# Solving non-linear Horn clauses using a linear Horn clause solver [*]

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 show that checking satisfiability of a set of non-linear Horn clauses (also called a non-linear Horn clause program) can be achieved using a solver for linear Horn clauses. We achieve this by interleaving a *program transformation* with a satisfiability checker for linear Horn clauses (also called a *solver* for linear Horn clauses). The program transformation is based on the notion of *tree dimension*, which we apply to a set of non-linear clauses, yielding a set whose derivation trees have bounded dimension. Such a set of clauses can be linearised. The main algorithm then proceeds by applying the linearisation transformation and solver for linear Horn clauses to a sequence of sets of clauses with successively increasing dimension bound. The approach is then further developed by using a solution of clauses of lower dimension to (partially) linearise clauses of higher dimension. We constructed a prototype implementation of this approach and performed some experiments on a set of verification problems, which shows some promise.

## 1 Introduction

Many software verification problems can be reduced to checking satisfiability of a set of Horn clauses (the *verification conditions*). In this paper we propose an approach for checking satisfiability of a set of non-linear Horn clauses (clauses whose body contains more than one non-constraint atom) using a linear Horn clause solver. A *program transformation* based on the notion of *tree dimension* is applied to a set of non-linear Horn clauses; this gives a set of clauses that can be linearised and then solved using a *linear solver* for Horn clauses. This combination of dimension-bounding, linearisation and then solving with a linear solver is repeated for successively increasing dimension. The dimension of a tree is a measure of its non-linearity – for example a linear tree (whose nodes have at most one child) has dimension zero while a complete binary tree has dimension equal to its height.

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 $k$. It is known that $P^{\leq k}$ can be transformed to a linear set of clauses preserving satisfiability; hence if we can find a model of the linear set of clauses then the original clauses $P^{\leq k}$ also have a model.

The algorithm terminates with success if a model (solution) $M$ of $P^{\leq k}$ is also a model (after appropriate translation of predicate names) of $P$. However if $M$ is not a solution of $P$, then we proceed to generate $P^{\leq k+1}$ and repeat the procedure. The algorithm terminates if $P^{\leq k}$ is shown to be unsatisfiable (unsafe) for some $k$, since this implies that $P$ is also unsatisfiable.

```
c1. fib(A, B):- A>=0,  A=<1, B=A.
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 defining a Fibonacci function.

A more sophisticated version of the algorithm attempts to use the model $M$ of $P^{\leq k}$ to (partially) linearise $P^{\leq k+1}$. We can exploit the model of $P^{\leq k}$ in the following way; if $P^{\leq k+1}$ has a counterexample that does not use the (approximate) solution $M$ for $P^{\leq k}$, then $P$ is unsatisfiable. We continue this process successively for increasing value of $k$ until we find a solution or a counterexample to $P$, or until resources are exhausted.

As an example program, we consider a set of constrained Horn clauses $P$ in Figure 1 which defines the Fibonacci function. It is an interesting problem since its derivations are trees whose dimensions depend on an input argument. The last clause represents a property of the Fibonacci function expressed as an integrity constraint.

We have made a prototype implementation of this approach and performed some experiments on a set of software verification problems, which shows some promise. The main contributions of this paper are as follows.

1. We present a linearisation procedure for dimension-bounded Horn clauses using partial evaluation (Section 3).

2. We give an algorithm for solving a set of non-linear Horn clauses using a linear Horn clause solver (Section 4).

3. We demonstrate the feasibility of our approach in practice applying it to non-linear Horn clause problems (Section 5).

## 2   Preliminaries

A constrained Horn clause (CHC) is a first order formula of the form $p(X) \leftarrow \mathscr{C}, p_1(X_1), \dots, p_k(X_k)$ ($k \geq 0$) (using Constraint Logic Programming syntax), where $\mathscr{C}$ is a conjunction of *constraints* with respect to some constraint theory, $X_i, X$ are (possibly empty) vectors of distinct *variables*, $p_1, \dots, p_k, p$ are *predicate symbols*, $p(X)$ is the *head* of the clause and $\mathscr{C}, p_1(X_1), \dots, p_k(X_k)$ is the *body*. An atomic formula, or simply *atom*, is a formula $p(t)$ where $p$ is a non-constraint predicate symbol and $t$ a tuple of arguments. Atoms are sometimes written as $A$, $B$ or $H$, possibly with sub- or superscripts.

A clause is called *non-linear* if it contains more than one atom in the body, otherwise it is called *linear*. A set of Horn clauses $P$ is called linear if $P$ only contains linear clauses, otherwise it is called non-linear. *Integrity constraints* are a special kind of Horn clauses whose head is *false* where *false* is always interpreted as FALSE. A set of Horn clauses is sometimes called a *(constraint logic) program*.

An *interpretation* of a set of CHCs is represented as a set of *constrained facts* of the form $A \leftarrow \mathscr{C}$ where $A$ is an atomic formula $p(Z)$ where $Z$ is a tuple of distinct variables and $\mathscr{C}$ is a constraint over $Z$ with respect to some constraint theory. An interpretation that makes each clause in $P$ TRUE is called a *model* of $P$. We say a set of Horn clause $P$ (including integrity constraints) is safe (solvable) iff it has a model. In some works e.g. [5, 29], a model is also called a *solution* and we use them interchangeably in this paper.

A labeled tree $c(t_1, \ldots, t_k)$ $(k \geq 0)$ is a tree whose nodes are labeled by identifiers, where $c$ is the label of the root and $t_1, \ldots, t_k$ are labeled trees, the children of the root.

**Definition 1 (Tree dimension (adapted from [12]))** *Given a labeled tree* $t = c(t_1, \ldots, t_k)$, *the tree dimension of* $t$ *represented as* $dim(t)$ *is defined as follows:*

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

Given a set of Horn clauses, we associate with each clause $p(X) \leftarrow \mathscr{C}, p_1(X_1), \ldots, p_k(X_k)$ a unique identifier $c$ whose *arity* is $k$.

Labelled trees can represent Horn clause derivations, where node labels are clause identifiers.

**Definition 2 (Trace tree)** *A* trace tree *for an atom $A$ in a set of Horn clauses $P$ is a labelled tree* $c(t_1, \ldots, t_k)$ *if $c$ is a clause identifier for a clause $A \leftarrow \mathscr{C}, A_1, \ldots, A_k$ in $P$ (with variables suitably renamed) and $t_1, \ldots, t_k$ are trace trees for $A_1, \ldots, A_k$ in $P$ respectively.*

There is a one-one correspondence between *trace trees* and derivation trees of Horn clauses up to variable renaming. Thus when we speak about the dimension of a Horn clause derivation, we refer to the dimension of its corresponding *trace tree*.

Using the clauses shown in Figure 1 along with their identifiers, Figure 2 (a) shows a *trace tree* $t = c_3(c_2(c_2(c_1, c_1), c_1))$ and Figure 2 (b) shows its tree dimension. It can be seen that $dim(t) = 1$.
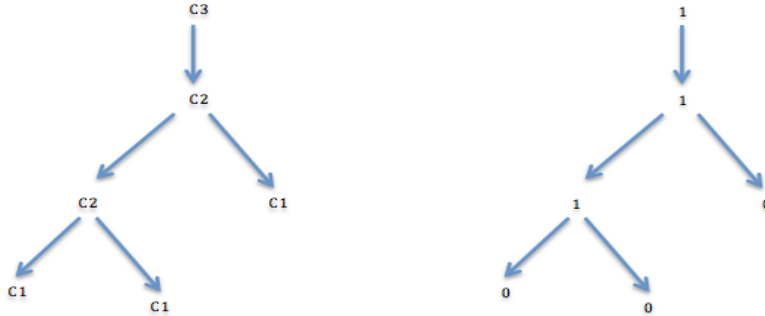


Figure 2: (a) a *trace tree* and (b) its tree dimension.

To make the paper self contained, we describe the transformation to produce a dimension-bounded set of clauses. Given a set of CHCs $P$ and $k \in \mathbb{N}$, we split each predicate $p$ occurring in $P$ into the predicates $p^{\leq d}$ and $p^{=d}$ where $d \in \{0, 1, \ldots, k\}$. An atom with predicate $p^{\leq d}$ or $p^{=d}$ is denoted $H^{\leq d}$ or $H^{=d}$ respectively. Such atoms have derivation trees of dimension at most $d$ and exactly $d$ respectively.

**Definition 3 ( At-most-$k$-dimension program $P^{\leq k}$)** *Let $P$ be a set of CHCs. $P^{\leq k}$ consists of the following clauses (adapted from [28, 24]):*

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 $0 \leq d \leq k$.*

```
%linear clauses
1. fib(0)(A,B) :- A>=0, A=<1, B=A.
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.

2. *Non-linear clauses:*

   *If $H \leftarrow \mathscr{C}, B_1, B_2, \ldots, B_r \in P$ with $r > 1$ and one of the following holds:*

   - *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 $Z_i = B_i^{\leq d-1}$ 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 $0 \leq e \leq d \leq k$.*

$P^{\leq k}$ is also called the *k-dimension-bounded program* corresponding to $P$. When the value of $k$ is not important, any program generated using the Definition 3 is called a dimension-bounded program. The relation between $P$ and its $k$-dimensional program is given by in the Proposition 1 where $\models$ is the usual "logical consequence" operator.

**Proposition 1 (Relation between $P$ and $P^{\leq k}$)** *Let $P$ be a program and $P^{\leq k}$ ($k \geq 0$) be its k-dimension-bounded program. Let $p(t)$ be an atom where $p$ is a predicate of $P$ and $p^\star(t)$ ($\star \in \{=, \leq\}$) be an atom where $p^\star$ is a predicate of $P^{\leq k}$. Then we have: $P^{\leq k} \models p^\star(t) \implies P \models p(t)$.*

In other words, Proposition 1 says that the set of facts that can be derived from $P^{\leq k}$ is a subset of the set of facts that can be derived from $P$, taking the predicate renaming into account. In this sense $P^{\leq k}$ is an *under-approximation* of $P$. In particular, if $P^{\leq k} \models \texttt{false}^\star$ then $P \models \texttt{false}$.

Let $S$ be an interpretation of a dimension-bounded set of clauses $P^{\leq k}$. That is, $S$ is a set of constrained facts of the form $H^{=d} \leftarrow \mathscr{C}$ or $H^{\leq d} \leftarrow \mathscr{C}$. An interpretation of $P$ is constructed from $S$ as follows.

**Definition 4 ($S^{\uparrow P^{\leq k}}$: an interpretation of $P$ constructed from an interpretation of $P^{\leq k}$)** *Let $S$ be an interpretation of $P^{\leq k}$. Then $S^{\uparrow P^{\leq k}}$ is the following set of constrained facts.*

$$S^{\uparrow P^{\leq k}} = \{p(X) \leftarrow \bigvee \{\mathscr{C} \mid p^{=d}(X) \leftarrow \mathscr{C} \in S \vee p^{\leq d}(X) \leftarrow \mathscr{C} \in S\} \mid d \in \{0...k\} \text{ and } p \text{ is a predicate in } P\}$$

*The set $S^{\uparrow P^{\leq k}}$ is a disjunctive interpretation of $P$ where the interpretation of $p$ is the disjunction of the interpretations of the corresponding dimension-bounded versions of $p$ in $P^{\leq k}$.*

The at-most-0-dimension program of Fib in Figure 1 is depicted in Figure 3. In textual form we represent a predicate $p^{\leq k}$ by p[k] and a predicate $p^{=k}$ by p(k). The at-most-1-dimension program of Fib in Figure 1 is depicted in Figure 4. Note that 0-dimension program is included in 1-dimension program. In general, all the clauses in $P^{\leq k}$ are also in $P^{\leq k+1}$. This provides a basis for an iterative

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

Figure 4: Fib$^{\leq 1}$ : at-most-1-dimension program of Fib.

strategy for a bounded set of Horn clauses. Since some programs have derivation trees of unbounded dimension, trying to verify a property for its increasing dimension separately is not a practical strategy. It only becomes a viable approach if a solution of $p^{\leq k}$ for some $k \geq 0$ is general enough to hold for all dimensions of $P$.

# 3   Linearisation strategies for dimension-bounded set of Horn clauses

In this section, we present linearisation strategies for set of clauses of bounded dimension. It is known [1] that a dimension-bounded set of clauses can be linearised, preserving satisfiability. In this section we describe a practical technique for linearisation, based on partial evaluation of an interpreter.

## 3.1   Linearisation based on partial evaluation

*Partial evaluation* (PE) has been studied for a variety of languages including logic programs [21, 14, 25, 22, 27]. We follow the pattern of transforming a program (a set of Horn clauses) by specialising an interpreter for that program [13, 22]. Let *PE* be a partial evaluator, *I* an interpreter and *P* an object program. Then the partial evaluation of *I* with respect to *P*, denoted $PE(I,P)$, represents the "compilation" of *P* using the semantics given by *I*.

We first write an interpreter for Horn clause programs, which is also written as a set of Horn clauses. Given a (possibly empty) conjunction of atoms (called a *goal*) the interpreter constructs a derivation, implementing a standard left-to-right, depth-first search. In the interpreter predicate solve(Gs), Gs is the goal, represented as a list of atoms. The basic step of the interpreter is represented by the clauses for solve(Gs) shown in Figure 5. If the conjunction is not empty, its first atom G is selected along with a matching Horn clause $G \leftarrow Cs, B$ in the program being interpreted, where *Cs* is a conjunction of constraints and *B* is a conjunction of atoms. This clause is represented by hornClause(G,Cs,B) in the interpreter. The body of the clause is conjoined with the remaining goal atoms, and the derivation continues with the new goal Gs1. If the conjunction is empty, the derivation is successful (second clause).

```
solve([G|Gs]) :-
   hornClause(G,Cs,B), solveConstraints(Cs), append(B,Gs,Gs1),
   solve(Gs1).
solve([]).
```

Figure 5: Depth-first interpreter for Horn clauses

To interpret a dimension-bounded set of clauses (say the bound is *k*), we use the fact that in all successful runs of the interpreter in which goals are selected in increasing order of dimension, the size of the conjunction of goals (that is, the length of the argument of `solve`) has an upper bound related to *k*. This bound is known as the *index* of the set of clauses and is given as $(i-1)*k+1$, where *i* is the maximum number of non-constraint atoms in the body of clauses [11]. Given this index, we can augment the interpreter with a check on the size of the conjunction, ensuring that it never exceeds the index. In addition, due to the requirement of increasing dimension in the selection of atoms, a left-to-right computation rule is not sufficient; therefore we permute the set of atoms in each clause body, since in at least one permutation the goals will be ordered by dimension. With these changes the interpreter remains complete for clauses of the given maximum index, at the possible cost of some redundancy in the search.

These additions result in the interpreter whose top level is shown in Figure 6. Let the interpreter predicate `solve(Gs,Index,L)` mean that the conjunction of goals `Gs` is to be solved, and `L`, `Index` are numbers representing the size of `Gs` and the maximum size of the stack of goals.

```
go(Index) :-
   solve([false],Index,1).
solve([G|Gs],Index,L) :-
   hornClause(G,Cs,B), solveConstraints(Cs),
   length(B,L1), L2 is L1+L-1, L2 =< Index,
   perm(B,B1), append(B1,Gs,Gs1),
   solve(Gs1,Index,L2).
solve([],_,_).
```

Figure 6: Interpreter for linearisation

**Partial evaluation of the interpreter.**   Given a set of facts of the form `hornClause(G,Cs,B)` representing the Horn clauses to be linearised, and some value of `Index`, the interpreter can be partially evaluated. We use Logen [26] to perform the partial evaluation with respect to a call to `go(Index)`, which initiates a proof of the goal `false` (see first clause of interpreter). All interpreter computations are partially evaluated except for the calls to `solve(Gs,Index,L)` and the execution of constraints within the goal `solveConstraints(Cs)`. Furthermore Logen performs standard structure-flattening and predicate renaming operations, yielding a set of clauses of the form `solve_i(X) :- Cs, solve_j(Y)`, where `solve_i(X)` and `solve_j(Y)` are instantiations of `solve(Gs,Index,L)` and `Cs` is a constraint. Thus the resulting clauses are linear, and furthermore preserve the meaning of the original clauses as given by the interpreter, by correctness of the partial evaluation procedure. The linearisation procedure is independent of the constraint theory underlying the clauses.

**Proposition 2** *Let P be a program and $P^{\leq k}$ ($k \geq 0$) be its k-dimension-bounded program. Let i be the maximum number of atoms in clause bodies of P. Let* `Index` $= (i - 1) * k + 1$. *Let $P'$ be a partial evaluation of the interpreter in Figure 6, with respect to P and the goal* `go(Index)`. *Then $P^{\leq k} \models false^{\leq k}$ iff $P' \models$* `go(Index)`.

Furthermore $P'$ is linear if the partial evaluator follows the strategy described above. Combining Propositions 2 and 1, we conclude that $P' \models$ `go(Index)` $\Rightarrow P \models false$.

  Note that linearisation required partial evaluation of the `perm` predicate, giving a blow-up in program size related to the length of the clause bodies. This is further discussed at the end of Section 5.

## 3.2   Obtaining linear over-approximations with a partial model

First we note that the set of predicates in $P^{\leq k}$ is a subset of the set of predicates in $P^{\leq k+1}$. Given a model $M$ for the predicates in $P^{\leq k}$, $P^{\leq k+1}$ can be linearised if we replace each occurrence of a predicate from $P^{\leq k}$ in the body of a clause in $P^{\leq k+1}$ with the corresponding constraint from the model $M$. The resulting set of clauses is linear since $P^{\leq k+1}$ contains at most one predicate in its body from $P^{\leq k+1}$ which is not in $P^{\leq k}$. Furthermore if $P^{\leq k+1}$ has a model then so does the set of clauses resulting from the replacement; the converse is however not the case since the model $M$ represents an over-approximation of $P^{\leq k}$. An example is given in Section 4.

  More generally, we can replace any subset of the occurrences of predicates from $P^{\leq k}$ in $P^{\leq k+1}$. We summarise this in the following lemma.

**Lemma 1 (Linear over-approximation)** *Let M be a model of the predicates in $P^{\leq k}$, represented by a set of "constrained facts" $p(X) \leftarrow \mathscr{C}$ where p is a predicate in $P^{\leq k}$. Let $P'$ be any set of clauses obtained from $P^{\leq k+1}$ by replacing some of the occurrences of predicates $p(X)$ from $P^{\leq k}$ in the bodies of clauses in $P^{\leq k+1}$ with their corresponding interpretation $\mathscr{C}$ in M. Then*

  1. *If $P^{\leq k+1}$ has a model then so does $P'$;*

  2. *If $P'$ contains no predicate from $P^{\leq k}$, then $P'$ is linear.*

# 4   Algorithm for solving sets of non-linear Horn clauses

A basic procedure for solving a set of non-linear Horn clauses using a linear Horn clause solver is presented in Algorithms 1 and 2. We use the term "linear solver" for linear Horn clause solver for brevity. The main procedure **SOLVE**(*P*) takes a set of non-linear Horn clauses *P* as input and outputs (upon termination) *(safe, solution)* if *P* is solvable or *(unsafe, counterexample)* otherwise. We represent counterexample as a *trace tree*. For a linear program it corresponds to a sequence of clauses used to derive a counterexample.

**Definition 5** ($S_{|t}$) *Let S be an interpretation of a set of Horn clauses P. Let t be any trace tree for some atom A in P (Definition 2) and let $A_t$ be the set of heads of clauses with identifiers in t. Then $S_{|t}$ is defined to be the set*

$$S_{|t} = \{(H \leftarrow \mathscr{C}) \mid (H \leftarrow \mathscr{C}) \in S \wedge H \notin A_t\}.$$

Informally, the derivation corresponding to *t* does not use any predicate interpreted by $S_{|t}$. This notion is used in Algorithm 2.

  Algorithm 2 is an extended version of Algorithm 1, which uses the solution for $P^{\leq k}$ to help to linearise $P^{\leq k+1}$ and also allows a more refined termination condition based on whether or not the solution for $P^{\leq k}$ is used in constructing a counterexample for $P^{\leq k+1}$.

  The procedures make use of several sub-procedures which will be described next.

### 4.1   Components of the algorithm

- **KDIM($P$,$k$)**: produces an at-most-$k$-dimension program $P^{\leq k}$ (Definition 3). By definition, $P^{\leq k}$ is linear for $k = 0$. For our example program presented in Figure 1, $Fib^{\leq 0}$ is shown in Figure 3 which is linear since there is at-most one non-constraint atom in the body of each clauses.

- **SOLVE_LINEAR($P'$)**: solves a set of linear Horn clauses $P'$. We assume the following about a linear solver: (i) if it terminates on $P'$, then it returns either *safe* and a *solution* or *unsafe* and a *counterexample*; (ii) it is sound, that is, if it returns a *solution S* for $P'$ then $P'$ has a model and $S$ is a solution (model) of $P'$; if it returns *unsafe* and a counterexample *cEx* then $P'$ is unsafe and *cEx* is a witness. In our setting (Algorithms 1 and 2), $P'$ corresponds to a linearised version of $P^{\leq k}$ for some $P$ and $k \geq 0$. For technical reasons, the top level predicate *false$^{=k}$* of $P^{\leq k}$ if any, is renamed to *false* before passing to a linear solver.

  In essence, any Horn clause solver which complies with our assumption, for example QARMC [17], Convex polyhedral analyser [23], ELDARICA [20] etc. can be used in a black-box fashion but in this paper, we make use of a solver described in [23], which is based on abstract interpretation [6] over the domain of convex polyhedra [7] but without refinement using finite tree automata. The solver produces the following solution for the program in Figure 3. We can check it is in fact a solution (model).

  ```
  fib(0)(A,B) :- [-A>= -1,A>=0,B=A].
  fib[0](A,B) :- [-A>= -1,A>=0,B=A].
  false[0] :- <>. % <>  means that there is no model for false[0],
  %so we can discard it
  ```

- **LINEARISE($P$,$k$,$S$)** generates a linear set of clauses from $P^{\leq k}$ and an interpretation $S$ for $P^{\leq k}$. Let $S$ be a set of constrained facts of the form $p(X) \leftarrow \mathscr{C}$, where $p$ is a predicate from $P^{\leq k}$, the procedure replaces every clause from $P^{\leq k}$ with head $p(X)$ by $p(X) \leftarrow \mathscr{C}$. This produces a set of clauses say $P'$. Then the procedure **LINEARISE_PE($P'$,$Index$)** is called, which is the linearisation procedure based on partial evaluation described in Section 3 where *Index* is a bound for the stack usage for linearising $P^{\leq k}$.

An excerpt from $Fib^{\leq 1}$ is shown below.

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

After reusing the solution obtained for $Fib^{\leq 0}$ and linearising, we obtain the following set of linear clauses.

```
false(1) :-  A>5, B<A, fib(1)(A,B).
fib(1)(A,B) :-  -A>= -2, A>1, A-C=2,  B-D=1, fib(1)(C,D).
```

Continuing to run our algorithm, the following solution obtained for $Fib^{\leq 2}$ becomes a solution for the program in Figure 1 (the original program) and the algorithm terminates.

```
fib(0)(A,B) :- [-A>= -1,A>=0,B=1].
fib[0](A,B) :- [-A>= -1,A>=0,B=1].
```

```
fib(1)(A,B) :- [A>=2,A+ -B=0].
fib[1](A,B) :- [A+ -B>= -1,B>=1,-A+B>=0].
fib(2)(A,B) :- [A>=4,-2*A+B>= -3].
fib[2](A,B) :- [A>=0,B>=1,-A+B>=0].
```

---

**Algorithm 1:** Algorithm for solving a set of Horn clauses

1   Procedure **SOLVE**($P$)
    **Input**: A set of CHCs $P$
    **Output**: *(safe, solution), (unsafe, cex)*
2   $k \leftarrow 0$;
3   $P' \leftarrow$ **LINEARISE**$(P, k, \emptyset)$;
4   $(status, Result) \leftarrow$ **SOLVE_LINEAR**$(P')$ ;        /* Result is a solution or a cex */
5   **if** *status = safe* **then**
6      **if** *($Result^{\uparrow P^{\leq k}}$ is a solution of P)* **then return** $(safe, Result^{\uparrow P^{\leq k}})$ ;
7      $k \leftarrow k+1$;
8   **else**
9      **return** $(unsafe, Result)$ ;                 /* Result is a cex */
10   **goto** 3;

---

**Algorithm 2:** Algorithm for solving a set of Horn clauses, with reuse of lower dimension solutions

1   Procedure **SOLVE**($P$)
    **Input**: A set of CHCs $P$
    **Output**: *(safe, solution), (unsafe, cex)*
2   $k \leftarrow 0$;
3   $S \leftarrow \emptyset$;
4   $P' \leftarrow$ **LINEARISE**$(P, k, S)$;
5   $(status, Result) \leftarrow$ **SOLVE_LINEAR**$(P')$ ;      /* Result is a solution or a cex */
6   **if** *status = safe* **then**
7      **if** *($Result^{\uparrow P^{\leq k}}$ is a solution of P)* **then return** $(safe, Result^{\uparrow P^{\leq k}})$ ;
8      $k \leftarrow k+1$;
9      $S \leftarrow Result$;
10   **else**
11      **if** $S = S_{|Result}$ **then return** $(unsafe, Result)$; ;      /* Result is a linear cex */
12      $S \leftarrow S_{|Result}$;                   /* $S_{|Result}$: Definition 5 */
13   **goto** 4;

---

## 4.2   Reuse of solutions, refinement and linearisation

Algorithm 2 solves non-linear Horn clauses $P$ in essentially the same way as Algorithm 1, but incorporates a *refinement* phase in the case that the linear solver finds a counterexample. This counterexample

---

**Algorithm 3:** Algorithm for linearising a set of clauses

---

1 Procedure **LINEARISE**(*P, k, S*)

  **Input**: A set of CHCs *P*, an integer *k* and a set of constrained facts *S*

  **Output**: A linearised set of clauses $P_{lin}$

2  $P^{\leq k} \leftarrow$ **KDIM**(*P,k*) ;                                        `/* Definition 3 */`

3  $P' \leftarrow$ **SUBSTITUTE**($P^{\leq k}$,*S*);        `/* substitute atoms of` $P^{\leq k}$ `with their`

  `interpretations from S */`

4  $Index \leftarrow (i-1)*k+1$ ;    `/* where i is the maximal number of body atoms of P */`

5  $P_{lin} \leftarrow$ **LINEARISE_PE**($P'$, *Index*) ;                    `/* Section 3.1 */`

6 **return** $P_{lin}$

---

possibly uses some of the model of the lower-dimension predicates *S*, in which case it is not certain whether it is a false alarm or a real counterexample. If the counterexample did use some of the predicate solutions from *S*, then we discard those solutions (Algorithm 2, line 12) and return to the linearisation step. If the counterexample does not use any predicate solutions from *S*, then it is a real counterexample (Algorithm 2, line 12). We will clarify this with an example program (linear for simplicity) shown below.

```
c1. false:- X=0, p(X).
c2. false:- q(X).
c3. p(X):- X>0.
c4. q(X):-X=0.
```

Suppose we have an approximate solution $S = \{p(X) : -true\}$ for the predicate $p(X)$. Using this solution, the above program is transformed into the following program.

```
c1. false:- X=0, p(X).
c2. false:- q(X).
c3. p(X):- true. (approximate solution)
c4. q(X):-X=0.
```

The trace $c_1(c_3)$ is a counterexample for this transformed program but not to the original program (since it uses approximate solution for the predicate *p*). However the trace $c_2(c_4)$ is a counterexample for this program as well as to the original since it does not use any approximate solution for the predicates appearing in the counterexample.

A schematic overview of Algorithm 2 is shown in Figure 7. At each iteration of the *abstraction-refinement* loop, the at-most-*k*-dimension under-approximation of *P* is computed, then linearised and solved using a solver for linear Horn clauses.
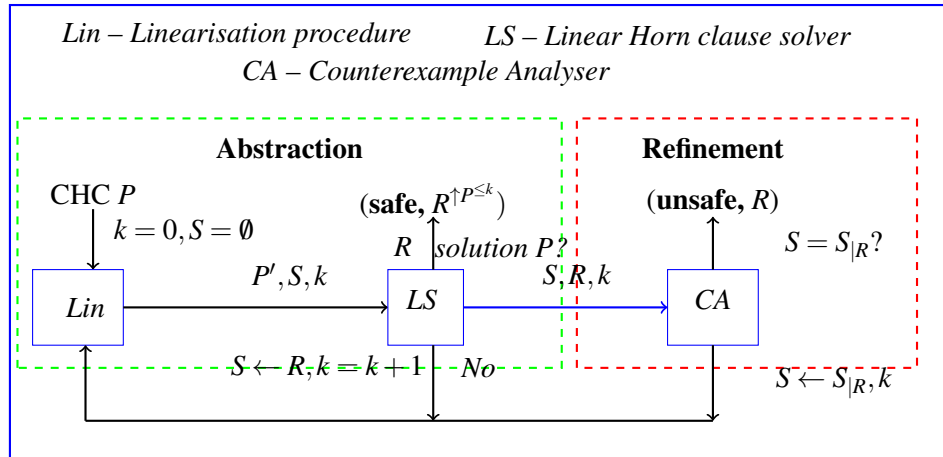
Figure 7: *Abstraction-refinement scheme for solving non-linear Horn clauses using a solver for linear Horn clauses. $P'$ is a set of linear CHC obtained by linearising the at-most-k-dimension underapproximation, $P^{\leq k}$, of P.*

The soundness of Algorithms 1 and 2 is captured by Proposition 3.

**Proposition 3 (Soundness)** *If Algorithm 1 or 2 returns safe and a solution S for a set of clauses P then P is safe and S is in fact a solution of P; if it returns unsafe and a counterexample cEx then P is unsafe and cEx is a witness.*

Another property of the Algorithm 2 is that of progress, that is, the same counterexample does not arise more than once.

## 5   Experimental results

We made a prototype implementation of Algorithm 2 in the tool called *LHornSolver*[1]. It uses the solver described in [23] without refinement as a linear Horn clause solver. *LHornSolver* is written in Ciao prolog [19] and is interfaced with Parma Polyhedra Library [2] and Yices SMT solver [10] for handling constraints. Then we carried out an experiment on a set of 44 non-linear CHC verification problems taken from the repository[2] of software verification benchmarks, the recursive category of SV-COMP[3] [4] and the tool QARMC. The experiments were run on a MAC computer running OS X on 2.3 GHz Intel core i7 processor and 8 GB memory. The benchmarks that we use in the experiments are not beyond the capabilities of existing solvers, but they are challenging. These programs are first translated to Prolog syntax using the tools ELDARICA[4] [20] and SeaHorn [18]. Our aim with these experiments is to explore: (1) whether using a linear solver for non-linear problem solving is practical; (2) the relationship between the solvability of a problem and its dimension; and (3) how the current results compare with the results using the state of the art non-linear Horn clause verification tool (in our case RAHFT [23]). The results are summarized in Table 1.

---

[1]https://github.com/bishoksan/LHornSolver
[2]https://svn.sosy-lab.org/software/sv-benchmarks/trunk/clauses/LIA/Eldarica/RECUR/
[3]http://sv-comp.sosy-lab.org/2015/benchmarks.php
[4]https://github.com/uuverifiers/eldarica

| Program | RAHFT | | | LHornSolver | | |
|---|---|---|---|---|---|---|
| | Safety | # iter. | Time (s) | Safety | #iter. | Time (s) |
| Addition03_false-unreach | safe | 2 | < 1 | ? | ? | ? |
| McCarthy91_false-unreach | unsafe | 0 | < 1 | ? | ? | ? |
| addition.nts.pl | safe | 0 | < 1 | safe | 1 | < 1 |
| bfprt.nts.pl | safe | 0 | < 1 | safe | 2 | 4 |
| binarysearch.nts.pl | safe | 0 | < 1 | safe | 1 | 1.1 |
| countZero.nts.pl | safe | 0 | < 1 | safe | 1 | < 1 |
| eq.horn | unsafe | 0 | < 1 | unsafe | 2 | < 1 |
| fib.pl | safe | 0 | < 1 | ? | ? | ? |
| identity.nts.pl | safe | 0 | < 1 | safe | 1 | < 1 |
| merge.nts.pl | safe | 0 | < 1 | safe | 1 | 1.7 |
| palindrome.nts.pl | safe | 0 | < 1 | safe | 1 | < 1 |
| parity.nts.pl | unsafe | 1 | < 1 | ? | ? | ? |
| remainder.nts.pl | unsafe | 0 | < 1 | unsafe | 1 | < 1 |
| revlen.pl | safe | 0 | < 1 | safe | 1 | < 1 |
| running.nts.pl | unsafe | 1 | < 1 | ? | ? | ? |
| sum_10x0_false-unreach | unsafe | 10 | 10 | ? | ? | ? |
| sum_non_eq_false-unreach | unsafe | 0 | < 1 | ? | ? | ? |
| suma1.horn | unsafe | 0 | < 1 | unsafe | 1 | < 1 |
| suma2.horn | unsafe | 0 | < 1 | unsafe | 2 | < 1 |
| summ_SG1.r.horn | safe | 0 | < 1 | ? | ? | ? |
| summ_SG2.r.horn | safe | 8 | 78 | ? | ? | ? |
| summ_SG3.horn | safe | 0 | < 1 | safe | 1 | < 1 |
| summ_b.horn | safe | 2 | 1.7 | ? | ? | ? |
| summ_binsearch.horn | safe | 1 | 3 | ? | ? | ? |
| summ_cil.casts.horn | safe | 0 | < 1 | safe | 1 | < 1 |
| summ_formals.horn | safe | 0 | < 1 | safe | 1 | < 1 |
| summ_g.horn | safe | 0 | < 1 | ? | ? | ? |
| summ_globals.horn | safe | 0 | < 1 | safe | 1 | < 1 |
| summ_h.horn | safe | 0 | < 1 | safe | 2 | < 1 |
| summ_local-ctx-call.horn | safe | 0 | < 1 | safe | 1 | < 1 |
| summ_locals.horn | safe | 0 | < 1 | ? | ? | ? |
| summ_locals2.horn | safe | 0 | < 1 | safe | 1 | < 1 |
| summ_locals3.horn | safe | 0 | < 1 | safe | 1 | < 1 |
| summ_locals4.horn | safe | 0 | < 1 | safe | 2 | 2.2 |
| summ_mccarthy2.horn | safe | 3 | 5 | ? | ? | ? |
| summ_multi-call.horn | safe | 0 | < 1 | safe | 1 | < 1 |
| summ_nested.horn | safe | 0 | < 1 | safe | 1 | < 1 |
| summ_ptr_assign.horn | safe | 0 | < 1 | safe | 1 | < 1 |
| summ_recursive.horn | safe | 0 | < 1 | ? | ? | ? |
| summ_rholocal.horn | safe | 0 | < 1 | safe | 1 | < 1 |
| summ_rholocal2.horn | safe | 0 | < 1 | safe | 1 | < 1 |
| summ_slicing.horn | safe | 0 | < 1 | ? | ? | ? |
| summ_summs.horn | safe | 0 | < 1 | ? | ? | ? |
| summ_typedef.horn | safe | 0 | < 1 | safe | 1 | < 1 |
| summ_x.horn | safe | 0 | < 1 | ? | ? | ? |
| Average | | 0.64 | 2.3 | | 1.185 | < 1 |

Table 1: Experimental results on non-linear CHC verification problems with a timeout of 5 minutes.

In the table *Program* represents a program, *Safety* represents a verification result, *#iter.* and *Time (s)* successively represent the number of refinement iterations and the time in seconds need to solve a program using both RAHFT and LHornSolver. It is to note that the underlying abstract interpreter, that is, the convex polyhedral analyser (CPA) is the same for both *RAHFT* and *LHornSolver* but *LHornSolver* uses it to solve linear Horn clauses though the CPA is not optimised for linear problems. The column *#iter.* for LHornSolver represents a value of $k$ for which a solution of $P^{\leq k}$ (under-approximation) of a set of clauses $P$ becomes a solution for $P$ or $P^{\leq k}$ becomes unsafe. The symbol "?" means that the result is unknown within the given time bound. The result "safe" means that the program is safe (solvable) and "unsafe" means it is unsafe.

*LHornSolver* solves 27 out of 44 (about 61%) problems within a second. In most of these problems, a solution of an under approximation ($P^{\leq k}$) becomes a solution for the original program or $P^{\leq k}$ becomes unsafe for a fairly small value of $k$ (1 or 2). This suggests that the solvability of a problem is shallow with respect to its dimension. This demonstrates the feasibility of solving a set of non-linear Horn clauses using a solver for linear Horn clauses.

In contrast, RAHFT solves all the problem. The difference in results maybe due to the following reason: the linear solver that is used in LHornSolver is the CPA (without refinement in contrast to [23]). The solver terminates but produces *false alarms*. If we use CPA with refinement as in [23], then we lose predicates names (due to program transformation), so the solution or counterexamples produced by the tool do not correspond to the original program (it is very hard to keep track of the changes). This hinders the reuse of solution from lower dimension to linearise program of higher dimension or refine it using the counterexample trace. Other solvers which don't modify the programs but produce solutions or counterexamples can be used as a linear solver in principle and we leave it for the future work. Another disadvantage of using CPA is that, if it cannot solve a linear program, then it emits an abstract trace which is checked for a feasibility. If it is spurious then LHornSolver returns with *unknown* (in principle we can refine the program but the refinement will have the problem as mentioned above). So it is highly unlikely that the trace picked by the tool non-deterministically results to be a real counterexample. We noticed in our experiments that the trace picked was spurious most of the times and *LHornSolver* immediately returned "unknown" answer before the timeout. This also explains why solving time of *LHornSolver* is less than that of *RAHFT*.

The interpreter described in Figure 6 computed a *permutation* of the atoms in a clause body; partial evaluation of the permutation procedure can cause a blow-up of the size of the linearised program, relative to the number of atoms in clause bodies. During our experiment we found that the maximum number of atoms in the bodies of the clauses in our benchmark programs was 5 and the value of $k$ was relatively small ($k = 0 \ldots 2$). The permutation procedure can be avoided if we first generate an at-most-$k$-dimension program whose body atoms are ordered by increasing dimension. This needs unfolding of the $\varepsilon$-clauses, since atoms whose predicate is $p^{\leq d}$ cannot be ordered directly; only atoms with predicates of the form $p^{=d}$ can be ordered. We have not yet evaluated the trade-offs in these two approaches.

## 6 Related Work

In the world of Horn clause solvers, after fixing a constraint theory, we can distinguish solvers depending on whether they can handle general non-linear Horn clauses or not. A majority of solvers [18, 16, 30, 29, 23] handle non-linear Horn clauses but there are notable exceptions like VeriMAP [8] or Sally[5]. For both VeriMAP and Sally, their underlying reasoning engine handles only linear Horn clauses which

---

[5]`https://github.com/SRI-CSL/sally`

restricts, in principle, their applicability. Our contribution is to lift this restriction by allowing those tools to be applied on arbitrary sets of Horn Clauses, linear or not, through a linearisation procedure that underapproximates the set of solutions. We give empirical evidence that this underapproximation often provides enough coverage to enable the verification of the original set of Horn clauses. To summarize, we allow solvers with restrictions to be applied on any input at the price of an under-approximation which often results in full coverage.

Our linearisation method based on partial evaluation described in Section 3.1 is related to the linearisation method based on *fold-unfold transformations* described by De Angelis *et al.* [9]. While their procedure transforms the target set of clauses directly, we transform an interpreter for the clauses using a generic partial evaluation procedure. Any clause transformation procedure could be formulated as a meta-program and partial evaluation applied to that program to yield the specified transformation. Thus neither approach offers any more power than the other. However the use of partial evaluation is arguably more flexible. The interpreter that is partially evaluated in our procedure is a standard interpreter for Horn clauses, modified with a bound on the size of goals, directly incorporating a general result that there is an upper bound on the size of goals in derivations with dimension-bounded programs. This provides a very generic starting point for the transformation with an explicit relation to the semantics of the clauses. A whole family of similar transformations could be formulated by varying the interpreter (for example using breadth-first search). The procedure in [9] is tailored to a restrictive setting where only goal clauses (integrity constraints) are non-linear and rest of the clauses are linear; correctness has to be established for that case.

Ganty, Iosif and Konečný [15] used the notion of *tree dimension* for computing summaries of procedural programs by underapproximating them. Roughly speaking, they compute procedure summaries iteratively, starting from the program behaviors captured by derivation trees of dimension 0. Then they reuse these summaries to compute summaries for program behaviors captured by derivation trees of dimension 1 and so on for 2, 3, etc. Kafle, Gallagher and Ganty [24] adapted the idea of dimension-based underapproximations to the setting of Horn clause systems. They gave empirical evidence supporting the thesis that for small values of the dimension the solutions are general enough to hold for every dimension. Their approach still required the use of general Horn-clause solvers capable of handling non-linear clauses. In this paper, we lift this requirement and allow the use of solvers for linear clauses only. Moreover, we provide an abstraction refinement loop that enables the solutions for lower dimension to be reused when searching for solutions in higher dimension.

# 7   Conclusion and future work

We presented an *abstraction-refinement* approach for solving a set of non-linear Horn clauses using an off-the-shelf linear Horn clause solver. It was achieved through a linearisation of a dimension bounded set of Horn clauses (which are known to be linearisable) using partial evaluation and the use of a linear Horn clause solver. Experiment on a set of non-linear Horn clause verification problems using our approach shows that the approach is feasible (a linear solver can be used for solving non-linear problems) and the solvability of a problem is shallow with respect to its dimension.

A linear set of clauses is essentially a transition system. Many tools exist whose input languages have a form such as C programs (without procedure calls), control flow graphs, Boogie programs, and such formalisms whose semantics is usually given as a transitions system. The results of this paper suggest that such tools could be applied to the verification of non-linear Horn clauses.

In the future, we plan to compare our results with the results from a specialised linear Horn clause

solver like VeriMap and other non-linear Horn clause solvers. We also plan to experiment with different linearisation strategies for Horn clauses and study their effects in Horn clause verification.

## Acknowledgement

The authors would like to thank José F. Morales for his help with Ciao Prolog foreign language interface and some parts of the implementation.

## References

[1] Foto N. Afrati, Manolis Gergatsoulis & Francesca Toni (2003): *Linearisability on Datalog programs. Theor. Comput. Sci.* 308(1-3), pp. 199–226, doi:10.1016/S0304-3975(02)00730-2.

[2] Roberto Bagnara, Patricia M. Hill & Enea Zaffanella (2008): *The Parma Polyhedra Library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. Sci. Comput. Program.* 72(1-2), pp. 3–21, doi:10.1016/j.scico.2007.08.001.

[3] Christel Baier & Cesare Tinelli, editors (2015): *Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings. Lecture Notes in Computer Science* 9035, Springer, doi:10.1007/978-3-662-46681-0.

[4] Dirk Beyer (2015): *Software Verification and Verifiable Witnesses - (Report on SV-COMP 2015).* In Baier & Tinelli [3], pp. 401–416, doi:10.1007/978-3-662-46681-0_31.

[5] Nikolaj Bjørner, Kenneth L. McMillan & Andrey Rybalchenko (2013): *On Solving Universally Quantified Horn Clauses.* In Francesco Logozzo & Manuel Fähndrich, editors: *SAS, LNCS* 7935, Springer, pp. 105–125. Available at `http://dx.doi.org/10.1007/978-3-642-38856-9_8`.

[6] Patrick Cousot & Radhia Cousot (1977): *Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints.* In Robert M. Graham, Michael A. Harrison & Ravi Sethi, editors: *POPL*, ACM, pp. 238–252. Available at `http://doi.acm.org/10.1145/512950.512973`.

[7] Patrick Cousot & Nicolas Halbwachs (1978): *POPL.* ACM Press, pp. 84–96, doi:10.1145/512760.512770. Available at `http://dl.acm.org/citation.cfm?id=512760`.

[8] Emanuele De Angelis, Fabio Fioravanti, Alberto Pettorossi & Maurizio Proietti (2014): *VeriMAP: A Tool for Verifying Programs through Transformations.* In Erika Ábrahám & Klaus Havelund, editors: *TACAS, LNCS* 8413, Springer, pp. 568–574. Available at `http://dx.doi.org/10.1007/978-3-642-54862-8_47`.

[9] Emanuele De Angelis, Fabio Fioravanti, Alberto Pettorossi & Maurizio Proietti (2015): *Proving correctness of imperative programs by linearizing constrained Horn clauses.* TPLP 15(4-5), pp. 635–650, doi:10.1017/S1471068415000289.

[10] Bruno Dutertre (2014): *Yices 2.2.* In Armin Biere & Roderick Bloem, editors: *Computer-Aided Verification (CAV'2014), Lecture Notes in Computer Science* 8559, Springer, pp. 737–744, doi:10.1007/978-3-319-08867-9_49.

[11] Javier Esparza, Pierre Ganty, Stefan Kiefer & Michael Luttenberger (2011): *Parikh's theorem: A simple and direct automaton construction. Inf. Process. Lett.* 111(12), pp. 614–619, doi:10.1016/j.ipl.2011.03.019.

[12] Javier Esparza, Stefan Kiefer & Michael Luttenberger (2007): *On Fixed Point Equations over Commutative Semirings.* In: *STACS 2007, 24th Annual Symposium on Theoretical Aspects of Computer Science, Proceedings, LNCS* 4393, Springer, pp. 296–307, doi:10.1007/978-3-540-70918-3_26.

[13] J. P. Gallagher (1986): *Transforming Logic Programs by Specialising Interpreters.* In: *Proceedings of the 7th European Conference on Artificial Intelligence (ECAI-86), Brighton*, pp. 109–122.

[14] John P. Gallagher (1993): *Tutorial on Specialisation of Logic Programs*. In David A. Schmidt, editor: *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation, PEPM'93, Copenhagen, Denmark, June 14-16, 1993*, ACM, pp. 88–98, doi:10.1145/154630.154640. Available at `http://dl.acm.org/citation.cfm?id=154630`.

[15] Pierre Ganty, Radu Iosif & Filip Konečný (2013): *Underapproximation of Procedure Summaries for Integer Programs*. In Nir Piterman & Scott A. Smolka, editors: *TACAS 2013. Proceedings*, *Lecture Notes in Computer Science* 7795, Springer, pp. 245–259, doi:10.1007/978-3-642-36742-7_18.

[16] Sergey Grebenshchikov, Ashutosh Gupta, Nuno P. Lopes, Corneliu Popeea & Andrey Rybalchenko (2012): *HSF(C): A Software Verifier Based on Horn Clauses - (Competition Contribution)*. In Cormac Flanagan & Barbara König, editors: *TACAS*, *LNCS* 7214, Springer, pp. 549–551. Available at `http://dx.doi.org/10.1007/978-3-642-28756-5_46`.

[17] Sergey Grebenshchikov, Nuno P. Lopes, Corneliu Popeea & Andrey Rybalchenko (2012): *Synthesizing software verifiers from proof rules*. In Jan Vitek, Haibo Lin & Frank Tip, editors: *ACM SIGPLAN PLDI*, ACM, pp. 405–416, doi:10.1145/2254064.2254112. Available at `http://dl.acm.org/citation.cfm?id=2254064`.

[18] Arie Gurfinkel, Temesghen Kahsai & Jorge A. Navas (2015): *SeaHorn: A Framework for Verifying C Programs (Competition Contribution)*. In Baier & Tinelli [3], pp. 447–450, doi:10.1007/978-3-662-46681-0_41.

[19] Manuel V. Hermenegildo, Francisco Bueno, Manuel Carro, Pedro López-García, Edison Mera, José F. Morales & Germán Puebla (2012): *An overview of Ciao and its design philosophy*. *TPLP* 12(1-2), pp. 219–252, doi:10.1017/S1471068411000457.

[20] Hossein Hojjat, Filip Konecný, Florent Garnier, Radu Iosif, Viktor Kuncak & Philipp Rümmer (2012): *A Verification Toolkit for Numerical Transition Systems - Tool Paper*. In Dimitra Giannakopoulou & Dominique Méry, editors: *FM. Proceedings*, *Lecture Notes in Computer Science* 7436, Springer, pp. 247–251, doi:10.1007/978-3-642-32759-9_21.

[21] N. Jones, C.K. Gomard & P. Sestoft (1993): *Partial Evaluation and Automatic Software Generation*. Prentice Hall.

[22] Neil D. Jones (2004): *Transformation by interpreter specialisation*. *Sci. Comput. Program.* 52, pp. 307–339, doi:10.1016/j.scico.2004.03.010.

[23] Bishoksan Kafle & John P Gallagher (2015): *Horn clause verification with convex polyhedral abstraction and tree automata-based refinement*. *Computer Languages, Systems & Structures*, doi:10.1016/j.cl.2015.11.001.

[24] Bishoksan Kafle, John P. Gallagher & Pierre Ganty (2015): *Decomposition by tree dimension in Horn clause verification*. In Alexei Lisitsa, Andrei P. Nemytykh & Alberto Pettorossi, editors: *VPT.*, *EPTCS* 199, pp. 1–14, doi:10.4204/EPTCS.199.1.

[25] Michael Leuschel (1994): *Partial Evaluation of the "Real Thing"*. In Laurent Fribourg & Franco Turini, editors: *LOPSTR, Proceedings*, *Lecture Notes in Computer Science* 883, Springer, pp. 122–137, doi:10.1007/3-540-58792-6_8.

[26] Michael Leuschel, Daniel Elphick, Mauricio Varea, Stephen-John Craig & Marc Fontaine (2006): *The Ecce and Logen partial evaluators and their web interfaces*. In John Hatcliff & Frank Tip, editors: *PEPM 2006*, ACM, pp. 88–94, doi:10.1145/1111542.1111557.

[27] Michael Leuschel & Germán Vidal (2014): *Fast offline partial evaluation of logic programs*. *Inf. Comput.* 235, pp. 70–97, doi:10.1016/j.ic.2014.01.005.

[28] Michael Luttenberger & Maximilian Schlund (2016): *Convergence of Newton's Method over Commutative Semirings*. *Inf. Comput.* 246, pp. 43–61, doi:10.1016/j.ic.2015.11.008.

[29] Kenneth L. McMillan & Andrey Rybalchenko (2013): *Solving Constrained Horn Clauses using Interpolation*. Technical Report, Microsoft Research.

[30] Philipp Rümmer, Hossein Hojjat & Viktor Kuncak (2013): *Disjunctive Interpolants for Horn-Clause Verification*. In Natasha Sharygina & Helmut Veith, editors: *CAV*, *Lecture Notes in Computer Science* 8044, Springer, pp. 347–363, doi:10.1007/978-3-642-39799-8. Available at `10.1007/978-3-642-39799-8_24`.