# Decomposition by tree dimension in Horn clause verification [*]

Bishoksan Kafle
Roskilde University, Denmark
kafle@ruc.dk

John P. Gallagher
Roskilde University, Denmark
IMDEA Software Institute, Spain
jpg@ruc.dk

Pierre Ganty
IMDEA Software Institute, Spain
pierre.ganty@imdea.org

In this paper we investigate the use of the concept of *tree dimension* in Horn clause analysis and verification. The dimension of a tree is a measure of its non-linearity – for example a list of any length has dimension zero while a complete binary tree has dimension equal to its height. We apply this concept to trees corresponding to Horn clause derivations. A given set of Horn clauses $P$ can be transformed into a new set of clauses $P^{\leq k}$, whose derivation trees are the subset of $P$'s derivation trees with dimension at most $k$. Similarly, a set of clauses $P^{>k}$ can be obtained from $P$ whose derivation trees have dimension at least $k+1$. In order to prove some property of all derivations of $P$, we systematically apply these transformations, for various values of $k$, to decompose the proof into separate proofs for $P^{\leq k}$ and $P^{>k}$ (which could be executed in parallel). We show some preliminary results indicating that decomposition by tree dimension is a potentially useful proof technique. We also investigate the use of existing automatic proof tools to prove some interesting properties about dimension(s) of feasible derivation trees of a given program.

**Keywords:** Tree dimension, proof decomposition, program transformation, Horn clauses.

## 1 Introduction

In this paper, we study the role of *tree dimension* in Horn clause analysis and verification. The dimension of a tree is a measure of its non-linearity – for example a list of any length has dimension zero while a complete binary tree has dimension equal to its height. We apply this concept to trees corresponding to Horn clause derivations. A given set of Horn clauses $P$ can be transformed into a new set of clauses $P^{\leq k}$ (whose derivation trees are the subset of $P$'s derivation trees with dimension at most $k$) and $P^{>k}$ (whose derivation trees have dimension at least $k+1$). Each such set of clauses represents an under-approximation of the original set of clauses and the proof for the original clauses can be constructed from their individual proofs. In order to prove some property of all derivations of $P$, we systematically apply these transformations, for various values of $k$, to decompose the proof into separate proofs for $P^{\leq k}$ and $P^{>k}$ (which could be executed in parallel). We prove each such set of clauses using abstract interpretation [4] over the domain of convex polyhedra [5] as described in [18]. Finally, the preliminary results in a set of Horn clause verification benchmarks show that this is a useful program transformation. This decomposition can also be viewed as refinement where one eliminates possibly infinite sets of program traces. As a result of this, the proof for the remaining part becomes simpler. To motivate readers, we present an example set of constrained Horn clauses (CHCs) $P$ in Figure 1 which defines the Fibonacci function. This is an interesting problem whose dimension depends on the input number and its computations are trees rather than linear sequences. The main contributions of this paper are the following.

[*]The research leading to these results has been supported by the EU FP7 project 318337, *ENTRA - Whole-Systems Energy Transparency* and Danish Research Council grant FNU-10-084290.

```
c1. fib(A, A):- A>=0,  A=<1.
c2. fib(A, B) :- A > 1, A2 = A - 2, fib(A2, B2),
             A1 = A - 1, fib(A1, B1), B = B1 + B2.
c3. false:- A>5, fib(A,B), B<A.
```

Figure 1: Example CHCs Fib: it defines a Fibonacci function.

1. We describe how to generate at-most k-dimension program and at-least k-dimension program from a given program using the notion of tree dimension (Section 2);
2. We give a verification algorithm for Horn clauses program based on its proof decomposition (Section 3);
3. We give an alternative way of generating the at-least k-dimension program using the theory of finite tree automata (Section 4);
4. We demonstrate the feasibility of our approach in practice applying it to non-linear Horn clause verification problems (Section 7);
5. We instrument a program with its dimension and use existing automatic verification tools to prove some interesting properties about its dimension (Section 5).

## 2 Preliminaries

A constrained Horn clause is a first order formula of the form $p(X) \leftarrow \mathscr{C}, p_1(X_1), \ldots, p_k(X_k)$ $(k \geq 0)$ (using Constraint Logic Programming (CLP) syntax), where $\mathscr{C}$ is a conjunction of constraints with respect to some background theory, $X_i, X$ are (possibly empty) vectors of distinct variables, $p_1, \ldots, p_k, p$ are predicate symbols, $p(X)$ is the head of the clause and $\mathscr{C}, p_1(X_1), \ldots, p_k(X_k)$ is the body. A clause is called non-linear if it contains more than one atom in the body ($k > 1$), otherwise it is called linear. A set of Horn clauses is sometimes called a program.

A labeled tree $c(t_1, \ldots, t_k)$ is a tree with its nodes labeled, where $c$ is a node label and $t_1, \ldots, t_k$ are labeled trees rooted at the children of the node and leaf nodes are denoted by $c$.

**Definition 1 (Tree dimension (adapted from [7]))** *Given a labeled tree $t = c(t_1, \ldots, t_k)$, the tree dimension of $t$ represented as $dim(t)$ is defined as follows:*

$$dim(t) = \begin{cases} 0 & \text{if } k = 0 \\ \max_{i \in [1..k]} dim(t_i) & \text{if there is a unique maximum} \\ \max_{i \in [1..k]} dim(t_i) + 1 & \text{otherwise} \end{cases}$$

Figure 2 (a) shows a derivation tree $t$ for Fibonacci number 3 and Figure 2 (b) shows its tree dimension. It can be seen that $dim(t) = 1$. This number is a measure of its non-linearity, the smaller the number the closer the tree is to a list. Since it is not a perfect binary tree, the height of $t$ (3) is greater than its dimension.

Given a set of CHCs $P$ and $k \in \mathbb{N}$, we split each predicate $H$ occurring in $P$ into the predicates $H^{\leq d}$ and $H^{=d}$ where $d \in \{0, 1, \ldots, k\}$. Here $H^{\leq d}$ and $H^{=d}$ generate trees of dimension at most $d$ and exactly $d$ respectively.

**Definition 2 ( At-most-k-dimension program $P^{\leq k}$)** *It consists of the following clauses (adapted from [20]):*

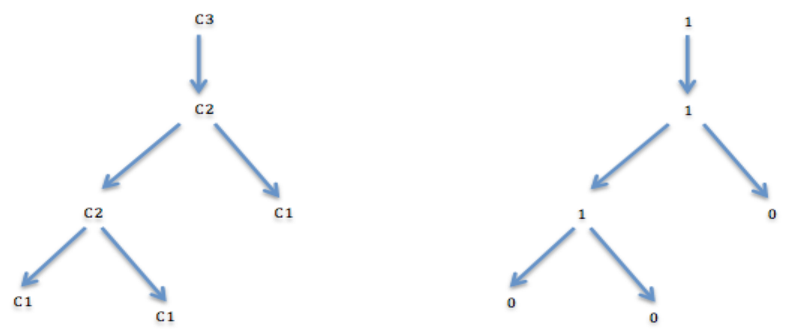

Figure 2: (a) derivation tree of Fibonacci 3 and (b) its tree dimension.

```
%linear clauses
1. fib(0)(A,A) :- A>=0, A=<1.
2. false(0) :- A>5, B<A, fib(0)(A,B).
%epsilon-clauses
3. false[0] :- false(0).
4. fib[0](A,B) :- fib(0)(A,B).
```

Figure 3: Fib$^{\leq 0}$ : at-most 0-dimension program of Fib.

1. *Linear clauses:*
   *If $H \leftarrow \mathscr{C} \in P$ , then $H^{=0} \leftarrow \mathscr{C} \in P^{\leq k}$.*
   *If $H \leftarrow \mathscr{C}, B_1 \in P$ then $H^{=d} \leftarrow \mathscr{C}, B_1^{=d} \in P^{\leq k}$ for $0 \leq d \leq k$.*
2. *Non-linear clauses:*
   *If $H \leftarrow \mathscr{C}, B_1, B_2, \ldots, B_r \in P$ with $r > 1$:*
   - *For $1 \leq d \leq k$, and $1 \leq j \leq r$:*
     *Set $Z_j = B_j^{=d}$ and $Z_i = B_i^{\leq d-1}$ for $1 \leq i \leq r \wedge i \neq j$. Then: $H^{=d} \leftarrow \mathscr{C}, Z_1, \ldots, Z_r \in P^{\leq k}$.*
   - *For $1 \leq d \leq k$, and $J \subseteq \{1, \ldots, r\}$ with $|J| = 2$:*
     *Set $Z_i = B_i^{=d-1}$ if $i \in J$ and $Z_i = B_i^{\leq d-2}$ if $i \in \{1, \ldots, r\} \setminus J$. If all $Z_i$ are defined, i.e., $d \geq 2$ if $r > 2$, then: $H^{=d} \leftarrow \mathscr{C}, Z_1, \ldots, Z_r \in P^{\leq k}$.*
3. *$\varepsilon$-clauses:*
   *$H^{\leq d} \leftarrow H^{=e} \in P^{\leq k}$ for $0 \leq d \leq k$ , and every $0 \leq e \leq d$.*

The at-most 0-dimension program of Fib in Figure 1 is depicted in Figure 3 (where the numbers on the first column are not clause identifiers and are there for future reference). In textual form we represent a predicate $p^{\leq k}$ by `p[k]` and a predicate $p^{=k}$ by `p(k)`. Since some programs have derivation trees of unbounded dimension, trying to verify a property for its increasing dimension separately is not a practical strategy. To deal with this, we need some construction which characterises derivation trees of at-least $k$-dimension. Next we define this construction (*at-least k-dimension program*). For this, we split each predicate $H$ occurring in $P$ into the predicates $H^{>d}$ and $H^{\geq 0}$ where $d \in \{0, 1 \ldots, k\}$. Here $H^{>d}$ generates trees of dimension at-least $d+1$ and $H^{\geq 0}$ generates trees of any dimension.

**Definition 3 (At-least k+1-dimension program $P^{>k}$)** *In addition to the linear, non-linear and $\varepsilon$-clauses from Definition 2 (with each predicate $H^{\leq k}$ and $H^{=k}$ from $P^{\leq k}$ renamed to $H^{>k}$ and $H^{\geq 0}$ respectively), the at-least k+1-dimension program $P^{>k}$ consists of the following clauses:*

```
%linear clauses
fib<0>(A,A) :- A>=0, A=<1.
false<0> :- A>5, B<A,fib<0>(A,B).
%epsilon-clauses
false{0} :- false<0>.
fib{0}(A,B) :- fib<0>(A,B).
%link clauses
false<0> :- false.
fib<0>(A,B) :- fib(A,B).
%original clauses (all clauses)
fib(A, A):- A>=0,  A=<1.
false:- A>5, fib(A,B), B<A.
fib(A, B) :- A > 1, A2 = A - 2, fib(A2, B2),
           A1 = A - 1, fib(A1, B1), B = B1 + B2.
```

Figure 4: Fib$^{>0}$ : at-least 1-dimension program of Fib.

1. *Link-clauses:*
   *For each $H \leftarrow B \in P$ there is a clause $H^{\geq 0} \leftarrow H \in P^{>k}$.*
2. *Original clauses:*
   *All clauses in $P$ are also in $P^{>k}$.*

The at-least 1-dimension program of Fib in Figure 1 is depicted in Figure 4. In textual form we represent a predicate $p^{>k}$ by `p{k}` and a predicate $p^{\geq 0}$ by `p<k>`.

## 3 Procedure for verification

Given a set of CHCs $P$ (including clauses with false head, also known as *integrity constraints*), the CHC verification problem is to check whether there exists a model of $P$. This is equivalent to checking whether there is any feasible derivation tree for false; if there is such a derivation then there is no model. We say $P$ is safe if it has a model and unsafe if it has no model. The procedure VERIFY($P$) is described in algorithm 1. VERIFY makes use of the procedure SAFE($P$) in the Algorithm 1, which is an oracle that returns *safe*, *unsafe* or *unknown*. The oracle is sound: if SAFE($P$) returns *safe* (*unsafe*) then $P$ is safe (unsafe). SAFE could be any existing automatic Horn clause solver [12, 19, 18, 17, 6]. When it cannot verify a program within a given time limit, the *unknown* answer is emitted. A given set of Horn clauses $P$ can be transformed into a new set of clauses $P^{\leq k}$ and $P^{>k}$. In order to prove some property of all derivations of $P$, we systematically apply these transformations, for various values of $k$, to decompose the proof into separate proofs for $P^{\leq k}$ *(line 4)* and $P^{>k}$ *(line 9)*. If both are safe then $P$ is *safe*. If one of them is unsafe then $P$ is *unsafe*. If an oracle cannot prove whether $P^{\leq k}$ is *safe/unsafe* then we return an *unknown* answer (we assume that the oracle would also return *unknown* for larger values of $k$). But if it cannot prove whether $P^{>k}$ is *safe/unsafe* then we try the *while loop* in the algorithm 1 with $k = k+1$.

One possible optimisation that we can make in Algorithm 1 is to consider $P^{>k}$ instead of $P$ in the next iteration of the *while loop* if we reach *line 14*. This is because at this stage we have already proven the safety of $P^{\leq k}$.

The soundness of Algorithm 1 is captured by the following lemma and proposition.

**Algorithm 1:** Verification algorithm for Horn clauses

```
1  Procedure VERIFY ($P$)
   Input: Set of CHCs $P$
   Output: safe, unsafe, unknown
2  initialization: $k \leftarrow 0$
3  while true do
4      generate $P^{\leq k}$
5      $r_1 \leftarrow$ SAFE($P^{\leq k}$)
6      if $r_1 \neq safe$ then
7          return $r_1$
8      end
9      generate $P^{\{k\}}$
10     $r'_1 \leftarrow$ SAFE( $P^{>k}$)
11     if $r'_1 \neq unknown$ then
12         return $r'_1$
13     end
14     $k \leftarrow k+1$
15 end
```

**Lemma 1 (Decomposition by dimension)** *For all k, program P is safe if and only if both $P^{\leq k}$ and $P^{>k}$ are safe.*

**Proposition 1 (Soundness)** *If Algorithm 1 returns* safe *then the input program is safe. If it returns* unsafe *then the program is unsafe.*

## 4 Dimension decomposition using finite tree automata

In this section, we show an alternative method for constructing an at-least k-dimension program, using operations on finite tree automata (FTAs). We first describe the connection between Horn clauses and FTAs and show how to construct an FTA from a set of Horn clauses.

### 4.1 Trace automata for CHCs

We add identifiers to clauses, whose purpose is to act as constructors of trace trees representing derivations. The identifiers are chosen from a set $\Sigma$ of ranked function symbols. If $P$ is a set of CHCs, let $\mathsf{id}_P : P \rightarrow \Sigma$ be an assignment of function symbols to clauses, such that for every clause $cl \in P$, the arity of $\mathsf{id}_P(cl)$ equals the number of atoms in the body of $cl$. We allow the same symbol to be assigned by $\mathsf{id}_P$ to more than one clause. We can also identify the predicates whose derivations are of interest (the *accepting* predicates in Definition 4).

**Definition 4 (Trace FTA for a set of CHCs)** *Let P be a set of CHCs, $\Sigma$ be a set of ranked function symbols and $\mathsf{id}_P : P \rightarrow \Sigma$ be a mapping from clauses to function symbols of appropriate arity. Let F be a set of predicates from P called the* accepting *predicates. Define the trace FTA for P as $\mathscr{A}_P^F = (Q, F, \Sigma, \Delta)$ where*

- *Q is the set of predicate symbols of P;*

- *$F \subseteq Q$ is the set of accepting predicate symbols;*
- *$\Sigma$ is a set of function symbols;*
- *$\Delta = \{c(p_1,\ldots,p_k) \to p \mid cl \in P,\ cl = p(X) \leftarrow \mathscr{C}, p_1(X_1),\ldots,p_k(X_k),\ c = \mathsf{id}_P(cl)\}$.*

*If $F$ is the set of all predicate symbols occurring in the clauses we omit the superscript $F$ from $\mathscr{A}_P^F$.*

The set of trees accepted by $\mathscr{A}_P^F$ is written $\mathscr{L}(\mathscr{A}_P^F)$. Elements of $\mathscr{L}(\mathscr{A}_P^F)$ are called the trace trees for $P$. $\mathscr{L}(\mathscr{A}_P^F)$ is isomorphic to the set of (successful and unsuccessful) derivation trees (for atomic formulas with accepting predicates) constructible from $P$ and from now on we identify trace trees with derivations. We do not define derivation trees formally here, but refer to the notion of an AND-tree in the literature [22, 10].

**Example 1** *Let $P$ be the set of CHCs in Figure 1 and let $F = \{\texttt{fib},\texttt{false}\}$. Let $\mathsf{id}_P$ map the clauses to $c_1, c_2, c_3$ respectively. Then $\mathscr{A}_P^F = (Q,F,\Sigma,\Delta)$ where:*

$$\begin{array}{rcl@{\qquad}rcl} Q & = & \{\texttt{fib},\texttt{false}\} & \Delta & = & \{c_1 \to \texttt{fib}, \\ \Sigma & = & \{c_1,c_2,c_3\} & & & c_2(\texttt{fib},\texttt{fib}) \to \texttt{fib}, \\ & & & & & c_3(\texttt{fib}) \to \texttt{false}\} \end{array}$$

*Figure 2(a) shows a trace tree recognised by this FTA. The tree can also be written $c_3(c_2(c_2(c_1,c_1),c_1))$.*

If a mapping $\mathsf{id}_P : P \to \Sigma$ assigns a unique identifier to each clause, that is, $\mathsf{id}_P$ is injective, then there is an inverse mapping $\mathsf{id}^{-1} : \mathsf{range}(\mathsf{id}_P) \to P$.

**Definition 5** ($\mathsf{chc}_{\mathsf{id}}(\mathscr{A})$) *Given an FTA $\mathscr{A} = (Q,F,\Sigma,\Delta)$ and an injective mapping $\mathsf{id}$ such that $\Sigma \subseteq \mathsf{range}(\mathsf{id})$, we can construct a set of CHCs from $\mathscr{A}$, called $\mathsf{chc}_{\mathsf{id}}(\mathscr{A})$, defined as follows:*

$$\begin{array}{rl} \mathsf{chc}_{\mathsf{id}}(\mathscr{A}) = \{q(X) \leftarrow \mathscr{C}, q_1(X_1),\ldots,q_n(X_n) \mid & c(q_1,\ldots,q_n) \to q \in \Delta, \\ & \mathsf{id}^{-1}(c) = q(X) \leftarrow \mathscr{C}, q_1(X_1),\ldots,q_n(X_n)\} \end{array}$$

*The set of accepting predicates of $\mathsf{chc}_{\mathsf{id}}(\mathscr{A})$ is defined to be $F$.*

In the definitions we reuse the states in the FTA as predicate symbols in the constructed clauses. In practice we use some injective renaming function from states to predicates in the constructed program. Further discussion of the mappings between CHCs and FTAs can be found in [19]. By construction, the derivations of $\mathsf{chc}_{\mathsf{id}}(\mathscr{A})$ (for the accepting predicates) correspond to the elements of $\mathscr{L}(\mathscr{A})$.

## 4.2 Construction of the at-least k-dimensional program using FTA operations

In the construction of the at-least k-dimension program $P^{>k}$ in Definition 4, the original program clauses from $P$ are included in the generated clauses. The presence of the original clauses suggests that the "decomposed" verification problem for $P^{>k}$ is as hard as the original problem for $P$, since it contains the clauses of $P$ as well as others, and so this form might not lend itself to verification.

Thus in the following construction we build $P^{>k}$ based on FTA language difference, and the original clauses are not copied to the at-least k-dimension program. We first define a general FTA-difference for CHCs.

**Definition 6 (FTA-difference for CHCs)** *Let $P$ and $Q$ be sets of CHCs, $F_1$ and $F_2$ their respective accepting predicates and $\mathsf{id}_P : P \to \Sigma$ and $\mathsf{id}_Q : Q \to \Sigma$ their respective identifier assignments, where $\mathsf{id}_P$ is injective. Let $\mathscr{A}_P^{F_1}$ and $\mathscr{A}_Q^{F_2}$ be the trace FTAs constructed from $P, Q$ respectively. Then the FTA-difference of $P$ and $Q$ (with their respective accepting predicates) written $P^{F_1} - Q^{F_2}$, is given as $\mathsf{chc}_{\mathsf{id}_P}(\mathscr{A}_P^{F_1} \setminus \mathscr{A}_Q^{F_2})$ where $\setminus$ is the difference of FTAs [3]. The set of accepting predicates is the set of accepting states for the difference FTA .*

The set of derivations for $P^{F_1} - Q^{F_2}$ contains, by construction, those derivations of $P^{F_1}$ that are not derivations of $Q^{F_2}$. We now apply these notions to the verification procedure based on decomposition. We are given a set of CHCs $P$, with accepting predicates $F = \{\mathsf{false}\}$. In the program $P^{\leq k}$, the set of accepting predicates is $F^k = \{\mathsf{false}^{\leq k}\}$. Note that we can ignore the derivations for the other predicates of the form $\mathsf{false}^{\leq j}$ or $\mathsf{false}^{=j}$ since $\mathsf{false}^{\leq k}$ by construction accumulates their derivations, for all $j \leq k$.

### 4.2.1 Assignment of identifiers in the at-most-k-dimension program

Given a program $P$ and the at-most-k-dimension program $P^{\leq k}$, we intend to construct the difference $P^{\{\mathsf{false}\}} - P^{\leq k\{\mathsf{false}\}}$ using Definition 6. In order to do so, we first need to construct the identifier assignment $\mathsf{id}_{P^{\leq k}}$ so as to preserve trace trees from $P$. This requires the modification of $P^{\leq k}$ to eliminate the $\varepsilon$-clauses, as follows.

**Definition 7 (Unfolding of $\varepsilon$-clauses in $P^{\leq k}$)** *Let $P^{\leq k}$ be the at-most-k-dimension program obtained from $P$ using Definition 2. Replace each $\varepsilon$-clause of form $H^{\leq d} \leftarrow H^{=e}$ by the set of clauses $H^{\leq d} \leftarrow B$, where $H^{=e} \leftarrow B$ is either a linear or non-linear clause in $P^{\leq k}$.*

The elimination of $\varepsilon$-clauses is an instance of the well-known unfolding transformation which preserves the derivability of atomic formulas. In other words an atom $A$ is derivable from a program $P$ if and only if it is derivable after applying the unfolding transformation [21].

In the following definition, the clause identifiers are chosen for clauses in $P^{\leq k}$ Informally, every clause of $P^{\leq k}$ inherits the clause identifier for the clause in $P$ from which it originates. More precisely we define the clause identifiers for $P^{\leq k}$ as follows.

**Definition 8 (Assignment of clause identifiers in $P^{\leq k}$)** *Let $P^{\leq k}$ be the at-most-k-dimension program obtained from $P$ using Definition 2, with $\varepsilon$-clauses eliminated according to Definition 7. Each clause of $P^{\leq k}$ is a linear, non-linear or an $\varepsilon$-unfolded-clause. The clause identifiers are assigned in two steps as follows.*

1. *Assign to each linear or non-linear clause the clause identifier from the clause in $P$ from which it is derived in Definition 2.*
2. *Assign to each each unfolded $\varepsilon$-clause the clause identifier for the linear or non-linear clause used to unfold it using Definition 7.*

We are now in a position to compare the sets of trace trees for $P$ and $P^{\leq k}$ using their respective FTAs.

**Lemma 2** *Let $P$ be a set of CHCs and let $\mathsf{id}_P : P \rightarrow \Sigma$ be an injective function assigning clause identifiers to $P$. Let $F_1 = \{\mathsf{false}\}$. Let $k \geq 0$ and let $P^{\leq k}$ be the at-most-k-dimension program obtained from $P$ using Definition 2 with $\varepsilon$-clauses unfolded using Definition 7 and let $F_2 = \{\mathsf{false}^{\leq k}\}$. Then $\mathscr{L}(\mathscr{A}^{F_2}_{P^{\leq k}}) = \{t \mid t \in \mathscr{L}(A^{F_1}_P), dim(t) \leq k\}$.*

The proof is by induction on derivations in $P^{\leq k}$ and uses the correspondence of the clause identifiers as set up in Definition 8.

**Theorem 1** *Let $P$ be a set of CHCs and let $\mathsf{id}_P : P \rightarrow \Sigma$ be an injective function assigning clause identifiers to $P$. Let $k \geq 0$ and let $P^{\leq k}$ be the at-most-k-dimension program obtained from $P$ using Definition 2 with $\varepsilon$-clauses unfolded using Definition 7. Then $\mathsf{false}$ is derivable from $P - P^{\leq k}$ if and only if $\mathsf{false}^{>k}$ is derivable from $P^{>k}$.*

```
c1. fib(0)(A,A) :- A>=0, A=<1.
c3. false(0) :- A>5, B<A, fib(0)(A,B).
c3. false[0] :- A>5, B<A, fib(0)(A,B).
c1. fib[0](A,B) :-A>=0, A=<1.
```

Figure 5: Fib$^{\leq 0}$ after unfolding $\varepsilon$-clauses and assigning clause identifiers.

Thus we have shown a different method of constructing the at-least k-dimension program $P^{>k}$, namely by taking the difference of $P$ with $P^{\leq k}$, which contains only derivations (for its accepting predicates) that have dimension greater than $k$.

Details on difference construction can be found in [19]. We construct the difference of two FTAs by (1) standardising apart the predicate names; (2) forming the union of the two FTAs; (3) determinising the union; (4) removing from the determinised FTA all states (and transitions that contain them) that contain an accepting state of the second FTA. Note that the set of states of the determinised FTA is a subset of the powerset of the original states. Note that determinisation of FTAs is often considered prohibitively complex even for small FTAs. We use a recent optimised FTA determinisation algorithm [9], returning a compact form of the determinised called product form, which can be used directly in constructing the resulting clauses.

**Example 2** *We illustrate this through an example using Fib$^{\leq 0}$ (Figure 3). The clauses 1 and 2 in Fib$^{\leq 0}$, will have $c_1$ and $c_3$ as identifiers since they were derived respectively from the clauses $c_1$ and $c_3$ in Fib (Figure 1). By unfolding $\varepsilon$-clauses (clauses 3 and 4) using respectively clauses 2 and 1 in Figure 3, we obtain* `false[0] :- A>5, B<A, fib(0)(A,B) and fib[0](A,B) :-A>=0, A=<1`. *They will have identifiers $c_3$ and $c_1$ respectively. Therefore, the clauses in Fib$^{\leq 0}$ will have the identifiers assigned as shown in Figure 5.*

After assigning identifiers to each of the clauses in Fib$^{\leq 0}$, we can construct an FTA corresponding to it using Definition 4, and obtain the FTA shown in Figure 6: as before we represent a predicate $p^{\leq k}$ by `p[k]` and a predicate $p^{=k}$ by `p(k)`.

$$Q = \{\texttt{fib(0)},\ \texttt{false(0)},\ \texttt{false[0]},\ \texttt{fib[0]}\} \qquad \Delta = \{c_1 \to \texttt{fib(0)},$$
$$F = \{\texttt{false[0]}\} \qquad c_3(\texttt{fib(0)}) \to \texttt{false(0)},$$
$$\Sigma = \{c_1, c_3\} \qquad c_3(\texttt{fib(0)}) \to \texttt{false[0]},$$
$$c_1 \to \texttt{fib[0]}\}$$

Figure 6: FTA $(Q,F,\Sigma,\Delta)$ corresponding to Fib$^{\leq 0}$.

The difference FTA between $\mathscr{A}_{Fib}^{\{\mathsf{false}\}}$ and $\mathscr{A}_{Fib^{\leq 0}}^{\{\mathsf{false}^{\leq 0}\}}$ accepts trees rooted at false which have dimension greater than 0. The determinised FTA (DFTA) constructed as explained above is shown in the Figure 7. DFTA states are sets of predicates, and we represent a set using square brackets instead of curly brackets in the code, e.g. `[fib(0), fib[0], fib]`. Furthermore the product form referred to above contains set of DFTA states, such as `[[fib(0), fib[0], fib], [fib]]`.

We can generate a new program from this DFTA together with the original program Fib following the approach taken in [19] obtaining the program in Figure 8. It should be noted that the derivation trees rooted at false have dimension at-least 1. Now verification of the original program Fib is decomposed into verifying the program in Figure 3 (where `false[0]` is replaced by `false` and the program in Figure 8.

```
c1 -> [fib(0), fib[0], fib].
c2([[fib(0), fib[0], fib], [fib]],
                     [[fib(0), fib[0], fib], [fib]]) -> [fib].
c3([[fib]]) -> [false].
```

Figure 7: Transitions of the determinised FTA.

```
fib_0(A,A) :- A>=0, A=<1.
fib(A,B) :-  A>1,C=A-2,D=A-1,B=E+F,fib_1(C,F), fib_1(D,E).
false :- A>5, B<A, fib(A,B).
fib_1(A,B) :- fib_0(A,B).
fib_1(A,B) :-  fib(A,B).
```

Figure 8: At-least 1-dimension program of Fib produced using the difference of FTAs

## 5 Program instrumentation with dimension

The dimension of successful derivations in a set of CHCs is not always obvious from the text of the clauses. In some cases a bound on the dimension is clear from the form of the clauses; for instance all derivations using a set of linear clauses clearly have dimension zero. But consider the well known 91-function of McCarthy[1], represented in Figure 9 using Horn clauses.

Although it is possible to construct derivation trees of arbitrary dimension using the clauses in Figure 9, the dependencies between the two recursive calls to `mc91` imply that no *successful* derivation has dimension greater than 2. We now show how to establish this using a transformation to instrument the clauses with dimension information, and then use automatic verification tools to establish properties of the dimension.

**Definition 9 (Dimension-instrumented clauses)** *Let $P$ be a set of CHCs. Define the set $P_{dim}$ of CHC as follows.*

- *For each predicate $p$ of arity $m$ define a predicate $p'$ of arity $m+1$.*
- *For each clause in $P$ of the form*

$$p(X) \leftarrow \mathscr{C}, p_1(X_1), \ldots, p_n(X_n)$$

*construct a clause*

$$p'(X,K) \leftarrow \mathscr{C}, p_1^{'}(X_1,K_1), \ldots, p_n^{'}(X_n,K_n), dim_n(K_1,\ldots,K_n,K)$$

*in $P_{dim}$, where $K_1,\ldots,K_n,K$ are variables added as the final argument for their respective predicates, and $dim_n(K_1,\ldots,K_n,K)$ is defined according to the rules in Definition 1 for determining the dimension of a tree.*

[1]http://en.wikipedia.org/wiki/McCarthy_91_function

```
mc91(N,X) :-  N > 100,  X = N-10.
mc91(N,X) :-  N =< 100,  Y = N+11,
              mc91(Y,Y2), mc91(Y2,X).
```

Figure 9: McCarthy's 91-function defined as Horn clauses

```
fib(A, A, K):-  A>=0,  A=<1, dim0(K).
fib(A, B, K) :- A > 1, A2 = A - 2, fib(A2, B2, K1),
            A1 = A - 1, fib(A1, B1, K2), B = B1 + B2, dim2(K1, K2, K).
dim0(K):-K=0.
dim2(K1, K2, K3):-K1>=K2+1, K3=K1.
dim2(K1, K2, K3):-K2>=K1+1, K3=K2.
dim2(K1, K2, K3):- K1=K2, K3 = K1+1.
```

Figure 10: Fib program instrumented with its dimension

**Example 3** *The dimension-instrumented version of the McCarthy 91-function contains the following clauses.*

```
mc91(N,X,K) :-  N > 100,  X = N-10, dim0(K).
mc91(N,X,K) :-  N =< 100,  Y = N+11,
     mc91(Y,Y2,K1), mc91(Y2,X,K2), dim2(K1,K2,K).
dim0(K):-K=0.
dim2(K1, K2, K3):-K1>=K2+1, K3=K1.
dim2(K1, K2, K3):-K2>=K1+1, K3=K2.
dim2(K1, K2, K3):- K1=K2, K3 = K1+1.
```

Using the instrumented program we can try to prove information about the dimension, such as upper or lower bounds or other relationships between the dimension and other predicate arguments. It follows from the undecidability result of Gruska [14] on context-free grammars, that the problem of determining whether the dimension of set of CHC is bounded by a constant is, in general, undecidable.

**Example 4** *To establish that the upper bound of successful derivations is 2, for facts* `mc91(X,Y)`*, we add the following integrity constraint to the dimension-instrumented clauses.*

```
false :- K > 2, mc91(X,Y,K).
```

*The clauses together with the integrity constraint are given to an automatic solver for Horn clauses [12, 19], which are able to prove the safety of the clauses and thus establish the upper bound of 2.*

In the next example, we show that the dimension can depend on the values of other predicate arguments.

**Example 5** *The dimension-instrumented version of the* Fib *clauses is shown in Figure 10. The property to be proved is that the dimension of* Fib *is lesser or equal to the half of its input value, expressed by the integrity constraint* `false:- fib(A,B, K), 2*K -1>=A`*. Again, this property is established by applying a Horn clause solver to prove the safety of the clauses together with the integrity constraint.*

**Example 6** *We present the well known counting change example taken from [1, Chapter 1]. The Figure 11 shows its CLP encoding and the Figure 12 shows the dimension-instrumented version in CLP. The property of interest is to relate the number of different coins (counts) with the program dimension. We can establish that the dimension is at most the number of different coins as expressed by the integrity constraint* `false :- B>=1, K > B, cc(A, B, C, K)`.

In general, verifying whether a program has a certain dimension is as challenging as proving any other properties of the program. But in some cases the knowledge of program dimension is useful for proving other program properties. For instance, using the knowledge that the McCarthy 91-function has dimension at most 2 would allow us to restrict the proof of any program property relating to successful derivations to the program $P^{[2]}$ where $P$ is the set of clauses for the McCarthy 91-function.

```
% base case: that is a hit
cc(0, Y, 1) :- Y>0.
% base case: that is a miss
cc(X, _, 0) :- X<0.
cc(_, Y, 0) :- Y=<0.
%inductive case
cc(X, Y, Z) :- X>0, kinds_of_coins(Y,A),
                    X1 = X-A, cc(X1, Y, Z1),
                    Y1 = Y-1, cc(X, Y1, Z2), Z = Z1 + Z2.
kinds_of_coins(A,B) :- A >= 1, B >= 1.
```

Figure 11: Counting change example encoded as CLP clauses

```
cc(0, Y, 1,K) :- Y>0, dim0(K).
cc(X, _, 0,K) :- X<0, dim0(K).
cc(_, Y, 0,K) :- Y=<0, dim0(K).
cc(X, Y, Z,K) :-
        X>0, kinds_of_coins(Y,A, K0), X1 = X-A,
        cc(X1, Y, Z1,K1), Y1 = Y-1, cc(X, Y1, Z2,K2),
        Z = Z1 + Z2, dim3(K0, K1,K2,K).
kinds_of_coins(A,B, K) :- A >= 1, B >= 1, dim0(K).
dim3(K0, K1,K2,K):-
        dim2(K0, K1, K3), dim2(K3,K2, K).
%predicates dim0(K) and dim2(K1, K2, K) are defined as above
```

Figure 12: Counting change example instrumented with its dimension

## 6 Related Work

The notion of dimension of a tree has a long history in science (starting with Geology) which has been detailed by Esparza *et al.* [8]. However, the use of dimension for program verification is more recent. Ganty and Iosif used it [11] for computing summaries of programs with procedures whose variables (global, local and parameters) take their value from the set of integers. Roughly speaking, the method they define first computes procedure summaries for all derivation trees of dimension 0, then they compute summaries for derivation trees of dimension 1 reusing the summaries computed for dimension 0 and so on.

Decomposition can be compared to refinement techniques based on automata [15, 16, 19] in which the aim is to eliminate sets of program traces that have been shown to be safe. Proof of the safety of a given dimension or dimensions of a set of clauses allows those dimensions to be eliminated, focusing the proof on the remaining dimensions. Our decomposition technique offers a very precise and practical approach to checking and eliminating infinite sets of traces.

## 7 Experimental results

We carried out an experiment on a set of 16 non-linear CHC verification problems taken from the repository[2] of software verification benchmarks. Our aim in the current paper is not to make a systematic comparison with other verification techniques; these are exploratory experiments to establish whether dimension-based decomposition is practical. The results are summarized in Table 1. Columns **Program**, **Result**, **Time** and **dim(k)** respectively represent a program, its verification result using our approach, time in seconds taken to generate the programs and solve it and a value of a proof decomposition parameter $k$.

Table 1: Experimental results on non-linear CHC verification problems

| **Program** | **Result** | **Time(s)** | **dim(k)** |
|---|---|---|---|
| addition | safe | 4 | 0 |
| bfprt | safe | 4 | 0 |
| binarysearch | safe | 4 | 0 |
| countZero | safe | 3 | 0 |
| floodfill | safe | 3 | 0 |
| identity | safe | 4 | 0 |
| merge | safe | 5 | 0 |
| palindrome | safe | 3 | 0 |
| fib | safe | 4 | 0 |
| mc91 | safe | 4 | 0 |
| revlen | safe | 4 | 0 |
| running | unsafe | 6 | 1 |
| triple | unsafe | - | - |
| buildheap | unsafe | - | - |
| parity | unsafe | 4 | 0 |
| remainder | unsafe | 4 | 0 |
| **avg. time(s)** | | 4 | |

For the safety check (the procedure *SAFE* in Algorithm 1) we use the verification procedure described in [18] which uses abstract interpretation over the domain of convex polyhedra, with a timeout of 5 minutes. The symbol "-" in Table 1 denotes that we were unable to solve these problems within the given time. Our approach solves 14 out of 16 problems with an average time of 4 seconds (over the solved problems). Our previous approach based on refinement with finite tree automata described in [19] solves 1 more additional problem, that is, *triple* than our current approach. These examples were also run on QARMC [13] which solves all the problems (much faster). Most of the problems are solved when we decompose the proof with the value of $k = 0$. This indicates that separating the proofs for linear programs eases the verification task. The splitting induced as a result of separating a set of traces has an effect on delaying join and widening operations during convex polyhedra analysis which increases its precision. In addition to this, some of the case base proofs (for example conditionals) becomes a normal proof without conditionals due to proof separation and the process of finding invariants becomes easier.

## 8 Conclusion and future work

We presented a program transformation approach to Horn clause verification using the notion of *tree dimension* to decompose the verification problem by separating dimensions. We presented one algorithm based on this idea which yielded preliminary results on set of non-linear Horn clause verification benchmarks, showing that the approach is feasible and this transformation is useful both for proving safety of a program as well as for finding bugs.

Other ideas of program verification based on tree-dimension are worth investigating, including proof by induction based on tree dimension, and further investigation of proof strategies that could exploit knowledge of dimension bounds (such as those discussed in Section 5).

[2]https://svn.sosy-lab.org/software/sv-benchmarks/trunk/clauses/LIA/Eldarica/RECUR/

Although it is formulated in the context of Datalog, it is known from Afrati *et al.* [2] that a set of CHC of bounded dimension can be turned into an equivalent set of linear CHC. The exact complexity of their procedure is still open.

## References

[1] H. Abelson and G. J. Sussman. *Structure and Interpretation of Computer Programs, Second Edition*. MIT Press, 1996.

[2] F. N. Afrati, M. Gergatsoulis, and F. Toni. Linearisability on datalog programs. *Theor. Comput. Sci.*, 308(1-3):199–226, 2003.

[3] H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. Available on: `http://www.grappa.univ-lille3.fr/tata`, 2007. release October, 12th 2007.

[4] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252. ACM, 1977.

[5] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the 5th Annual ACM Symposium on Principles of Programming Languages*, pages 84–96, 1978.

[6] E. De Angelis, F. Fioravanti, A. Pettorossi, and M. Proietti. Verimap: A tool for verifying programs through transformations. In E. Ábrahám and K. Havelund, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings*, volume 8413 of *Lecture Notes in Computer Science*, pages 568–574. Springer, 2014.

[7] J. Esparza, S. Kiefer, and M. Luttenberger. On fixed point equations over commutative semirings. In *STACS 2007, 24th Annual Symposium on Theoretical Aspects of Computer Science, Proceedings*, volume 4393 of *LNCS*, pages 296–307. Springer, 2007.

[8] J. Esparza, M. Luttenberger, and M. Schlund. A brief history of strahler numbers. In *LATA '14, 8th Int. Conf. on Language and Automata Theory and Applications*, volume 8370 of *LNCS*, page 113. Springer, 2014.

[9] J. P. Gallagher, M. Ajspur, and B. Kafle. An Optimised Algorithm for Determinisation and Completion of Finite Tree Automata. Technical Report 145, Roskilde University, Denmark, 2014. Available from http://akira.ruc.dk/∼jpg/dfta.pdf.

[10] J. P. Gallagher and L. Lafave. Regular approximation of computation paths in logic and functional languages. In *Partial Evaluation*, volume 1110 of *LNCS*, pages 115–136. Springer, 1996.

[11] P. Ganty, R. Iosif, and F. Konečný. Underapproximation of procedure summaries for integer programs. In *TACAS '13: Proc. 19th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, volume 7795 of *LNCS*, pages 247–261. Springer, 2013.

[12] S. Grebenshchikov, A. Gupta, N. P. Lopes, C. Popeea, and A. Rybalchenko. Hsf(c): A software verifier based on Horn clauses - (competition contribution). In *TACAS*, volume 7214 of *LNCS*, pages 549–551. Springer, 2012.

[13] S. Grebenshchikov, N. P. Lopes, C. Popeea, and A. Rybalchenko. Synthesizing software verifiers from proof rules. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, pages 405–416. ACM, 2012.

[14] J. Gruska. A few remarks on the index of context-free grammars and languages. *Information and Control*, 19(3):216–223, 1971.

[15] M. Heizmann, J. Hoenicke, and A. Podelski. Refinement of trace abstraction. In *Static Analysis, 16th International Symposium, SAS 2009*, volume 5673 of *LNCS*, pages 69–85. Springer, 2009.

[16] M. Heizmann, J. Hoenicke, and A. Podelski. Software model checking for people who love automata. In *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, volume 8044 of *LNCS*, pages 36–52. Springer, 2013.

[17] K. Hoder and N. Bjørner. Generalized property directed reachability. In A. Cimatti and R. Sebastiani, editors, *Theory and Applications of Satisfiability Testing - SAT 2012 - 15th International Conference, Trento, Italy, June 17-20, 2012. Proceedings*, volume 7317 of *Lecture Notes in Computer Science*, pages 157–171. Springer, 2012.

[18] B. Kafle and J. P. Gallagher. Constraint specialisation in horn clause verification. In *Proceedings of the 2015 Workshop on Partial Evaluation and Program Manipulation, PEPM, Mumbai, India, January 15-17, 2015*, pages 85–90. ACM, 2015.

[19] B. Kafle and J. P. Gallagher. Tree automata-based refinement with application to horn clause verification. In *Verification, Model Checking, and Abstract Interpretation - 16th International Conference, VMCAI 2015, Mumbai, India, January 12-14, 2015. Proceedings*, volume 8931 of *LNCS*, pages 209–226. Springer, 2015.

[20] M. Luttenberger. An extension of parikh's theorem beyond idempotence. *CoRR*, abs/1112.2864, 2011.

[21] A. Pettorossi and M. Proietti. Synthesis and transformation of logic programs using unfold/fold proofs. *J. Log. Program.*, 41(2-3):197–230, 1999.

[22] R. F. Stärk. A direct proof for the completeness of SLD-resolution. In *CSL '89, 3rd Workshop on Computer Science Logic, Kaiserslautern, Germany, October 2-6, 1989, Proceedings*, volume 440 of *LNCS*, pages 382–383. Springer, 1989.