Tree dimension in verification of constrained Horn clauses

Bishoksan Kafle
The University of Melbourne
E-mail: bishoksank@unimelb.edu.au

John P. Gallagher
Roskilde University and IMDEA Software Institute
E-mail: jpg@ruc.dk

Pierre Ganty
IMDEA Software Institute
Email: pierre.ganty@imdea.org

submitted 1 January 2003; revised 1 January 2003; accepted 1 January 2003

Abstract

In this paper we show how the notion of tree dimension can be used in the verification of constrained Horn clauses (CHCs). The dimension of a tree is a numerical measure of its branching complexity and the concept here applies to Horn clause derivation trees. Derivation trees of dimension zero correspond to derivations using linear CHCs, while trees of higher dimension arise from derivations using non-linear CHCs. We show how to instrument CHCs predicates with an extra argument for the dimension, allowing a CHC verifier to reason about bounds on the dimension of derivations. Given a set of CHCs P, we define a transformation of P yielding a *dimension bounded* set of CHCs $P^{\leq k}$. The set of derivations for $P^{\leq k}$ consists of the derivations for P that have dimension at most k. We also show how to construct a set of clauses denoted $P^{>k}$ whose derivations have dimension exceeding k. We then present algorithms using these constructions to decompose a CHC verification problem. One variation of this decomposition considers derivations of successively increasing dimension. The paper includes descriptions of implementations and experimental results.

1 Introduction

The dimension of a tree, also known as the Horton-Strahler number of a tree[1] is a numerical measure of a tree's branching complexity. The concept was originally applied to analyse flows in rivers and their tributaries and to other naturally occurring tree structures (Esparza et al. 2014). Recently it has found several applications in program analysis and verification (Esparza et al. 2010; Reps et al. 2016). In this paper we apply the notion of tree dimension in the verification of CHCs, where

[1] https://en.wikipedia.org/wiki/Strahler_number

the trees whose dimension we consider are derivation trees. Derivation trees of dimension zero correspond to derivations using linear CHCs, while trees of higher dimension arise from derivations using non-linear CHCs.

The verification of a property of a set of CHCs often involves implicitly the set of all derivation trees for that set. For example, a safety property is typically formalised as the consistency of a set of clauses, which amounts to establishing the absence of a derivation of a contradiction and requires the consideration of all derivations for the given clauses. CHCs provide a convenient representation for the statement of invariant properties of various systems including imperative programs (Grebenshchikov et al. 2012), which is again usually formalised as the absence of a derivation of some statement representing the violation of the invariant. An automated tool for finding such derivations might benefit from a divide-and-conquer strategy, decomposing the set of all derivations into smaller more manageable sets.

Tree dimension provides one such approach to decompose verification problems that involve the set of all derivations. Given a set of CHCs P and a dimension $k \geq 0$, we define a transformation yielding a set of CHCs $P^{\leq k}$ whose derivations have dimension of at most k. We can also obtain the complementary set of clauses (called $P^{>k}$) whose derivation trees have dimension at least $k+1$. Each such set of clauses ($P^{\leq k}$ and $P^{>k}$) represents an under-approximation of the original set P in the sense that they give rise to a subset of P's derivations.

Why might decomposition by dimension be useful? Firstly, the overall verification problem is reduced into simpler, but still non-trivial parts, each with an infinite number of derivations. By contrast, if one of the parts were finite, say the set of derivations of bounded depth, then the complementary part would arguably be no simpler than the original. Secondly, the particular properties of bounded dimension can be exploited. Any dimension-bounded set of clauses $P^{\leq k}$ can be linearised, while preserving key semantic properties including consistency (Kafle et al. 2016). This allows the use of tools designed and optimised for linear clauses.

We also show how to reason directly about the dimension of derivations using any CHC verification system, by instrumenting the clauses, adding an extra argument to each predicate representing the dimension.

In Section 2 we introduce the technical background of the paper. We review the notion of tree dimension and introduce the syntax and semantics of CHCs. We relate the concept of *tree dimension* to CHCs derived from imperative programs in Section 3; and present a method for instrumenting CHCs predicates with an extra argument for the dimension and verify dimension related properties using the standard CHCs solvers. In Section 3.4 we present partial evaluation algorithms to construct two versions of dimension-bounded clauses constructed from a given set of CHCs: one whose derivations are bounded in dimension from above and one whose derivations are bounded from below. The dimension-bounded sets of clauses are exploited by verification algorithms presented in Section 4. Section 5 contains a description of a prototype implementation and discusses the results obtained. Section 6 presents a discussion of related work as well as the role of dimension in using CHCs for safety verification of imperative programs. Finally, Section 7 concludes.

2 Preliminaries and formal background

A labelled tree $c(t_1, \ldots, t_k)$ $(k \geq 0)$ is a tree whose nodes are labelled by identifiers, where c is the label of the root and $t_1, \ldots, t_k$ are labelled trees, the children of the root. In this paper, all trees we consider are finite.

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. Formally, the dimension of a tree is defined as follows.

Definition 1 (*Tree dimension adapted from Esparza et al. (2007)*)
Given a labelled tree $t = c(t_1, \ldots, t_k)$, the tree dimension of t represented as $\mathit{dim}(t)$ is defined as follows:

$$\mathit{dim}(c(t_1,\ldots,t_k)) = \begin{cases} 0 & \text{if } k = 0 \\ \mathit{dim}(t_i) & \text{if } k > 0 \wedge |\{i \mid \forall j\colon \mathit{dim}(t_j) \leq \mathit{dim}(t_i)\}| = 1 \\ \mathit{dim}(t_i) + 1 & \text{if } k > 0 \wedge |\{i \mid \forall j\colon \mathit{dim}(t_j) \leq \mathit{dim}(t_i)\}| > 1 \end{cases}$$

Figure 1 shows a labelled tree $t = c_3(c_2(c_2(c_1, c_1), c_1))$ (each c_i is a node label) in graphical form and the dimension of each of its subtrees. The dimension of the root node (1 in this case) is the dimension of the tree.

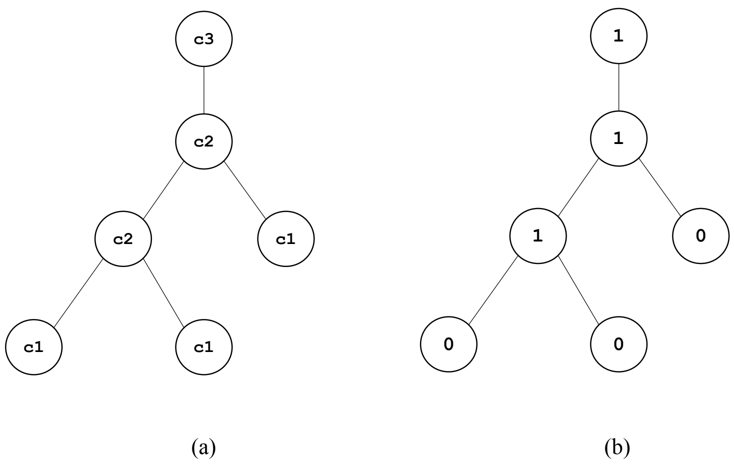

Fig. 1. (a) a labelled tree $c_3(c_2(c_2(c_1, c_1), c_1))$ and (b) the dimension of each subtree.

A constrained Horn clause (CHC) is a first-order predicate logic formula of the form $\forall \mathbf{x_0} \ldots \mathbf{x_k} (p_1(\mathbf{x_1}) \wedge \ldots \wedge p_k(\mathbf{x_k}) \wedge \phi \rightarrow p_0(\mathbf{x_0}))$, where ϕ is a finite conjunction of *constraints* with respect to some constraint theory, $\mathbf{x_0}, \ldots, \mathbf{x_k}$ are (possibly empty) tuples of *variables*, $p_0, \ldots, p_k$ are *predicate symbols*, $p_0(\mathbf{x_0})$ is the *head* of the clause and $p_1(\mathbf{x_1}) \wedge \ldots \wedge p_k(\mathbf{x_k}) \wedge \phi$ is the *body*. Following the conventions of Constraint Logic Programming (CLP), such a clause is written as $p_0(\mathbf{x_0}) \leftarrow \phi, p_1(\mathbf{x_1}), \ldots, p_k(\mathbf{x_k})$. An atomic formula, or simply *atom*, is a formula $p(\mathbf{x})$ where p is a predicate symbol

$$F_n = \begin{cases} n & \text{if } n = 0 \text{ or } 1 \\ F_{n-1} + F_{n-2} & \text{if } n > 1 \end{cases}$$

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

Fig. 2. Fibonacci function (left), its encoding as CHCs and a property *Fib* (right).

and $\mathbf{x}$ 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 CHCs P is called linear if P only contains linear clauses, otherwise it is called non-linear. *Integrity constraints* are a special kind of clause whose head is the predicate `false`. A set of constrained Horn clauses can also be regarded as a *constraint logic program*, though in this paper CHCs are not regarded as executable programs; we are concerned with verifying logical properties of CHCs.

For concrete examples of CHCs we use Prolog syntax and typewriter font, writing the implication $\leftarrow$ as `:-` and using capital letters for variable names. The constraints can also be intermixed with the body atoms. Figure 2 (right) contains an example of a set of constrained Horn clauses, called *Fib*, which encodes the Fibonacci function. The first two clauses `c1` and `c2` define the Fibonacci function and clause `c3` represents a property of the Fibonacci function expressed as an integrity constraint. `c2` is a non-linear clause while `c1` and `c3` are linear. Each CHC in a given set of CHCs is associated with an identifier, as illustrated in Figure 2.

CHC semantics. The semantics of CHCs is obtained using standard concepts from predicate logic semantics. An *interpretation* assigns to each predicate a relation over the domain of the constraint theory $\mathbb{T}$, whereas constraints have interpretations in the theory itself. In particular, the predicate `false` is always interpreted as *false*. An interpretation *satisfies* a set of formulas if each formula in the set evaluates to *true* in the interpretation in the standard way. In particular, *a model* of a set of CHCs is an interpretation in which each clause evaluates to *true*. A set of CHCs P is *consistent* if and only if it has a model. Otherwise it is *inconsistent.*

In the algorithms developed in Section 4, we consider only interpretations representable within the constraint theory by a set of *constrained facts* of the form $p(\mathbf{x}) \leftarrow \phi$ where $\mathbf{x}$ is a tuple of distinct variables and ϕ a constraint (free variables of ϕ are subset of $\mathbf{x}$) in the constraint theory underlying the CHCs. There is exactly one constrained fact for each predicate p in the set of CHCs. Such a constrained fact defines the interpretation of p as the relation $\{\mathbf{x}\theta \mid \mathbf{x}\theta \text{ is ground, and } \phi\theta \text{ holds in } \mathbb{T}\}$. We call such a set of constrained facts a *syntactic interpretation*, and if it is a model, we call it a *syntactic model.* If a set of CHCs has a syntactic model, then it has a model, but the reverse is not necessarily true. In particular, a syntactic interpretation satisfies a clause $A_0 \leftarrow \phi, A_1, \ldots, A_k$ if for constrained facts (with variables suitably renamed) $A_0 \leftarrow \phi_0$, $A_1 \leftarrow \phi_1$, ..., $A_k \leftarrow \phi_k$ in the interpretation, the formula $\phi \wedge \phi_1 \wedge \ldots \wedge \phi_k \rightarrow \phi_0$ holds in the underlying constraint theory.

In some works e.g. (Bjørner et al. 2013; McMillan and Rybalchenko 2013) a syntactic model is also called a *solution* and we use these terms interchangeably in this paper when the context is clear. When modelling safety properties of systems using CHCs, the consistency of a set of CHCs corresponds to *safety* of the system. Thus we also refer to CHCs as being *safe* or *unsafe* when they are consistent or inconsistent respectively.

AND-trees and trace trees. Derivations for CHCs are represented by AND-trees. The following definitions of derivations and trace trees are adapted from Gallagher and Lafave (1996). From now on, we assume that each clause has a unique identifier.

Definition 2 (*AND-tree or derivation tree*)
An *AND-tree* for a set of CHCs is a tree each of whose nodes is labelled by an atom, a constraint and a clause identifier such that

1. each non-leaf node corresponds to a clause (with variables suitably renamed) $A \leftarrow \phi, A_1, \ldots, A_k$ and is labelled by an atom A, constraint ϕ and has children labelled by atoms $A_1, \ldots, A_k$;
2. each leaf node corresponds to a clause $A \leftarrow \phi$ (with variables suitably renamed) and is labelled by an atom A and constraint ϕ;
3. each node is labelled with the clause identifier of the clause corresponding to the node.

The phrase "with variables suitably renamed" here and elsewhere in the paper means that variables occurring in the body but not in the head do not occur in the labels of any ancestor node. An example of an AND-tree is shown in Figure 3 (right).

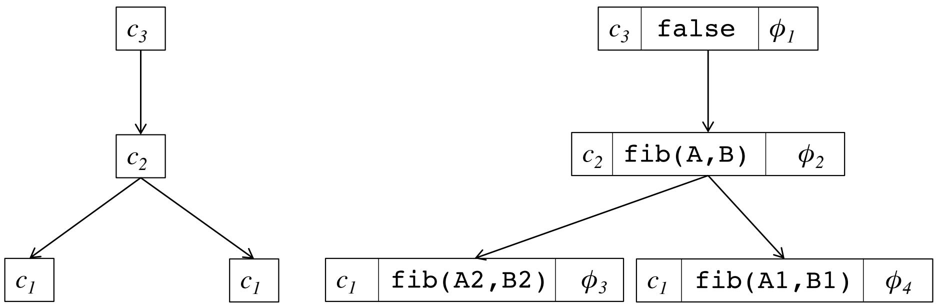

Fig. 3. A trace-term $c_3(c_2(c_1, c_1))$ of Fib (left) and its AND-tree (right), where $\phi_1 \equiv \texttt{A} > 5 \wedge \texttt{B} < \texttt{A}$; $\phi_2 \equiv \texttt{A} > 1 \wedge \texttt{A2} = \texttt{A} - 2 \wedge \texttt{A1} = \texttt{A} - 1 \wedge \texttt{B} = \texttt{B1} + \texttt{B2}$; $\phi_3 \equiv \texttt{A2} \geq 0 \wedge \texttt{A2} \leq 1 \wedge \texttt{B2} = \texttt{A2}$; $\phi_4 \equiv \texttt{A1} \geq 0 \wedge \texttt{A1} \leq 1 \wedge \texttt{B1} = \texttt{A1}$.

Definition 3 ($\mathsf{constr}(t)$)
Given an AND-tree t, the conjunction of the constraints in its node labels is represented by $\mathsf{constr}(t)$. t is *feasible* or *successful* if and only if $\mathsf{constr}(t)$ is satisfiable under $\mathbb{T}$.

Definition 4
For an atom $p(\mathbf{x})$ and a set of CHCs P we write $P \vdash p(\mathbf{x})$ if there exists a feasible AND-tree with root labelled by $p(\mathbf{x})$.

Definition 5
A feasible AND-tree with root node labelled by `false` is called a *counterexample.*

The soundness and completeness of derivation trees (Jaffar et al. 1998) implies that P is inconsistent if and only if $P \vdash$ `false`, that is, P has a counterexample. AND-trees in this paper, unless otherwise stated, are counterexamples.

An AND-tree t can be associated with a more abstract structure called a *trace tree*, which is the result of removing all node labels from t apart from the clause identifiers. The identifiers can be treated as constructors whose arity is the number of atoms in the clause body of the clause associated with the identifier. In this way we can write trace trees as terms (as in Figure 1(a)).

Thus a trace tree, together with a mapping from clause identifiers to clauses, uniquely defines an AND-tree (up to renaming of variables). Namely, $c(t_1, \dots, t_k)$ corresponds to the AND-tree whose root is labelled by the atom A and the clause $A \leftarrow \phi, A_1, \dots, A_k$ whose identifier is c, and whose children are the AND-trees corresponding to $t_1, \dots, t_k$ respectively.

Definition 6 (*Dimension of a CHC derivation*)
The dimension of a derivation for a set of CHCs is the tree dimension of the AND-tree (or associated trace tree) for the derivation.

It is clear from these definitions that the dimension of derivations is closely related to the syntactic structure of CHCs. For instance, a set of linear clauses can give rise only to derivations of dimension zero, since the corresponding trace trees are linear.

3 Tree dimension and CHCs

3.1 Programs as CHCs and their dimension

In this subsection we discuss the notion of tree dimension in relation to CHCs representing imperative programs. CHCs provide a suitable language for expressing the semantics of imperative languages (Peralta et al. 1998; Bjørner et al. 2015; Grebenshchikov et al. 2012), enabling the use of CHC tools for verification of properties of imperative programs. The clauses resulting from the translation may give rise to derivations of different dimension, depending on the style of semantic specification underlying the translation. For example, procedures call can be encoded as linear (consider inline) or non-linear CHCs giving rise to different dimensions.

Imperative programs without procedures. Consider first a language with no procedures. Let S be an imperative statement such as an assignment, conditional or loop and let a configuration $\langle S, \sigma \rangle$ stand for statement S executing in state σ. In *structural operational semantics* (Nielson and Nielson 1992) (sometimes called small-step semantics), the meaning of statements is expressed by transitions of the form $\langle S, \sigma \rangle \Rightarrow \langle S', \sigma' \rangle$, which means that executing S in state σ yields (in one execution step) the configuration $\langle S', \sigma' \rangle$. A translation based on small-step semantics then yields a corresponding linear clause $p_S(\sigma) \leftarrow \phi(\sigma, \sigma'), p_{S'}(\sigma')$, where p_S and $p_{S'}$ are predicates corresponding to statements S and S' respectively, and $\phi(\sigma, \sigma')$ is a constraint relating the variables in states σ and σ'. (Alternatively, we could choose $p_{S'}(\sigma') \leftarrow \phi(\sigma, \sigma'), p_S(\sigma)$, reversing the direction of the transition, depending on the purpose of the encoding).

By contrast, in *natural semantics* (sometimes called big-step semantics), the meaning of a statement S is expressed by a transition $\langle S, \sigma \rangle \Rightarrow \sigma'$, where this means that the execution of statement S in state σ terminates with final state σ'. A translation based on big-step semantics yields clauses that break down such a "big step" into smaller steps, using the syntactic structure of the statement.

The difference between the two styles can be clearly seen for the translation of a statement sequence $S_1; S_2$. The small-step semantics would yield linear clauses of the following form, in which the computation of S_1 is carried out step by step until S_1 terminates, and then S_2 is executed.

$$
\begin{array}{l}
p_{S_1;S_2}(\sigma) \leftarrow \phi_1, p_{S_1';S_2}(\sigma'). \\
\ldots \\
p_{S_1'';S_2}(\sigma) \leftarrow \phi_2, p_{S_2}(\sigma'). \\
p_{S_2}(\sigma) \leftarrow \phi_3, p_{S_2'}(\sigma'). \\
\ldots
\end{array}
$$

The clauses resulting from small-step semantics closely correspond to the control-flow graph of the statement, where each clause corresponds to an edge in the graph.

The big-step semantics of $S_1; S_2$ yields a clause of the form:

$$
\begin{array}{l}
p_{S_1;S_2}(\sigma, \sigma'') \leftarrow p_{S_1}(\sigma, \sigma'), p_{S_2}(\sigma', \sigma''). \\
p_{S_1}(\sigma, \sigma') \leftarrow \ldots. \\
p_{S_2}(\sigma', \sigma'') \leftarrow \ldots. \\
\ldots
\end{array}
$$

Here the first clause is non-linear, chaining the two big steps corresponding to the execution of S_1 and S_2 together to make one big step for $S_1; S_2$.

A translation from imperative code to CHCs may mix big- and small-step styles. In both styles, a loop results in a recursive predicate (that is, one that calls itself directly or indirectly). Regarding the dimension of derivations in the two styles, however, it is clear that small-step semantics yields linear clauses and hence zero-dimensional derivations, that is, all derivation trees will be linear. Big-step semantics, on the other hand, yields non-linear clauses. However, although the clauses contain recursive predicates for the loops, it can be shown that derivations using the non-linear clauses derived from big-step semantics have bounded dimension, with the bound determined by the level of statement nesting. Since clauses whose derivations are of bounded dimension can be linearised (Kafle et al. 2016), these non-linear clauses can be transformed to linear clauses. It may be asked whether the result is the same as the clauses resulting from the small-step-based translation. The answer is "not exactly". While the linearised clauses resulting from big-step semantics would correspond to the same small execution steps, there are more arguments of the predicates than in the clauses resulting from small-step semantics, representing the intermediate states that are created in the clause bodies resulting from big-step semantics.

Example 1
Given the program $P:\ \ \mathtt{x}=1;\mathtt{y}=2;$ the small-step encoding gives:

```
s(X,Y):- X1=1, Y1=Y, s1(X1,Y1).
s1(X,Y):- X1=X, Y1=2, s2(X1,Y1).
s2(X,Y):- true.
```

The big-step encoding gives:

```
b(X0,Y0,X2,Y2):- b1(X0,Y0,X1,Y1), b2(X1,Y1,X2,Y2).
b1(X0,Y0,X1,Y1):- X1=1, Y1=Y0.
b2(X1,Y1,X2,Y2):- X2=X1, Y2=2.
```

This can be straightforwardly linearised to the following, where each predicate represents the remaining computation.

```
p(X0,Y0,X2,Y2):- p1(X0,Y0,X1,Y1,X2,Y2).
p1(X0,Y0,X1,Y1,X2,Y2):- X1=1, Y1=Y0, p2(X1,Y1,X2,Y2).
p2(X1,Y1,X2,Y2):- X2=X1, Y2=2.
```

This is similar to the small-step encoding, but contains more arguments, partly due to the fact that the final state of the small-step encoding is not explicitly returned, but it is returned in the big-step encoding, and partly due to the variables representing intermediate states (for example in the predicate `p1`).

Imperative programs with procedures. Turning to a language with procedures, the small-step semantics requires the state to include a stack, whose height is unbounded in the presence of recursive procedures. The *call* and *return* statements respectively push and pop the stack. Thus the clauses, though still linear, are interpreted over a richer domain than that of the program variables themselves. In the big-step semantics no explicit stack is needed; a procedure call is represented, as other statements, with a big-step predicate expressing the relation between the states before and after the call (in effect, the predicate is a procedure summary).

As regards dimension, clauses resulting from big-step semantics of programs with recursive procedures can give rise to derivations of unbounded dimension due to the presence of recursive procedures of the form `proc p() {...p();...p();...}`, which yields a non-linear clause of this form.

$$p(\sigma_0, \sigma_n) \leftarrow \ldots, p(\sigma_1, \sigma_2), \ldots, p(\sigma_3, \sigma_4), \ldots$$

$$M(n) = \begin{cases} n - 10 & \text{if } n > 100 \\ M(M(n+11)) & \text{if } n \leq 100 \end{cases}$$

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

Fig. 4. McCarthy's 91-function and its encoding as CHCs.

We note that the clauses due to big-step semantics could still be linearised (in effect a transformation to continuation-passing form in which a stack is introduced) but this transformation is different from the linearisation of bounded-dimension clauses.

In summary, CHCs representing single imperative procedures with no calls to external procedures are naturally linear, either by direct translation based on small-step semantics (or equivalently, control-flow graphs) or by translating to dimension-bounded clauses using big-step semantics and then linearising using techniques presented in Kafle et al. (2016) and Afrati et al. (2003). On the other hand, imperative programs with procedure calls can be given a straightforward translation into CHCs using big-step semantics, but the dimension of derivations in the clauses is not in general bounded. The techniques described in this paper for decomposition based on dimension are hence mostly relevant for verification and analysis of imperative programs with recursive procedures. Other techniques for obtaining linear clauses from such programs do so at the cost of introducing a stack as a predicate argument.

3.2 Construction of dimension instrumented set of clauses

In some sets of CHCs, the dimension of derivation trees is not bounded, but there is a bound on the dimension of *feasible* derivations. Figure 4 shows the well known 91-function of McCarthy[2] together with its constrained Horn clauses representation.

Although it is possible to construct derivation trees of arbitrary dimension using the clauses in Figure 4, the dependencies between the two recursive calls to `mc91` imply that no *feasible* derivation tree for `mc91(N,X)` has dimension greater than 2. This is a meta-property of the set of clauses; however, as we now show, by instrumenting the clauses with dimensions, such properties can be expressed as safety properties of CHCs.

Definition 7 (*Dimension-instrumented clauses*)
Let P be a set of CHCs. The dimension instrumented set P_{dim} of CHCs is defined 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

$$p(\mathbf{x}) \leftarrow \phi, p_1(\mathbf{x_1}), \ldots, p_n(\mathbf{x_n})$$

  construct a clause

$$p'(\mathbf{x}, k) \leftarrow \phi, p'_1(\mathbf{x_1}, k_1), \ldots, p'_n(\mathbf{x_n}, k_n), dim([k_1, \ldots, k_n], k)$$

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

in P_{dim}, where $k_1, \ldots, k_n, k$ are fresh variables added as the final argument for their respective predicates, and $dim([k_1, \ldots, k_n], k)$ is defined according to the rules in Definition 1 for determining the dimension k of a tree from the dimensions $k_1, \ldots, k_n$ of the subtrees of the root node.

Proposition 1
Let P be a set of CHCs and P_{dim} be the set of clauses defined from P using Definition 7. Then $P_{dim} \vdash p(\mathbf{t}, k)$ if and only if the atom $p(\mathbf{t})$ has a derivation of dimension k in P.

Example 2
Figure 5 lists the dimension-instrumented version of the McCarthy 91-function.

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

Fig. 5. Dimension instrumented CHCs for the McCarthy 91-function.

3.3 Verification of dimension properties

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.

Example 3
To establish that successful derivations for the atom `mc91(X,Y)` have dimension at most 2 we add the integrity constraint `false:- mc91(N,X,K), K>2.` to the dimension-instrumented clauses of Fig 5. The clauses together with the integrity constraint are given to an automatic solver for Horn clauses, e.g. (Grebenshchikov et al. 2012; Kafle et al. 2016), which is 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 4
The dimension-instrumented version of the *Fib* clauses is shown in Figure 6. The property to be proved is that the dimension of the trees rooted at `false` of *Fib* is less than or equal to the half of *Fib*'s input value, expressed by the integrity constraint `false:- fib(A,B,K), 2K-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.

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

Fig. 6. Dimension instrumented CHCs for the Fib program.

```
%
cc(0,Y,1):- Y>0.
%
cc(X,_,0):- X<0.
cc(_,Y,0):- Y=<0.
%
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(1,1).  kinds_of_coins(2,5).  kinds_of_coins(3,10).
kinds_of_coins(4,25). kinds_of_coins(5,50).
```

Fig. 7. Counting change example encoded as a set of CHCs.

Example 5
We present the well known counting change example taken from Abelson and Sussman (1996, Chapter 1). Figure 7 shows its encoding in CHCs and the Figure 8 shows the dimension-instrumented version of the clauses. The property of interest is to relate the number of different coins (counts) with the dimension of the derivation of the predicate `cc`. 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)`.

```
cc(0,Y,1,K):- Y>0, dim([],K).
cc(X,_,0,K):- X<0, dim([],K).
cc(_,Y,0,K):- Y=<0, dim([],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, dim([K0,K1,K2],K).

kinds_of_coins(1,1,K):- dim([],K).
kinds_of_coins(2,5,K):- dim([],K).
kinds_of_coins(3,10,K):- dim([],K).
kinds_of_coins(4,25,K):- dim([],K).
kinds_of_coins(5,50,K):- dim([],K).
```

Fig. 8. Dimension instrumented CHCs for the Counting change example.

In general, verifying whether all the feasible derivation trees of a predicate in the program has a certain dimension is as challenging as proving any other non-trivial properties of the program. But in some cases the knowledge of dimension of derivation trees of a program is useful for verifying other program properties. For instance, using the knowledge that the derivation trees of McCarthy 91-function have dimension at most 2 would allow us to restrict the verification of any program property relating to successful derivations to the derivations in the dimension-bounded program $P^{\leq 2}$ (see Section 3.4.1) where P is the set of clauses for the McCarthy 91-function.

3.4 Derivation of dimension-bounded CHCs by partial evaluation

Definition 7 showed how to construct P_{dim}, an "instrumented" version of a set of CHCs P, such that $P_{dim} \vdash p(\mathbf{t}, k)$ if and only if the atom $p(\mathbf{t})$ has a derivation of dimension k in P.

In this section we apply *partial evaluation* (Jones et al. 1993) to *specialise* P_{dim} with respect to dimension constraints. In particular, from a given set of CHCs P, and a dimension bound $k \geq 0$, we generate from P_{dim} sets of clauses $P^{\leq k}$ and $P^{>k}$, whose derivations have dimension at most k and at least $k+1$ respectively.

For instance, suppose we wish to generate a set of clauses whose derivations for predicate p have dimension at most 2. Let $p(\mathbf{x}, k)$ be an atom and let $\phi(k)$ be a constraint restricting the value of the dimension argument k, where in this case $\phi(k) \equiv k \leq 2$. The goal of specialisation is to derive a set of clauses $P^{\leq 2}$, whose derivations for $p(\mathbf{x}, k)$ satisfy $\phi(k)$.

Specialisation for this example could be achieved just by replacing each clause in P_{dim} of the form $p(\mathbf{x}, k) \leftarrow Body$ by $p(\mathbf{x}, k) \leftarrow k \leq 2 \wedge Body$ in $P^{\leq 2}$. However, a derivation for $Body$ for which $k > 2$ gives an infeasible derivation for $p(\mathbf{x}, k)$; we would like to eliminate as many such infeasible derivations as possible from $P^{\leq 2}$ by partially evaluating the atom $p(\mathbf{x}, k)$ and propagating the given constraint throughout the clauses. The presence of clauses leading to infeasible derivations tends to cause analysis tools to make coarser approximations. Hence partial evaluation can increase the precision obtained when analysing or verifying dimension-constrained clauses.

Instantiation of a standard algorithm for partial evaluation. There are many variants of partial evaluation algorithms for CHCs. We present here an instantiation of the "basic algorithm" for partial evaluation of logic programs (Gallagher 1993), which is parameterised by an "unfolding rule" and an abstraction operation.

The $\mathsf{pe\_step}_P$ operation is applied to a set of constrained facts S representing goals, and returns a set of constrained facts representing subgoals obtained from the leaves of partial AND-trees for each element of S, constructed using the given unfolding rule. More precisely,

$$\begin{array}{rcl}\mathsf{pe\_step}_P(S) & = & \{p_i(\mathbf{x_i}) \leftarrow (\phi \wedge \theta)|_{\mathbf{x_i}} \mid \\ & & \quad p(\mathbf{x}) \leftarrow \theta \in S, \\ & & \quad p(\mathbf{x}) \leftarrow \phi, p_1(\mathbf{x_1}), \ldots, p_m(\mathbf{x_m}) \in P, \\ & & \quad \mathsf{SAT}(\theta \wedge \phi), \\ & & \quad 1 \leq i \leq m\}.\end{array}$$

$\phi|_{\mathbf{v}}$ stands for the constraint $\exists \mathbf{w}.\phi$, where $\mathbf{w} = \mathsf{vars}(\phi) \setminus \mathbf{v}$.

Given a set of constrained facts S_0 representing initial goals, the set $\mathsf{lfp}\ \lambda S.(S_0 \cup \mathsf{pe\_step}_P(S))$ is the set of all constrained facts obtained from nodes in AND-trees for elements of S_0. That is, if $p(\mathbf{x}) \leftarrow \theta \in S_0$, t is a feasible AND-tree with root labelled by $p(\mathbf{x})$, and $q(\mathbf{y})$ is the label of a node in t, then $q(\mathbf{y}) \leftarrow (\mathsf{constr}(t) \wedge \theta)|_{\mathbf{y}} \in \mathsf{lfp}\ \lambda S.(S_0 \cup \mathsf{pe\_step}_P(S))$. This set is usually infinite, and so we introduce an abstraction operation $\mathsf{abstract}_\Psi$ implementing a property-based abstraction (Grebenshchikov et al. 2012). This is based on a fixed set of constrained facts Ψ, and abstracts a set of constrained facts according to which properties in Ψ they satisfy. Formally, $\mathsf{abstract}_\Psi$ is defined as follows.

$$\begin{array}{lcl}
\mathsf{abstract}_\Psi(S) & = & \{\mathsf{rep}_\Psi(p(\mathbf{x}) \leftarrow \theta) \mid p(\mathbf{x}) \leftarrow \theta \in S\} \\
 & & \textit{where} \\
\mathsf{rep}_\Psi(p(\mathbf{x}) \leftarrow \theta) & = & p(\mathbf{x}) \leftarrow \bigwedge\{\psi \mid p(\mathbf{x}) \leftarrow \psi \in \Psi \wedge \theta \models \psi\}
\end{array}$$

Here, rep_ψ is applied to a constrained fact, returning its abstract "representative" with respect to the set Ψ. $\mathsf{abstract}_\Psi(S)$ generalises the constrained facts in S; for every constrained fact $p(\mathbf{x}) \leftarrow \theta \in S$, there exists a (renamed) constrained fact $p(\mathbf{x}) \leftarrow \phi \in \mathsf{abstract}_\Psi(S)$ such that $\theta \models \phi$. The maximum size of $\mathsf{abstract}_\Psi(S)$ is $2^{|\Psi|}$ and so the closure $S^* = \mathsf{lfp}\ \lambda S.(S_0 \cup \mathsf{abstract}_\Psi(\mathsf{pe\_step}_P(S)))$ is finite.

The partial evaluation algorithm returns a set of clauses, $\mathsf{pe\_cls}_{\Psi,P}(S^*)$. The predicates in the clauses are renamed according to their versions; that is, if S^* contains constrained facts $p(\mathbf{x}) \leftarrow \theta_1$ and $p(\mathbf{x}) \leftarrow \theta_2$, then two renamed versions of p are produced. Formally, $\mathsf{pe\_cls}_{\Psi,P}$ is defined as follows.

$$\begin{array}{lll}
\mathsf{pe\_cls}_{\Psi,P}(S) = & \{p^{v_0}(\mathbf{x}) \leftarrow \theta \wedge \phi, p_1^{v_1}(\mathbf{x_1}), \ldots, p_m^{v_m}(\mathbf{x_m}) \mid \\
 & \quad p(\mathbf{x}) \leftarrow \theta \in S, \\
 & \quad p(\mathbf{x}) \leftarrow \phi, p_1(\mathbf{x_1}), \ldots, p_m(\mathbf{x_m}) \in P, \\
 & \quad \mathsf{SAT}(\theta \wedge \phi)\}
\end{array}$$

where p^{v_0} is the version of p corresponding to its representative $\mathsf{rep}_\Psi(p(\mathbf{x}) \leftarrow \theta)$ and for $1 \leq j \leq m$, $p_j^{v_j}$ is the version of p corresponding to the representative $\mathsf{rep}_\Psi(p_j(\mathbf{x_j}) \leftarrow (\theta \wedge \phi)|_{\mathbf{x_j}})$.

As we will see in Sections 3.4.1 and 3.4.2, partial evaluation can return clauses, whose dimension is bounded from above or below, depending on the initial set S_0 and the set Ψ.

Proposition 2
Let P be a set of CHCs, S_0 and Ψ be sets of constrained facts, where $S_0 \subseteq \Psi$. Let $p(\mathbf{x}) \leftarrow \theta(\mathbf{x}) \in S_0$. Let P' be the set of clauses $\mathsf{pe\_cls}_{\Psi,P}(S^*)$ where $S^* = \mathsf{lfp}\ \lambda S.(S_0 \cup \mathsf{abstract}_\Psi(\mathsf{pe\_step}_P(S)))$.

Then there exists a renamed version of p, say p^m in P' such that for all $\mathbf{t}$, $P' \vdash p^m(\mathbf{t})$ if and only if $P \vdash p(\mathbf{t}) \wedge \theta(\mathbf{t})$

Proof

- There exists p^m such that $P \vdash p(\mathbf{t}) \wedge \theta(\mathbf{t}) \Rightarrow P' \vdash p^m(\mathbf{t})$. This follows from the soundness of the basic algorithm for partial evaluation, namely that it preserves the derivations that satisfy the input constraint. The proof is by induction on the iterations of the computation of the fixpoint, and we do not give a proof here.

- There exists p^m such that $P' \vdash p^m(\mathbf{t}) \Rightarrow P \vdash p(\mathbf{t}) \wedge \theta(\mathbf{t})$. By assumption, $p(\mathbf{x}) \leftarrow \theta(\mathbf{x}) \in \Psi$. Hence S^* contains a constrained fact of the form $p(\mathbf{x}) \leftarrow \theta(\mathbf{x}) \wedge \psi_1(\mathbf{x}) \wedge \ldots \wedge \psi_j(\mathbf{x})$, where $\{p(\mathbf{x}) \leftarrow \theta(\mathbf{x}), p(\mathbf{x}) \leftarrow \psi_1(\mathbf{x}), \ldots p(\mathbf{x}) \leftarrow \psi_j(\mathbf{x})\} \subseteq \Psi$ $(j \geq 0)$ and $\theta(\mathbf{x}) \models \psi_i(\mathbf{x})$, $0 \leq i \leq j$. Let p^m be the renamed predicate corresponding to $p(\mathbf{x}) \leftarrow \theta(\mathbf{x}) \wedge \psi_1(\mathbf{x}) \wedge \ldots \wedge \psi_j(\mathbf{x})$. For every clause with head $p^m(\mathbf{x})$, its body contains $\theta(\mathbf{x})$.
  Let u' be a feasible AND-tree for $p^m(\mathbf{t})$ in P'. By construction, for every clause in P', there is a clause in P that is identical except for (a) predicate names and (b) the clause constraint, which is weaker in P than in P'. Hence there is a feasible AND-tree u for $p(\mathbf{t})$ in P, that is identical to u' except for predicate names, and $\mathsf{constr}(u') \rightarrow \mathsf{constr}(u)$. Furthermore, $\mathsf{constr}(u') \rightarrow \theta(\mathbf{t})$ (as $\theta(\mathbf{x})$ is in all clauses with head $p^m(\mathbf{x})$), hence $\mathsf{SAT}(\theta(\mathbf{t}) \wedge \mathsf{constr}(u))$. Hence there is a feasible AND-tree for $p(\mathbf{t}) \wedge \theta(\mathbf{t})$, that is, $P \vdash p(\mathbf{t}) \wedge \theta(\mathbf{t})$.

$\square$

Example 6
Let P be the following clauses (containing no constraints in order to simplify the example).

```
p:- true.        p:- p,p.
```

P_{dim} is the following set of clauses, after unfolding the *dim* predicates.

```
p(K):- K=0.
p(K):- p(K1), p(K2), K1>=K2+1, K=K1.
p(K):- p(K1), p(K2), K2>=K1+1, K=K2.
p(K):- p(K1), p(K2), K1=K2, K=K1+1.
```

Let Ψ in the algorithm be {`p(K):-K=<1`, `p(K):-K=<0`} and $S_0 =$ {`p(K):-K=<1`}. To compute $\mathsf{lfp}\ \lambda S.(S_0 \cup \mathsf{abstract}_\Psi(\mathsf{pe\_step}_P(S)))$, the algorithm computes sets $S_0, S_1, \ldots$ where $S_{i+1} = S_i \ \cup\ \mathsf{abstract}_\Psi(\mathsf{pe\_step}_P(S_i))$. The lfp is the limit of this sequence, which is reached when $S_{i+1} = S_i$. The key steps in the execution are as follows.

- $\mathsf{pe\_step}_P(S_0)$ first constructs the set of clauses in P with `K=<1` added to each body. For example, from the second clause in P_{dim} we obtain:

  ```
  p(K):- p(K1), p(K2), K1>=K2+1, K=K1, K=<1.
  ```

  The two constrained atoms in the above clause, after checking satisfiability and projecting the constraints onto their variables, are

  {`p(K1):- K1=<1, p(K2):- K2=<0`} .

  Applying $\mathsf{abstract}_\Psi$ to this set yields $S_1 =$

  {`p(K):- K=<1, p(K):- K=<1,K=<0`} .

```
fib(A,B,0) :- A>=0, A=<1, A=B.
fib(A,B,K) :- A>1, D=A-2, E=A-1, B=F+G,
          fib(D,G,K2), fib(E,F,K1), K1+1=<K, K2=K.
fib(A,B,K) :- A>1, D=A-2, E=A-1, B=F+G, fib(D,G,K1),
          fib(E,F,K2), K1+1=<K, K=K2.
fib(A,B,K) :- A>1, D=A-2, E=A-1, B=F+G,
          fib(D,G,K1), fib(E,F,K2),  K1=K-1, K2=K1.
```

Fig. 9. Dimension instrumented *Fib* program after unfolding dimension predicates.

The other clauses are treated similarly but no other new constrained facts are returned.

- Since $S_0 \neq S_1$, we compute $S_2 = S_1 \cup \mathsf{abstract}_\Psi(\mathsf{pe\_step}_P(S_1))$. Since no new constrained facts are generated by this step (that is, $S_2 = S_1$), the limit of the sequence is reached and so $S_2 = \mathsf{lfp}\ \lambda S.(S_0 \cup \mathsf{abstract}_\Psi(\mathsf{pe\_step}_P(S)))$.
- $\mathsf{pe\_cls}_{\Psi,P}(S_2)$ returns the following set of clauses. The renaming distinguishes the two atoms in S_2, renaming the predicate `p` as `p_1`, corresponding to `p(A,K):- K=<1,K=<0`, and `p_2` corresponding to `p(A,K):- K=<1`.

```
p_2(B):- B=0.
p_2(B):- B>=F+1, B=<1, B=D, p_2(D), p_1(F).
p_2(B):- B>=D+1, B=<1, B=F, p_1(D), p_2(F).
p_2(B):- B=<1, B=D+1, B=F+1, p_1(D), p_1(F).
p_1(B):- B=0.
p_1(B):- B>=F+1, B=<0, B=D, p_1(D), p_1(F).
p_1(B):- B>=D+1, B=<0, B=F, p_1(D), p_1(F).
p_1(B):- B=<0, B=D+1, B=F+1, p_1(D), p_1(F).
```

We notice that for predicate `p_1` the last three clauses cannot succeed since they would yield a derivation whose dimension is greater than 0 and hence the constraints in those clauses would not be satisfied. However, we can see that the successful derivations of `p_1(K)` have `K=<0` and the successful derivations of `p_2(K)` have `K=<1`.

In Sections 3.4.1 and 3.4.2, the partial evaluation algorithm is applied to P_{dim} after first unfolding the *dim* atoms (as shown in Figure 9 for the clauses for *Fib*), suitably instantiating the inputs S_0 and Ψ, to generate clauses whose derivations have dimensions that are bounded from above and below respectively.

3.4.1 Construction of at-most-k-dimension set of clauses

Given an instrumented set of CHCs P_{dim} and $k \geq 0$, we apply the partial evaluation algorithm to obtain $P^{\preceq k}$, the at-most-k dimension clauses. Let $\lhd \in \{=, \leq\}$ and in the algorithm, let $S_0 = \{p(\mathbf{x}, z) \leftarrow z \lhd k \mid p \text{ is a predicate in } P\}$ and $\Psi = \{p(\mathbf{x}, z) \leftarrow z \lhd d \mid 0 \leq d \leq k,\ p \text{ is a predicate in } P\}$.

Figure 10 shows the at-most-1-dimension clauses for *Fib*. The predicate names have been chosen to reflect the dimension constraints. The final argument is the dimension, as in the instrumented clauses.

```
false^{=1}(A):- C>5,C-D>0,A=1,fib^{=1}(C,D,A).
false^{≤1}(A):- A>=0,C>5,C-D>0,-A>= -1,fib^{≤1}(C,D,A).

fib^{=1}(A,B,C):- A>1,C=1,A-E=2,A-F=1,
                              B-G-H=0,I=0,fib^{=1}(E,H,C),fib^{=0}(F,G,I).
fib^{=1}(A,B,C):- A>1,C=1,A-E=2,A-F=1,
                              B-G-H=0,I=0,fib^{=0}(E,H,I),fib^{=1}(F,G,C).
fib^{=1}(A,B,C):- A>1,C=1,A-E=2,A-F=1,
                              B-G-H=0,I=0,fib^{=0}(E,H,I),fib^{=0}(F,G,I).
fib^{=0}(A,B,C):- A>=0,-A>= -1,A-B=0,C=0.
fib^{≤1}(A,B,C):- A>=0,-A>= -1,A-B=0,C=0.
fib^{≤1}(A,B,C):- A>1,C=1,A-E=2,A-F=1,
                              B-G-H=0,I=0,fib^{=1}(E,H,C),fib^{=0}(F,G,I).
fib^{≤1}(A,B,C):- A>1,C=1,A-E=2,A-F=1,
                              B-G-H=0,I=0,fib^{=0}(E,H,I),fib^{=1}(F,G,C).
fib^{≤1}(A,B,C):- A>1,C=1,A-E=2,A-F=1,
                              B-G-H=0,I=0,fib^{=0}(E,H,I),fib^{=0}(F,G,I).
```

Fig. 10. $Fib^{\leq 1}$: at-most-1-dimension version of *Fib*.

Note that for each predicate p and each d, $0 \leq d \leq k$, the partial evaluation produces versions for both $p^{\leq d}$ and $p^{=d}$ (though the set of clauses for some of these versions might be empty).

By Proposition 2, for each predicate p of P, $P^{\leq k}$ contains a predicate (which we call $p^{\leq k}$) all of whose derivations have dimension at most k.

3.4.2 Construction of at-least-k-dimension set of clauses

We obtain $P^{>k-1}$, the at-least-k dimension clauses in a similar way. In the algorithm, let $S_0 = \{p(\mathbf{x}, z) \leftarrow z \geq k \mid p \text{ is a predicate in } P\}$. Let $\Psi = \{p(\mathbf{x}, z) \leftarrow z \geq d \mid 0 \leq d \leq k,\ p \text{ is a predicate in } P\}$.

Figure 11 shows the at-least-1-dimension clauses for *Fib*. The predicate names have been chosen to reflect the dimension constraints.

By Proposition 2, for each predicate p of P, $P^{>k-1}$ contains a predicate (which we call $p^{\geq k}$ or sometimes $p^{>k-1}$) all of whose derivations have dimension at least k.

4 Verification Algorithms

In this section, we describe two algorithms for verification of CHCs, based on the notion of tree dimension. The verification problem we address is to decide whether a given set of CHCs has a model. In case it has no model, the problem is to find a counterexample. A set of CHCs has a model if and only if there is no derivation of `false` from the clauses (or of $\texttt{false}^{\leq k}$ *or* $\texttt{false}^{>k}$ for some dimension bounded version of `false`). Such a derivation exists only if the set contains at least one integrity

```
false^{≥1}(A):- A>=1,C>5,C-D>0,fib^{≥1}(C,D,A).

fib^{≥1}(A,B,C):- A>1,C-I>=1,I>=0,A-E=2,A-F=1,
                      B-G-H=0,fib^{≥1}(E,H,C,J),fib^{≥0}(F,G,I,K).
fib^{≥1}(A,B,C):- A>1,C-I>=1,I>=0,A-E=2,A-F=1,
                        B-G-H=0,fib^{≥0}(E,H,I),fib^{≥1}(F,G,C).
fib^{≥1}(A,B,C):- A>1,C>=1,A-E=2,A-F=1,
                     B-G-H=0,C-I=1,fib^{≥0}(E,H,I),fib^{≥0}(F,G,I).
fib^{≥0}(A,B,C):- A>=0,-A>= -1,A-B=0,C=0.
fib^{≥0}(A,B,C):- A>1,C-I>=1,I>=0,A-E=2,A-F=1,
                        B-G-H=0,fib^{≥1}(E,H,C),fib^{≥0}(F,G,I).
fib^{≥0}(A,B,C):- A>1,C-I>=1,I>=0,A-E=2,A-F=1,
                        B-G-H=0,fib^{≥0}(E,H,I),fib^{≥1}(F,G,C).
fib^{≥0}(A,B,C):- A>1,C>=1,A-E=2,A-F=1,
                     B-G-H=0,C-I=1,fib^{≥0}(E,H,I),fib^{≥0}(F,G,I).
```

Fig. 11. $Fib^{>0}$: at-least-1-dimension version of *Fib*.

constraint (clause with head `false` (or $\texttt{false}^{\leq k}$ *or* $\texttt{false}^{>k}$)). A set containing no integrity constraints has at least one model, namely the interpretation consisting of $p(x) \leftarrow true$ for every predicate p in the clauses.

4.1 Decomposition by dimension of verification problem

We first present an algorithm exploiting the decomposition of a set P of CHCs into complementary sets $P^{\leq k}$ and $P^{>k}$. For each k, these two sets can be solved separately (possibly in parallel).

Proposition 3 (*Decomposition by dimension*)
A set of CHCs P is safe if and only if for some k, both $P^{\leq k}$ and $P^{>k}$ are safe.

Proof
Let both $P^{\leq k}$ and $P^{>k}$ be safe, for some k. Equivalently, $P^{\leq k} \not\models \texttt{false}$ and $P^{>k} \not\models \texttt{false}$. By the constructions in Sections 3.4.1 and 3.4.2 and Proposition 2, there is no derivation of `false` in P of dimension $\leq k$ or of dimension $> k$, which is equivalent to $P \not\models \texttt{false}$, i.e. P is safe. $\square$

The essence of the algorithm based on tree dimension is to decompose P into $P^{\leq k}$ and $P^{>k}$ for successive values of k. If for some k, both of them are safe, then P is also safe, by Proposition 3. P is unsafe if we find a k such that one of them is unsafe.

Lifting interpretations. We introduce a lifting which constructs a syntactic interpretation for a set of CHCs given a syntactic interpretation for an annotated version of the same set of CHCs.

Definition 8 ($S^{\uparrow}$*: Lifting of an interpretation*)
Let $Pred$ be a set of predicates, I be a finite set. Define $Pred^I = \{p^{\triangle} \mid p \in Pred, \triangle \in I\}$ and let S be an interpretation of $Pred^I$ given by constrained facts. Then $S^{\uparrow}$ is the following set of constrained facts:

$$S^{\uparrow} = \{p(x) \leftarrow \bigvee_{(p^{\triangle}(x)\leftarrow\phi)\in S} \phi \mid p^{\triangle} \in Pred^I\} \ .$$

The procedure SolvePartition defined in Algorithm 1 makes use of a procedure $\mathsf{Safe}(P)$, which is a sound oracle: if it returns (safe, $\mathsf{solution}$) then P is safe; if it returns (unsafe, $\mathsf{counterexample}$) then P is unsafe and the counterexample proves it; else we know nothing about P and $\mathsf{unknown}$ is returned. The oracle Safe could be any existing automatic Horn clause solver (Grebenshchikov et al. 2012; Kafle et al. 2016; Hoder and Bjørner 2012; De Angelis et al. 2014; Mordvinov and Fedyukovich 2017) possibly with a timeout. When it cannot verify a program within a given time limit, it returns $\mathsf{unknown}$.

Consider a call SolvePartition($P, k, \emptyset$), the algorithm checks first (using the oracle) whether $P^{\leq k}$ is safe and if so then it proceeds to check whether $P^{>k}$ is safe. If, for either set of CHCs, the oracle returns unsafe then the algorithm returns unsafe. Similarly, if, for both sets of CHCs, the oracle returns safe then the algorithm returns safe together with the interpretation built so far augmented with the current $\mathsf{solution}$ R' (*line 13*), defining a model for P. Otherwise, the Safe oracle returns $\mathsf{unknown}$. The $\mathsf{unknown}$ for $P^{\leq k}$ is propagated and the $\mathsf{unknown}$ for $P^{>k}$ causes the algorithm to proceed by calling itself recursively on the set $P^{>k}$, with $k+1$ and with the interpretation built so far.

Algorithm 1 SolvePartition(P,k,S)

```
 1: Input: A set of CHCs $P$, an integer $k \geq 0$, and an interpretation $S$ (init $\emptyset$)
 2: Output: (safe, solution) | (unsafe, counterexample) | unknown
 3: (status, $R$) ← Safe($P^{\leq k}$)
 4: if status=unsafe then
 5:     return (unsafe $R$)                    ▷ $P^{\leq k}$ is unsafe, hence $P$ is unsafe
 6: if status=unknown then
 7:     return unknown                         ▷ $P^{\leq k}$ may be safe or unsafe, so is $P$
 8: $P^{>} \leftarrow P^{>k}$                  ▷ We turn to $P^{>k}$ as $P^{\leq k}$ is safe
 9: (status, $R'$) ← Safe($P^{>}$)
10: if status=unsafe then
11:     return (unsafe, $R'$)                  ▷ $P^{>k}$ is unsafe, hence $P$ is unsafe
12: if status=safe then
13:     return (safe, $(S \cup R \cup R')^{\uparrow}$)   ▷ $P^{\leq k}$ and $P^{>k}$ are safe, hence $P$ is safe
14: return SolvePartition($P^{>}$,$k+1$,$S \cup R$)  ▷ recurse: $P^{>k}$ may be safe or unsafe
```

Example 7
Applying the algorithm to our example program *Fib*, the oracle Safe finds that both $Fib^{\leq 0}$ and $Fib^{>0}$ are safe, and thus *Fib* is safe.

```
applen(A,B,C):- A=0, B=C, B>=0.
applen(A,B,C):- applen(A1,B,C1), A=A1+1, C=C1+1.

revlen(A,B):- A=0, B=0.
revlen(A,B):- revlen(A1,C), applen(C,D,B), A=A1+1, D=1.

false :- revlen(A,B), A≠B.
```

Fig. 12. Length abstracted version of reverse of a list.

The soundness of the above algorithm follows from the soundness of the oracle and the properties of dimension bounded set of clauses, which is formally stated by the following proposition.

Proposition 4 (*Soundness*)
If Algorithm 1 returns (*un*)*safe* on a set of CHCs P then P is (un)safe.

4.2 Verification by successive iteration of bounded dimension CHCs

For an unsafe program P, there exists some $k_0 \geq 0$ such that $P^{\leq k}$ is *unsafe* for all $k \geq k_0$. So for discovering a bug, we can generate $P^{\leq k}$ successively for $k = 0, 1, 2, \ldots$ and check its safety, as in bounded model checking (BMC). In BMC, the original program and the bounded underapproximations are decidable. By contrast, the under-approximations obtained by dimension bounding are themselves undecidable and there is no upper bound on the dimension.

However, a solution of a bounded dimension program can extend to a solution of the original problem as we shall see. The example in Figure 12 is the at-most-1-dimension version of the example in Figure 12. The oracle Safe derives the following invariant for the predicates $\mathtt{applen}^{\leq 1}$ and $\mathtt{revlen}^{\leq 1}$ from it. This invariant (mapped to the original program using Definition 8) is in fact an invariant of the original program (in Figure 12). Thus the solution of an underapproximation is the solution of the original program.

$$\mathtt{applen}^{\leq 1}(\mathtt{A},\mathtt{B},\mathtt{C}) \leftarrow \mathtt{B} \geq 0 \wedge \mathtt{A} \geq 0 \wedge \mathtt{A} + \mathtt{B} = \mathtt{C}.$$
$$\mathtt{revlen}^{\leq 1}(\mathtt{A},\mathtt{B}) \leftarrow \mathtt{B} \geq 0 \wedge \mathtt{A} = \mathtt{B}.$$

Therefore, the safety of $P^{\leq k}$ also can be checked successively for increasing value of k starting from 0 until $P^{\leq k}$ (for some k) is proven unsafe or its solution generalises to P or the results for $P^{\leq k}$ is unknown. A solution for $P^{\leq k}$ generalises to P if the lifted model of $P^{\leq k}$ using Definition 8 is also a model of P. The iteration done this way does not make any reuse of solutions of lower dimension while verifying a program of higher dimension, which could save some verification effort. The iteration in which the iterates of higher dimension reuses solutions from lower dimensions is reminiscent of *Newtonian iteration* (Esparza et al. 2010). However, reuse introduces a new problem since solutions are approximate. If a counterexample is found for $P^{\leq k+1}$ (where solutions from lower dimensions are used), it needs to be further examined since it may not be a counterexample for $P^{\leq k+1}$ (in which no

```
applen^{=0}(A,B,C):- A=0,  B=C,  B>=0.
applen^{=1}(A,B,C):- A=D+1, C=E+1, applen^{=1}(D,B,E).
applen^{=0}(A,B,C):- A=D+1, C=E+1, applen^{=0}(D,B,E).
applen^{≤1}(A,B,C):- applen^{=1}(A,B,C).
applen^{≤1}(A,B,C):- applen^{=0}(A,B,C).
applen^{≤0}(A,B,C):- applen^{=0}(A,B,C).

revlen^{=0}(A,B) :- A=0, B=0.
revlen^{=1}(A,B) :- A=C+1, E=1, applen^{≤0}(D,E,B), revlen^{=1}(C,D).
revlen^{=1}(A,B) :- A=C+1, E=1, revlen^{≤0}(C,D), applen^{=1}(D,E,B).
revlen^{=1}(A,B) :- A=C+1, E=1, revlen^{=0}(C,D), applen^{=0}(D,E,B).
revlen^{≤1}(A,B) :- revlen^{=1}(A,B). revlen^{≤1}(A,B) :- revlen^{=0}(A,B).
revlen^{≤0}(A,B) :- revlen^{=0}(A,B).

false^{=1}:- A≠B, revlen^{=1}(A,B). false^{=0} :- A≠B, revlen^{=0}(A,B).
false^{≤1} :- false^{=1}. false^{≤1} :- false^{=0}. false^{≤0} :- false^{=0}.
```

Fig. 13. At-most-1-dim version of reverse list example in Figure 12

solutions are used from lower dimensions). We present a solution to this problem in Algorithm 2 via refinement of approximations. Before presenting the algorithm, we first introduce Definition 9 which defines the subset of S of constrained facts involved in a derivation t.

Definition 9 ($S_{|t}$)
Let S be an interpretation of a set of CHCs P given by constrained facts and let t be any derivation in P. Define $S_{|t}$ to be

$$S_{|t} = \{(A \leftarrow \phi) \mid (A \leftarrow \phi) \in S \wedge \text{atom } A \text{ labels a node of } t\} \ .$$

Next, we define the auxiliary procedure subst() as follows.

Definition 10 (subst(P,S))
Given a set of CHCs P and an interpretation S, define subst(P,S) as the set of CHCs obtained as follows: for every constrained fact $A \leftarrow \phi$ in S, replace all the clauses from P whose head is A with the clause $A \leftarrow \phi$.

We now turn to Algorithm 2. Consider a call SolveInc($P, k, \emptyset$); the algorithm checks first (using the oracle) whether $P^{\leq k}$ with the information provided by S "plugged in" using subst is safe (*line 3*). If the oracle returns unknown, the algorithm returns unknown (*line 4-5*). Else if the oracle returns unsafe, the counterexample R is further examined (*line 6-9*). If it uses no constrained facts of S then the counterexample is also a counterexample for P (*line 7-8*). In the case that some constrained facts of S are used in R then the algorithm recurses with those facts removed from S (*line 9*). Finally, if the oracle returns safe the algorithm checks whether the model extends to P and returns safe if so (*line 10-11*). Should the check fail the algorithm recurses with k increased (*line 12*).

Removing the over-approximations used by the counterexample ensures progress (as we shall see in the example below) in the sense that the same counterexample

Algorithm 2 SolveInc(P, k, S)

```
1: Input: A set of CHCs $P$, an integer $k \geq 0$, and an interpretation $S$ (init $\emptyset$)
2: Output: (safe, solution) | (unsafe, counterexample) | unknown
3: (status, $R$) $\leftarrow$ Safe(subst($P^{\preceq k}$, $S$))   $\triangleright$ substitute syntactic model $S$ into $P^{\leq}$
4: if status=unknown then
5:     return unknown
6: if status=unsafe then   $\triangleright$ $R$ is a counterexample for subst($P^{\leq}$, $S$)
7:     if $S_{|R} = \emptyset$ then   $\triangleright$ $R$ uses no predicate defined by $S$
8:         return (unsafe, $R$)   $\triangleright$ hence $R$ is a counterexample for $P$
9:     return SolveInc($P$, $k$, $S \setminus S_{|R}$)   $\triangleright$ recurse with the facts of $S$ not used in $R$
10: if ($R^{\uparrow}$ is a solution of $P$) then   $\triangleright$ subst($P^{\preceq k}$, $S$) is safe
11:     return (safe, $R^{\uparrow}$)
12: return SolveInc($P$, $k+1$, $R$)   $\triangleright$ $R^{\uparrow}$ does not solve $P$, recurse
```

does not arise again in the next iteration. This is because if the same trace arises again and does not use any over-approximations, then it must be a counterexample. In the worst case, all the solutions from the lower dimensions are removed.

Consider 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 $S = \{p(X) \leftarrow \mathit{true}\}$ 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. %
c4. q(X):- X=0.
```

The trace `c1(c3)` is a counterexample for this transformed program but not for the original program (since it uses an approximate solution for the predicate p). However the trace `c2(c4)` is a counterexample for this program as well as for the original since it does not use any approximate solution for the predicates appearing in the counterexample.

5 Experimental results

5.1 Verification of safety properties

Implementation and experimental setting. Algorithms 1 and 2 are implemented in Ciao Prolog (Hermenegildo et al. 2012), interfaced with the Parma Polyhedra Library (Bagnara et al. 2008) and the Yices 2.2 SMT solver (Dutertre 2014) for the manipulation of constraints. The experiments are carried out on a set of 45 (36 safe and 9 unsafe) CHC verification problems taken from three sources: the repository of NTS benchmarks[3], the recursive category of SV-COMP[4] (Beyer 2015) and the benchmarks from the QARMC tool (Grebenshchikov et al. 2012). Examples were chosen that potentially have derivations of unbounded dimension (that is, they are sets of non-linear clauses). Some of these benchmarks are first translated to Prolog syntax using the tools ELDARICA[5] (Hojjat et al. 2012) and SeaHorn (Gurfinkel et al. 2015). The benchmarks are not beyond the capabilities of the existing Horn clause solvers, but they are typically used for testing the performance of new tools. The experimental evaluation is done on a MacBook Pro running OS X on 2.3 GHz Intel core i7 processor, 4 cores and 8 GB memory. The results of these algorithms are compared with that of Rahft (Kafle et al. 2016), a Horn clause verifier which refines an abstract interpretation by eliminating infeasible derivations.

Implementation of $P^{\leq k}$ and $P^{>k}$. For the experiments, we constructed the set of clauses $P^{\leq k}$ and $P^{>k}$ using the procedures described in Section 3.4.

The experiments are intended to establish (i) whether the dimension-based decomposition is practical, (ii) the relationship between the dimension and the solvability of a problem and (iii) how this approach compares other approaches.

Discussion. The results are summarised in Table 1. We report results for three verification algorithms, namely Algorithm 1 and Algorithm 2, and the Safe oracle that is used in those algorithms. For Algorithm 1, we report the result returned, the dimension bound, and the time. For the Safe oracle we report the result and time, and for Algorithm 2 we return the dimension when a result is returned (if at all) and the time.

The oracle Safe used in both algorithms is an abstract interpreter over the domain of convex polyhedra (Kafle et al. 2016), which returns unsafe if a feasible derivation of the predicate `false` exists, safe if a syntactic model can be found within a time bound, and unknown otherwise.

Firstly, the results show that implementation of decomposition based on tree dimension is practical. Algorithm 1 solves 43 out of 45 problems and Algorithm 2 solves about 27 out of 45 problems. There are 7 examples where Algorithm 1 with dimension $k \leq 2$ was enough to prove safety but the Safe oracle was not able to return safe or unsafe. That is, with a given oracle Safe, there are examples for which Safe returns unknown on the original clauses, but there is a low dimension (say $k = 0$ or $k = 1$) where Safe returns safe on both $P^{\leq k}$ and $P^{>k}$. This is evidence that decomposition by dimension is useful with respect to that particular oracle and is an effective refinement heuristic for these cases. While in other refinement approaches, a spurious counterexample is the basis of refinement, Algorithms 1 and 2 can be viewed as performing refinement in which clauses are refined by eliminating safe derivations of lower dimensions, thereby removing a possibly infinite number of traces that have already been shown to be safe.

[3] `https://github.com/pierreganty/NTSLib/`
[4] `http://sv-comp.sosy-lab.org/2015/benchmarks.php`
[5] `https://github.com/uuverifiers/eldarica`

Table 1. Experimental results on 45 CHC verification problems with a timeout of 5 minutes. Times are in seconds.

		Alg. 1		**Safe oracle**		**Alg. 2**	
Program	safety	dim	time	result	time	dim	time
Addition03_false-unreach	safe	0	3	safe	< 1	?	?
McCarthy91_false-unreach	unsafe	1	6	unsafe	< 1	?	?
addition.nts.pl	safe	0	3	safe	< 1	1	< 1
bfprt.nts.pl	safe	0	5	safe	< 1	2	4
binarysearch.nts.pl	safe	0	3	safe	< 1	1	1.1
countZero.nts.pl	safe	0	4	safe	< 1	1	< 1
eq.horn	unsafe	0	3	unsafe	< 1	2	< 1
fib.pl	safe	1	6	?	?	?	?
identity.nts.pl	safe	0	4	safe	< 1	1	< 1
merge.nts.pl	safe	0	5	safe	< 1	1	1.7
palindrome.nts.pl	safe	0	3	safe	< 1	1	< 1
parity.nts.pl	unsafe	0	3	?	?	?	?
remainder.nts.pl	unsafe	0	3	unsafe	< 1	1	< 1
revlen.pl	safe	0	3	safe	< 1	1	< 1
running.nts.pl	unsafe	1	4	?	?	?	?
sum_10x0_false-unreach	unsafe	?	?	?	?	?	?
sum_non_eq_false-unreach	unsafe	0	3	unsafe	< 1	?	?
suma1.horn	unsafe	0	3	unsafe	< 1	1	< 1
suma2.horn	unsafe	0	3	unsafe	< 1	2	< 1
summ_SG1.r.horn	safe	0	2	safe	< 1	?	?
summ_SG2.r.horn	safe	?	?	?	?	?	?
summ_SG3.horn	safe	0	3	safe	< 1	1	< 1
summ_b.horn	safe	2	12	?	?	?	?
summ_binsearch.horn	safe	?	?	?	?	?	?
summ_cil.casts.horn	safe	0	3	safe	< 1	1	< 1
summ_formals.horn	safe	0	4	safe	< 1	1	< 1
summ_g.horn	safe	0	3	safe	< 1	?	?
summ_globals.horn	safe	0	3	safe	< 1	1	< 1
summ_h.horn	safe	0	3	safe	< 1	2	< 1
summ_local-ctx-call.horn	safe	0	2	safe	< 1	1	< 1
summ_locals.horn	safe	0	4	safe	< 1	?	?
summ_locals2.horn	safe	0	2	safe	< 1	1	< 1
summ_locals3.horn	safe	0	3	safe	< 1	1	< 1
summ_locals4.horn	safe	0	3	safe	< 1	2	2.2
summ_mccarthy2.horn	safe	?	?	?	?	?	?
summ_multi-call.horn	safe	0	3	safe	< 1	1	< 1
summ_nested.horn	safe	0	3	safe	< 1	1	< 1
summ_ptr_assign.horn	safe	0	3	safe	< 1	1	< 1
summ_recursive.horn	safe	0	3	?	?	?	?
summ_rholocal.horn	safe	0	3	safe	< 1	1	< 1
summ_rholocal2.horn	safe	0	3	safe	< 1	1	< 1
summ_slicing.horn	safe	0	3	safe	< 1	?	?
summ_summs.horn	safe	0	3	safe	< 1	?	?
summ_typedef.horn	safe	0	4	safe	< 1	1	< 1
summ_x.horn	safe	0	3	safe	< 1	?	?
# solved (safe/unsafe)		43 (35/8)		36 (30/6)		27 (23/4)	

Most of the problems solved using Algorithm 1 are solved when they are decomposed with dimension $k = 0$. The separation of the derivations ($k = 0$) eases the verification task. Only 4 problems that were solved needed decomposition greater than 0. Similarly, for Algorithm 2, the solution of an under-approximation ($P^{\leq k}$) for a fairly small value of $k = 1$ or $k = 2$ was sufficient for finding a syntactic model for those problems that were solved. Though this observation may be related to this particular set of examples, we suspect that many application problems resulting from encoding imperative programs have derivation trees of low dimension.

Use of linearisation. As mentioned previously, $P^{\leq k}$ can be linearised, potentially allowing the use of specialised verification procedures for linear clauses. Although our implementation of the oracle Safe contains no special facilities for dealing with linear clauses, we applied a linearisation procedure in Algorithm 2. For this purpose we used a procedure based on partial evaluation (Kafle et al. 2016). We did not observe that linearisation in itself offers any advantages, although one might expect that linear clauses were in some way a simpler case for verification. In order to exploit linearisation, it would be necessary to use a verification procedure with more specialised procedures for recognising and solving particular classes of linear recursive predicates amenable to precise solution, for example as described by Gonnord and Halbwachs (2006).

Limitations of our experiments and possible improvements. It would be possible to combine dimension-bounded decomposition with refinement-based solving. For example, our oracle Safe is limited in that it does not attempt any refinement after computing a convex polyhedra abstract interpretation of the clauses. It returns `unknown` if the over-approximation allows a derivation of `false`, which might, however, be infeasible. Thus Algorithm 2 tends to return `unknown` before the timeout, in cases where a more sophisticated oracle would allow the procedure to continue.

6 Discussion and 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. (2014). However, the use of dimension for program verification is more recent. Ganty et al. (2016) used the notion of *tree dimension* for computing summaries of procedural programs by under-approximating them. Roughly speaking, they compute procedure summaries iteratively, starting from the program behaviours captured by derivation trees of dimension 0. Then they reuse these summaries to compute summaries for program behaviours captured by derivation trees of dimension 1 and so on for 2, 3, etc. We adapt the idea of dimension-based under-approximations to the setting of CHCs.

Decomposition can be compared to refinement techniques based on automata (Heizmann et al. 2009; Heizmann et al. 2013; Kafle and Gallagher 2017) in which the aim is to eliminate sets of program traces that have been shown to be safe. In our case, establishing the safety of clauses whose derivations are of a given dimension allows us to eliminate those dimensions, and focus on the remaining dimensions.

Our decomposition technique offers a practical way to checking and eliminating infinite sets of traces.

In the world of constrained Horn clause verification tools (solvers) we can distinguish solvers depending on whether they can handle general non-linear Horn clauses or not. A majority of solvers (Gurfinkel et al. 2015; Grebenshchikov et al. 2012; Rümmer et al. 2013; McMillan and Rybalchenko 2013; Kafle and Gallagher 2017) handle non-linear Horn clauses but there are notable exceptions like VeriMAP (De Angelis et al. 2014) or Sally[6]. For both VeriMAP and Sally, their underlying reasoning engine handles only linear Horn clauses which appears to restrict, in principle, their applicability. However, prior work on consistency preserving linearisation of dimension-bounded sets of CHCs (Kafle et al. 2016) shows solvers for linear CHCs can be used to check consistency of non-linear sets of CHCs. Another work on linearisation of CHCs based on *fold-unfold transformations* is described by De Angelis et al. (2015).

7 Conclusion and Future Work

We applied the notion of *tree dimension* to decompose constrained Horn clause verification problems by dimensions. We presented algorithms based on this idea; whose results on a set of non-linear Horn clause verification benchmarks show its feasibility and usefulness both for proving safety as well as for finding bugs in programs. We also looked into the problem of instrumenting clauses with dimension predicates and reason about the dimension directly from the resulting clauses.

Other ideas for program verification based on tree dimension are worth investigating, including induction based on tree dimension, and further investigation of strategies that could exploit knowledge of dimension bounds (such as those discussed in Section 3.3).

Acknowledgements

The research leading to these results has been supported by EU FP7 project 318337, ENTRA - Whole-Systems Energy Transparency, EU FP7 project 611004, coordination and support action ICT-Energy, EU FP7 project 610686, POLCA - Programming Large Scale Heterogeneous Infrastructures, Madrid Regional Government project S2013/ICE-2731, N-Greens Software - Next-GeneRation Energy-EfficieNt Secure Software, and the Spanish Ministry of Economy and Competitiveness project No. TIN2015-71819-P, RISCO - RIgorous analysis of Sophisticated COncurrent and distributed systems. The first author is supported by the Australian Research Council Discovery Project grant DP140102194.

We thank the anonymous reviewers for their comments and suggestions, which greatly improved the paper.

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

References

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

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

Bagnara, R., Hill, P. M., and Zaffanella, E. 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, 3–21.

Baier, C. and Tinelli, C., Eds. 2015. *TACAS. Proceedings.* LNCS, vol. 9035. Springer.

Beyer, D. 2015. Software verification and verifiable witnesses - (report on SV-COMP 2015). See Baier and Tinelli (2015), 401–416.

Bjørner, N., Gurfinkel, A., McMillan, K. L., and Rybalchenko, A. 2015. Horn clause solvers for program verification. In *Fields of Logic and Computation II - Essays Dedicated to Yuri Gurevich on the Occasion of His 75th Birthday*, L. D. Beklemishev, A. Blass, N. Dershowitz, B. Finkbeiner, and W. Schulte, Eds. LNCS, vol. 9300. Springer, 24–51.

Bjørner, N., McMillan, K. L., and Rybalchenko, A. 2013. On solving universally quantified Horn clauses. In *SAS*, F. Logozzo and M. Fähndrich, Eds. LNCS, vol. 7935. Springer, 105–125.

De Angelis, E., Fioravanti, F., Pettorossi, A., and Proietti, M. 2014. Verimap: A tool for verifying programs through transformations. In *TACAS*, E. Ábrahám and K. Havelund, Eds. LNCS, vol. 8413. Springer, 568–574.

De Angelis, E., Fioravanti, F., Pettorossi, A., and Proietti, M. 2015. Proving correctness of imperative programs by linearizing constrained Horn clauses. *TPLP 15,* 4-5, 635–650.

Dutertre, B. 2014. Yices 2.2. In *CAV*, A. Biere and R. Bloem, Eds. LNCS, vol. 8559. Springer, 737–744.

Esparza, J., Kiefer, S., and Luttenberger, M. 2007. On fixed point equations over commutative semirings. In *STACS, Proceedings.* LNCS, vol. 4393. Springer, 296–307.

Esparza, J., Kiefer, S., and Luttenberger, M. 2010. Newtonian program analysis. *J. ACM 57,* 6, 33.

Esparza, J., Luttenberger, M., and Schlund, M. 2014. A brief history of strahler numbers. In *LATA. Proceedings*, A. H. Dediu, C. Martín-Vide, J. L. Sierra-Rodríguez, and B. Truthe, Eds. LNCS, vol. 8370. Springer, 1–13.

Gallagher, J. P. 1993. Specialisation of logic programs: A tutorial. In *Proceedings PEPM'93, ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation.* ACM Press, Copenhagen, 88–98.

Gallagher, J. P. and Lafave, L. 1996. Regular approximation of computation paths in logic and functional languages. In *Partial Evaluation*, O. Danvy, R. Glück, and P. Thiemann, Eds. Springer-Verlag LNCS, vol. 1110. 115–136.

Ganty, P., Iosif, R., and Konečný, F. 2016. Underapproximation of procedure summaries for integer programs. *STTT 19,* 5 (apr), 565–584.

Gonnord, L. and Halbwachs, N. 2006. Combining widening and acceleration in linear relation analysis. In *SAS*, K. Yi, Ed. LNCS, vol. 4134. Springer, 144–160.

Grebenshchikov, S., Gupta, A., Lopes, N. P., Popeea, C., and Rybalchenko, A. 2012. HSF(C): A software verifier based on Horn clauses - (competition contribution). In *TACAS*, C. Flanagan and B. König, Eds. LNCS, vol. 7214. Springer, 549–551.

Grebenshchikov, S., Lopes, N. P., Popeea, C., and Rybalchenko, A. 2012. Synthesizing software verifiers from proof rules. In *PLDI*, J. Vitek, H. Lin, and F. Tip, Eds. ACM, 405–416.

Gurfinkel, A., Kahsai, T., and Navas, J. A. 2015. SeaHorn: A framework for verifying C programs (competition contribution). See Baier and Tinelli (2015), 447–450.

Heizmann, M., Hoenicke, J., and Podelski, A. 2009. Refinement of trace abstraction. In *SAS*, J. Palsberg and Z. Su, Eds. LNCS, vol. 5673. Springer, 69–85.

Heizmann, M., Hoenicke, J., and Podelski, A. 2013. Software model checking for people who love automata. See Sharygina and Veith (2013), 36–52.

Hermenegildo, M. V., Bueno, F., Carro, M., López-García, P., Mera, E., Morales, J. F., and Puebla, G. 2012. An overview of ciao and its design philosophy. *TPLP 12,* 1-2, 219–252.

Hoder, K. and Bjørner, N. 2012. Generalized property directed reachability. In *SAT. Proceedings*, A. Cimatti and R. Sebastiani, Eds. LNCS, vol. 7317. Springer, 157–171.

Hojjat, H., Konecný, F., Garnier, F., Iosif, R., Kuncak, V., and Rümmer, P. 2012. A verification toolkit for numerical transition systems - tool paper. In *FM*, D. Giannakopoulou and D. Méry, Eds. LNCS, vol. 7436. Springer, 247–251.

Jaffar, J., Maher, M., Marriott, K., and Stuckey, P. 1998. The semantics of constraint logic programs. *Journal of Logic Programming 37,* 1–3, 1–46.

Jones, N., Gomard, C., and Sestoft, P. 1993. *Partial Evaluation and Automatic Software Generation.* Prentice Hall.

Kafle, B. and Gallagher, J. P. 2017. Horn clause verification with convex polyhedral abstraction and tree automata-based refinement. *Computer Languages, Systems & Structures 47*, 2–18.

Kafle, B., Gallagher, J. P., and Ganty, P. 2016. Solving non-linear horn clauses using a linear horn clause solver. In *HCVS*, J. P. Gallagher and P. Rümmer, Eds. EPTCS, vol. 219. 33–48.

Kafle, B., Gallagher, J. P., and Morales, J. F. 2016. RAHFT: A Tool for Verifying Horn Clauses Using Abstract Interpretation and Finite Tree Automata. In *CAV*, S. Chaudhuri and A. Farzan, Eds. Lecture Notes in Computer Science, vol. 9779. Springer, 261–268.

McMillan, K. L. and Rybalchenko, A. 2013. Solving constrained Horn clauses using interpolation. Tech. rep., Microsoft Research.

Mordvinov, D. and Fedyukovich, G. 2017. Synchronizing constrained horn clauses. In *LPAR-21. 21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning*, T. Eiter and D. Sands, Eds. EPiC Series in Computing, vol. 46. EasyChair, 338–355.

Nielson, H. R. and Nielson, F. 1992. *Semantics with applications - a formal introduction.* Wiley professional computing. Wiley.

Peralta, J., Gallagher, J. P., and Sağlam, H. 1998. Analysis of imperative programs through analysis of constraint logic programs. In *SAS*, G. Levi, Ed. Springer-Verlag LNCS, vol. 1503. 246–261.

Reps, T. W., Turetsky, E., and Prabhu, P. 2016. Newtonian program analysis via tensor product. In *POPL*, R. Bodík and R. Majumdar, Eds. ACM, 663–677.

Rümmer, P., Hojjat, H., and Kuncak, V. 2013. Disjunctive interpolants for Horn-clause verification. See Sharygina and Veith (2013), 347–363.

Sharygina, N. and Veith, H., Eds. 2013. *CAV.* LNCS, vol. 8044. Springer.