Solving Constrained Horn Clauses Using Dependence-Disjoint Expansions

Qi Zhou, David Heath, and William Harris

Recursion-free Constrained Horn Clauses (CHCs) are logic-programming problems that can model safety properties of programs with bounded iteration and recursion. In addition, many CHC solvers reduce recursive systems to a series of recursion-free CHC systems that can each be solved efficiently.

In this paper, we define a novel class of recursion-free systems, named *Clause-Dependence Disjoint* (CDD), that generalizes classes defined in previous work. The advantage of this class is that many CDD systems are smaller than systems which express the same constraints but are part of a different class. This advantage in size allows CDD systems to be solved more efficiently than their counterparts in other classes. We implemented a CHC solver named SHARA. SHARA solves arbitrary CHC systems by reducing the input to a series of CDD systems. Our evaluation indicates that SHARA outperforms state-of-the-art implementations in many practical cases.

1 Introduction

Many critical problems in program verification can be reduced to solving systems of Constrained Horn Clauses (CHCs), a class of logic-programming problems [4, 6, 20, 21]. A CHC is a logical implication with the following form:

$$R_1(\vec{v_1}) \leftarrow R_2(\vec{v_2}) \wedge R_3(\vec{v_3}) \wedge \ldots \wedge \varphi(\vec{v_0}, \vec{v_1}, \vec{v_2}, \vec{v_3}, \ldots)$$

Here, the left side of the implication, called the head, contains an uninterpreted relational predicate applied to a vector of variables. The right side has any number of such predicates conjoined together with a *constraint* (φ). The constraint is a logical formula in a background theory and may use variables named by the predicates. A CHC system is a set of CHCs. The goal of the CHC solving problem is to find suitable interpretations for each predicate such that each CHC is logically consistent in isolation.

In this work we focus on the subclass of CHC systems which are known as *recursion-free*. In a recursion-free CHC system, no derivation of a predicate will invoke that predicate. Less formally, a recursion-free CHC system is one where following implication arrows through the system will never reach the same clause twice. Recursion-free CHC systems are an important subclass for two reasons. First, recursion-free systems can be used to model safety properties for hierarchical programs [15, 16] (programs with only bounded iteration and recursion). Second and most importantly, a well-known approach for solving a general CHC system reduces the input problem to solving a sequence of recursion-free systems. Such approaches attempt to synthesize a solution for the original system from the solutions of recursion-free systems [4]. The performance of such solvers relies heavily on the performance of solving recursion-free CHC systems.

Typically, even recursion-free CHC systems are not solved directly. Instead, they are reduced to a more specific subclass of recursion-free CHC system. These classes include those of *body-disjoint* (or *derivation tree*) systems [4, 11, 19–21] and of *linear* systems [1]. We will discuss these classes in §2 and §6. Such classes can be solved by issuing *interpolation queries* to find suitable definitions for the uninterpreted predicates.

Submitted to:
HCVS 2018

In general, solving a recursion-free CHC system for propositional logic and the theory of linear integer arithmetic is co-NEXPTIME-complete [21]. In contrast, solving a linear system or body-disjoint system with the same logic and theories is in co-NP [21]. We refer to such classes that are solvable in co-NP time as *directly solvable*. Because solving an arbitrary recursion-free system is harder than solving a directly solvable system, solvers which reduce to directly solvable systems are highly reliant on the size of the reductions.

The first contribution of this paper is the introduction of a novel class of directly solvable systems that we refer to as *Clause-Dependence Disjoint* (CDD). The formal definition of CDD is given at Defn. 5. CDD is a strict superset of the union of previously introduced classes of directly solvable systems. The key characteristic of this class is that when an arbitrary recursion-free system is reduced to a CDD system and to a system from a different directly solvable class, the CDD system is frequently the smaller of the two. Therefore, solving recursion-free systems by reducing them to CDD form is often less computationally expensive than reducing them to a system in a different class.

The second contribution of this paper is a solver for CHC systems, named SHARA. Given a recursion-free system S, SHARA reduces the problem of solving S to solving a CDD system S'. In the worst case, it is possible that the size of S' may be exponential in the size of S. However, empirically we have found that the size of S' is usually close enough to the size of S that SHARA frequently outperforms DUALITY, one of the best known CHC solvers. The procedure implemented in SHARA is a generalization of existing techniques that synthesize compact verification conditions for hierarchical programs [7, 15]. Given a general (possibly recursive) CHC system, SHARA solves a sequence of recursion-free systems. Each subsystem is a bounded unwinding of the original system. SHARA attempts to combine the solutions of these recursion-free systems to synthesize a solution to the original problem, as has been proposed in previous work [21].

We implemented SHARA within the DUALITY CHC solver [4], which is implemented within the Z3 automatic theorem prover [5]. We evaluated the effectiveness of SHARA on standard benchmarks drawn from SVCOMP15 [8]. The results indicate that SHARA outperforms modern solvers many cases. Futhermore, the results indicate that combining the strengths of SHARA with that of other existing approaches (as discussed in §5) is a promising direction for the future of CHC solving.

The rest of this paper is organized as follows. §2 illustrates the operation of SHARA on a recursion-free CHC system. §3 reviews technical work on which SHARA is based. §4 describes SHARA in technical detail. §5 gives the results of our empirical evaluation of SHARA. §6 compares SHARA to related work.

2 Overview

In §2.1, we describe a recursion-free CHC system, $\mathtt{S_{DA}}$ (Figure 2), that models the safety of the program `dblAbs` (Figure 1). In §2.2, we show that $\mathtt{S_{DA}}$ is a CDD system and how SHARA can solve it by encoding it into binary interpolants. In §2.3, we illustrate that $\mathtt{S_{DA}}$ is not in directly solvable classes introduced in previous work.

2.1 Verifying `dblAbs`: an example hierarchical program

`dblAbs` is a procedure that doubles the absolute value of its input and stores the result in `res`. The program also asserts that `res` is greater than or equal to 0 before exiting. Verifying this assertion reduces to solving a recursion-free CHC system over a set of uninterpreted predicates that represent the control locations in `dblAbs`. In particular, one such system, $\mathtt{S_{DA}}$, is shown in Figure 2. While $\mathtt{S_{DA}}$ has been presented as the

```
1  def dbl(int x)
2    return 2 * x
3  def main(n)
4    abs = 0
5    if (n >= 0)
6      abs = n
7    else
8      abs = -n
9    res = dbl(abs)
10   assert(res >= 0)
```

Figure 1: `dblAbs`: an example hierarchical program.

$$\texttt{dbl}(\texttt{x},\texttt{d}) \leftarrow \texttt{d} = 2 * \texttt{x} \quad (1)$$
$$\texttt{L}_4(\texttt{n},\texttt{abs}) \leftarrow \texttt{abs} = 0 \quad (2)$$
$$\texttt{L}_6(\texttt{n},\texttt{abs}) \leftarrow \texttt{L}_4(\texttt{n},\texttt{abs}) \wedge \texttt{n} \geq 0 \quad (3)$$
$$\texttt{L}_8(\texttt{n},\texttt{abs}) \leftarrow \texttt{L}_4(\texttt{n},\texttt{abs}) \wedge \texttt{n} < 0 \quad (4)$$
$$\texttt{L}_9(\texttt{n},\texttt{abs'}) \leftarrow \texttt{L}_6(\texttt{n},\texttt{abs}) \wedge \texttt{abs'} = \texttt{n} \quad (5)$$
$$\texttt{L}_9(\texttt{n},\texttt{abs'}) \leftarrow \texttt{L}_8(\texttt{n},\texttt{abs}) \wedge \texttt{abs'} = -\texttt{n} \quad (6)$$
$$\texttt{main}(\texttt{n},\texttt{res}) \leftarrow \texttt{L}_9(\texttt{n},\texttt{abs'}) \wedge \texttt{dbl}(\texttt{x},\texttt{d}) \wedge \texttt{abs'} = \texttt{x} \wedge \texttt{res} = \texttt{d} \quad (7)$$
$$\mathit{False} \leftarrow \texttt{main}(\texttt{n},\texttt{res}) \wedge \texttt{res} < 0 \quad (8)$$

Figure 2: A CHC system that models the safety condition of `dblAbs`, named $\texttt{S}_{\texttt{DA}}$.

result of a translation from `dblAbs`, SHARA is purely a solver for CHC systems: it does not require access to the concrete representation of a program, or for a given CHC system to be the result of translation from a program at all.

2.2 $\texttt{S}_{\texttt{DA}}$ as a Clause-Dependence Disjoint System

The recursion-free CHC system $\texttt{S}_{\texttt{DA}}$ is a *Clause-Dependence Disjoint* (CDD) system. A CHC system can be classified as CDD when each clause satisfies the following rules: **(1)** no two predicates in the body of the same clause share any transitive dependencies on other predicates and **(2)** no clause has more than one occurrence of a given predicate in the body. As an example, clause (7) is dependence disjoint. Two predicates $\texttt{L}_9$ and `dbl` are in its body. The transitive dependency of $\texttt{L}_9$ is the set $\{\texttt{L}_4, \texttt{L}_6, \texttt{L}_8\}$ while the transitive dependency of `dbl` is the empty set. Therefore, their transitive dependencies are disjoint: $\{\texttt{L}_4, \texttt{L}_6, \texttt{L}_8\} \cap \varnothing = \varnothing$. All other clauses in $\texttt{S}_{\texttt{DA}}$ have at most one uninterpreted predicate in the body, so they are trivially disjoint dependent. Therefore $\texttt{S}_{\texttt{DA}}$ is a CDD system. The formal definition of CDD and its key properties are given in §4.1. The formal definition of transitive dependency is given in §3.1.

SHARA solves CDD systems directly by issuing a binary interpolation query for each uninterpreted predicate in topological order. Each interpretation of a predicate P can be computed by interpolating **(1)** the *pre*-formula, constructed from clauses where P is the head, and **(2)** the *post*-formula, constructed from all clauses where the head transitively depends on P.

For example, consider $\texttt{L}_9$. By the time SHARA attempts to synthesize an interpretation for $\texttt{L}_9$ it will have solutions for $\texttt{L}_4$, $\texttt{L}_6$, $\texttt{L}_8$. Possible interpretations of these predicates are shown in Figure 3. The pre-formula is constructed from the bodies of clauses where $\texttt{L}_9$ is the head. Each relational predicate, P, is replaced by a corresponding boolean indicator variable, $\texttt{b}_P$. Each boolean indicator variable implies the solution for its predicate, encoded as the disjunction of the negation of the boolean indicator variable and the solution. In particular, the pre-formula for $\texttt{L}_9$ is constructed from clauses (5) and (6):

$$((\texttt{b}_{\texttt{L}_6} \wedge \texttt{abs'} = \texttt{n}) \vee (\texttt{b}_{\texttt{L}_8} \wedge \texttt{abs'} = -\texttt{n})) \wedge (\neg \texttt{b}_{\texttt{L}_6} \vee \texttt{n} \geq 0) \wedge (\neg \texttt{b}_{\texttt{L}_8} \vee \texttt{n} < 0) \quad (9)$$

The post-formula is constructed from clauses that transitively depend on $\texttt{L}_9$. Again, we replace relational predicates by corresponding boolean indicators. However, we omit the boolean indicator for $\texttt{L}_9$. The post-formula is composed from clauses (1), (7), and (8):

$$(\neg \texttt{b}_{\texttt{dbl}} \vee \texttt{d} = 2 * \texttt{x}) \wedge (\neg \texttt{b}_{\texttt{main}} \vee (\texttt{b}_{\texttt{dbl}} \wedge \texttt{abs'} = \texttt{x} \wedge \texttt{res} = \texttt{d})) \wedge (\texttt{b}_{\texttt{main}} \wedge \texttt{res} < 0) \quad (10)$$

Interpolating the pre and post formulas yields an interpretation of L_9: `abs'` ≥ 0. The procedure for solving a CDD system is described in formal detail in §4.2.

2.3 S_{DA} is not in other recursion-free classes

In this section, we show that S_{DA} is not in other known classes of recursion-free CHC systems. Specifically, we will discuss body-disjoint systems and linear systems.

Body-disjoint (or derivation tree) systems [4, 11, 19–21] are a class of recursion-free CHC system where each uninterpreted predicate appears in the body of at most one clause and appears in such a clause exactly once. Such systems cannot model a program with multiple control paths that share a common prefix, typically modeled as a CHC system with an uninterpreted predicate that occurs in the body of multiple clauses. S_{DA} is not a body-disjoint system because L_4 appears in the body of both clause (3) and clause (4). In order to handle S_{DA}, a solver that uses body-disjoint systems would have to duplicate L_4. Worse, if L_4 had dependencies, then each dependency would also need to be duplicated.

Previous work has also introduced the class of linear systems [1], where the body of each clause has at most one uninterpreted predicate. However, such systems cannot directly model the control flow of a program that contains procedure calls. S_{DA} is not a linear system because the body of clause (7) has two predicates, `L9` and `dbl`. CHC solvers that use linear systems effectively inline the constraints for relational predicates that occur in non-linear clauses [2]. In the case of S_{DA}, inlining the constraints of `dbl` is efficient, but in general such approaches can generate systems that are exponentially larger than the input. For example, if the procedure `dbl` were called more than once in `dblAbs` then multiple copies of the body of the procedure would be inlined. And if the body of this procedure were large, the inlining could become prohibitively expensive.

The hierarchy of discussed classes of recursion-free systems is depicted in Figure 4. As shown, the class of CDD systems is a superset of both the class of body-disjoint systems and the class of linear systems. So any recursion-free system that is efficiently expressible in body-disjoint or linear form is also efficiently expressible in CDD form. In addition, some systems which are expensive to express in body-disjoint or linear form are efficiently expressible in CDD form. SHARA takes advantage of this fact when solving input systems. Given an arbitrary recursion-free CHC system S, SHARA reduces S to a CDD system S' and solves S' directly. In general, S' may have size exponential in the size of S. However, SHARA generates CDD systems via heuristics analogous to those used to generate compact verification conditions of hierarchical programs [7, 15]. In practice these heuristics often yield CDD systems which are small with respect to the input system. A general procedure for constructing a CDD expansion of a given CHC system is given in Appendix A.

3 Background

3.1 Constrained Horn Clauses

3.1.1 Structure

A Constrained Horn Clause is a logical implication where the antecedent is called the body and the consequent is called the head. The body is a conjunction of a logical formula, called the constraint, and a vector of uninterpreted predicates. The constraint is an arbitrary formula in some background logic, such as linear integer arithmetic. The uninterpreted predicates are applied to variables which may or may not appear in the constraint. A head can be either an uninterpreted predicate applied to variables or *False*. A

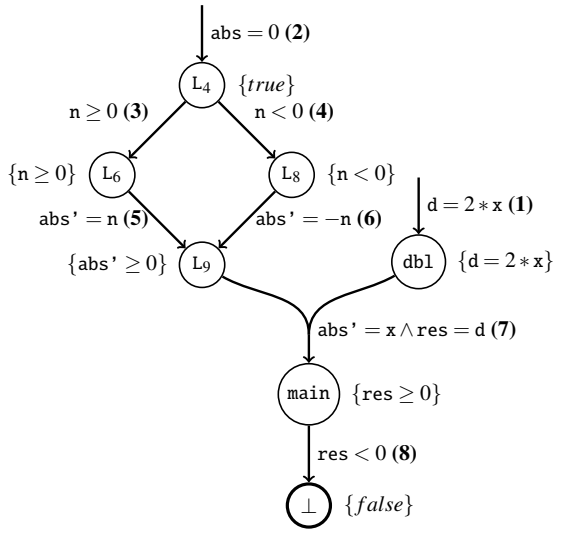

Figure 3: S_{DA} as a directed hypergraph. Each relational predicate is depicted as a graph node while each clause is represented by a hyperedge. Each hyperedge is labeled by the constraint in the corresponding CHC. Each node has a valid corresponding interpretation, written in braces.

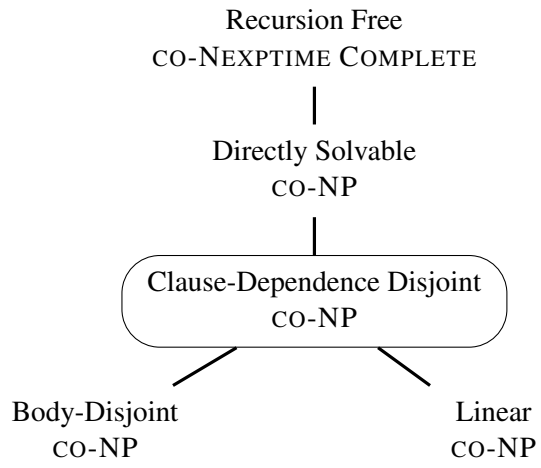

Figure 4: The hierarchy of classes of recursion-free CHC systems. Body Disjoint and Linear systems are subsumed by CDD systems. Solving Directly Solvable CHC systems is in co-NP while solving general, recursion-free systems is co-NEXPTIME Complete.

clause where the head is *False* is called a query. A CHC can be defined structurally:

$$
\begin{aligned}
\texttt{chc} &::= \texttt{head} \leftarrow \texttt{body} \\
\texttt{head} &::= \mathit{False} \\
&\quad\;\; \mid \texttt{pred} \\
\texttt{body} &::= \varphi \wedge \texttt{preds} \\
\texttt{preds} &::= \mathit{True} \\
&\quad\;\; \mid \texttt{pred} \wedge \texttt{preds} \\
\texttt{pred} &::= \textit{an uninterpreted predicate applied to variables} \\
\varphi &::= \textit{a formula}
\end{aligned}
$$

For a given CHC C, $\mathsf{Body}(C)$ denotes the vector of uninterpreted predicates in the body and $\mathsf{Constraint}(C)$ denotes the constraint in the body. If C is not a query, then $\mathsf{Head}(C)$ denotes the uninterpreted predicate in the head. A CHC system is a set of CHCs where exactly one clause is a query. For a given CHC system S, $\mathsf{Pred}(S)$ denotes the set of all uninterpreted predicates and `query` denotes the body of the query clause.

To explain the structure of a CDD system, we need terminology that relates predicates in a CHC system including the terms *predicate dependency*, *transitive predicate dependency*, and *sibling*.

Definition 1. *Given a CHC system* S *and two uninterpreted predicates* P *and* $Q \in \mathsf{Pred}(S)$*, if* $\exists C \in S$ *such that* $P = \mathsf{Head}(C)$ *and* $Q \in \mathsf{Body}(C)$*, then* Q *is a* predicate dependency *of* P*.*

Example 1. *In* S_{DA}*, because* L_4 *is in the body of clause (4) and* L_8 *is the head of clause (4),* L_4 *is a predicate dependency of* L_8*.*

Given a CHC system S and an uninterpreted predicate P, $\mathsf{Deps}(P)$ denotes the set of all predicate dependencies of P in S.

Definition 2. *Given a CHC system S and three uninterpreted predicates P, Q, and R* $\in \mathsf{Pred}(S)$*, if Q* $\in \mathsf{Deps}(P)$ *then Q is a* transitive predicate dependency *of P. If Q is a transitive predicate dependency of P and R is a transitive predicate dependency of Q, then R is a* transitive predicate dependency *of P.*

Example 2. *In* S_{DA}*, because* L_4 *is a predicate dependency of* L_8*,* L_4 *is a transitive predicate dependency of* L_8*. And because* L_8 *is an transitive predicate dependency of* L_9*,* L_4 *is a transitive predicate dependency of* L_9*.*

Given a CHC system S and an uninterpreted predicate P, $\mathsf{TrDeps}(P)$ denotes the set of all transitive predicate dependencies of P in S.

Definition 3. *Given a CHC system S and two uninterpreted predicates P and* $Q \in \mathsf{Pred}(S)$*, if* $\exists C \in S$ *such that* $P \in \mathsf{Body}(C)$ *and* $Q \in \mathsf{Body}(C)$*, then Q and P are siblings.*

Example 3. *Because uninterpreted predicates* L_9 *and* dbl *both appear in the body of clause (7),* L_9 *and* dbl *are siblings.*

For a given CHC system S, if there is no uninterpreted predicate $P \in \mathsf{Pred}(S)$ such that $P \in \mathsf{TrDeps}(P)$, then S is a *recursion-free* CHCs system.

A solution to a CHC system S is a map from each predicate $P \in \mathsf{Pred}(S)$ to its corresponding interpretation which is a formula. For a solution to be valid, each clause in S must be valid after substituting each predicate by its interpretation.

3.2 Logical interpolation

All logical objects in this paper are defined over a fixed space of first-order variables, X. For a theory T, the space of T formulas over X is denoted $\mathsf{Forms}(T)$. For each formula $\varphi \in \mathsf{Forms}(T)$, the set of free variables that occur in φ (i.e., the *vocabulary* of φ) is denoted $\mathsf{Vocab}(\varphi)$. For formulas $\varphi_0, \ldots, \varphi_n, \varphi \in \mathsf{Forms}(T)$, the fact that $\varphi_0, \ldots, \varphi_n$ *entail* φ is denoted $\varphi_0, \ldots, \varphi_n \models \varphi$.

An interpolant of a pair of mutually inconsistent formulas φ_0 and φ_1 in $\mathsf{Forms}(T)$ is a formula I in $\mathsf{Forms}(T)$ over their common vocabulary that explains their inconsistency.

Definition 4. *For* $\varphi_0, \varphi_1, I \in \mathsf{Forms}(T)$*, if (1)* $\varphi_0 \models I$*, (2)* $I \wedge \varphi_1 \models \mathsf{False}$*, and (3)* $\mathsf{Vocab}(I) \subseteq \mathsf{Vocab}(\varphi_0) \cap \mathsf{Vocab}(\varphi_1)$*, then I is an* interpolant *of* φ_0 *and* φ_1*.*

For the remainder of this paper, all spaces of formulas will be defined for a fixed, arbitrary theory T that supports interpolation, such as the theory of linear arithmetic. Although determining the satisfiability of formulas in such theories is NP-complete in general, decision procedures [5] and interpolating theorem provers [17] for such theories have been proposed that operate on such formulas efficiently.

We define SHARA in terms of an abstract interpolating theorem prover for T named ITP. Given two formulas φ_0 and φ_1, if φ_0 and φ_1 are mutually inconsistent, ITP returns the interpolant of φ_0 and φ_1. Otherwise, ITP returns None.

4 Technical Approach

This section presents the technical details of our approach. §4.1 presents the class of Clause-Dependence Disjoint systems and its key properties. §4.2 describes how SHARA solves CDD systems directly. §4.3 describes how SHARA solves a given recursion-free system by solving an CDD system. Proofs of all theorems stated in this section are in the appendix.

Input : A CDD System S.
Output : If S is solvable, then a solution of S; otherwise, the value None.

```
1  Procedure SolveCDD(S)
2      σ := ∅
3      Preds := TopologicalSort(Pred(S))
4      for P ∈ Preds do
5          interpolant := Itp(Pre(P, σ), Post(P, σ))
6          switch interpolant do
7              case SAT: do return None ;
8              case I: do σ [P] := I ;
9          end
10     end
11     return σ
```

Algorithm 1: SolveCDD: for a CDD system S, returns a solution to S or the value None to denote that S has no solution.

4.1 Clause-Dependence Disjoint Systems

The key contribution of our work is the introduction of the class of Clause-Dependence Disjoint (CDD) CHC systems:

Definition 5. *For a given recursion-free CHC system S, if for all sibling pairs, $P, Q \in \mathsf{Pred}(S)$, the transitive dependencies of P and Q are disjoint ($\mathsf{TrDeps}(P) \cap \mathsf{TrDeps}(Q) = \emptyset$) and no predicate shows more than once in the body of a single clause, then S is* Clause-Dependence Disjoint (CDD).

CDD systems model hierarchical programs with branches and procedure calls such that each execution path invokes each statement at most once.

Example 4. *The CHC system S_{DA} is a CDD system. An argument is given in §2.2.*

As discussed in §2, CDD is a superset of the union of the class of body-disjoint systems and the class of linear systems. For a given recursion-free system S, if each uninterpreted predicate $Q \in \mathsf{Pred}(S)$ appears in the body of at most one clause and no predicate appears more than once in the body of a single clause, then S is *body-disjoint* [20, 21]. If the body of each clause in S contains at most one relational predicate, then S is *linear* [1].

Theorem 1. *The class of CDD systems is a strict superset of the union of the class of body-disjoint systems and the class of linear systems.*

Proof is given in Appendix B.

4.2 Solving a CDD system

Alg. 1 presents SolveCDD, a procedure designed to solve CDD systems. Given a CDD system S, SolveCDD topologically sorts the uninterpreted predicates in S based on their dependency relations (line 3). Then, the algorithm calculates interpretations for each predicate in this order by invoking Itp (line 5). Itp computes a binary interpolant of the pre and post formulas of the given predicate, where these formulas are based on the current, partial solution. The pre and post formulas are computed respectively by Pre and Post, which we define in §4.2.2 and §4.2.3. It is possible that the pre and post formulas may be mutually satisifiable, in which case Itp returns SAT (line 7). In this case, SolveCDD returns

None to indicate that S is not solvable. Otherwise, SOLVECDD updates the partial solution by setting the interpretation of P to I (line 8). Once all predicates have been interpolated, SOLVECDD returns the complete solution, σ (line 11).

Example 5. *Given the CDD system* S_{DA}*,* SOLVECDD *may generate interpolation queries in any topological ordering of the dependency relations. One such ordering is* L_4*,* L_6*,* L_8*,* L_9*,* dbl*,* main*.*

Theorem 2. *Given a CDD system S over the theory of linear integer arithmetic,* SOLVECDD *either returns the solution of S or* None *in co-NP time.*

Proof is given in Appendix C.

In order to solve a CDD system, we construct efficiently sized pre and post formulas for each relational predicate and interpolate over these formulas. These pre and post formulas are built from **(1)** the *constraint* of a given predicate, which explains under what conditions the predicate holds, and **(2)** the *counterexample characterization*, which explains what condition must be true if the predicate holds.

4.2.1 Constructing constraints for predicates

In order to construct efficiently sized pre and post formulas for relational predicates, we use a method for compactly expressing the constraints on a given predicate. For a CDD system S, a predicate $P \in \mathsf{Pred}(S)$, and a partial solution σ that maps predicates to their solutions, the formula $\mathsf{Ctr}(P,\sigma)$ is a compact representation of the constraints of P. If σ does not contain P, then the constraint of P is constructed from the clauses where P is the head. When σ does contain P, $\mathsf{Ctr}(P,\sigma)$ is a lookup from σ. Each $P \in \mathsf{Pred}(S)$ has a corresponding boolean variable $\mathtt{b}_P$:

$$\mathsf{Ctr}(P,\sigma) = \begin{cases} \bigvee_{(C_i \in \mathsf{S}) \wedge (\mathsf{Head}(C_i)=P)} \left(\mathsf{Constraint}(C_i) \wedge \bigwedge_{Q \in \mathsf{Body}(C_i)} \mathtt{b}_Q \right), & \text{if } P \notin \sigma \\ \sigma[P], & \text{if } P \in \sigma \end{cases}$$

The *counterexample characterization* of P is a small extension of the compact constraint of P. It states that if P is used (meaning $\mathtt{b}_P = True$), then the constraint of P must hold:

$$\textsc{Cex}(P,\sigma) = \neg \mathtt{b}_P \vee \mathsf{Ctr}(P,\sigma)$$

Example 6. *When* SOLVECDD *solves predicate* L_9 *in* S_{DA}*, it generates a constraint based on clauses (5) and (6):*

$$\mathsf{Ctr}(L_9,\sigma) = (\mathtt{abs'} = \mathtt{n} \wedge b_{L_6}) \vee (\mathtt{abs'} = -\mathtt{n} \wedge b_{L_8})$$

The counterexample characterization for L_9 *is based on its boolean indicator and its constraint:*

$$\textsc{Cex}(L_9,\sigma) = \neg b_{L_9} \vee \mathsf{Ctr}(L_9,\sigma)$$

4.2.2 Constructing pre-formulas for predicates

$\textsc{Pre}(P,\sigma)$ denotes the pre-formula for an arbitrary predicate P with respect to the partial solution map, σ. Due to the topological ordering, when SOLVECDD attempts to solve P, the interpretations for all dependencies of P will be stored in σ. The pre-formula is built from these interpretations together with boolean indicators of the dependencies and the constraint of P:

$$\textsc{Pre}(P,\sigma) = \mathsf{Ctr}(P,\sigma) \wedge \left(\bigwedge_{Q \in \mathsf{Deps}(P)} (\neg b_Q \vee \sigma[Q]) \right)$$

Example 7. *When* SOLVECDD *solves predicate* L_9 *in* S_{DA}, σ *maps* L_6 *to* $n \geq 0$ *and* L_8 *to* $n < 0$. *The pre-formula for* L_9 *under* σ *is therefore:*

$$\mathsf{Ctr}(L_9) \wedge (\neg b_{L_6} \vee n \geq 0) \wedge (\neg b_{L_8} \vee n < 0)$$

The formula $\mathsf{Ctr}(L_9, \sigma)$ *is given in Ex. 6.*

4.2.3 Constructing post-formulas for predicates

$\textsc{Post}(P, \sigma)$ denotes the post-formula for an arbitrary predicate P with respect to the partial solution map, σ. A valid post-formula is mutually inconsistent with the solution of P. and is constructed based on the predicates which depend on P. Let D_0 be the *transitive dependents* of P in S (i.e the predicates that have P as a transitive dependency), let D_1 be the siblings in S of $(D_0 \cup P)$, let D_2 be all transitive dependencies of D_1, and let $D = D_0 \cup D_1 \cup D_2$. The post-formula for P under σ is the conjunction of counterexample characterization of all predicates $Q \in D$ and the query clause:

$$\textsc{Post}(R, \sigma) = \texttt{query} \wedge (\bigwedge_{Q \in D} \textsc{Cex}(Q, \sigma))$$

Example 8. *When* SOLVECDD *solves* L_9 *in* S_{DA}, *it must consider the dependents of* L_9. *The* transitive dependents D_0 *of* L_9 *is {*`main`*}. The siblings set* D_1 *is {*`dbl`*}. The set of transitive dependencies of* D_1 *is* $\varnothing$. *Therefore, D is {*`main,dbl`*}. The* `query` *is* $b_{main} \wedge res < 0$. *The post-formula for* L_9 *under* σ *is:*

$$\texttt{query} \wedge \textsc{Cex}(\texttt{main}, \sigma) \wedge \textsc{Cex}(\texttt{dbl}, \sigma)$$

4.3 Solving recursion-free systems using CDD systems

Given a recursion-free CHC system S, SHARA constructs a CDD system S'. SHARA then directly solves S' and, from this solution, constructs a solution for S. For two given recursion-free CHC systems S and S', if there is a homomorphism from $\mathsf{Pred}(S')$ to $\mathsf{Pred}(S)$ that preserves the relationship between the clauses of S' in the clauses of S, then S' is an *expansion* of S (all definitions in this section will be over fixed S, and S').

Definition 6. *Let* $\eta : \mathsf{Pred}(S') \to \mathsf{Pred}(S)$ *be such that* ***(1)*** *for all* $P' \in \mathsf{Pred}(S')$, P' *has the same parameters as* $\eta(P')$; ***(2)*** *for each clause* $C' \in S'$, *the clause C, constructed by substituting all predicates* P' *by* $\eta(P')$, *is in S; and* ***(3)*** *each predicate P in S has at least one predicate* P' *in* S' *such that* $\eta(P') = P$. *Then* η *is a* correspondence *from* S' *to S.*

If there is a correspondence from S' to S, then S' is an *expansion* of S, denoted $S \preceq S'$.

Definition 7. *If* S' *is CDD,* $S \preceq S'$, *and there is no CDD system* S'' *such that* $S \preceq S' \preceq S''$ *and* $S'' \neq S'$, *then* S' *is a minimal CDD expansion of S.*

SHARA (Alg. 2), given a recursion-free CHC system S (line 1), returns a solution to S or the value None to denote that S is unsolvable. SHARA first runs a procedure EXPAND on S to obtain a CDD expansion S' of S (EXPAND is given in Appendix A). SHARA then invokes SOLVECDD on S'. When SOLVECDD returns that S' has no solution, SHARA propagates None (line 4). Otherwise, SHARA constructs a solution from the CDD solution, σ', by invoking $\textsc{Collapse}(\eta, \sigma')$ (line 5). COLLAPSE is designed to convert the solution for the CDD system back to a solution for the original problem. It does this by taking the conjunction of all interpretations of predicates which correspond to the same predicate in the original problem. That is, given a CDD solution σ' and a correspondence η from $P' \in \mathsf{Pred}(S')$ to $P \in \mathsf{Pred}(S)$, $\textsc{Collapse}(\eta, \sigma')$ generates an entry in σ for each predicate in the original system: $\sigma[P] := \bigwedge_{\eta(P')=P} \sigma'(P')$.

```
Input  : A recursion-free CHC system S.
Output : A solution to S or None.
Procedure SHARA(S)
    (S', η) := EXPAND(S) ;
    switch SOLVECDD(S') do
        case None: do return None ;
        case σ': do return COLLAPSE(η, σ') ;
    end
```

Algorithm 2: SHARA: a solver for recursion-free CHCs, which uses procedures EXPAND (see Appendix A) and SOLVECDD (see §4.2).

Theorem 3. *S is solvable if and only if* SHARA *returns a solution* σ.

Proof is given in Appendix D.

5 Evaluation

We performed an empirical evaluation of SHARA to determine how it compares to existing CHC solvers. To do so, we implemented SHARA as a modification of DUALITY CHC solver, which is included in the Z3 theorem prover [9]. We modified DUALITY to use SHARA as its solver for recursion-free CHC systems. We modified the algorithm used by DUALITY to generate recursion-free unwindings of a given recursive system so that, in each iteration, it generates an unwinding which is converted to CDD form. In the following context, "SHARA" refers to this modified version of DUALITY.

We evaluated SHARA and an unmodified version of DUALITY on 4,309 CHC systems generated from programs in the SV-COMP 2015 [8] verification benchmark suite. To generate CHC systems, we ran the SEAHORN [10] verification framework with its default settings (procedures are not inlined and each loop-free fragment is a clause), set to timeout at 90 seconds. We used the benchmarks in SV-COMP 2015 [8] because they were used to evaluate DUALITY in previous work [19].

We also compared SHARA to the ELDARICA CHC solver, but ELDARICA could not parse the CHC systems generated by SEAHORN. We compared SHARA and ELDARICA on an alternative set of benchmarks generated by the UFO model checker [3], and found that SHARA outperformed ELDARICA by at least an order of magnitude on an overwhelming number of cases. As a result, we focus our discussion on a comparison of SHARA and DUALITY.

All experiments were run using a single thread on a machine with 16 1.4 GHz processors and 128 GB of RAM. We ran the solvers on each benchmark, timing out each implementation after 180 seconds.

Out of 4,309 benchmarks, SHARA solved or refuted 2,408 while DUALITY solved or refuted 2,321. SHARA timed out on 762 benchmarks and DUALITY timed out on 1,145. On the remaining benchmarks, some constraint caused Z3's interpolating theorem prover to fail, meaning the result was neither a solve nor a refutation. SHARA reached this failure on 1,139 benchmarks while DUALITY failed on 843. The two solvers can induce a failure in Z3 on different systems because in attempting to solve a given system, they generate different interpolation queries.

The results of our evaluation are shown in Figure 5. Of the 4,040 benchmarks on which both solvers took a short amount of time—less than five seconds—SHARA solved the benchmarks in an average of 0.51 seconds and DUALITY solved them in an average of 0.42 seconds. Figure 5 contains data for benchmarks which took longer than five second for both systems to solve. Out of these 269 benchmarks,

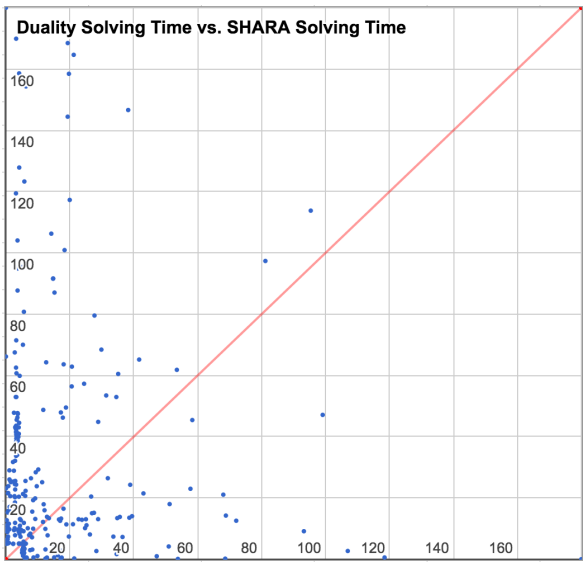

Figure 5: Solving times of SHARA vs. DUALITY. The x and y axes range over the solving times in seconds of SHARA and DUALITY, respectively. Each point depicts the performance of a benchmark. The line $y = x$ is shown in red.

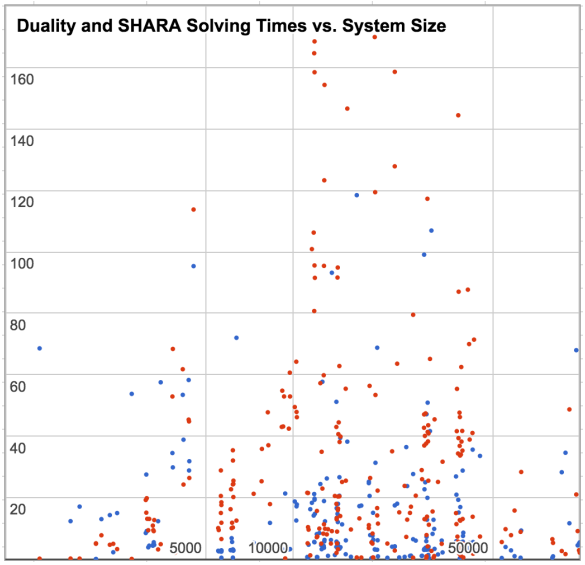

Figure 6: Times of SHARA and DUALITY vs. system size. The x-axis ranges over the size of a given system, and the y-axis ranges over solvers' times. Measurements of SHARA and DUALITY are shown in blue and red, respectively.

SHARA solved 185 in less time than DUALITY, and solved 159 in less than half the time of DUALITY. DUALITY solved 84 in less time than SHARA, and solved 53 in less than half the time of SHARA. Of the 762 benchmarks on which SHARA timed out, DUALITY solved or found a counterexample to 185. Of the 1,145 benchmarks on which DUALITY timed out, SHARA solved or found a counterexample to 470.

Figure 6 shows the relationship between the solving times of DUALITY and SHARA and the size of a given system, measured as lines of code in the format generated by SEAHORN. The majority of files have between 1,000 and 100,000 lines, so Figure 6 is restricted to this range. The data indicates that the performance improvement of SHARA compared to DUALITY is consistent across systems of all sizes available.

The results indicate that **(1)** on a significant number of different verification problems, SHARA can perform significantly better than DUALITY, but **(2)** there are some cases in which the strengths of each algorithm yield better results. Specifically, the unmodified version of DUALITY uses a technique called lazy annotation to avoid enumerating all derivation trees. SHARA does not use lazy annotation. We collected the differences between sizes of a given system and its minimal CDD expansion generated by SHARA and found that they were independent of SHARA's performance compared to DUALITY. Thus, while DUALITY may in the worst case enumerate exponentially many derivation trees, it appears to enumerate far fewer than the worst-case bound in some cases, causing it to perform better than SHARA. Our results indicate that a third approach that combines the strengths of both DUALITY and SHARA, perhaps by *lazily* unwinding a given system into a series of CDD systems, could yield further improvements.

6 Related Work

A significant body of previous work has presented solvers for different classes of Constrained Horn Clauses, or finding inductive invariants of programs that correspond to solutions of CHCs. IMPACT attempts to verify a given sequential procedure by iteratively selecting paths and synthesizing invariants for each path. This approach corresponds to solving a recursive linear CHC system [18].

Previous work also proposed a verifier for recursive programs [11]. The proposed approach selects interprocedural paths of a program and synthesizes invariants for each as nested interpolants. Such an approach corresponds to attempting to solve a recursive CHC system S by selecting derivation trees of S and solving each tree.

Previous work has proposed solvers for recursive systems that, given a system S, attempt to solve S by generating and solving a series of recursion-free unwindings of S. In particular, ELDARICA attempts to solve each unwinding S' by reducing to and solving body-disjoint systems [20, 21]. DUALITY attempts to avoid solving all derivation-trees (i.e body-disjoint systems) by using lazy annotation [4]. Other optimizations select derivation trees to solve using symbolic analogs of Prolog evaluation with tabling [13, 19].

WHALE attempts to verify sequential recursive programs by generating and solving hierarchical programs, which correspond to recursion-free CHC systems [2]. To solve a particular recursion-free system, WHALE solves a linear inlining of the input using a procedure named VINTA [1]. In general, the linear inlining may be exponentially larger than the input.

SHARA is similar to the recursion-free CHC approaches given above in that it reduces the problem to solving a CHC system in a directly-solvable class. SHARA is distinct in that it reduces to solving Clause-Dependent Disjoint (CDD) systems. As discussed in §2, the class of CDD systems is a superset of classes used by the approaches above. CDD systems can also be solved directly.

SHARA solves general CHC systems using the same strategy as proposed by the above approaches. Specifically, it solves a series of recursion-free unwindings of the original system, and tries to synthesize a general solution from the recursion-free solutions.

Previous work describes solvers for non-linear Horn clauses over particular theories. In particular, verifiers have been proposed for recursion-free systems over the theory of linear arithmetic [14]. Because the verifier relies on quantifier elimination, it is not clear if it can be extended to richer theories that support interpolation, such as the combination of linear arithmetic with uninterpreted functions. Other work describes a solver for the class of *timed pushdown systems*, a subclass of CHC systems over the theory of linear real arithmetic [12]. Unlike these approaches, SHARA can solve systems over any theory that supports interpolation.

DAG inlining attempts to generate compact verification conditions for hierarchical programs [15]. SHARA attempts to solve recursion-free CHC systems by reducing them to compact CDD systems. Because hierarchical programs and recursion-free CHC systems are closely related, algorithms that operate on hierarchical programs correspond to algorithms that operate on recursion-free Horn Clauses. However, it is not apparent whether such algorithms can be used directly to synthesize solutions.

References

[1] A. Albarghouthi, A. Gurfinkel, and M. Chechik. Craig interpretation. In *SAS*, 2012.

[2] A. Albarghouthi, A. Gurfinkel, and M. Chechik. Whale: An interpolation-based algorithm for inter-procedural verification. In *VMCAI*, 2012.

[3] A. Albarghouthi, Y. Li, A. Gurfinkel, and M. Chechik. UFO: A framework for abstraction- and interpolation-based software verification. In *CAV*, 2012.

[4] N. Bjørner, K. L. McMillan, and A. Rybalchenko. On solving universally quantified Horn clauses. In *SAS*, 2013.

[5] L. M. de Moura and N. Bjørner. Z3: an efficient SMT solver. In *TACAS*, 2008.

[6] C. Flanagan. Automatic software model checking using CLP. In *ESOP*, 2003.

[7] C. Flanagan and J. B. Saxe. Avoiding exponential explosion: generating compact verification conditions. In *POPL*, 2001.

[8] Github - sosy-lab/sv-benchmarks: svcomp15. `https://github.com/sosy-lab/sv-benchmarks/releases/tag/svcomp15`, 2016. Accessed: 2016 Dec 13.

[9] Github - z3prover/z3: The z3 theorem prover. `https://github.com/Z3Prover/z3`, 2016. Accessed: 2016 Dec 13.

[10] A. Gurfinkel, T. Kahsai, A. Komuravelli, and J. A. Navas. The SeaHorn verification framework. In *CAV*, 2015.

[11] M. Heizmann, J. Hoenicke, and A. Podelski. Nested interpolants. In *POPL*, 2010.

[12] K. Hoder and N. Bjørner. Generalized property directed reachability. In *SAT*, 2012.

[13] J. Jaffar, A. E. Santosa, and R. Voicu. An interpolation method for CLP traversal. In *CP*, 2009.

[14] A. Komuravelli, A. Gurfinkel, and S. Chaki. SMT-based model checking for recursive programs. In *CAV*, 2014.

[15] A. Lal and S. Qadeer. DAG inlining: a decision procedure for reachability-modulo-theories in hierarchical programs. In *PLDI*, 2015.

[16] A. Lal, S. Qadeer, and S. K. Lahiri. A solver for reachability modulo theories. In *CAV*, 2012.

[17] K. L. McMillan. An interpolating theorem prover. In *TACAS*, 2004.

[18] K. L. McMillan. Lazy abstraction with interpolants. In *CAV*, 2006.

[19] K. L. McMillan. Lazy annotation revisited. In *CAV*, 2014.

[20] P. Rümmer, H. Hojjat, and V. Kuncak. Classifying and solving Horn clauses for verification. In *VSTTE*, 2013.

[21] P. Rümmer, H. Hojjat, and V. Kuncak. Disjunctive interpolants for Horn-clause verification. In *CAV*, 2013.

Input : A recursion-free CHC system S.
Output : A minimal CDD expansion S' of S and a correspondence from S' to S.
1 **Procedure** EXPAND(S)
2 **Procedure** EXPAUX(S')
3 **switch** SHAREDREL(S') **do**
4 **case** None: **do return** S' ;
5 **case** $C \in S', P \in \mathsf{Pred}(S')$: **do return** EXPAUX(COPYREL(S', C, P)) ;
6 **end**
7 **return** (EXPAUX(S), CORR)
Algorithm 3: EXPAND: given a recursion-free CHC system S, returns a minimal CDD expansion S' of S and its correspondence.

A Generating a Minimal CDD Expansion

Given a recursion-free CHC system S, Alg. 3 returns a minimal CDD expansion of S (Defn. 7). EXPAND defines a procedure EXPAUX (line 2—line 6) that takes a CHC system S and returns a minimal CDD expansion of S. EXPAND runs EXPAUX on S and returns the result, paired with the map CORR : $\mathsf{Pred}(S') \to \mathsf{Pred}(S)$ (line 7).

EXPAUX, given a recursion-free CHC system S', runs a procedure SHAREDREL on S', which tries to find a clause $C \in S'$ and a predicate $P \in \mathsf{Body}(C)$ such that P is in the transitive dependencies of two sibling predicates. In such a case, we say that (C, P) is a *sibling-shared dependency*.

If SHAREDREL determines that no sibling-shared dependency exists, then EXPAUX returns S' (line 4).

Otherwise, SHAREDREL must have located a sibling-shared dependency (C, P). In this case, EXPAUX runs COPYREL on S', C, and P, which returns an expansion of S' by creating a fresh copy of P and updating $\mathsf{Body}(C)$ to avoid the shared dependency. EXPAUX recurses on this expansion and returns the result (line 5).

EXPAND always returns a CDD expansion of its input (see Appendix D, Lemma 3) that is minimal. EXPAND is certainly not unique as an algorithm for generating a minimal CDD expansion. In particular, feasible variations of EXPAND can be generated from different implementations of SHAREDREL, each of which chooses clause-relation pairs to return based on different heuristics. We expect that other expansion algorithms can also be developed by generalizing algorithms introduced in previous work on generating compact verification conditions of hierarchical programs [15].

B Proof of characterization of CDD systems

The following is a proof of Thm. 1.

Proof. To prove that CDD is a strict superset of the union of the class of linear systems and the class of body-disjoint systems, we prove **(1)** CDD contains the class of linear systems, **(2)** CDD contains the class of body-disjoint systems, and **(3)** there is some CDD system that is neither linear nor body-disjoint.

For goal **(1)**, let S be an arbitrary linear system. S is CDD if for each clause C in S **(1)** and each pair of distinct predicates in the body of C has disjoint transitive dependencies and **(2)** no predicate appears more than once in the body of C. (Defn. 5). Let C be an arbitrary clause in S. Since C is a linear clause, it has at most one relational predicate in its body. And since the system is recursion-free, the transitive dependencies are trivially disjoint and there can be no repeated predicate. Therefore, S is CDD.

For goal **(2)**, let S be an arbitrary body-disjoint system. The dependence relation of S is a tree T, by the definition of a body-disjoint system. Let C be an arbitrary clause in S, with distinct relational predicates R_0 and R_1 in its body. All dependencies of R_0 and R_1 are in subtrees of T, which are disjoint by the definition of a tree. Thus, S is CDD, by Defn. 5.

For goal **(3)**, the system $\mathtt{S_{DA}}$ is CDD, but is neither linear nor body-disjoint. □

C Proof SOLVECDD is in co-NP

The following is a proof of Thm. 2. Namely, SOLVECDD is in co-NP.

Proof. PRE and POST construct formulas linear in the size of the CHC system. The satisfiability problem for the constructed formulas are in NP for linear arithmetic. SOLVECDD issues (at worst) a linear number of interpolation queries in terms of number of predicate. Therefore, the upper bound of SOLVECDD is co-NP. □

D Proof of Correctness

In this section, we prove that SHARA is correct when applied to recursion-free CHC systems. We first establish lemmas for the correctness of each procedure used by SHARA, namely COLLAPSE (Lemma 1 and Lemma 2), EXPAND (Lemma 3), and SOLVECDD (Lemma 4, Lemma 6, and Lemma 7). We combine the lemmas to prove SHARA is correct (Thm. 3).

For two recursion-free CHC systems S and S', if S' is an expansion of S, then the result of collapsing a solution of S' is a solution of S.

Lemma 1. *For two recursion-free CHC system S' and S such that σ' is a solution of S' and η is a correspondence from S' to S, $\textsc{Collapse}(\eta,\sigma')$ is a solution of S.*

Proof. For each predicate $P' \in \mathsf{Pred}(S')$ such that $\eta(P') = P$, there must exist some clause $C' \in S'$ such that $P' \in \mathsf{Body}(C')$ because S' is an expansion of S. Let predicate $Q' \in \mathsf{Pred}(S')$ be the head of C'. $\sigma'[P'] \wedge \mathsf{Constraint}(C') \models \sigma'[Q']$ by the fact that σ' is a solution of S'. Therefore,

$$\textsc{Collapse}(\eta,\sigma')[P] \wedge \mathsf{Constraint}(C') \models \left(\bigwedge_{\substack{Q' \in \mathsf{Pred}(S') \\ \eta(Q')=Q}} \sigma(Q') = \textsc{Collapse}(\eta,\sigma')[Q'] \right)$$

Therefore, $\textsc{Collapse}(\eta,\sigma')$ has a solution for P. Since $\textsc{Collapse}(\eta,\sigma')$ has a solution for each predicate in S, $\textsc{Collapse}(\eta,\sigma')$ is a solution of S. □

Every expansion of a solvable recursion-free CHC system is also solvable.

Lemma 2. *If a recursion-free CHC system S is solvable and S' is an expansion of S, then S' is solvable.*

Proof. Let σ be a solution of S, and let η be a correspondence from S' to S. Let σ' be such that for each $P' \in \mathsf{Pred}(S')$, $\sigma'(P') = \sigma(\eta(P'))$. Then σ' is a solution of S'. □

EXPAND always returns a CDD expansion of its input.

Lemma 3. *For two recursion-free CHC systems S and S' and a correspondence from S' to S, η, such that $(S',\eta) = \textsc{Expand}(S)$, S' is a CDD system and an expansion of S.*

Proof. By induction over the evaluation of EXPAND on an arbitrary recursion-free CHC system S. The inductive fact is that for each evaluation step CORR is a correspondence from argument S' to S. In the base case, EXPAUX is called initially on S, by Alg. 3. CORR is a correspondence from S to itself, by the definition of CORR (Appendix A).

In the inductive case, EXPAUX constructs an argument $\textsc{CopyRel}(S,C,P)$ where C is a clause and P is a predicate in S. EXPAUX recusively invokes itself with this argument. For each recursion-free CHC system S' generated by $\textsc{CopyRel}(S,C,P)$, CORR is a correspondence from S' to S by definition of COPYREL (Appendix A). By this fact and the inductive hypothesis, CORR is a correspondence from $\textsc{CopyRel}(S',C,P)$ to S.

EXPAUX returns its parameter at some step, by Alg. 3. Therefore, EXPAUX returns an expansion of S.

For a given recursion-free CHC system S', if $(S',\eta) = \textsc{Expand}(S')$, then $\textsc{SharedRel}(S') = \mathsf{None}$, by the definition of EXPAND. If $\textsc{SharedRel}(S') = \mathsf{None}$, then S' is CDD, by the definition of SHAREDREL and CDD systems (Defn. 5). Therefore, S' is CDD. □

Furthermore, EXPAND returns a *minimal* CDD expansion of its input. This fact is not required to prove Thm. 3, and thus a complete proof is withheld.

For each recursion-free CHC system S, S has a solution if and only if all interpolation queries return interpolants.

Lemma 4. *Given a recursion-free CHC system S that is CDD and solvable, for all predicates $P \in \mathsf{Pred}(S)$,* ITP *returns an interpolant I.*

Proof. Assume that S has a solution σ and there are some predicates $P \in \mathsf{Pred}(S)$ such that ITP returns *SAT*. This means there must be a model m for the conjunction of $\textsc{Pre}(\mathtt{P}, \sigma')$ and $\textsc{Post}(\mathtt{P}, \sigma')$. But $\textsc{Post}(\mathtt{P}, \sigma) = \mathsf{False}$, by the definition of a solution of a CHC system. Therefore, there can be no such model m. Therefore, ITP always returns interpolant. □

Lemma 5. *Given a recursion-free CHC system S that is CDD, if for all predicates $P \in \mathsf{Pred}(S)$,* ITP *returns an interpolant I, then* $\textsc{SolveCdd}(S)$ *is a solution of S.*

Proof. By induction on the $\textsc{SolveCdd}(S)$ calls to ITP over all predicates $P \in \mathsf{Pred}(S)$ in topological order. The inductive fact is that after each call to ITP, σ is a partial solution of S. In the base case, $\mathsf{Pred}(S) = \varnothing$, Therefore, σ is a solution of S.

In the inductive case, SOLVECDD calls ITP on a predicate $P \in \mathsf{Pred}(S)$ with partial solution σ. Due to the topological ordering, σ contains interpretations for each predicate $P \in \mathsf{Deps}(S)$. Based on the definition of an interpolant (Defn. 4), $\textsc{Pre}(\mathtt{P}, \sigma)$ and $\textsc{Post}(\mathtt{P}, \sigma)$ are inconsistent. The interpolant of these two formulas returned by ITP, I, is entailed by each clause C where P is the head where the predicates in $\mathsf{Body}(C)$ are substituted by their interpretations in σ. I is also inconsistent with all constraints that appear after P that support the query clause. Therefore, when SOLVECDD updates σ by binding P to I the result is a partial solution of S. □

The output of SOLVECDD is correct for a given input CDD system.

Lemma 6. *For a given CDD system S, and $\sigma = \textsc{SolveCdd}(S)$, σ is a solution of S.*

Proof. The fact that $\textsc{SolveCdd}(S)$ returns σ implies that for each predicate $P \in \mathsf{Pred}(S)$, ITP returns a valid interpolant (Alg. 1). Therefore, Lemma 5 implies that SOLVECDD returns a complete solution of S. □

Lemma 7. *For a CDD system S such that S is solvable, there is some σ such that $\sigma = \textsc{SolveCdd}(S)$.*

Proof. For all predicates $P \in \mathsf{Pred}(S)$, ITP returns a interpolant I, by Lemma 4 and the fact that S is solvable. Therefore, by Lemma 5 and the fact that S is solvable, $\textsc{SolveCdd}(S)$ returns a solution of S. □

The output of SHARA is correct for a given input CDD system. (§4, Thm. 3).

Proof. Given two recursion-free CHC systems S and S' and a η such that $(S', \eta) = \textsc{Expand}(S)$, S' is minimal CDD expansion of S and η is a correspondence from S' to S (Lemma 3). Assume that S is solvable. Then so is S', by Lemma 2. Therefore, there exists some σ' such that $\sigma' = \textsc{SolveCdd}(S')$, by the definition of SHARA. σ' is a solution of S', by Lemma 6. $\textsc{Collapse}(\eta, \sigma')$ is a solution of S, by Lemma 1. Therefore, SHARA returns a valid solution of S. □