T(z)$ belongs to C, then $\varphi(1, 2, 3) \wedge P(1) \wedge Q(2) \Rightarrow T(3)$ and $\varphi(-8, 23, 9) \wedge P(-8) \wedge Q(23) \Rightarrow T(0)$ are example instances. We are interested only in example instances whose constraints (i.e. $\varphi(1, 2, 3)$ and $\varphi(-8, 23, 9)$ in the previous examples) are true, and call $P(1) \wedge Q(2) \Rightarrow T(3)$ and $P(-8) \wedge Q(23) \Rightarrow T(0)$ examples instances as well (provided

$$
\begin{aligned}
\Delta' \triangleq\ & \{D_Q\colon S_P \times S_Q \to \mathit{Prop} \mid Q \in \mathrm{dom}(\Delta)\}\\
\uplus\ & \{W\colon (S_P \times (\textstyle\sum_{Q\in\mathrm{dom}(\Delta)} S_Q)) \times (S_P \times (\textstyle\sum_{Q\in\mathrm{dom}(\Delta)} S_Q)) \to \mathit{Prop}\}\\
\uplus\ & \{f^{\mathcal{R}}\colon S_P \times S_{\mathit{Head}(\mathcal{R})} \to S_{\mathit{Body}(\mathcal{R})} \mid \mathcal{R} \in C[Q] \text{ for some } Q \in \mathrm{dom}(\Delta)\}\\
\uplus\ & \{g^{\mathcal{R}}\colon S_P \to S_{\mathit{Body}(\mathcal{R})} \mid \mathcal{R} \in C[\bot]\}\\
C' \triangleq\ & \{\mathit{WF}(W),\quad \forall x\colon S_P.\ \neg\theta(P)(x) \Rightarrow \textstyle\bigvee_{\mathcal{R}\in\Delta[\bot]} \mathit{FailAt}_{\mathcal{R}}(x)\}\\
\cup\ & \{\forall x\colon S_P. \forall z\colon S_Q.\ \neg\theta(P)(x) \wedge D_Q(x,z) \Rightarrow \hat{\theta}(Q)(x,z) \vee \textstyle\bigvee_{\mathcal{R}\in C[Q]} \mathit{ProvableBy}_{\mathcal{R}}(x,z)\\
& \hfill \mid Q \in \mathrm{dom}(\Delta)\}
\end{aligned}
$$

$$
\begin{aligned}
\mathit{ProvableBy}_{\mathcal{R}}(x,z) \triangleq\ & \varphi(z, f^{\mathcal{R}}(x,z))\\
& \wedge \bigwedge_{i=1}^{n} D_{L_i}\big(\pi_i(f^{\mathcal{R}}(x,z)\big) \wedge W\big((x, \mathrm{inj}_Q(z)), (x, \mathrm{inj}_{L_i}(\pi_i(f^{\mathcal{R}}(x,z))))\big)\big)\\
& \Big[\text{if } \mathcal{R} \text{ is } \forall x y_1 \dots y_n. \varphi(x, (y_1,\dots,y_n)) \wedge L_1(y_1) \wedge \cdots \wedge L_n(y_n) \Rightarrow Q(x)\Big]
\end{aligned}
$$

$$
\begin{aligned}
\mathit{FailAt}_{\mathcal{R}}(x) \triangleq\ & \varphi(g^{\mathcal{R}}(x)) \wedge \bigwedge_{i=1}^{n} D_{L_i}\big(x, \pi_i(g^{\mathcal{R}}(x)\big)\big)\\
& \Big[\text{if } \mathcal{R} \text{ is } \forall y_1 \dots y_n. \varphi(y_1,\dots,y_n) \wedge L_1(y_1) \wedge \cdots \wedge L_n(y_n) \Rightarrow \bot\Big]
\end{aligned}
$$

$$
\hat{\theta}(Q)(x,z) \triangleq \begin{cases} \theta(P)(z) \vee (x = z) & \text{if } Q = P\\ \theta(Q)(z) & \text{if } Q \in \mathcal{X}\\ \bot & \text{otherwise} \end{cases}
$$

Fig. 4. pfwCSP Constraint Generation $\mathit{IsOpt}(\uparrow, \Delta, C, P, \mathcal{X}, \theta) \triangleq (\Delta', C')$ for CHC Maximality Checking

that $\varphi(1,2,3)$ and $\varphi(-8,23,9)$ are true). Note that all example instances come from clauses; non-emptiness constraints and well-foundedness constraints do not generate example instances.

The main iteration of CEGIS consists of the following two phases:

- **Synthesis Phase**: find a candidate solution θ that satisfies all the examples in $\mathcal{E}$;
- **Validation Phase**: check whether the candidate θ also satisfies the original constraints (Δ, C), and if so, return θ as a genuine solution, and otherwise repeat the procedure with $\mathcal{E}$ extended with counterexamples.

The synthesis phase of our procedure Solve is template-based. A *template* of a predicate of arity n is a formula $\psi(x_1,\dots,x_n;c_1,\dots,c_m)$ with extra variables $c_1,\dots,c_m$. The extra variables are regarded as constants that have not yet been determined, and fixing their values determines a predicate. Similarly a template of a function is an expression with extra variables. In the synthesis phase, the solver prepares a template for each predicate variable and function variable. Substituting each predicate/function variable in $\mathcal{E}$ to the template results in a set of constraints over extra variables. The solver chooses templates carefully so that constraints after substitution can be solved by an SMT solver, and hence appropriate values of extra variables can be computed by an SMT solver.

The templates are chosen so that non-emptiness constraints and well-foundedness constraints are satisfied independent of the values of extra variables. If $WF(P) \in C$, then a template for $P(x, y)$ is $f(x) \geq 0 \land f(y) \geq 0 \land f(x) > f(y)$ for a fresh function variable f.[14] It is easy to see that this predicate is well-founded for every function f. A template for Q with the singleton set constraint $singleton(Q)$ is $x = e$, where e is an expression with extra variables.

The choice of template significantly affects the performance of the solver. The strategy of the solver by Unno et al. [2021] is based on stratified families of templates, which are designed for enumerating candidate predicates in a prioritized manner: starting from a less expressive initial template for each predicate variable, the solver gradually refines it in a counterexample-guided manner if no solution exists in the current template. The stratification of templates and the counterexample-guided refinement thus automatically adjust the expressiveness of templates depending on the complexity of solutions for the target constraint set. See Unno et al. [2021] for details.

The validation phase is straightforward since what we need to check is only the validity of clauses. We just to need to invoke an SMT solver.

7 IMPLEMENTATION AND EVALUATION

Section 4 has shown that the CHC optimization problem is Π_2^0-complete (Theorem 4.2). A natural question to ask is whether there is any possibility to realize a practical CHC optimization tool despite the theoretical hardness. The goal of this section is to provide an initial positive answer via preliminary experiments.

To this end, we implemented a prototype CHC optimization tool called OptPCSat based on Algorithm 1 using the Multicore OCaml language for concurrent programming. We extended an existing tool PCSat [Unno et al. 2021] for solving pfwCSP constraints with non-emptiness constraints and adopted it as our backend pfwnCSP solver (invoked in Lines 2, 9, 10, and 20). For optimization problems that only involve maximization, OptPCSat supports 3 different modes: no-side-condition mode, non-trivial mode, and non-vacuous mode. In the no-side-condition mode, OptPCSat solves the given CHC optimization problem as it is. In the non-trivial and the non-vacuous modes, OptPCSat finds an optimal solution of the given CHC constraints extended with side-conditions: the two modes require any ordinary predicate variable to be non-empty and the non-vacuous mode further requires the body of any definite clause to be non-empty. These additional conditions have been observed in the literature [Albarghouthi et al. 2016; Prabhu et al. 2021] to improve the "usefulness" of synthesized specifications. It is easy to extend Algorithm 1 to CHC with additional non-emptiness constraints (only for the maximization mode): we enforce the initial solution obtained in Line 2 of Algorithm 1 to satisfy the side-conditions, and the remaining optimization steps preserve the conditions. OptPCSat also supports the best-effort optimization within a specified time limit: it returns the best solution found so far when the time limit is exceeded: It happens if an initial solution is obtained in Line 2 but the call to Aux does not return, repeatedly obtaining improved solutions or waiting for a return of the parallel calls to Solve. OptPCSat returns "Fail" if it fails to obtain an initial solution within the time limit.

We tested OptPCSat on (1) the benchmark set consisting of 65 CHC maximization problems, originally introduced to evaluate HornSpec [Prabhu et al. 2021], and (2) a new benchmark set consisting of 10 CHC optimization problems to further differentiate our method with related ones compared in Section 2. The set includes 7 new CHC maximization problems that work as counterexamples against the soundness of the maximality checking method for HornSpec: Since

[14]Following [Unno et al. 2021], we actually use more complicated templates, supporting piecewise linear ranking functions and lexicographic ordering.

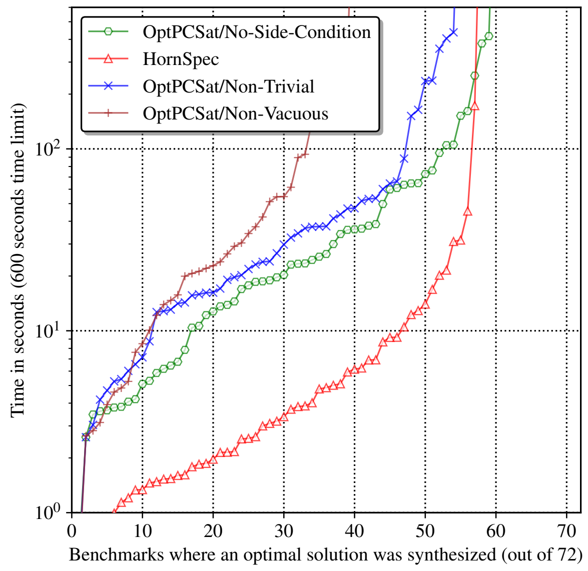

Fig. 5. Cactus plot for the experiments on 72=(65+7) CHC maximization problems

it (incorrectly) checks for maximality whether any one-point extension of the current solution violates the given CHC constraints, to fool HornSpec, each of the 7 problems is designed to have at least one non-maximal solution whose any one-point extension becomes a non-solution. The benchmark set also contains the CHC optimization problem obtained from the running program in Section 2 that cannot in principle be solved by the maximal specification synthesis method based on repeated multi-abduction [Albarghouthi et al. 2016]. Other 2 benchmarks involve both maximization and minimization, and hence beyond the scope of the existing methods.

The experiments were conducted on Amazon Linux 2, Intel(R) Xeon(R) Platinum 8259CL CPU @ 2.50GHz CPU and 32 GiB RAM. The time limit was 600 seconds. The experiment results on CHC maximization benchmarks are summarized as the cactus plot in Figure 5. Out of 72 (=65 + 7) benchmarks, OptPCSat obtained 59 optimal solutions and 12 (possibly non-optimal) solutions in the no-side-condition mode. In the non-trivial mode, our tool obtained 54 optimal solutions and 17 (possibly non-optimal) solutions. In the non-vacuous mode, OptPCSat obtained 39 optimal solutions and 26 (possibly non-optimal) solutions. We observed that OptPCSat "failed" more with the additional non-vacuity and non-triviality conditions because they added an extra overhead on Solve (Lines 2,12). We also found that the performance decrease was mainly due to the strategy adopted for synthesizing non-empty predicates, which is again orthogonal to our optimality checking method. Overall the experimental results show that our optimization method can indeed find optimal solutions for non-trivial CHC optimization problems, despite the theoretical hardness of the class of problems. It is also worth mentioning here that initial solutions found by PCSat were often not optimal and required refinements: the median, minimum, and maximum values of the numbers of refinements in the no-side-condition mode are 2, 0, and 19. The percentage of the optimal solutions that never required refinement is 24%. The same numbers in the non-trivial mode are 2, 0, 17, and 14%. In the non-vacuous mode, the numbers are 2, 0, 19, and 25%.

The cactus plot also compares OptPCSat and HornSpec. HornSpec returned solutions for 57 (out of 72) instances. The plot shows that HornSpec is generally faster than OptPCSat as expected: note here that we cannot directly compare the results of OptPCSat and HornSpec since HornSpec does not construct a certificate of the optimality of the returned solution, which is in stark contrast to OptPCSat that consumes time for constructing such a certificate in the form of ordinary, well-founded, functional, and non-empty predicates.

We then have applied OptPCSat to check the optimality of the 57 solutions returned by HornSpec. OptPCSat has successfully shown that 47 solutions are optimal and 2 solutions are non-optimal. The optimality of the other 8 solutions is unknown. The non-optimal solutions returned by HornSpec are for 2 problem instances from the new 7 CHC maximization problems that are designed to work as counterexamples for HornSpec. That said, HornSpec efficiently found maximal solutions (but without a certificate) for at least 47 (out of 65) problems from the original benchmark set for HornSpec. It is therefore an interesting future work to combine the approaches of OptPCSat and HornSpec to complement each other.

We conclude this section by discussing the results on the new benchmark set designed to differentiate OptPCSat from the other related tools. For the new 7 CHC maximization problems, OptPCSat obtained maximal solutions for 6 problems. The problem instance that OptPCSat was not able to solve required synthesis of a predicate that involves integer modulo operators that did not appear in the original constraint set. By manually providing the predicate, OptPCSat solved the instance, Thus, the failure is not due to a limitation of our optimality checking method but the predicate synthesis engine of the backend solver PCSat. PCSat successfully solved the other 3 problems: 1 problem that encodes the verification example discussed in Section 1 and 2 problems that involve both maximization and minimization.

8 RELATED WORK

Sections 2 to 4 have discussed subclasses of CHC optimization and existing optimization methods [Albarghouthi et al. 2016; Hashimoto and Unno 2015; Prabhu et al. 2021; Zhou et al. 2021] in details. This section compares our work with these work from different perspectives and with other related work.

Optimality beyond Linear-Time Safety. For modular verification of open programs, the usefulness of *angelic* non-determinism has been observed by various researchers in the context of angelic verification [Blackshear and Lahiri 2013; Das et al. 2015; Lahiri et al. 2020], necessary precondition inference [Cousot et al. 2013], and refinement type optimization [Hashimoto and Unno 2015]. These methods however do not guarantee the semantic optimality of synthesized specifications. By incorporating their ideas, our approach could be extended, for example, to synthesize a maximally-weak precondition for the non-termination of the given program and to synthesize maximal specifications of blackbox library functions under an *angelic* interpretation of non-determinism exhibited by the target program. To this end, our approach needs to be extended to optimization of pCSP (resp. CHCs with functionality constraints) for finitely (resp. infinitely) branching non-determinism. [Hashimoto and Unno 2015] has also discussed synthesis of syntactically-optimal liveness specifications. We expect that synthesis of liveness specifications with semantic optimality is within our reach by extending our approach to optimization of CHCs with well-foundedness constraints.

Existentially quantified CHCs vs. CHCs with non-emptiness. Hashimoto and Unno [2015] reduces non-optimality checking to satisfiability checking of existentially quantified CHCs [Beyene et al. 2013]. By contrast, we reduce non-optimality to CHCs with non-emptiness constraints, which is exact (i.e. Σ_2^0-complete), while existentially quantified CHCs is hard. This suggests that focusing on CHCs with non-emptiness may be beneficial in practice.

Multi-Abduction vs. CHC Optimization. Zhou et al. [2021] reduces a maximal specification synthesis problem of library functions invoked from the given recursive function with a postcondition ϕ to a multi-abduction problem instead of CHC optimization. Their method thus amounts to synthesizing maximally-weak postconditions of the library functions that make the postcondition ϕ to be an *inductive invariant*. By contrast we properly reduce maximal specification synthesis to CHC optimization.

Prabhu et al. [2021] proposed a method for CHC maximization. We have mentioned the incorrectness of their maximality checking algorithm, and here we discuss other aspects of their work. Their paper proposed a notion of *non-vacuous* solutions, as a formal criterion for a "practically useful" solution, and developed heuristic methods for efficiently constructing non-vacuous and weak (i.e. close to maximal) solutions. These contributions, which are independent of the flaw in the maximality checking algorithm, are actually complementary to our development: The combination of their efficient candidate solution construction method and our correct maximality checker would provide us with an efficient CHC optimization tool with maximality guarantee. Although their maximality checking algorithm is incorrect, the experimental results show that their tool often returns a maximal solution (of which maximality is automatically provable by our tool) at least for their benchmark set.

9 CONCLUSION

We presented a computational theoretic analysis of CHC optimization and its subclasses, and based on the observations, developed a new CHC optimization method that addressed the unsoundness and the incompleteness of existing predicate constraint optimization methods: our method can soundly and completely guarantee the semantic optimality of synthesized specifications with respect to the user-specified preference order via automated termination proofs. Besides the theoretical developments, we worked toward a practical application to optimal specification synthesis and obtained promising experimental results. Interesting future directions include an extension to pfwnCSP optimization, which enables optimal synthesis of not only linear-time safety specifications but also branching-time and/or liveness specifications of non-deterministic programs.

ACKNOWLEDGMENTS

We thank anonymous reviewers for useful comments. This work was supported by JSPS KAKENHI Grant Numbers JP20H05703, JP22H03564, JP20H04162, JP22H03570 and JP19H04084.

REFERENCES

Aws Albarghouthi, Isil Dillig, and Arie Gurfinkel. 2016. Maximal Specification Synthesis. In *POPL '16* (St. Petersburg, FL, USA). ACM, 789–801.

Christophe Alias, Alain Darte, Paul Feautrier, and Laure Gonnord. 2010. Multi-dimensional Rankings, Program Termination, and Complexity Bounds of Flowchart Programs. In *SAS '10* (Perpignan, France). Springer, 117–133.

Amir M. Ben-Amram and Samir Genaim. 2014. Ranking Functions for Linear-Constraint Loops. *J. ACM* 61, 4, Article 26 (July 2014), 55 pages.

Amir M. Ben-Amram and Samir Genaim. 2017. On Multiphase-Linear Ranking Functions. In *CAV '17*. Springer, 601–620.

Tewodros Beyene, Swarat Chaudhuri, Corneliu Popeea, and Andrey Rybalchenko. 2014. A Constraint-based Approach to Solving Games on Infinite Graphs. In *POPL '14* (San Diego, California, USA). ACM, 221–233.

Tewodros A. Beyene, Corneliu Popeea, and Andrey Rybalchenko. 2013. Solving Existentially Quantified Horn Clauses. In *CAV '13 (LNCS, Vol. 8044)*. Springer, 869–882.

Sam Blackshear and Shuvendu K. Lahiri. 2013. Almost-Correct Specifications: A Modular Semantic Framework for Assigning Confidence to Warnings. In *PLDI '13* (Seattle, Washington, USA) *(PLDI '13)*. ACM, 209–218.

Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. 2005. Linear Ranking with Reachability. In *CAV '05 (LNCS, Vol. 3576)*. Springer, 491–504.

Byron Cook, Andreas Podelski, and Andrey Rybalchenko. 2006. Termination proofs for systems code. In *PLDI '06*. ACM, 415–426.

Patrick Cousot, Radhia Cousot, Manuel Fähndrich, and Francesco Logozzo. 2013. Automatic Inference of Necessary Preconditions. In *VMCAI '13*. Springer, 128–148.

Ankush Das, Shuvendu K. Lahiri, Akash Lal, and Yi Li. 2015. Angelic Verification: Precise Verification Modulo Unknowns. In *CAV '15*. Springer, 324–342.

Grigory Fedyukovich, Yueling Zhang, and Aarti Gupta. 2018. Syntax-Guided Termination Analysis. In *CAV '18 (LNCS, Vol. 10981)*. Springer, 124–143.

Juergen Giesl, Cornelius Aschermann, Marc Brockschmidt, Fabian Emmes, Florian Frohn, Carsten Fuhs, Jera Hensel, Carsten Otto, Martin Pluecker, Peter Schneider-Kamp, Thomas Stroeder, Stephanie Swiderski, and Rene Thiemann. 2017. Analyzing Program Termination and Complexity Automatically with AProVE. *Journal of Automated Reasoning* 58 (2017), 3–31.

Laure Gonnord, David Monniaux, and Gabriel Radanne. 2015. Synthesis of Ranking Functions Using Extremal Counterexamples. In *PLDI '15* (Portland, OR, USA). ACM, 608–618.

Kodai Hashimoto and Hiroshi Unno. 2015. Refinement Type Inference via Horn Constraint Optimization. In *SAS '15 (LNCS, Vol. 9291)*. Springer, 199–216.

Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski. 2014. Termination Analysis by Learning Terminating Programs. In *CAV '14*. Springer, 797–813.

Satoshi Kura, Hiroshi Unno, and Ichiro Hasuo. 2021. Decision Tree Learning in CEGIS-Based Termination Analysis. In *CAV '21*. Springer, 75–98.

Takuya Kuwahara, Tachio Terauchi, Hiroshi Unno, and Naoki Kobayashi. 2014. Automatic Termination Verification for Higher-Order Functional Programs. In *ESOP '14 (LNCS, Vol. 8410)*. Springer, 392–411.

Shuvendu K. Lahiri, Akash Lal, Sridhar Gopinath, Alexander Nutz, Vladimir Levin, Rahul Kumar, Nate Deisinger, Jakob Lichtenberg, and Chetan Bansal. 2020. Angelic Checking within Static Driver Verifier: Towards high-precision defects without (modeling) cost. In *FMCAD '20*. IEEE, 169–178.

Chin Soon Lee, Neil D. Jones, and Amir M. Ben-Amram. 2001. The size-change principle for program termination. In *POPL '01*. ACM, 81–92.

Jan Leike and Matthias Heizmann. 2014. Ranking Templates for Linear Loops. In *TACAS '14 (LNCS, Vol. 8413)*. Springer, 172–186.

Saswat Padhi, Rahul Sharma, and Todd D. Millstein. 2016. Data-Driven Precondition Inference with Learned Features. In *PLDI '16*. 42–56.

Andreas Podelski and Andrey Rybalchenko. 2004. A Complete Method for the Synthesis of Linear Ranking Functions. In *VMCAI '04 (LNCS, Vol. 2937)*. Springer, 239–251.

Sumanth Prabhu, Grigory Fedyukovich, Kumar Madhukar, and Deepak D'Souza. 2021. Specification Synthesis with Constrained Horn Clauses. In *PLDI '21* (Virtual, Canada). ACM, 1203–1217.

Sriram Sankaranarayanan, Swarat Chaudhuri, Franjo Ivančić, and Aarti Gupta. 2008. Dynamic Inference of Likely Data Preconditions over Predicates by Tree Learning. In *ISSTA '08* (Seattle, WA, USA) *(ISSTA '08)*. ACM, 295–306.

Yuki Satake, Hiroshi Unno, and Hinata Yanagi. 2020. Probabilistic Inference for Predicate Constraint Satisfaction. *AAAI '20* 34, 02 (Apr. 2020), 1644–1651.

Mohamed Nassim Seghir and Daniel Kroening. 2013. Counterexample-Guided Precondition Inference. In *ESOP '13*. Springer, 451–471.

Raymond M. Smullyan. 1968. *First-order logic*. Springer. xii+158 pages.

Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. 2006. Combinatorial Sketching for Finite Programs. In *ASPLOS XII* (San Jose, California, USA). ACM, 404–415.

Saurabh Srivastava and Sumit Gulwani. 2009. Program verification using templates over predicate abstraction. In *PLDI '09* (Dublin, Ireland). ACM, 223–234.

Hiroshi Unno, Tachio Terauchi, and Eric Koskinen. 2021. Constraint-Based Relational Verification. In *CAV '21*. Springer, 742–766.

Caterina Urban. 2013. The Abstract Domain of Segmented Ranking Functions. In *SAS '13 (LNCS, Vol. 7935)*. Springer, 43–62.

Caterina Urban, Arie Gurfinkel, and Temesghen Kahsai. 2016. Synthesizing Ranking Functions from Bits and Pieces. In *TACAS '16*. Springer, 54–70.

Caterina Urban and Antoine Miné. 2014. An Abstract Domain to Infer Ordinal-Valued Ranking Functions. In *ESOP '14*. Springer, 412–431.

Zhe Zhou, Robert Dickerson, Benjamin Delaware, and Suresh Jagannathan. 2021. Data-Driven Abductive Inference of Library Specifications. *Proceedings of the ACM on Programming Languages* 5, OOPSLA, Article 116 (Oct. 2021), 29 pages.

Received 2022-07-07; accepted 2022-11-07