EPiC Series in Computing

Volume 46, 2017, Pages 338–355

LPAR-21. 21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning

Synchronizing Constrained Horn Clauses

Dmitry Mordvinov[1] and Grigory Fedyukovich[2]

[1] Saint-Petersburg State University, Department of Software Engineering, Russia,
dmitry.mordvinov@se.math.spbu.ru

[2] University of Washington Paul G. Allen School of Computer Science & Engineering, USA,
grigory@cs.washington.edu

Abstract

Simultaneous occurrences of multiple recurrence relations in a system of non-linear constrained Horn clauses are crucial for proving its satisfiability. A solution of such system is often inexpressible in the constraint language. We propose to *synchronize* recurrent computations, thus increasing the chances for a solution to be found. We introduce a notion of *CHC product* allowing to formulate a lightweight iterative algorithm of merging recurrent computations into groups and prove its soundness. The evaluation over a set of systems handling lists and linear integer arithmetic confirms that the transformed systems are drastically more simple to solve than the original ones.

1 Introduction

Inductive invariants describe useful properties of recurrent computations. Expected as solutions by the automated constrained Horn solvers [8, 9, 7, 12, 17, 11], they could never be delivered due to two reasons. First, multiple individual recurrences and a high level of non-determinism significantly enlarges the search space for each invariant. Second, the expressivity of the constraint language imposes extra requirements on the shape of the invariant, exacerbating the question of its existence.

The earlier idea, originated from the literature on constraint programming, was to *synchronize* recurrent computations by making them iterate simultaneously until either one terminates [2, 4] using folding/unfolding rules. More recently, De Angelis et al. [3] adapted this transformation to the field of constrained Horn clauses (CHC) and showed that it also relaxes the shape of the desired invariant, simplifying its discovery. However, synchronizing an arbitrary number of recurrent computations could complicate the system description due to extra relations and rules. Further reasoning about the transformed systems could be problematic, especially if the solver is linear.

We present an alternative CHC transformation which tends to produce smaller systems for any number of synchronizable recurrences. The key idea is to symbolically align the computations to have equal lengths and then to merge them together. The transformation itself is inexpensive: it requires a syntactical pass over the sets of rules that live in the CHC system. In the lower level, it performs an operation similar to the Cartesian product (thus, the introduced concept is called a *CHC product*). Such intuitive analogy lets us set up a theoretical

framework and prove the soundness of the transformation. To the best of our knowledge, it is the first self-contained description of the CHCs being synchronized which is independent on folding/unfolding rules.

Synchronization helps establish useful relational properties about recurrence relations to hold before and after each iteration. For instance, such step-by-step reasoning allows proving the monotonicity of Fibonacci numbers completely automatically. Furthermore, synchronization allows proving properties about multiple iterators over the same list of integers [5]. Propagating equalities of lists through the CHCs practically reduces the constraint language from lists to linear arithmetic, but keeps the same level of expressivity for the systems.

We implemented our CHC product transformation and integrated it to the symbolic model checking tool Rosette/Unbound [14] aiming to derive inductive invariants of functional programs in Racket. As a constrained Horn solver, Rosette/Unbound uses Spacer3 [12] that effectively checks satisfiability of the transformed systems. While nearly none of the original systems over linear arithmetic and lists was solvable, Spacer3 delivered the inductive invariants of most of the transformed systems within a second.

Related work. Synchronization of CHCs is typically achieved using folding / unfolding rules [3]. This transformation aims at replacing conjunctions of applications of uninterpreted symbols in a CHC with an application of a fresh uninterpreted symbol. All other applications of the original symbols, which are alone in some other CHC in the system, remain untouched. In contrast, our transformation tautologically extends CHCs, and in most of the cases it helps "cleaning", i.e., replacing the original symbols completely. Furthermore, our algorithm in general is less expensive itself and produces the systems that are less expensive to solve.

The systems of CHCs in question are non-linear, in a sense that their bodies contain applications of more than one uninterpreted symbol. The approaches for linearization include [10], which proceeds by iterative construction of a linear under-approximation of the system, solving it, and checking if such partial solution also fits the original system. In many cases, our transformation acts as linearization (more often than [3]). But instead of delivering an exact solution for the original system (since for the considered systems, as also noticed by [15], it barely exists), we accurately re-formulate each system and prove that it is consistent with the original one.

The concept of "product" of programs appears in the field of relational verification for pairs of programs [1, 13, 16]. Those approaches sequentially compose the programs, insert a pre- and a postcondition and proceed by applying Hoare-style derivations either manually or automatically. An interesting extension of this idea to the case of unbalanced recursion was proposed by [16]. However, since it potentially increases a search space of possible non-straightforward products, the approach currently remains manual.

Simulation is another, yet orthogonal, approach to deal with pairs of systems of recurrent computations specified with CHCs. Instead of proving whether a given relational property holds, it aims to discover a fresh property that actually holds using simulation. A recently proposed approach for such synthesis [6] is limited to the case of one-way simulation, so the discovered facts are not guaranteed to hold as relational properties in our setting. In future, it would be interesting to see how our approach can complement the simulation discovery.

2 Preliminaries

Throughout the paper, we consider a fixed signature $(\mathcal{F}, \mathcal{P}, ar)$, where $\mathcal{F}$ and $\mathcal{P}$ are disjoint sets of function and predicate symbols, and ar is a mapping of symbols to their arities. We

assume that interpretation of symbols from $\mathcal{F}$ and $\mathcal{P}$ is given by structure $(\mathcal{D}, \sigma)$ with non-empty domain $\mathcal{D}$, for each n-ary $f \in \mathcal{F}$, $\sigma(f) : \mathcal{D}^n \to \mathcal{D}$, and for each n-ary $p \in \mathcal{P}$, $\sigma(p) \subseteq \mathcal{D}^n$. Given a countable set of variables $\mathcal{X}$, a first-order language $\mathcal{A}$ of quantifier-free formulae over $\mathcal{F}$, $\mathcal{P}$, and $\mathcal{X}$ is called a *constraint language* (or an *assertion language*). *Constraint* is a formula in $\mathcal{A}$.

We assume a standard definition of satisfiability of constraints. We denote a set of free variables of formula Φ by $fv(\Phi)$. Restriction of function f to set X is denoted $f\big|_X$.

Definition 1. Constrained Horn clause (*CHC*) is a formula C in a first-order logic of the form

$$\phi \wedge p_1(\vec{x_1}) \wedge \ldots \wedge p_k(\vec{x_k}) \implies H \tag{1}$$

Here we consider a fixed set $\mathcal{R}$ of uninterpreted *relation symbols*, such that $\mathcal{R} \cap (\mathcal{F} \cup \mathcal{P}) = \varnothing$. Expression H, called *head* of the clause, is either an application $p_0(\vec{x_0})$, or constant $\bot$. Every p_i is an uninterpreted relation symbol: $p_i \in \mathcal{R}$, and ϕ is a constraint. Each $\vec{x_i}$ is a vector of variables, $\vec{x_i}[j] \in \mathcal{X}$ for all $0 \leqslant j < |\vec{x_i}|$. It is convenient to introduce the following notation for the left and right sides of C, respectively:

$$body(C) \stackrel{\text{def}}{=} \{\phi, p_1(\vec{x_1}), \ldots, p_k(\vec{x_k})\} \qquad\qquad head(C) \stackrel{\text{def}}{=} H$$

In spirit of constraint logic programming [4], throughout the paper we write CHCs as rules, i.e., $H \leftarrow \phi, p_1(\vec{x_1}), \ldots, p_k(\vec{x_k})$ for (1). We distinguish three disjoint parts in the body of each clause C, i.e., $body(C) = \{\phi\} \cup L \cup R$, where ϕ is a constraint, R is a set of recursive applications: $R = \{r \in body(C) \mid rel(r) = rel(head(C))\}$, and L is a set of non-recursive applications in the body: $L = body(C) \backslash (\{\phi\} \cup R)$. Given an expression $a = p(\vec{x})$ for some $p \in \mathcal{R}$, $rel(a)$ mnemonically denotes the applied relation symbol p, and $args(a)$ denotes the vector of arguments $\vec{x}$. For convenience, we extend functions rel and $args$ for all non-application or non-relation application expressions e: $rel(e) = \bot$, and $args(e) = \vec{\varnothing}$ (where $\vec{\varnothing}$ is an empty vector). This allows us write $rel(head(C))$ for an arbitrary clause C.

Definition 2. Given a set of CHCs P and $p \in \mathcal{R}$, *rules defining* p is a subset of P, such that:

$$rules(p) \stackrel{\text{def}}{=} \{C \in P \mid rel(head(C)) = p\}$$

Definition 3. A *system* of CHCs is pair (P, q), where P is a set of CHCs, and $q \in P$ is the only clause, called *query*, with $head(q) = \bot$.

Remark 1. Without loss of generality, for a fixed system of CHCs we assume that different clauses have disjoint sets of free variables.

Derivations of heads in CHCs are defined using trees (see Sect. 3.1). Throughout the paper, we write capital T (possibly, subscripted or superscripted) for a set of trees (possibly, sharing nodes) and lowercase t for a single tree. Each tree t is treated as pair $[r, T]$, where r is a root node (denoted $root(t)$), and T is a set of the partial subtrees. A *path* in tree t is a partial subtree of the form $\pi = [v, \varnothing]$ or $\pi = [v, \{\pi'\}]$, where π' is a path as well. The latter case permits also a simplified notation $\pi = [v, \pi']$. A set of all paths from the root to some leaf of t is denoted by $paths(t)$. Finally, $paths(T) \stackrel{\text{def}}{=} \bigcup_{t \in T} paths(t)$, and $roots(T) \stackrel{\text{def}}{=} \bigcup_{t \in T} \{root(t)\}$.

It is easy to see that $paths(\varnothing) = \varnothing$, and $\pi \in paths([r, T]) \iff \pi = [r, \pi'] \wedge \pi' \in paths(T)$. Also, it can be that $|roots(T)| < |T|$, for example if $T = paths(t)$ for some tree t with more than one leaf (obviously $|roots(T)|$ in this case is always 1).

Definition 4. A set of trees $merge(T)$ of T is the one of minimal cardinality, such that $paths(merge(T)) = paths(T)$.

Proposition 1. $merge(T) = \left\{ [r, merge(T')] \;\middle|\; r \in roots(T), T' = \{T_i \mid [r, T_i] \in T\} \right\}$

Remark 2. Prop. 1 implies that if $|roots(T)| = 1$, then $|merge(T)| = 1$.

3 Exploring Unsatisfiability of CHCs

Systems of CHCs serve as specifications for synthesis of inductive invariants. Due to decidability restrictions of the constraint language and a large search space, inductive invariants cannot be discovered even for certain linear systems. Disproving satisfiability, however, could be easier since it is enough to find a single, finite counterexample as opposed to an invariant satisfying the entire system universally. In this section, we formally discuss both notions.

3.1 Satisfiability vs unsatisfiability

Definition 5. Let $S = (P, q)$ be a system of CHCs over relation symbols $\mathcal{R}$, and $P = \{C_1, \ldots, C_n\}$. S is called *satisfiable*, denoted sat_S, if there is a mapping $I : \mathcal{R} \to \mathcal{A}$, called *relational symbol assignment*, such that $\bigwedge_{i=1}^{n} Cl_\forall(J(C_i))$ is satisfiable. Here $Cl_\forall(\Phi)$ is the universal closure of Φ, and $J(C)$ is an expression in $\mathcal{A}$ defined for C having form (1):

$$J(C) \stackrel{\text{def}}{=} \left(\phi \wedge I(p_1)(\vec{x_1}) \wedge \ldots \wedge I(p_k)(\vec{x_k}) \implies \begin{cases} I(p)(\vec{x}), & \text{if } head(C) = p(\vec{x}) \\ \bot, & \text{if } head(C) = \bot \end{cases} \right)$$

Relational symbol mapping I maps each uninterpreted relation symbol from $\mathcal{R}$ to its interpretation. It defines a *solution* of the system, also referred to as *inductive invariant*.

Definition 6. Given set P of CHCs, $C \in P$ of form (1), let $\vec{x_0} = args(head(C))$. Let $\vec{v_0}$ be a vector of values of variables in $\vec{x_0}$. We say that C is *realizable* on $\vec{v_0}$, if there is an assignment $v : fv(C) \to \mathcal{D}$ that extends $\vec{v_0}$ (i.e., $v(\vec{x_0}[j]) = \vec{v_0}[j]$ for all j), such that:

1. constraint $\phi \in body(C)$ evaluates to $\top$ on v;
2. for each application $p_i(\vec{x_i}) \in body(C)$, there exists $C_i \in rules(p_i)$, such that C_i is *realizable* on $\vec{v_i}$, where $\vec{v_i}[j] = v(\vec{x_i}[j])$ for all $0 \leqslant j < |\vec{x_i}|$.

Def. 6 is inductive: to realize each CHC we should "unroll" it up to the CHCs whose bodies consist of constraints only (called *facts*). This unrolling induces some tree structure (finite or infinite). We are interested only in finite ones. This lets us reformulate Def. 6 in terms of trees:

Definition 7. A tree $t = [r, T]$ *realizes* CHC C if

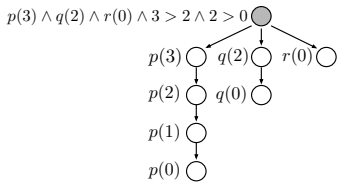
1. $r = (C, \vec{v_0}, v)$, where $\vec{v_0}$ is a vector of values from $\mathcal{D}$ with $|\vec{v_0}| = |args(head(C))|$, and $v : fv(C) \to \mathcal{D}$ is an assignment of all free variables in C extending $\vec{v_0}$;
2. constraint $\phi \in body(C)$ evaluates to $\top$ on v;
3. for each application $p_i(\vec{x_i}) \in body(C)$, there exists a tree $t_i \in T$, such that $t_i = [(C_i, \vec{v_i}, \cdot), T_i]$, where $C_i \in rules(p_i)$, and $\vec{v_i}[j] = v(\vec{x_i}[j])$ for all $0 \leqslant j < |\vec{x_i}|$.
4. each $t_i = [(C_i, \cdot, \cdot), T_i]$ in T realizes C_i.

Definition 8. The system $S = (P, q)$ of CHCs is called *unsatisfiable*, denoted $unsat_S$, if there exists a tree, called *refutation tree*, $t = [(q, \varnothing, \cdot), T]$ realizing q.

Note that if a refutation tree for a system S of CHCs can be built, then the bodies of all CHCs from S are evaluated to $\top$ independently on an interpretation of the relation symbols. In other words, for every interpretation there is an assignment of variables that make the body of the query evaluate to $\bot$, thus witnessing unsatisfiability of S.

Example 1. System S_{p+q+r} of CHCs is unsatisfiable which is witnessed by the following refutation tree. The root of the refutation tree is labeled by the body of the query with values $\{3, 2, 0\}$ substituted respectively for variables $\{x_7, x_8, x_9\}$. Three paths correspond to the sequences of recursive applications of p, q, and r until the facts are reached.

$$\begin{cases} p(x_1) \leftarrow x_1 = 0 \\ p(x_2) \leftarrow p(x_1), x_2 = x_1 + 1 \\ q(x_3) \leftarrow x_3 = 0 \\ q(x_4) \leftarrow q(x_3), x_4 = x_3 + 2 \\ r(x_5) \leftarrow x_5 = 0 \\ \quad \bot \leftarrow p(x_7), q(x_8), r(x_9), (x_7 > x_8), (x_8 > x_9) \end{cases}$$



□

Lemma 1. $sat_S \implies \neg unsat_S$

To prove that a system of CHCs is unsatisfiable it is enough to find a refutation tree for it. Interestingly, a refutation tree can be constructed by merging the individual paths (see Def. 4).

3.2 Aligning paths using tautologies

An extension $S^\top = (P^\top, q)$ of $S = (P, q)$ with tautological clauses is equivalent to S, i.e., both systems are either *sat* or *unsat*. In particular, let $P^\top$ contain all CHCs from P, and for each fact $p(\vec{x}) \leftarrow \phi$ in P, let $P^\top$ additionally contain a CHC $p(\vec{x}) \leftarrow \phi, p(\vec{x})$. This *tautologically-extended* system $S^\top$ has the following property.

Lemma 2. *If $unsat_S$, then there exists some natural number N and a refutation tree of $S^\top$, such that all paths from the root of the refutation to its leaves have length N.*

Proof. If $unsat_S$, then there exists some finite refutation tree t of S. Let N be the maximal length among paths from $root(t)$ to the leaves. Each leaf in t is reachable from $root(t)$ in n steps, where $n \leqslant N$, and it corresponds to a fact. Thus, if the last edge in t is copied $(N - n)$ times, then the resulting tree witnesses unsatisfiability of $S^\top$ due to tautological clauses. □

In the rest of the paper, we assume that every refutation tree is aligned.

4 Product of CHCs

Multiple recurrence relations appearing in the same CHC complicate discovery of inductive invariants. This can be overcome by a modification or a simplification of the system, as long as it preserves the original semantics. The key contribution of the paper is the new CHC transformation which relaxes the shape of the expected invariant and in many non-trivial cases makes the transformed systems solvable.

4.1 Defining product transformation formally

Definition 9. Non-empty sets $A_1, \ldots, A_n$ are covered by A, denoted $A \in \lfloor A_1, \ldots, A_n \rfloor$, if

1. $A \subseteq A_1 \times \ldots \times A_n$, and
2. each element of each A_i appears at position i of at least one tuple $x \in A$.

Example 2. The following holds for the sets of natural numbers:

$$\{(1,3,5),(2,4,5)\} \in \lfloor \{1,2\},\{3,4\},\{5\} \rfloor \qquad \{(1,3,5),(2,3,5)\} \notin \lfloor \{1,2\},\{3,4\},\{5\} \rfloor$$

We define products gradually, for all ingredients of systems of CHCs.

Definition 10 (product of predicate applications)**.** Given predicates $p_1, \ldots, p_n$ and predicate r such that $ar(r) = \sum_{i=1}^{n} ar(p_i)$, an *r-product of applications* $p_1(\vec{x_1}), \ldots, p_n(\vec{x_n})$, denoted $\prod_{i=1,r}^{n} p_i(\vec{x_i})$, is an expression $r(\vec{x_1} \cdot \ldots \cdot \vec{x_n})$. Here $\vec{x} \cdot \vec{y}$ denotes the concatenation of vectors $\vec{x}$ and $\vec{y}$.

We may abuse the notation and write $\prod_r p_i(\vec{x_i})$ or just $\prod_r A$ for set A, if the order of elements in the product is clear from the context.

Definition 11. Given CHC C with $body(C) = \{\phi\} \cup L \cup R$, a set $R_{/_{head}}(C)$ is defined as:

$$R_{/_{head}}(C) \stackrel{\text{def}}{=} \begin{cases} head(C), & \text{if } R = \varnothing \\ R, & \text{otherwise} \end{cases}$$

Def. 11 embodies the concept of the tautological extension: using $R_{/_{head}}(C)$ instead of R turns every fact $p(\vec{x}) \leftarrow \phi$ into $p(\vec{x}) \leftarrow \phi, p(\vec{x})$.

Definition 12 (product of CHCs)**.** Let $C_1, \ldots, C_n$ be CHCs and $p_1, \ldots, p_n$ be different relation symbols, such that $C_i \in rules(p_i)$. Let p be a fresh relation symbol, such that $p \notin \mathcal{R}$ and $ar(p) = \sum_{i=1}^{n} ar(p_i)$. CHC C over $\mathcal{R} \cup \{p\}$ is called *product* of $C_1, \ldots, C_n$ on p, denoted $C = C_1 \times_p \ldots \times_p C_n$, if:

$$head(C) = \prod_p head(C_i); \qquad body(C) = \{\phi\} \cup L \cup R \backslash \{head(C)\} \qquad \text{where:}$$

$$\phi = \bigwedge_{i=1}^{n} \phi_i; \quad L = \bigcup_{i=1}^{n} L_i;$$

$$R = \left\{ \prod_p r_i \;\middle|\; (r_1, \ldots r_n) \in \lfloor R_{/_{head}}(C_1), \ldots, R_{/_{head}}(C_n) \rfloor \right\}$$

Definition 13 (product of relation symbols)**.** Let $P = \{C_1, \ldots, C_m\}$ be a set of CHCs over $\mathcal{R}$, and $p_1, \ldots, p_n$ be relation symbols from $\mathcal{R}$. Product of $p_1, \ldots, p_n$, denoted $p_1 \times \ldots \times p_n$, is a pair $(p, rules(p))$, where p is a fresh relation symbol with $ar(p) = \sum_{i=1}^{n} ar(p_i)$, and $rules(p) = \{C_1 \times \ldots \times C_n \mid (C_1, \ldots, C_n) \in rules(p_1) \times_p \ldots \times_p rules(p_n)\}$.

Now we are ready to formally define the product transformation.

Definition 14. Let $P = \{C_1, \ldots, C_m\}$ be a set of CHCs, and some $C_a \in P$ has form:

$$C_a = H \leftarrow \phi, p_1(\vec{x_1}), \ldots, p_n(\vec{x_n}), W$$

such that for all $1 \leqslant i < j \leqslant n$: $p_i \neq p_j \neq rel(H)$, and W may contain other applications of uninterpreted relation symbols $\mathcal{R}$. Let $(p, rules(p)) = p_1 \times \ldots \times p_n$. A *p-transformation* of P is a set P' of CHCs over $\mathcal{R}' = \mathcal{R} \cup \{p\}$, obtained from P by adding new rules and replacing applications of p_i in C_a with their p-product:

$$P' \stackrel{\text{def}}{=} P \backslash \{C_a\} \cup rules(p) \cup \{C'_a\},$$

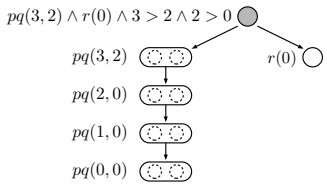
where C'_a has form:

$$C'_a \stackrel{\text{def}}{=} H \leftarrow \phi, \prod_p p_i(\vec{x_i}), W$$

Remark 3. In certain cases (e.g., when the $rules(p_1), \ldots, rules(p_n)$ have the same topologies and $p_1, \ldots, p_n$ are not mutually recursive), the applications of $p_1, \ldots, p_n$ might completely disappear from the transformed system, thus making $rules(p_1), \ldots, rules(p_n)$ useless.

Example 3. System of CHCs S_{pq+r} transformed from S_{p+q+r} (shown in Ex. 1):

$$\begin{cases} pq(x_1, x_3) \leftarrow x_1 = 0 \wedge x_3 = 0 \\ pq(x_2, x_3) \leftarrow pq(x_1, x_3) \wedge x_2 = x_1 + 1 \wedge x_3 = 0 \\ pq(x_1, x_3) \leftarrow pq(x_1, x_3) \wedge x_1 = 0 \wedge x_4 = x_3 + 2 \\ pq(x_2, x_4) \leftarrow pq(x_1, x_3) \wedge x_2 = x_1 + 1 \wedge x_4 = x_3 + 2 \\ r(x_5) \leftarrow x_5 = 0 \\ \bot \leftarrow pq(x_7, x_8) \wedge r(x_9) \wedge (x_7 > x_8) \wedge (x_8 > x_9) \end{cases}$$



The pq-product gives rise to four new CHCs in $rules(pq)$ that do not have applications of p and q, and thus $rules(p)$ and $rules(q)$ are not needed. S_{pq+r} preserves unsatisfiability, and the original refutation tree could be obtained by *splitting* the pq-path (see more details in Sect. 4.2). □

4.2 Properties of product transformation

The goal of this subsection is to demonstrate that our product transformation preserves satisfiability and unsatisfiability for any system of CHCs.

Theorem 1. $sat_P \implies sat_{P'}$

Proof. By Def. 5 there is $I : \mathcal{R} \to \mathcal{A}$ such that all clauses from P are satisfied for all values of free variables. Our goal is to find $I' : \mathcal{R}' \to \mathcal{A}$ satisfying transformed system P'. Such assignment is constructed in the following way:

$$I'(p)(\vec{x}) \stackrel{\text{def}}{=} \begin{cases} I(p)(\vec{x}), & \text{if } p \in \mathcal{R} \\ I(p_1)(\vec{x_1}) \wedge \ldots \wedge I(p_n)(\vec{x_n}), & \text{if } p = p_1 \times \ldots \times p_n, p_i \in \mathcal{R}, \vec{x} = \vec{x_1} \cdot \ldots \cdot \vec{x_n} \end{cases}$$

Now it is easy to see that all clauses from P' are satisfied with I'. Let C' be some clause in P'. It either does not have occurrences of p or does. In the former case, it is satisfied by I' because it is satisfied by I. In the latter case, we again consider two situations.

1. $C' \notin rules(p)$. In this case, C' is obtained by replacing applications $p_1(\vec{x_1}), \ldots, p_n(\vec{x_n})$ in the body of some clause $C \in P$ with $\prod_p p_i(\vec{x_i})$. In expression $J(C)$, it corresponds to $I(p_1)(\vec{x_1}) \wedge \ldots \wedge I(p_1)(\vec{x_1})$, which is exactly $I'(p)(\vec{x_1} \cdot \ldots \cdot \vec{x_n})$ by construction of I' (i.e., $J'(C')$ is equivalent to $J(C)$).

2. $C' \in rules(p)$. By Def. 13 there exist clauses $C_1, \ldots, C_m$ from P such that $C' = C_1 \times_p \ldots \times_p C_n$. Thus, for every C_i, by Def. 5, $Cl_\forall(J(C_i))$ is satisfiable. For some fixed variable values, this implies that either the premise of implication in $J(C_i)$ is false, or both the premise and the conclusion are true. If the premise of $J(C_i)$ is false then at least one of its conjuncts are false. But by Def. 12 every conjunct of every $J(C_i)$ occurs in the premise of $J'(C')$ making the product implication true. Finally, if both the premise and the conclusion of each $J(C_i)$ is true, then $J'(C')$ also holds. Indeed, every constraint or application of relation symbol in C' is either taken from some C_i, or it is an application of p (the interpretation of which is also true as the conjunction of true expressions). □

Interestingly, Th. 1 is not applicable in the backward direction. The fact that inductive invariant for the transformed system P' exists and expressible in $\mathcal{A}$ does not imply that an invariant for the original system P is also expressible in $\mathcal{A}$, as can be seen from the following example.

Example 4. The following system of CHCs over the theory of lists encodes the property that the result of summing all elements of any list to 0 is the additive inverse to the result of subtracting all elements of the same list from 0:

$$\begin{cases} \text{FLD}_+(xs, n) \leftarrow xs = nil, n = 0 \\ \text{FLD}_-(ys, m) \leftarrow ys = nil, m = 0 \\ \text{FLD}_+(xs, n) \leftarrow \text{FLD}_+(xs', n'), xs = cons(x, xs'), n = n' + x \\ \text{FLD}_-(ys, m) \leftarrow \text{FLD}_-(ys', m'), ys = cons(y, ys'), m = m' - y \\ \bot \leftarrow \text{FLD}_+(xs, n) \wedge \text{FLD}_-(xs, m) \wedge (n + m \neq 0) \end{cases}$$

Despite this property seems intuitive, the corresponding inductive invariant is inexpressible in the quantifier-free theory of lists. In contrast, the transformed system has a simple inductive invariant: $\text{FLD}_{+/-}(xs, n, ys, m) = (xs = ys) \implies (n + m = 0)$ which can be obtained by propagating the negation of the linear part of the query towards the fact CHC:

$$\begin{cases} \text{FLD}_{+/-}(xs, n, ys, m) \leftarrow xs = nil, n = 0, ys = nil, m = 0 \\ \text{FLD}_{+/-}(xs, n, ys, m) \leftarrow \text{FLD}_{+/-}(xs', n', ys, m), xs = cons(x, xs'), n = n' + x, ys = nil, m = 0 \\ \text{FLD}_{+/-}(xs, n, ys, m) \leftarrow \text{FLD}_{+/-}(xs, n, ys', m'), xs = nil, n = 0, ys = cons(y, ys'), m = m' + y \\ \text{FLD}_{+/-}(xs, n, ys, m) \leftarrow \text{FLD}_{+/-}(xs', n', ys', m'), xs = cons(x, xs'), n = n' + x, ys = cons(y, ys'), m = m' - y \\ \bot \leftarrow \text{FLD}_{+/-}(xs, n, xs, m) \wedge (n + m \neq 0) \end{cases}$$

□

In general, it is impossible to show that $sat_{P'} \implies sat_P$. However, we are still able to demonstrate that whenever P' is satisfiable, then there does not exist a refutation tree of P.

Theorem 2. $unsat_P \iff unsat_{P'}$

Proof (1/2. Sufficiency)**.** Let t' be a refutation tree for P'. Recall that each node in t' is a triple $(C, \vec{v}_0, v)$. Here we use the notation from Def. 14: C'_a stands for clauses with replaced applications of $p_1, \ldots, p_n$ in some CHC C_a, and $(p, rules(p)) = p_1 \times \ldots \times p_n$. Our goal is to show that P is also unsatisfiable. This is done in two steps: first, we build tree t using t' (Part A), and then, we show that t is a refutation tree for P (Part B).

Part A (Building t).

We construct t by *merging* a set T (recall Def. 4), each element of which corresponds to some path from the root to a leaf in t'. Intuitively, T is built as follows. If a path of t' does not pass through C'_a, then we take it as is. Otherwise, we (1) replace node C'_a with C_a, (2) unroll $p_1, \dots, p_n$ simultaneously (thus adding n children, each of which corresponds to the unrolling of p_i) instead of unrolling p. To formally define it, we introduce mappings τ and *split* from paths to trees:

$$\tau([(C, \vec{v}_0, v), \pi]) \stackrel{\text{def}}{=} \begin{cases} [(C, \vec{v}_0, v), \{\tau(\pi)\}], \text{ if } C \neq C'_a & \text{(2a)} \\ [(C_a, \vec{v}_0, v), \mathit{split}(\pi)], \text{ if } C = C'_a & \text{(2b)} \end{cases}$$

Mapping *split* takes a path π and outputs a set of child trees for node $(C_a, \vec{v}_0, v)$. Let $\pi = [(C, \vec{v}_0, v), \pi']$. There could be two possible scenarios: either $C \in \mathit{rules}(p)$, or not. If $C \notin \mathit{rules}(p)$, let $\mathit{split}(\pi) \stackrel{\text{def}}{=} \tau(\pi)$. If $C \in \mathit{rules}(p)$, we continue "unrolling" π' until either a leaf or a node with clause $C' \notin \mathit{rules}(p)$ is reached. Formally, let

$$\pi = \Big[(C^{1\prime}, \vec{v}_0^{\,1}, v^1), \big[\dots [(C^{m\prime}, \vec{v}_0^{\,m}, v^m), \pi']\big]\Big], \tag{3}$$

where $C^{i\prime} = C^i_1 \times_p \dots \times_p C^i_n$, $C^i_j \in P$; and if $\pi' = \{[(C', \cdot, \cdot), \cdot]\}$ then $C' \notin \mathit{rules}(p)$. By Def. 12, $\mathit{head}(C^{i\prime}) = \prod_p p_j(\vec{x}^{\,i}_j)$, which allows us to split the vector of values: $\vec{v}^{\,i}_0$ to $\vec{v}^{\,i}_1 \cdot \dots \cdot \vec{v}^{\,i}_n$, such that for each j, $|\vec{x}^{\,i}_j| = |\vec{v}^{\,i}_j|$. To sum up, in case $C \in \mathit{rules}(p)$, let

$$\mathit{split}(\pi) \stackrel{\text{def}}{=} \left\{ \left[(C^1_i, \vec{v}^{\,1}_i, v^1\big|_{fv(C^1_i)}), \big[\dots [(C^m_i, \vec{v}^{\,m}_i, v^m\big|_{fv(C^m_i)}), \tau(\pi')]\big]\right] \,\middle|\, 1 \leqslant i \leqslant n \right\} \tag{4}$$

Both τ and *split* enjoy the freedom in our paths notation and are defined on $\varnothing$ as $\varnothing$. Now, when τ is formally defined, we can build a set of partial subtrees of t:

$$T \stackrel{\text{def}}{=} \{\tau(\pi) \mid \pi \in \mathit{paths}(t')\} \tag{5}$$

The root node of each tree in T is $\tau(\mathit{root}(t'))$, and thus $|\mathit{roots}(T)| = 1$. By Remark 2, $|\mathit{merge}(T)| = 1$. The only tree in $\mathit{merge}(T)$ is in fact the desired tree t.

Part B (Demonstrating unsat_P).

First, notice that despite (2) does not consider the case of $C \in \mathit{rules}(p)$, τ is still well-defined. In fact, τ is used in three places: for building T in (5), recursively applied in (2) and in (4). In all these three cases, τ is never applied to a path starting from CHC $C \in \mathit{rules}(p)$.

Next, by Def. 8, $\mathit{root}(t') = (q', \varnothing, \cdot)$. By Def. 14, if $q' = C'_a$ then $q = C_a$, or otherwise $q = q'$. In both cases, $\mathit{root}(\tau(\pi)) = (q, \varnothing, \cdot)$ for each $\pi \in \mathit{paths}(t')$ by (2). On the other hand, $\mathit{root}(t) = \mathit{root}(\tau(\pi))$ for some $\pi \in \mathit{paths}(t')$, implying that $\mathit{root}(t) = (q, \varnothing, \cdot)$. It remains to show that t realizes q, which by Def. 8 immediately implies unsat_P. Thus, in the rest of the proof, we demonstrate that conditions $\boxed{1}$, $\boxed{2}$, $\boxed{3}$, and $\boxed{4}$ of Def. 7 are fulfilled for t.

$\boxed{1}$ is obviously fulfilled by construction of t: every node added in (2) and (4) is a triple $(C, \vec{v_0}, v)$ with $C \in P$ and $\vec{v_0}$ is of the appropriate length; domain of v is always $fv(C)$: in (2) $fv(C_a) = fv(C'_a)$, and in (4) v is explicitly bound to $fv(C)$.

To ensure that $\boxed{2}$ is fulfilled, consider two cases: whether $C \in \mathit{rules}(p)$ or not. In case if $C \in P' \backslash \mathit{rules}(p)$, the corresponding CHC in P has the same constraint ϕ. In this case, (2)

implies that τ preserves variables assignment ($\vec{v_0}$ and v are the same in both sides of equation), keeping constraint evaluate to $\top$. In the other case, $C \in rules(p)$, (4) splits the path to n branches with separate variable assignments. Each v^i assigns value for every free variable of $C^{i'}$. By Def. 12 the constraint in each $C^{i'}$ is the conjunction of constraints in $\{C^i_j\}$. By condition 2 of Def. 7 for t', the product-constraint evaluates to $\top$ with assignment $\vec{v}^i$. Also by Remark 1 all variables in C^i_j are disjoint. This immediately implies that the constraint in each CHC C^i_j evaluates to $\top$ with $v^i\big|_{fv(C^i_1)}$. That is, all constraints appearing in t are satisfied by the variable assignments.

To confirm that 3 is fulfilled, given $t = [(C, \cdot, v), T]$ with $C \in P$, we must show that for each $r(\vec{x}) \in body(C)$ there is $t_r \in T$ with $root(t_r) = (C_r, \vec{v_i}, \cdot)$ for some $C_r \in rules(r)$ (the second condition $\vec{v_i}[j] = v(\vec{x_i}[j])$ is trivially fulfilled by construction). In other words, for each node $(C, \cdot, \cdot)$ and $r(\vec{x}) \in body(C)$, we should demonstrate that there is a path $\big[(C, \cdot, \cdot), [(C_r, \cdot, \cdot), \pi']\big] \in paths(t)$ with $C_r \in rules(r)$.

The main idea is as follows. Note that τ maps each node in t' to one or more nodes in t (by either preserving path or splitting it in applications of p). Thus for each node t there is a "source" node in t'. Since t' is a refutation tree for P', all applications of relation symbols are realized by some subtree; τ-image of some child subtree back to t gives us the desired path. More specifically, fix $c = (C, \cdot, \cdot)$ and $r(\vec{x}) \in body(C)$, and let c' be the "source" of c in t'. There are three possible cases:

(a) If $C \neq C_a$ and $rel(head(C)) \notin \{p_1, \ldots, p_n\}$, then by Def. 14, $C \in P \cap P'$, hence $C \in P'$. This implies that $c' = (C, \cdot, \cdot)$ (and c is produced by (2a)). As t' is a refutation tree for P', there is path $\pi = \big[(C, \cdot, \cdot), [(C_r, \cdot, \cdot), \pi']\big]$ in t' with $C_r \in rules(r)$. Partial subtree $\tau(\pi)$ gives us the desired path in t.

(b) If $C = C_a$ then t' has node $c' = (C'_a, \cdot, \cdot)$. If $r \notin \{p_1, \ldots, p_n\}$ then we proceed similarly to *(a)*. Otherwise, $r = p_i$, and thus, there is a path $\pi = \big[(C'_a, \cdot, \cdot), [(C_p, \cdot, \cdot), \pi']\big] \in paths(t')$, where $C_p \in rules(p)$. By (5), $\tau(\pi) = \Big[(C_a, \cdot, \cdot), \big\{[(C^j_1, \cdot, \cdot), T_1], \ldots, [(C^j_n, \cdot, \cdot), T_n]\big\}\Big]$ giving the desired path $\big[c, [(C_i, \cdot, \cdot), \ldots]\big] \in paths(t)$ with $C_i \in rules(p_i)$.

(c) If $rel(head(C)) \in \{p_1, \ldots, p_n\}$, then node c is added to t either in (2a) or in (4) as the result of splitting. The former case is exactly the same as *(a)*. In the latter case, $c' = (C', \cdot, \cdot)$ is an element of some path (3), where $C' = C_1 \times_p \ldots \times_p C_n$ and $C = C_i$ for some i (with $C_i \in rules(p_i)$). Now either $r = p_i$, and thus $r(\vec{x}) \in R_i \subseteq R_{/_{head}}(C_i)$ (r is the recursive application in C_i), or $r \neq p_i$, and thus $r \in L_i$. In both cases, by Def. 12 there is a corresponding relation application in C' (recall $R = \big\{\prod_p r_i \mid (r_1, \ldots r_n) \in \big[R_{/_{head}}(C_1) \times \ldots \times R_{/_{head}}(C_n)\big]\big\}$ and $L = \bigcup_{i=1}^{n} L_i$); denote it $r'(\vec{x'})$. If $r = p_i$ then $r' = p$, and otherwise $r' = r$. Since t' is a refutation tree, $r'(\vec{x'})$ is realizable by some subtree. If $r \neq p_i$ then c' is the last p-node of unrolling (3), and then $\tau(\pi')$ is the desired subtree of t. Otherwise, $r(\vec{x})$ is among n paths created by *split* in (4) for each application of p_i. This accomplishes the proof of 3.

Finally note that proofs for 1, 2, and 3 are formulated for an arbitrary clause C. This automatically implies that all partial subtrees of t realize their clauses. Thus, 4 also holds.

Proof (2/2. Necessity)**.** The general flow of proving necessity is similar to the one for sufficiency: first merge paths (possibly branched to n paths in C_a) of t to t' and then show that t'

Algorithm 1: CHCPRODUCT

Input: system (P, q) of CHCs, operator PARTITION from a set to a set of its disjoint subsets
Output: system (P', q') of CHCs
Data: worklist WL of CHCs

1 $P' \leftarrow \varnothing$;
2 $WL \leftarrow WL \cup q$;
3 **while** $(\neg$EMPTY$(WL))$ **do**
4 $\quad C_a \leftarrow$ GET(WL);
5 $\quad Prt \leftarrow$ PARTITION(GETNONRECPART(C_a));
6 $\quad$ **foreach** $\big(\{p_1(\vec{x_1}), \ldots, p_n(\vec{x_n})\} \in Prt\big)$ **do**
7 $\quad\quad (p, rules(p)) \leftarrow p_1 \times \ldots \times p_n$;
8 $\quad\quad C_a \leftarrow$ REPLACE$\Big(C_a,\ p_1(\vec{x_1}), \ldots, p_n(\vec{x_n}),\ \prod\limits_p p_i(\vec{x_i})\Big)$;
9 $\quad\quad$ **foreach** $(C \in rules(p))$ **do** $WL \leftarrow WL \cup C$;
10 $\quad$ **end**
11 $\quad P' \leftarrow P' \cup C_a$;
12 **end**

is refutation for P'. The main problem here is that different paths of t have different lengths: when fact clause is reached in one of n branches that does not mean that it is reached in other branches. Formally, case of $R = \varnothing$ in Def. 11 stays uncovered. This problem is solved by tautological extension of P and Lemma 2. We skip the text of the proof due to the similarity to the proof of sufficiency. □

Theorem 3. $sat_{P'} \implies \neg unsat_P$

Proof. This fact is immediately implied by Lemma 1 and Th. 2. □

5 Connecting with Practice

Our CHC transformation approach, called CHCPRODUCT, is outlined in Alg. 1. It takes an original system (P, q) of CHCs as input and outputs a merged system (P', q') of CHCs, that further can be processed with an off-the-shelf constrained Horn solver. The algorithm maintains a worklist WL of CHCs and initializes WL by the query q (line 2). In each iteration, the algorithm pops an element C_a from WL (line 4), modifies it (line 8), and inserts to P' (line 11).

For each C_a, the algorithm identifies the non-recursive part L from its body and splits it into disjoint subsets (line 5). Then, for each set $\{p_1, \ldots, p_n\}$, it performs the p-transformation of the original system (recall Def. 14), in which $(p, rules(p)) \leftarrow p_1 \times \ldots \times p_n$. It is likely that simultaneous applications of $\{p_1, \ldots, p_n\}$ appear along the algorithm's run several times (e.g., in Ex. 5). It is sufficient to replace each such appearance with a new application of the same p.

Importantly, the bodies of $rules(p)$ (line 9) could contain multiple applications of the other relation symbols, for which the new products could be potentially created. The algorithm considers them in the next iterations. Obviously, the algorithm can be optimized, so it inserts each of $rules(p)$ to WL exactly once.

Example 5. Consider the system of CHCs that encodes the recurrent computation of n^n and $n!$ for any number $n > 1$:

$$\begin{cases} \textsc{mlt}(u,v,v') \leftarrow u=0, v'=0 \\ \textsc{fct}(n,x') \leftarrow n=0, x'=1 \\ \textsc{pwr}(m,y,y') \leftarrow m=0, y'=1 \\ \textsc{mlt}(u,v,v') \leftarrow \textsc{mlt}(u-1,v,w), u\neq 0, v'=v+w \\ \textsc{fct}(n,x') \leftarrow \textsc{fct}(n-1,a), \textsc{mlt}(a,n,x'), n\neq 0 \\ \textsc{pwr}(m,y,y') \leftarrow \textsc{pwr}(m-1,y,b), \textsc{mlt}(b,y,y'), m\neq 0 \\ \bot \leftarrow \textsc{fct}(n,x'), \textsc{pwr}(n,n,y'), n>1, x'\geqslant y' \end{cases}$$

The system uses three relation symbols, MLT, FCT, and PWR, for the multiplication, factorial, and power, respectively. $\textsc{mlt}(u,v,v')$ defines the computation of $v'=0+\underbrace{v+\ldots+v}_{u}$, which is equivalent to $u\cdot v$; $\textsc{fct}(n,x')$ defines the computation of $x'=1\cdot\underbrace{1\cdot\ldots\cdot n}_{n}$, which is equivalent to $n!$, and $\textsc{pwr}(m,y,y')$ defines the computation of $y'=1\cdot\underbrace{y\cdot\ldots\cdot y}_{m}$, which is equivalent to y^m. The query establishes a property about the results of these computations. The interpretations for MLT, FCT, and PWR exist iff $\forall n$, s.t. $n>1$, $\neg(n!\geqslant n^n)$ holds.

Algorithm CHCPRODUCT makes the system solvable in linear arithmetic. The algorithm starts with the query, which has the conjunction of applications of FCT and PWR, produces a product $(\textsc{fp}, \mathit{rules}(\textsc{fp})) = \textsc{fct}\times\textsc{pwr}$, and rewrites the query accordingly. After the first iteration, WL contains the following CHCs:

$$\begin{cases} \textsc{fp}(n,x',m,y,y') \leftarrow n=0, x'=1, m=0, y'=1 \\ \textsc{fp}(n,x',m,y,y') \leftarrow \textsc{fp}(n-1,a,m,y,y'), \textsc{mlt}(a,n,x'), n\neq 0, m=0, y'=1 \\ \textsc{fp}(n,x',m,y,y') \leftarrow \textsc{fp}(n,x',m-1,y,b), n=0, x'=1, \textsc{mlt}(b,y,y'), m\neq 0 \\ \textsc{fp}(n,x',m,y,y') \leftarrow \textsc{fp}(n-1,a,m-1,y,b), \textsc{mlt}(a,n,x'), n\neq 0, \textsc{mlt}(b,y,y'), m\neq 0 \\ \bot \leftarrow \textsc{fp}(n,x',n,n,y'), n>1, x'\geqslant y' \end{cases}$$

In the next iteration, CHCPRODUCT processes all the CHCs in WL including the following:

$$\textsc{fp}(n,x',m,y,y') \leftarrow \textsc{fp}(n-1,a,m-1,y,b), \textsc{mlt}(a,n,x'), n\neq 0, \textsc{mlt}(b,y,y'), m\neq 0$$

The CHC has the conjunction of applications of MLT and MLT and thus, the algorithm produces a product $(\textsc{mlt}^2, \mathit{rules}(\textsc{mlt}^2)) = \textsc{mlt}\times\textsc{mlt}$ similarly to $(\textsc{fp}, \mathit{rules}(\textsc{fp})) = \textsc{fct}\times\textsc{pwr}$ from the previous iteration. Note that in order to use Def. 12, which disallows self-products, the algorithm preliminarily duplicates $\mathit{rules}(\textsc{mlt})$ and makes all variables unique (to comply with Remark 1) by adding indexes $\{1,2\}$ to the variables. The final system is as follows:

$$\begin{cases} \textsc{mlt}(u,v,v') \leftarrow u=0, v'=0 \\ \textsc{mlt}(u,v,v') \leftarrow \textsc{mlt}(u-1,v,w), u\neq 0, v'=v+w \\ \textsc{mlt}^2(u_1,v_1,v_1',u_2,v_2,v_2') \leftarrow u_1=0, v_1'=0, u_2=0, v_2'=0 \\ \textsc{mlt}^2(u_1,v_1,v_1',u_2,v_2,v_2') \leftarrow \textsc{mlt}^2(u_1-1,v_1,w_1,u_2,v_2,v_2'), u_1\neq 0, v_1'=v_1+w_1, u_2=0, v_2'=0 \\ \textsc{mlt}^2(u_1,v_1,v_1',u_2,v_2,v_2') \leftarrow \textsc{mlt}^2(u_1,v_1,v_1',u_2-1,v_2,w_2), u_1=0, v_1'=0, u_2\neq 0, v_2'=v_2+w_2 \\ \textsc{mlt}^2(u_1,v_1,v_1',u_2,v_2,v_2') \leftarrow \textsc{mlt}^2(u_1-1,v_1,w_1,u_2-1,v_2,w_2), u_1\neq 0, v_1'=v_1+w_1, u_2\neq 0, v_2'=v_2+w_2 \\ \textsc{fp}(n,x',m,y,y') \leftarrow n=0, x'=1, m=0, y'=1 \\ \textsc{fp}(n,x',m,y,y') \leftarrow \textsc{fp}(n-1,a,m,y,y'), \textsc{mlt}(a,n,x'), n\neq 0, m=0, y'=1 \\ \textsc{fp}(n,x',m,y,y') \leftarrow \textsc{fp}(n,x',m-1,y,b), n=0, x'=1, \textsc{mlt}(b,y,y'), m\neq 0 \\ \textsc{fp}(n,x',m,y,y') \leftarrow \textsc{fp}(n-1,a,m-1,y,b), \textsc{mlt}^2(a,n,x',b,y,y'), n\neq 0, m\neq 0 \\ \bot \leftarrow \textsc{fp}(n,x',n,n,y'), n>1, x'\geqslant y' \end{cases}$$

Note that each CHC has at most one application of the relation symbol in the non-recursive part. A solution to the system (i.e., interpretation to symbols MLT, MLT^2, and FP) is expressible in linear arithmetic. In contrast, original system would require an invariant for FCT and PWR over non-linear arithmetic, which is hard (if not practically impossible) to find. □

Discussion. A prominent feature of the algorithm is that it cleans the transformed system off the dead relation symbols and unused CHCs (recall Remark 3), likely simplifying the task of its further solving. Some degree of the algorithm's success is also due to an implementation of method PARTITION that specifies the grouping of the predicate applications. In our experience, it should be guided by the prior analysis of the structure of each set $rules(p_i)$. For instance, a product of a recurrent and a non-recurrent relation (e.g., p and r in Ex. 1) would not bring any benefits, since it is just easier to unfold the latter once [3].

The particular choice of coverage (Def. 9) in the p-transformation allows to overcome the challenge of merging non-trivial recurrences. That is, if the recursive part of the body of some CHC has more than two applications (like for Fibonacci numbers in Ex. 6) then the coverage (Def. 9) is not unique. However, it makes sense to choose the coverage consistently with how the inductive arguments evolve. This is closely referred to the concept of *synchronous product* that is discussed in details in the next section.

6 Synchronous Product of CHCs

The broadly defined product of CHCs does not prevent the situation when for the same original system of CHCs, there are multiple transformed systems that satisfy Def. 12. But not all such systems can be efficiently solved. In this section, we analyze the properties of CHCs and refine our theoretical concept of product into a more practical concept of synchronous product.

Example 6. Consider the system of CHCs that verifies the monotonicity of Fibonacci numbers:

$$\begin{cases} \text{FIB}(n, x) \leftarrow x = 1, n < 2 \\ \text{FIB}(n, x) \leftarrow \text{FIB}(n-1, x'), \text{FIB}(n-2, x''), x = x' + x'', n \geqslant 2 \\ \qquad \bot \leftarrow \text{FIB}(n_1, x_1), \text{FIB}(n_2, x_2), (n_1 < 2 \wedge n_2 < 2) \vee n_2 \geqslant n_1, x_2 < x_1 \end{cases}$$

In order to construct the product $(\text{FIB}^2, rules(\text{FIB}^2)) = \text{FIB} \times \text{FIB}$, we preliminarily duplicate $rules(\text{FIB})$ and add indexes $\{1, 2\}$ to all variables (otherwise Def. 12 would not be applicable). The difficulty starts when merging recursive clauses requiring to choose a particular coverage of $\big\lfloor(\text{FIB}(n_1 - 1, x_1'), \text{FIB}(n_1 - 2, x_1'')), (\text{FIB}(n_2 - 1, x_2'), \text{FIB}(n_2 - 2, x_2''))\big\rfloor$. This gives rise to two possible systems of CHCs, but only one of them is solvable by the constrained Horn solvers for linear arithmetic:

$$\begin{cases} \text{FIB}^2(n_1, x_1, n_2, x_2) \leftarrow x_1 = 1, n_1 < 2, x_2 = 1, n_2 < 2 \\ \text{FIB}^2(n_1, x_1, n_2, x_2) \leftarrow \text{FIB}^2(n_1, x_1, n_2{-}1, x_2'), \text{FIB}^2(n_1, x_1, n_2{-}2, x_2''), x_1 = 1, x_2 = x_2'{+}x_2'', n_1 < 2, n_2 \geqslant 2 \\ \text{FIB}^2(n_1, x_1, n_2, x_2) \leftarrow \text{FIB}^2(n_1{-}1, x_1', n_2, x_2), \text{FIB}^2(n_1{-}2, x_1'', n_2, x_2), x_1 = x_1'{+}x_1'', x_2 = 1, n_1 \geqslant 2, n_2 < 2 \\ \text{FIB}^2(n_1, x_1, n_2, x_2) \leftarrow \text{FIB}^2(n_1{-}1, x_1', n_2{-}1, x_2'), \text{FIB}^2(n_1{-}2, x_1'', n_2{-}2, x_2''), x_1 = x_1'{+}x_1'', x_2 = x_2'{+}x_2'', n_1 \geqslant 2, n_2 \geqslant 2 \\ \qquad \bot \leftarrow \text{FIB}^2(n_1, x_1, n_2, x_2), (n_1 < 2 \wedge n_2 < 2) \vee n_2 \geqslant n_1, x_2 < x_1 \end{cases}$$

□

Before we introduce the concept of synchronous product of relations that explains the choice of the coverage and some other details of FIB^2-product, we need a quick detour through the graph theory.

Definition 15. Given undirected unweighted graph $G = \langle V, E \rangle$, set $C \subseteq 2^V$ is called *n-clique-cover* of G iff:

1. $\forall c \in C, |c| = n$;
2. $\forall c \in C, \forall v_1, v_2 \in c, v_1 \neq v_2 \implies (v_1, v_2) \in E$, i.e., every subgraph induced by a set of nodes c is a subclique of G;
3. $\forall v \in V \implies \exists c \in C, v \in c$.

We denote a set of all n-clique-covers of G by $\mathcal{C}^n(G)$. We say that G is *n-clique-coverable*, if $\mathcal{C}^n(G) \neq \varnothing$.

Example 7. Three bipartite graphs in Fig. 1 are all 2-clique-coverable. Note that for each 2-clique-coverable bipartite graph $G = \langle V, E \rangle$, $E \in \mathcal{C}^2(G)$. □

Given two finite sets A_1 and A_2, such that $A_1 \cap A_2 = \varnothing$, then any symmetric binary relation $\omega \subseteq A_1 \times A_2$ defines an undirected unweighted bipartite graph $G = \langle A_1 \cup A_2, \omega \rangle$: $x_1 \in A_1$ and $x_2 \in A_2$ are connected in G iff $\omega(x_1, x_2)$ holds. More generally, for $1 \leqslant i < j \leqslant n$, finite sets $A_1, \ldots, A_n$ such that $A_i \cap A_j = \varnothing$ and symmetric binary relations $\omega_{ij} \subseteq A_i \times A_j$ determine an undirected n-partite graph $G = \bigcup\limits_{1 \leqslant i < j \leqslant n} \langle A_i \cup A_j, \omega_{ij} \rangle$. We denote $G = \langle \bigcup\limits_{i=1}^{n} A_i, \cdot \rangle$ if ω_{ij} are not important in the context.

Lemma 3. *Let $G = \langle \bigcup\limits_{i=1}^{n} A_i, \cdot \rangle$ be n-partite graph. Then*

$$A \in \mathcal{C}^n(G) \implies A \in \lfloor A_1, \ldots, A_n \rfloor$$

Now we are ready to formally introduce the concept of *synchronous product* of two relations.

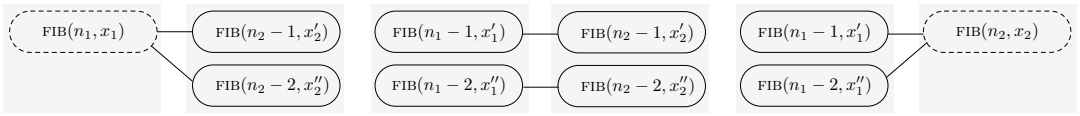
Definition 16. Let C_p and C_q be CHCs with heads $p(\vec{x})$ and $q(\vec{y})$ respectively. Let $body(C_p) = \{\phi_p\} \cup L_p \cup R_p$ and $body(C_q) = \{\phi_q\} \cup L_q \cup R_q$. Let binary relation ω_{pq} on recursive applications of relation symbols be defined as follows:

$$\omega_{pq}(a, b) \stackrel{\text{def}}{=} \vdash \phi_p \wedge \phi_q \wedge \bigwedge(L_p \cup L_q) \implies \Big(\rho_{pq}(\vec{x} \cdot \vec{y}) \implies \rho_{pq}\big(args(a) \cdot args(b)\big)\Big) \tag{6}$$

where ρ_{pq} is some fixed-arity relation. Clauses C_p and C_q are called *synchronized* by ρ_{pq} iff graph $G(C_p, C_q)$ is 2-clique-coverable:

$$G(C_p, C_q) \stackrel{\text{def}}{=} \langle R_{/_{head}}(C_p) \cup R_{/_{head}}(C_q), \omega_{pq} \rangle$$

We say that two relations are synchronized by ρ_{pq} if the recursive calls of p and q can iterate simultaneously, i.e., some arguments of p and arguments in q evolve (e.g., increment, decrement, or stay the same) consistently to relation ρ_{pq} that is used in (6) and could be derived from the query of the system of CHCs. That is, if p and q are initialized with two values related via ρ_{pq} and both terminate, then ρ_{pq} is preserved during the entire computation process. Binary relation ω_{pq} determines the preservation of ρ_{pq} of those values via checking the validity of implication in (6). In practice, computations are often synchronized, e.g., two iterators over lists are synchronized by the list-specifying parameter (in this case, ρ_{pq} equates lists). The same is true for any recursive data structure.



(a) (b) (c)

Figure 1: Applications of FIB connected while constructing the FIB^2-product in Ex. 6 (dashed ovals correspond to the cases when $R_{/_{head}}(C)$ is composed of $head(C)$).

Definition 17. Given CHCs $C_1, \ldots, C_n$ and family of relations $\{\rho_{ij}\}$, such that C_i and C_j are synchronized by ρ_{ij} for $1 \leqslant i < j \leqslant n$. A *synchronous product* of $C_1, \ldots, C_n$ on ρ_{ij} and p is defined similarly to product $C_1 \times_p \ldots \times_p C_n$ (recall Def. 12) with the only difference in how the recursive applications are matched:

$$R \stackrel{\text{def}}{=} \left\{ \prod_p r_i \;\middle|\; (r_1, \ldots, r_n) \in \mathcal{C}^n\Big(\bigcup_{1 \leqslant i < j \leqslant n} G(C_i, C_j) \Big) \right\}$$

Similarly, we extend the definition of product of relation symbols $p_1, \ldots, p_n$ (recall Def. 13) to the synchronous case, denoted $\bowtie_f (p_1, \ldots, p_n)$. Intuitively, by introducing synchronous product we use a subset of the synchronization information that can be mined from the given system of CHCs via the additional queries to the solver. This information identifies connections among recursive applications, thus simplifying the task of discovering an inductive invariant for the transformed system.

It can be noticed that (6) is a tautology in case when $\{\rho_{ij}\}$ are true on all inputs. In this case, synchronous product does not restrict any connections. Such weakest possible synchronous product is mnemonically denoted $\bowtie_\top (p_1, \ldots, p_n)$.

Remark 4. $\bowtie_\top (p_1, \ldots, p_n) = p_1 \times \ldots \times p_n$.

Further simplifications of the system are due to C_a from Def. 14, which drives the entire transformation. In our experiments, we mine relations $\{\rho_{ij}\}$ from the constraint of C_a, in particular, by picking a subset (or the entire set, if possible) of its conjuncts satisfying (6).

Example 8. Let us demonstrate how the synchronous product affects matching the recursive premises in Ex. 6, where C_a is the query CHC. Because C_a assumes that $(n_1 < 2 \wedge n_2 < 2) \vee n_2 \geqslant n_1$ is true, we let $\rho(x, y) \stackrel{\text{def}}{=} (x < 2 \wedge y < 2) \vee y \geqslant x$. In the initial state $\rho(n_1, n_2)$ is true, and the fact CHCs are merged trivially, by conjoining the constraints.

Two CHCs, FIB$(n_1, x_1) \leftarrow x_1 = 1, n_1 < 2$ and FIB$(n_2, x_2) \leftarrow$ FIB$(n_2 - 1, x_2')$, FIB$(n_2 - 2, x_2'')$, $x = x_2' + x_2'', n_2 \geqslant 2$, which have three applications of FIB, are matched respectively to Fig. 1(a): the edge between nodes FIB(n_1, x_1) and FIB$(n_2 - 1, x_2')$ is identified because the following implication holds:

$$\vdash n_1 < 2 \wedge n_2 \geqslant 2 \wedge \ldots \implies ((n_1 < 2 \wedge n_2 < 2) \vee n_2 \geqslant n_1 \implies (n_1 < 2 \wedge \boldsymbol{n_2 - 1} < 2) \vee \boldsymbol{n_2 - 1} \geqslant n_1)$$

the second edge between nodes FIB(n_1, x_1) and FIB$(n_2 - 2, x_2'')$ is identified similarly:

$$\vdash n_1 < 2 \wedge n_2 \geqslant 2 \wedge \ldots \implies ((n_1 < 2 \wedge n_2 < 2) \vee n_2 \geqslant n_1 \implies (n_1 < 2 \wedge \boldsymbol{n_2 - 2} < 2) \vee \boldsymbol{n_2 - 2} \geqslant n_1)$$

The next case is trickier since two non-fact CHCs are matched (Fig. 1(b)). Thus, there are four applications of FIB; two edges need to be identified, and there are four candidates for them.

We proceed with connecting $\textsc{fib}(n_1 - 1, x_1')$ to either $\textsc{fib}(n_2 - 1, x_2')$ or $\textsc{fib}(n_2 - 2, x_2'')$. There are two formulas. The first one holds:

$$\vdash n_1 \geqslant 2 \wedge n_2 \geqslant 2 \wedge \ldots \implies ((n_1 < 2 \wedge n_2 < 2) \vee n_2 \geqslant n_1 \implies (n_1 - 1 < 2 \wedge \boldsymbol{n_2 - 1} < 2) \vee \boldsymbol{n_2 - 1} \geqslant n_1 - 1)$$

while the second one does not hold (e.g., on $n_1 = n_2 = 3$):

$$\vdash n_1 \geqslant 2 \wedge n_2 \geqslant 2 \wedge \ldots \implies ((n_1 < 2 \wedge n_2 < 2) \vee n_2 \geqslant n_1 \implies (n_1 - 1 < 2 \wedge \boldsymbol{n_2 - 2} < 2) \vee \boldsymbol{n_2 - 2} \geqslant n_1 - 1)$$

These checks are enough to identify the only edge between the synchronized applications $\textsc{fib}(n_1 - 1, x_1')$ and $\textsc{fib}(n_2 - 1, x_2')$ (Fig. 1(b), upper part). By exclusion, we identify the remaining edge between the synchronized applications $\textsc{fib}(n_1 - 2, x_1'')$ and $\textsc{fib}(n_2 - 2, x_2'')$ (Fig. 1(b), lower part).

Finally, we match one non-fact CHC and one fact CHC (Fig. 1(c)). There are two formulas which hold because of the contradicting premises, thus identifying both edges:

$$\vdash n_1 \geqslant 2 \wedge n_2 < 2 \wedge \ldots \implies ((n_1 < 2 \wedge n_2 < 2) \vee n_2 \geqslant n_1 \implies (n_1 - 1 < 2 \wedge n_2 < 2) \vee n_2 \geqslant n_1 - 1)$$

and

$$\vdash n_1 \geqslant 2 \wedge n_2 < 2 \wedge \ldots \implies ((n_1 < 2 \wedge n_2 < 2) \vee n_2 \geqslant n_1 \implies (n_1 - 2 < 2 \wedge n_2 < 2) \vee n_2 \geqslant n_1 - 2)$$

□

7 Evaluation

We evaluated our transformation on a set of challenging systems of CHCs originated from the relational verification tasks over functional programs. Our transformation is implemented on top of the Rosette/Unbound verifier for Racket code. We compare it to the CHC transformation based on the folding / unfolding rules implemented in the VeriMAPrel [3] tool designed to handle C code. Since our goal is to compare performance of constrained Horn solving only, we use the model checking tools to produce and to transform the CHCs (by giving equivalent programs in Racket and C, respectively, as input to the tools). Finally, we use the off-the-shelf Horn solver, Spacer3 [12], to check satisfiability of both systems using the PDR engine over linear integer arithmetic (LIA).

We use 47 benchmarks over LIA (crafted and adapted from [3]) and 23 benchmarks over lists.[1] Since the underlying solver does not support the theory of lists yet, we rely on Rosette/Unbound to over-approximate systems of CHCs from lists to LIA. Intuitively, lists are abstracted by sequences of nondeterministically treated integer variables. That is, whenever a list element appears in the body of some rule, it is replaced by a fresh integer symbol. To support synchronization of several iterators over the same list, we explicitly propagate equalities of the heads and the tails to the product CHCs, and thus we decrease the level of nondeterminism in the transformed system. For all our examples, this abstraction was shown expressive enough to prove the properties of interest, such as the one shown in Ex. 4. We refer the reader to consider the Racket code of our benchmarks for more details.

The result of the experimental comparison is shown in Fig. 2. Each point in the scatterplot represents a pair of CHC solving runs for the same benchmark: after the product transformation

[1] The tool and the benchmarks can be found at https://github.com/dvvrd/rosette.

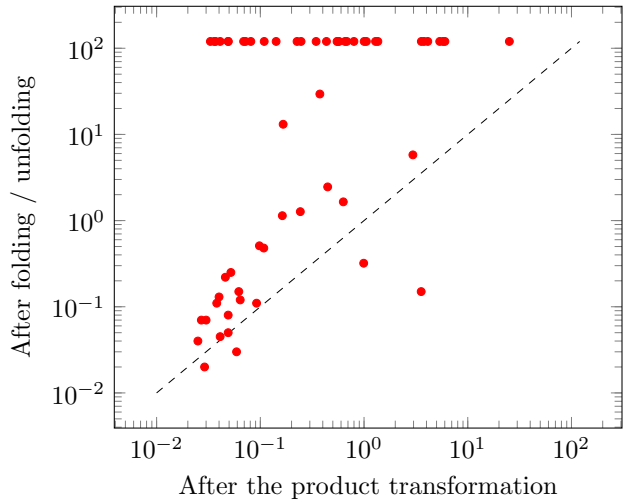


Figure 2: Solving the transformed systems of CHCs (sec).

(x-axis) and after applying the folding / unfolding rules (y-axis). Most of the original systems were not solvable before the transformation. Our approach outperforms the competitor in all but 4 cases. The transformed systems of CHCs appeared to be linear more often than the competing systems (however, we do not guarantee that the result of our transformation is small in general). Note that in most of the cases, our transformed systems were solved within a second, while for the corresponding competing system, the transformation was either unsupported, or the solving exceeded a timeout. That said, some of the performance gap can be justified also by the gap between the imperative and functional semantics of the programming languages (i.e., the same programs can be encoded using loops or recursion) and the low-level encoding routines hidden in the VeriMAPrel and Rosette/Unbound tools. In future, it would make sense to implement the folding / unfolding rules on top of the Rosette/Unbound framework and perform a more fair comparison with the synchronous product transformation.

8 Conclusion

We presented an approach to transform non-linear systems of CHCs by synchronizing recurrent computations. The approach is based on the notion of the CHC product that defines the strategies of replacing individual relation symbols and rules in the system of CHCs by fresh, joint relation symbols. Our transformation results in smaller systems of CHCs (compared to the systems transformed by the classical folding / unfolding rules) and enables their fast satisfiability checking by the external constrained Horn solver. We integrated the transformation in the Rosette/Unbound tool for verifying functional programs and empirically showed that it performs well on a set of non-trivial programs handling lists and linear arithmetic.

Acknowledgments. We are grateful to Fabio Fioravanti for insightful discussions and for providing the access to the VeriMAPrel tool. The work is supported in parts by the SNSF Fellowship P2T1P2_161971 and the CCC Postdoc Best Practices award from the NSF grant CCF-1136996.

References

[1] G. Barthe, J. M. Crespo, and C. Kunz. Relational verification using product programs. In *FM*, volume 6664 of *LNCS*, pages 200–214. Springer, 2011.

[2] R. M. Burstall and J. Darlington. A transformation system for developing recursive programs. *J. ACM*, 24(1):44–67, 1977.

[3] E. De Angelis, F. Fioravanti, A. Pettorossi, and M. Proietti. Relational Verification Through Horn Clause Transformation. In *SAS*, volume 9837 of *LNCS*, pages 147–169, 2016.

[4] S. Etalle and M. Gabbrielli. Transformations of CLP modules. *Theor. Comput. Sci.*, 166(1&2):101–146, 1996.

[5] G. Fedyukovich, M. B. S. Ahmad, and R. Bodík. Gradual Synthesis for Static Parallelization of Single-Pass Array-Processing Programs. In *PLDI*. ACM, 2017. to appear.

[6] G. Fedyukovich, A. Gurfinkel, and N. Sharygina. Property directed equivalence via abstract simulation. In *CAV*, volume 9780, Part II, pages 433–453. Springer, 2016.

[7] S. Grebenshchikov, N. P. Lopes, C. Popeea, and A. Rybalchenko. Synthesizing software verifiers from proof rules. In *PLDI*, pages 405–416. ACM, 2012.

[8] K. Hoder and N. Bjørner. Generalized property directed reachability. In *SAT*, volume 7317, pages 157–171. Springer, 2012.

[9] H. Hojjat, F. Konecný, F. Garnier, R. Iosif, V. Kuncak, and P. Rümmer. A verification toolkit for numerical transition systems - tool paper. In *FM*, volume 7436 of *LNCS*, pages 247–251. Springer, 2012.

[10] B. Kafle, J. P. Gallagher, and P. Ganty. Solving non-linear Horn clauses using a linear Horn clause solver. In *HCVS*, volume 219 of *EPTCS*, pages 33–48, 2016.

[11] B. Kafle, J. P. Gallagher, and J. F. Morales. Rahft: A tool for verifying Horn clauses using abstract interpretation and finite tree automata. In *CAV, Part I*, volume 9779 of *LNCS*, pages 261–268. Springer, 2016.

[12] A. Komuravelli, A. Gurfinkel, and S. Chaki. SMT-Based Model Checking for Recursive Programs. In *CAV*, volume 8559 of *LNCS*, pages 17–34, 2014. https://bitbucket.org/spacer/code/branch/spacer3.

[13] S. K. Lahiri, K. L. McMillan, R. Sharma, and C. Hawblitzel. Differential assertion checking. In *FSE*, pages 345–355. ACM, 2013.

[14] D. Mordvinov and G. Fedyukovich. Verifying Safety of Functional Programs with Rosette/Unbound. *CoRR*, abs/1704.04558, 2017. https://github.com/dvvrd/rosette.

[15] P. Rümmer, H. Hojjat, and V. Kuncak. Disjunctive interpolants for Horn-Clause verification. In *CAV*, volume 8044 of *LNCS*, pages 347–363. Springer, 2013.

[16] O. Strichman and M. Veitsman. Regression verification for unbalanced recursive functions. In *FM*, volume 9995 of *LNCS*, pages 645–658, 2016.

[17] H. Unno and T. Terauchi. Inferring simple solutions to recursion-free Horn Clauses via sampling. In *TACAS*, volume 9035 of *LNCS*, pages 149–163. Springer, 2015.