Verification of programs with ADTs using Shallow Horn Clauses

Théo LOSEKOOT[1], Thomas GENET[1], and Thomas JENSEN[2]

[1] Université de Rennes
[2] INRIA
theo.losekoot@irisa.fr, thomas.genet@irisa.fr, thomas.jensen@inria.fr

Abstract This paper considers verification of relational properties of programs over algebraic data types (ADTs) by translating programs and properties into Constrained Horn clauses (CHCs). Verification reduces to satisfiability of CHCs modulo the theory of algebraic data types, which can be done by exhibiting a Herbrand model of the clauses. Herbrand models of CHCs with *recursive* ADTs are unbounded, so we need a formalism to finitely represent such models. We propose Shallow Horn Clauses (SHoCs), a new formalism for finitely representing unbounded Herbrand models. SHoCs enjoy the usual Boolean closure properties and are strictly more expressive and compact than tree automata and convoluted tree automata, which have previously been used for this purpose. We propose an iterative procedure for inferring SHoCs. The model inference problem arising from relational verification is undecidable, so we propose an incomplete but sound inference procedure. Experiments show that this procedure performs well in practice w.r.t. state of the art tools, both for verifying properties and for finding counterexamples.

Keywords: Formal verification · Relational properties · Algebraic datatypes · Model inference

1 Introduction

Constrained Horn Clauses (CHCs) have attracted considerable attention as a formalism for program verification [4]. CHCs are Horn clauses with additional constraints expressed in an underlying theory. They unify the representation of the semantics of a program and of properties to verify on it, and proving a property on a program is reduced to checking satisfiability of the combined set of Horn clauses that represent the program and the property. Proving satisfiability can be done using an axiomatic proof or by exhibiting a model (e.g. with a finite model finder). However, when the underlying theory of the CHCs is the theory of *recursive* algebraic data types (ADTs), axiomatic proofs usually require advanced techniques, such as automatic induction [23]. In addition, since models over recursive ADTs are unbounded Herbrand models, a challenge in the model exhibition approach is to find finite representations of these models. Since these models are sets of atoms, this amounts to finitely representing an unbounded set of atoms. *Tree automata* recognize regular term languages and have proved to be a convenient way to finitely represent unbounded Herbrand models over ADTs [11, 13, 16, 20]. However, ordinary tree automata fail to capture relational information in the model. By recognizing tuples of terms instead of a single term, *convoluted tree automata* successfully represent Herbrand models where it is enough to relate *fixed branches* of a term [18]. This *ad hoc* extension of automata makes it possible to represent some relations that are important in program verification, e.g., the relation between lists and their size. However, the fixed-branch restriction prevents convolution from modelling relations that depend on unpredictable choice of branches, e.g., the relation between a tree and the height of its highest branch.

We overcome this limitation by using *Shallow Horn Clauses* (SHoCs) to represent Herbrand models with relational information. In our setting, programs and properties are still represented using CHCs but Herbrand models of programs and properties are represented using SHoCs instead of tree automata. SHoCs is an addition to the collection of formalisms obtained by imposing syntactic restrictions on Horn clauses. We compare SHoCs with CS-programs [17] in Section 4.1 and with SCF-programs [5] in Section 7. A SHoC is a Horn clause where the head is linear and only contains shallow terms, i.e. a function symbol applied to variables, atoms in the body only contain variables, and every such variable appears in the head. The body of a SHoC may be non-linear. For instance, here are the SHoCs representing the relation $R_{sh}(t, n)$ between a binary tree t and a natural number n, where (t, n) belongs to the relation if t is of height smaller or equal to n:

$$R_{sh}(leaf, z) \quad R_{sh}(leaf, s(N)) \quad R_{sh}(node(T_1, E, T_2), s(N)) \Leftarrow R_{sh}(T_1, N) \land R_{sh}(T_2, N)$$

The relation recognized by this SHoCs is the set of tuples of terms (t, n) such that $R_{sh}(t, n)$ belongs to the least fixpoint of the clauses. As shown in the next section, this kind of relation cannot be recognized by a (convoluted) tree automaton nor by CS-programs. Our contributions are:

– We formally define SHoCs and their closure properties, and show that SHoCs are strictly more expressive than (convoluted) tree automata.
– We define a Learner-Teacher procedure for learning a set of SHoCs that prove satisfiability of a set of CHCs. This procedure is the key element in proving a property on a set of CHCs representing a program.
– We have implemented and evaluated the Learner-Teacher procedure on a large set of benchmarks. On several examples, we observe that SHoCs are more compact and more generic than their equivalent tree automata or convoluted tree automata.

– We have compared our tool with other tools available for verifying properties over ADTs. Experiments show that verification using SHoCs succeeds on more benchmarks than other approaches. In particular, we succeed on the verification of relational properties on ADTs which are out of the scope of state-of-the-art SMT solvers with ADT support such as Spacer [12], Eldarica [14] and RInGen [21].

2 Verification using Tree Automata and Shallow Horn Clauses

This section provides examples of verification using tree automata and convolutions. They are intended to motivate the introduction of shallow Horn clauses. Let $nat ::= z \mid s(nat)$ and $natTree ::= leaf \mid node(natTree, nat, natTree)$ be the ADTs of natural numbers and binary trees of *nat*s. The following Horn clauses define the predicates *leq* (less than or equal), *isEmpty* (is a tree empty), *all*0 (is a tree full of zeros), *no*0 (is a tree free of zeros), *height* (the height of a binary tree), and *heightRB* (the height of the rightmost branch of a binary tree).

$$\begin{array}{lll}
leq(z,z) & false \Leftarrow leq(s(X),z) & isEmpty(leaf) \\
leq(z,s(X)) & leq(s(X),s(Y)) \Leftarrow leq(X,Y) & false \Leftarrow isEmpty(node(T_1,E,T_2)) \\
 & leq(X,Y) \Leftarrow leq(s(X),s(Y)) &
\end{array}$$

$$\begin{array}{ll}
all0(leaf) & all0(node(T_1,z,T_2)) \Leftarrow all0(T_1) \wedge all0(T_2) \\
false \Leftarrow all0(node(T_1,s(E),T_2)) & all0(T_1) \Leftarrow all0(node(T_1,z,T_2)) \\
 & all0(T_2) \Leftarrow all0(node(T_1,z,T_2))
\end{array}$$

$$\begin{array}{ll}
no0(leaf) & no0(node(T_1,s(E),T_2)) \Leftarrow no0(T_1) \wedge no0(T_2) \\
false \Leftarrow no0(node(T_1,z,T_2)) & no0(T_1) \Leftarrow no0(node(T_1,s(E),T_2)) \\
 & no0(T_2) \Leftarrow no0(node(T_1,s(E),T_2))
\end{array}$$

$$\begin{array}{l}
height(leaf,z) \\
height(node(T_1,E,T_2),s(N_1)) \Leftarrow height(T_1,N_1) \wedge height(T_2,N_2) \wedge leq(N_2,N_1) \\
height(node(T_1,E,T_2),s(N_2)) \Leftarrow height(T_1,N_1) \wedge height(T_2,N_2) \wedge leq(N_1,N_2) \\
N = M \Leftarrow height(T,N) \wedge height(T,M)
\end{array}$$

$$\begin{array}{l}
heightRB(leaf,z) \\
heightRB(node(T_1,E,T_2),s(N)) \Leftarrow heightRB(T_2,N) \\
N = M \Leftarrow heightRB(T,N) \wedge heightRB(T,M)
\end{array}$$

All variables are in uppercase and universally quantified. Let $\mathcal{P}$ denote this set of clauses that define the program to verify. Because our clauses intend to represent functional programs, the last clause of, e.g., the *height* definition ensures that the corresponding relation is, in fact, a function in the Herbrand model of $\mathcal{P}$. We recall that, given a set of function symbols, the Herbrand *structure* interprets each function symbol f as the function that takes terms $t_1, \ldots, t_n$ as arguments and maps them to the term $f(t_1, \ldots, t_n)$. This structure is unique. However, Herbrand *models* of a set of clauses may not be unique. A Herbrand model is a set of ground atoms satisfying all the clauses. Without the last clause of *height*, a Herbrand model where $height(t, n)$ is true for all trees t and all natural numbers n is a correct model for the three first clauses of *height*, as the clauses of $\mathcal{P}$ are not interpreted using standard first-order logics and not the least fixed-point semantics. For a terminating function, the added clause ensures that $\mathcal{P}$ can only be satisfied by a single Herbrand model M representing this function. To prove a property ϕ on $\mathcal{P}$ we have to prove $\mathcal{P} \models \phi$. Since there is only one model M of $\mathcal{P}$, this reduces to checking that $M \models \phi$. We motivate the need for SHoCs using three properties to prove on $\mathcal{P}$:

$$\begin{aligned}
\phi_1 &\overset{def}{=} \mathit{all0}(T) \wedge \mathit{no0}(T) \Rightarrow \mathit{isEmpty}(T)\\
\phi_2 &\overset{def}{=} \mathit{heightRB}(\mathit{node}(T_1, E, T_2), X) \wedge \mathit{heightRB}(T_2, Y) \Rightarrow \mathit{leq}(Y, X)\\
\phi_3 &\overset{def}{=} \mathit{heightRB}(T, N) \wedge \mathit{height}(T, M) \Rightarrow \mathit{leq}(N, M)
\end{aligned}$$

Property ϕ_1 can be proven by representing M with a tree automaton. Property ϕ_2 cannot be proven with tree automata but can be proven using convoluted tree automata. Finally, property ϕ_3 cannot be proven using convoluted tree automata but can be with SHoCs.

2.1 Tree automata

First, we focus on proving that $M \models \phi_1$. Recall that the Herbrand model M is the set of atoms satisfying $\mathcal{P}$. Let $M_{isEmpty} = \{\mathit{isEmpty}(t) \mid \mathit{isEmpty}(t) \in M\}$, $M_{all0} = \{\mathit{all0}(t) \mid \mathit{all0}(t) \in M\}$ and $M_{no0} = \{\mathit{no0}(t) \mid \mathit{no0}(t) \in M\}$ be sets of atoms and subsets of M. Thus, proving ϕ_1 is equivalent to showing that for all terms t,

(1) if $M \models \mathit{all0}(t) \wedge \mathit{no0}(t)$ then $M \models \mathit{isEmpty}(t)$, or equivalently

(2) if $\mathit{all0}(t) \in M_{all0} \wedge \mathit{no0}(t) \in M_{no0}$ then $\mathit{isEmpty}(t) \in M_{isEmpty}$

To prove (2), we need to pick *all* terms t such that $\mathit{all0}(t)$ belong to M_{all0}, $\mathit{no0}(t)$ belong to M_{no0} and check that the terms $\mathit{isEmpty}(t)$ (for the same t) belong to $M_{isEmpty}$. For proving this automatically, we need a finite representation of M_{all0}, M_{no0} and $M_{isEmpty}$. Those three sets are regular and tree automata recognizing them can be automatically inferred [11,13,16,20]. Tree automata rewrite terms to states. If a term t rewrites to a state q then it is said to be *recognized* in q. Here is a possible tree automaton recognizing terms t such that $\mathit{isEmpty}(t)$ belong to $M_{isEmpty}$: $\mathit{leaf} \rightarrow q_{isEmpty}$. This automaton only recognizes the empty tree leaf. We can do the same for $\mathit{no0}$ with the following tree automaton:

$$\begin{array}{rll}
\mathit{leaf} \rightarrow q_{no0} & z \rightarrow q_0 & s(q_1) \rightarrow q_1\\
\mathit{node}(q_{no0}, q_1, q_{no0}) \rightarrow q_{no0} & s(q_0) \rightarrow q_1 &
\end{array}$$

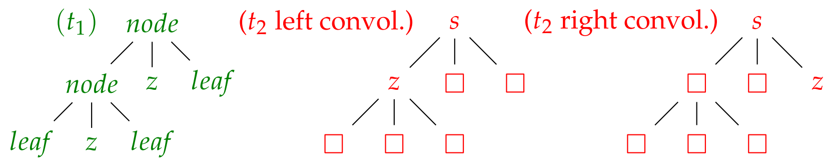
This automaton recognizes z in q_0 and recognizes all natural numbers greater than 0 in q_1, e.g., $s(z) \rightarrow s(q_0) \rightarrow q_1$. It also recognizes all trees of natural numbers greater to 0 in state q_{no0}. Thus, the language recognized by q_{no0} is the set of terms t such that $\mathit{no0}(t)$ belong to M_{no0}. Finally, we can do the same for $\mathit{all0}$ with the following tree automaton where state q_{all0} recognizes terms t such that $\mathit{all0}(t)$ belong to M_{all0}:

$$\begin{array}{rll}
\mathit{leaf} \rightarrow q_{all0} & z \rightarrow q_0 & s(q_1) \rightarrow q_1\\
\mathit{node}(q_{all0}, q_0, q_{all0}) \rightarrow q_{all0} & s(q_0) \rightarrow q_1 &
\end{array}$$

Thus, to prove (2) it is enough to prove that all terms recognized in both q_{all0} and q_{no0} are all recognized in $q_{isEmpty}$, i.e., prove that the intersection between the languages of q_{all0} and q_{no0} is included in the language recognised by $q_{isEmpty}$, which is decidable on tree automata.

2.2 Convoluted tree automata

Models that can be expressed using regular languages and tree automata have limitations. To prove the property ϕ_2 the model needs to preserve the relation that exists between the height of the rightmost branch of $node(T_1, E, T_2)$ and X. Verifying ϕ_2 cannot follow the previous approach. Losekoot *et al.* [18] use an extension of tree automata with convolutions to recognize regular languages of tuples to prove such properties. A convolution transforms a n-tuple of terms into a term built on n-tuples of symbols. To recognize the *binary* relation $heightRB$, a convoluted tree automaton recognizes the overlaying of terms t_1 and t_2 for all (t_1, t_2) belonging to the relation (i.e. such that $heightRB(t_1, t_2) \in M$). For instance, to overlay $t_1 = node(node(leaf, z, leaf), z, leaf)$ and $t_2 = s(z)$, one needs to define a new symbol $\langle node, s\rangle$ for the convolution of the top symbols of the two terms and then overlay subterms of t_1 and t_2. Since the number of children of $node$ and s are different, a direct overlaying is impossible. Convoluted tree automata solves this by extending the arity of symbols so that they are all the same, and replace non-existent subterms by a padding symbol $\square$. However, using such a representation, when overlaying t_1 and t_2 there are many possible trees representing t_2. As a result, convoluted tree automata are parameterized by a *fixed* overlaying strategy, e.g., the left (resp. right) convolution strategy convolutes the leftmost (resp. rightmost) branches of the two terms together.



Since the $heightRB$ relation relates the height of the rightmost branch of the tree with the natural number, the right convolution strategy has to be used. This leads to the following convoluted tree automaton for $heightRB$:

$$\begin{array}{r|r|r}
\langle node, s\rangle(q_t, q_{nat}, q_{hRB}) \to q_{hRB} & \langle node, \square\rangle(q_t, q_{nat}, q_t) \to q_t & \langle z, \square\rangle \to q_{nat} \\
\langle leaf, z\rangle \to q_{hRB} & \langle leaf, \square\rangle \to q_t & \langle s, \square\rangle(q_{nat}) \to q_{nat}
\end{array}$$

In particular, this automaton recognizes the right convolution of t_1 and t_2 that is $\langle node, s\rangle(\langle node, \square\rangle(\langle leaf, \square\rangle, \langle z, \square\rangle, \langle leaf, \square\rangle), \langle z, \square\rangle, \langle leaf, z\rangle)$ in state q_{hRB}. Similarly, it is possible to build a convoluted automaton recognizing the leq relation. This automaton has the following transitions $\langle z, z\rangle \to q_{leq}$, $\langle \square, z\rangle \to q_{leq}$, $\langle \square, s\rangle(q_{leq}) \to q_{leq}$, $\langle z, s\rangle(q_{leq}) \to q_{leq}$, $\langle s, s\rangle(q_{leq}) \to q_{leq}$. Then, using these two automata, it is possible to prove ϕ_2: we have to prove that for all t_1, t_2, e, x, y such that $(node(t_1, e, t_2), x)$ and (t_2, y) belong to the language recognized by

q_{hRB}, then the pair (y, x) is recognized by q_{leq}. This can easily be proven using standard algorithms on tree automata. Convoluted tree automata successfully model relations for which it is enough to relate a single *fixed* branch of a term (e.g. leftmost or rightmost branch) with another *fixed* branch in another term. Note that for another function, say *heightLB* computing the height of the *leftmost* branch of a binary tree, we would need to choose a different strategy for convolution than the right convolution.

2.3 Shallow Horn Clauses

Using the *fixed-branch* restriction enables efficient verification of many list processing programs [18], but it prevents convolution from modelling relations that depend on unpredictable choice of branches, such as the relation *height* between a binary tree and its height. *E.g.*, it is impossible to have a representation for M (the model of $\mathcal{P}$) sufficiently precise for proving $M \models \phi_3$. In this paper, we define a restriction of Horn clauses called Shallow Horn Clauses (SHoCs) to represent such models. In a set of SHoCs, the predicates symbols replace states of tree automata and the language "recognized" by a predicate P is the set of tuples of terms $(t_1, \dots, t_n)$ such that $P(t_1, \dots, t_n)$ belongs to the smallest Herbrand model of the SHoCs (equivalently to the least fixpoint of the SHoCs). For instance, the SHoCs recognizing the *heightRB* relation (and equivalent to the previous convoluted automaton) only consists of the following two simple clauses:

$$R_{heRB}(leaf, z) \qquad R_{heRB}(node(T_1, E, T_2), s(N)) \Leftarrow R_{heRB}(T_2, N)$$

Restricting Horn clauses to SHoCs yields decidability of boolean operations like intersection, complement, ..., which are necessary for verification. Besides, since SHoCs do not rely on a *fixed* convolution strategy they can precisely model the *height* function and prove property ϕ_3, as illustrated in Section 4.1.

3 Prerequisites

We use types, algebraic data types, typed variables, typed terms and typed relation symbols as in SMTLIB [1]. As in SMTLIB, we use a typed alphabet Σ associating symbols to types and $\mathcal{X}$ a set of typed variables. In the example of Section 1, Σ contains the symbol $z : nat$, $s : nat \rightarrow nat$, $leq : nat \times nat \rightarrow bool$, etc. The set $\mathcal{X}$ contains variables $X : nat, Y : nat, etc.$ and variables $T : natTree, T_1 : natTree, etc.$ A *relation* symbol is a function symbol whose co-domain is *bool*, e.g, $leq : nat \times nat \rightarrow bool$. *Patterns* (ranged over by p) are (typed) terms that may contain variables. In the following, we use *terms* (ranged over by t) to mean (typed) terms that do not contain variables. The set of all terms over an alphabet Σ is written $\mathcal{T}(\Sigma)$. An *atom* is a pattern of type *bool*, e.g., $leq(z, X)$ is an atom. A *ground* atom is an atom that does not contain variables, i.e. a term of type *bool*. A tuple $(e_1, \dots, e_n)$ can be written as $\vec{e}$ when size can stay implicit. When clear from the context, we omit the typing information on variables, patterns, terms, relations and atoms.

A clause is a first-order formula of the form $\forall \overrightarrow{X},\ R_1(\overrightarrow{p}_1) \vee \ldots \vee R_k(\overrightarrow{p}_k) \vee \neg R_{k+1}(\overrightarrow{p}_{k+1}) \vee \ldots \vee \neg R_n(\overrightarrow{p}_n)$ with $\overrightarrow{X}$ a tuple of typed variables, $1 \leq k \leq n$, and for all i in $[1 \ldots n]$, R_i is a relation symbol, $\overrightarrow{p}_i$ is a tuple of patterns, and $R_i(\overrightarrow{p}_i)$ is an atom whose variables are all in $\overrightarrow{X}$. We use first-order logic with equality, meaning the binary equality predicate is built-in and can be used. Clauses are also written $R_1(\overrightarrow{p}_1) \vee \ldots \vee R_k(\overrightarrow{p}_k) \Leftarrow R_{k+1}(\overrightarrow{p}_{k+1}) \wedge \ldots \wedge R_n(\overrightarrow{p}_n)$, where universal quantification is implicit, $H = R_1(\overrightarrow{p}_1) \vee \ldots \vee R_k(\overrightarrow{p}_k)$ is called the *head* and $B = R_{k+1}(\overrightarrow{p}_{k+1}) \wedge \ldots \wedge R_n(\overrightarrow{p}_n)$ the *body*. The head and body of a clause can be manipulated as sets of their atoms, as order usually does not matter and duplicates are useless. The function *Vars* fetches the set of variables contained in a term, pattern, atom, and clause. The height of a pattern p is written $ht(p)$ and extends to tuples, sets, and atoms containing patterns, as the maximum of each individual height. A substitution σ is a function replacing variables by patterns in a pattern, atom, or clause.

Definition 1 (Horn clauses and Strict Horn clauses). *A Horn clause is a clause $H \Leftarrow B$ with at most one head atom, i.e. $|H| \leq 1$. A Horn clause is called* strict *if $|H| = 1$ and its head does not contain the equality predicate.*

Proposition 1 (Smallest Herbrand model). *Let $\mathcal{C}$ be a set of strict Horn clauses. The unique* smallest Herbrand model *$\mathcal{H}(\mathcal{C})$ exists and is the smallest set of ground atoms which satisfies every clause of $\mathcal{C}$ [8].*

Definition 2 (Language of a relation). *Given a set of strict Horn clauses $\mathcal{C}$ and a relation symbol R, $\mathcal{L}(R, \mathcal{C})$ is the set of tuples of terms rooted by R in $\mathcal{H}(\mathcal{C})$, i.e., $\mathcal{L}(R, \mathcal{C}) = \{(t_1, \ldots, t_n) \mid R(t_1, \ldots, t_n) \in \mathcal{H}(\mathcal{C})\}$.*

Example 1. Let $\mathcal{C}$ be the set of clauses $R_0(a, b)$, $R_1(f(X), g(Y)) \Leftarrow R_0(X, Y)$, and $R_0(f(X), g(Y)) \Leftarrow R_1(X, Y)$. The smallest Herbrand model $\mathcal{H}(\mathcal{C})$ is defined by $\mathcal{H}(\mathcal{C}) = \{R_0(f^{2n}(a), g^{2n}(b)) \mid n \geq 0\} \cup \{R_1(f^{2n+1}(a), g^{2n+1}(b)) \mid n \geq 0\}$. The language of R_0 is $\mathcal{L}(R_0, \mathcal{C}) = \{(f^{2n}(a), g^{2n}(b)) \mid n \geq 0\}$.

In the paper, we use two different types of clauses: *constrained clauses* to represent the verification problem, i.e., the input program $\mathcal{P}$ and the property ϕ to verify, and *shallow Horn clauses* to represent the model M such that $M \models \mathcal{P}$ and $M \models \phi$, which are the focus of the next section. Usually, the constraints defining the programs are represented using constrained *Horn* clauses, but our translation yields clauses whose head might contain multiple atoms, so our method must handle non-Horn constrained clauses.

Constrained clauses are clauses with additional constraints from a theory T, i.e., formulas of the shape $\forall \overrightarrow{X}, H \Leftarrow B \wedge \psi(\overrightarrow{X})$ with $\forall \overrightarrow{X}, H \Leftarrow B$ a clause and $\psi(\overrightarrow{X})$ a formula over $\overrightarrow{X}$ adding theory-related constraints. In our setting, such constraints are over the theory of inductive data types [2] without tester predicates. Thus, constraints are of the form $X = f(\overrightarrow{X})$, where $f \in \Sigma$, X is a typed variable and $\overrightarrow{X}$ is a tuple of typed variables. For simplicity, we usually fold up such constraints, i.e., write $R(p)$ instead of $X = p \wedge R(X)$. A *ground* (constrained) clause is one that has no variables or where every variable's value is completely determined by $\psi(\overrightarrow{X})$, e.g, $R(X) \Leftarrow X = Nil$ is considered ground.

4 Shallow Horn Clauses (SHoCs)

This section formally defines the notion of Shallow Horn Clause (SHoC) as a strict Horn clause whose head atom is linear and shallow, and whose body consists of atoms that do not contain any constructors and whose variables are all introduced in the head atom.

Definition 3 (Linear, flat, and shallow patterns). *A pattern p is* linear *if every variable appears at most once in p. A pattern $f(\overrightarrow{X})$ is* flat *if $\overrightarrow{X}$ is a tuple of variables (with $\overrightarrow{X}$ empty when f is a constant). A pattern $f(\overrightarrow{p})$ is* shallow *if $\overrightarrow{p}$ is a tuple of flat patterns.*

For instance $f(X, g(Y))$ is linear but not flat (because of $g(Y)$) nor shallow (because of X), $f(X, X)$ is not linear but it is flat, and $f(g(X), h(Y, Z))$ is linear and shallow. By extension, a linear (resp. shallow, flat) atom is a pattern of type *bool* which is linear (resp. shallow, flat).

Definition 4 (Shallow Horn Clause – SHoC). *A* Shallow Horn Clause *is a strict Horn clause such that: (a) the head atom H is linear and shallow, (b) all atoms of the body B are flat, (c) the clause has no existential variables, i.e. $Vars(B) \subseteq Vars(H)$.*

In the following, we write SHoCs for a finite set of SHoC.

SHoC definition motivation The strictness of the clause allows to see them as an induction rule, and a SHoCs as an inductively-defined set of relations. Shallow heads and flat bodies in SHoCs greatly limits the expressivity of such clauses and only allows relations that do not need to keep the value of some parameter while exploring the others recursively. The absence of existential variables is what allows to have a simple top-down membership procedure, as the procedure does not need to find a value (i.e. prove the *existence*) for any existential variable. Linearity in the head simplifies SHoCs manipulation (e.g., removing epsilon clauses, see Section 4.3) while preserving the expressive power since equalities between variables, which are SHoCs-definable, can be used in the body. The body is not required to be linear. This allows to recognize relations that do not only need to build upon terms but also compare them (e.g. the R_{sh} relation from Section 1).

4.1 Expressivity of SHoCs

We illustrate the expressivity of SHoCs with the relations from the example of Section 1. Automata for *isEmpty*, *no*0 and *all*0 are replaced by the following seven SHoCs, where R_0 (resp. R_s) is an additional predicate recognizing only z (resp. natural numbers greater to 0).

$R_0(z) \quad R_s(s(X)) \quad R_{isEmpty}(leaf)$
$R_{all0}(leaf) \quad R_{all0}(node(T_1, E, T_2)) \Leftarrow R_0(E) \wedge R_{all0}(T_1) \wedge R_{all0}(T_2)$
$R_{no0}(leaf) \quad R_{no0}(node(T_1, E, T_2)) \Leftarrow R_s(E) \wedge R_{no0}(T_1) \wedge R_{no0}(T_2)$

The following set of SHoCs named S_3, with relations R_{leq}, R_{he}, and R_{heRB}, represent, respectively, the models for *leq*, *height*, and *heightRB* to prove property ϕ_3. The additional relation R_{sh} is necessary to express *height* has a SHoCs. Recall that the atom $R_{sh}(t, n)$ is true if tree t is of height smaller or equal to n.

$R_{leq}(z, z)$
$R_{leq}(z, s(X))$
$R_{leq}(s(X), s(Y)) \Leftarrow R_{leq}(X, Y)$

$R_{sh}(leaf, z)$
$R_{sh}(leaf, s(N))$
$R_{sh}(node(T_1, E, T_2), s(N))$
$\quad \Leftarrow R_{sh}(T_1, N) \wedge R_{sh}(T_2, N)$

$R_{he}(leaf, z).$
$R_{he}(node(T_1, E, T_2), s(N))$
$\quad \Leftarrow R_{he}(T_1, N) \wedge R_{sh}(T_2, N)$
$R_{he}(node(T_1, E, T_2), s(N))$
$\quad \Leftarrow R_{sh}(T_1, N) \wedge R_{he}(T_2, N)$

$R_{heRB}(leaf, z)$
$R_{heRB}(node(T_1, E, T_2), s(N))$
$\quad \Leftarrow R_{heRB}(T_2, N)$

SHoCs are strictly more expressive than convoluted tree automata because any convoluted tree automaton has an equivalent SHoCs and some SHoCs-definable relations such as $\{f(t, t) \mid t \in \mathcal{T}(\Sigma)\}$ cannot be represented by a convoluted tree automaton.

There exists a variety of clause-based formalism to represent languages, such as *CS-programs* [17]. A CS-program is a set of CS-clauses, which are strict Horn clauses whose body is flat and linear. A CS-program is also interpreted by the least fixed-point semantics. The expressivity of SHoCs and CS-programs are incomparable. The R_{he} and R_{sh} examples cannot be represented using a CS-program because of bodies linearity: CS-programs can only build relations from bottom to top but can not compare two subterms (needed for R_{he} and R_{sh}). On the other hand, since the head of CS-programs is not required to be shallow, it allows them to precisely represent some relations where symbols need to be stacked in the head of the clause. This is the case for the *double* function (Example 2) which is out of the scope of SHoCs but in the scope of CS-programs.

Example 2. Let $double(n, m)$ be the relation s.t. $m = 2 \times n$, defined by the CHCs:

$$double(z, z) \quad double(s(N), s(s(M))) \Leftarrow double(N, M)$$
$$N = M \Leftarrow double(X, N) \wedge double(X, M)$$

Note that the set consisting of the first two clauses is a valid CS-program for *double*. The relation *double* cannot be recognized by a SHoCs. First, note that the clause $double(s(N), s(s(M))) \Leftarrow double(N, M)$ is not a SHoC because of the double s symbol in the head. Then, the idea of the proof is very close to that of the word language $a^n b^n$ not being recognizable by a finite string automaton. For a SHoCs to recognize *double* we would need predicate symbols to record how many s symbols have been read on the first parameter while the head symbols for the two parameters were both s and then, once z is encountered for the first parameter, check that this number of s's is the same for the remainder of the second parameter. For instance, a possible SHoCs representing the finite relation $\{(s(s(z)), s(s(s(s(z)))))\}$ in R is $R(s(X), s(Y)) \Leftarrow R_1(X, Y)$, $R_1(s(X), s(Y)) \Leftarrow R_2(X, Y)$, $R_2(z, s(Y)) \Leftarrow R'_1(Y)$, $R'_1(s(X)) \Leftarrow R'_0(X)$, and $R'_0(z)$. However, representing the *double* relation for all natural numbers would require infinitely many predicate symbols and therefore an infinite SHoCs, which is forbidden.

4.2 Compactness of SHoCs

SHoCs enjoy a form of genericity which permits to represent relations in a more compact way than tree automata. This has already been illustrated in Section 1, where the automaton for *heightRB* has 6 convoluted transitions whereas the equivalent SHoCs has 2 clauses. Now, consider the relation $length(l, n)$ relating lists with their size. A SHoCs for this relation is:

$$length(nil, z) \qquad length(cons(E, L), s(N)) \Leftarrow length(L, N)$$

This SHoCs does not constrain the structure of elements in the list. Thus, this SHoCs is valid for lists of elements of any type. This results in a more generic and more compact representation than what tree automata can do for regular languages [11, 13, 16, 20] and for regular relations [18], where the complete structure of elements of the list has to be described explicitly by transitions. This is a general phenomenon that we have observed during the experiments, see Section 6. It significantly improves the efficiency of inference of SHoCs w.r.t. inference of (convoluted) tree automata.

4.3 ϵ-clauses and their elimination

In order to ease the definition of operations such as union or intersection of SHoCs, we introduce a new type of clauses, called ϵ-clauses. The ϵ-clauses are similar to ϵ-transitions of automata that define transitions between states without recognizing any symbol. However, though ϵ-clauses simplify the definition of some operations on SHoCs, they are not valid SHoCs (because their heads do not contain any function symbols, *i.e.*, they are flat). Thus, we need a procedure to generate an equivalent SHoCs without ϵ-clauses, similar to the removal of ϵ-transition in tree automata. For SHoCs, we call this procedure *Extend* and it is defined below. The *Extend* procedure is close to the *unfolding* rule from [17].

Definition 5 **(ϵ-clause and ϵ-definition).** *An ϵ-clause is a strict Horn clause $H \Leftarrow B$ such that (a) H and B are flat and linear (b) $Vars(H) = Vars(B)$.*
Given a relation symbol R and a SHoCs S, an ϵ-definition of R is a set $\mathcal{C}_R^\epsilon$ of ϵ-clauses $R(\overrightarrow{X}) \Leftarrow B$ such that R does not appear in S nor in B.

Definition 6 **($Extend(S, \mathcal{C}_R^\epsilon)$).** *Let S be a SHoCs and suppose that no two clauses in S share variables and let $\mathcal{C}_R^\epsilon$ be an ϵ-definition, then $Extend(S, \mathcal{C}_R^\epsilon) = S \cup \bigcup_{\varphi \in \mathcal{C}_R^\epsilon} Ext_\varphi$ where $Ext_{R(\overrightarrow{X}) \Leftarrow R_1(\overrightarrow{X}_1) \wedge \ldots \wedge R_n(\overrightarrow{X}_n)}$ is the following set of SHoCs:*

$$\left\{ \sigma(R(\overrightarrow{X}) \Leftarrow B_1 \cup \ldots \cup B_n) \;\middle|\; [\textstyle\bigwedge_{i \in [1 \ldots n]} (R_i(\overrightarrow{t}_i) \Leftarrow B_i) \in S] \wedge [\sigma = MGU(U)] \right\}$$

with $U = \{(\overrightarrow{X}_i, \overrightarrow{t}_i) \mid i \in [1 \ldots n]\}$.

Example 3 (ϵ-definition and $Extend(S, \mathcal{C}_R^\epsilon)$). Let S be a SHoCs with clauses:

$$\begin{array}{llll} (a) & R_1(h(X)) \Leftarrow A(X) & (b) & R_1(h(X)) \Leftarrow B(X) \\ (c) & R_2(f(X), g(Y)) \Leftarrow C(X) & (d) & R_2(f(X), h(Y)) \Leftarrow D(X) \end{array}$$

Let $\mathcal{C}_R^\epsilon$ be the ϵ-definition $\{R(X,Y) \Leftarrow R_1(X) \wedge R_2(Y,X)\}$. The set $Extend(S, \mathcal{C}_R^\epsilon)$ contains S and the two new SHoCs:

$$R(h(X), f(Y)) \Leftarrow A(X) \wedge D(Y) \quad R(h(X), f(Y)) \Leftarrow B(X) \wedge D(Y)$$

Note that every clause with head relation R_1, (a) and (b), has for parameter a pattern the form $h(X')$, which constrains X to be of the form $h(X')$ for the atom $R_1(X)$. This forbids to use the clause (c) for $R_2(Y,X)$, as it constrains X to be of the form $g(X')$. The two new clauses of $Extend(S, \mathcal{C}_R^\epsilon)$ result from the combination of the clause (d) with the two clauses (a) and (b).

4.4 Closure properties and decision procedures of SHoCs

This section defines the union, intersection, and complement of SHoCs. We also show that the membership problem is decidable but emptiness is not. First, note that a set $\{S_1, \ldots, S_n\}$ of SHoCs can always be combined into one by taking the set-union $S_1 \cup \ldots \cup S_n$ of the clauses composing them, possibly with some renaming if two SHoCs define different relations with the same name.

Definition 7 (Closure by intersection of relations). *Let S be a SHoCs defining a set of relations $\mathcal{R}$ and let $\mathcal{R}' \subseteq \mathcal{R}$ a subset of same-type relations. The intersection of $\mathcal{R}'$ in a fresh relation symbol R is a SHoCs $S_\cap$ defined as*

$$S_\cap = Extend(S, \mathcal{C}_R^\epsilon) \text{ with } \mathcal{C}_R^\epsilon = \{R(\vec{X}) \Leftarrow \bigwedge_{R' \in \mathcal{R}'} R'(\vec{X})\}$$

Definition 8 (Closure by union of relations). *Let S be a SHoCs defining a set of relations $\mathcal{R}$ and let $\mathcal{R}' \subseteq \mathcal{R}$ a subset of same-type relations. The union of $\mathcal{R}'$ in a fresh relation symbol R is a SHoCs $S_\cup$ defined as*

$$S_\cup = Extend(S, \mathcal{C}_R^\epsilon) \text{ with } \mathcal{C}_R^\epsilon = \{R(\vec{X}) \Leftarrow R'(\vec{X}) \mid R' \in \mathcal{R}'\}$$

We have $\mathcal{L}(R, S_\cap) = \bigcap_{R' \in \mathcal{R}'} \mathcal{L}(R', S)$ and $\mathcal{L}(R, S_\cup) = \bigcup_{R' \in \mathcal{R}'} \mathcal{L}(R', S)$. Given a typed set E of tuples of terms, we denote by $\overline{E}$ the typed complement of E, i.e., tuples of terms of the same type but not belonging to E.

Definition 9 (Complement specification). *Let S be a SHoCs defining the set of relations $\mathcal{R}$. The complement of S is a SHoCs S^c such that $\forall R \in \mathcal{R}, \mathcal{L}(R, S^c) = \overline{\mathcal{L}(R, S)}$.*

When S is clear from context, we may simply write R^c to refer to R in S^c. Now, we introduce the projector notation for SHoCs which simplifies the definition of the complement construction by replacing every occurrence of the j^{th} variable of the function f_i by the projector $\pi_{(i,j)}$.

Definition 10 (Projector notation for SHoCs). *Any SHoC*

$$R(f_1(X_{(1,1)},\dots,X_{(1,|f_1|)}),\dots,f_n(X_{(n,1)},\dots,X_{(n,|f_n|)})) \Leftarrow B$$

can be equivalenty written as

$$R(f_1,\dots,f_n) \Leftarrow \sigma(B)$$

with $\sigma = \{X_{(i,j)} \mapsto i \cdot j \mid i \in [1\dots n] \wedge j \in [1\dots |f_i|]\}$.

Example 4. The clause $R_{sh}(node(T_1,E,T_2),s(N)) \Leftarrow R_{sh}(T_1,N) \wedge R_{sh}(T_2,N)$ can be written as $R_{sh}(node,s) \Leftarrow R_{sh}(\pi_{(1,1)},\pi_{(2,1)}) \wedge R_{sh}(\pi_{(1,3)},\pi_{(2,1)})$

Definition 11 (Complement construction). *Let S be a SHoCs and $\mathcal{R}_S$ be the set of relations that S defines. For a given relation R of type $\tau_1 \times \dots \times \tau_n \to bool$, let $F(R)$ be the set of tuples of constructors $(f_1,\dots,f_n)$ such that each constructor f_i has τ_i as output type. Then, the complement of S is defined as*

$$S^c = \bigcup_{R\in\mathcal{R}_S,\vec{f}\in F(R)} \{R(\vec{f}) \Leftarrow B' \mid B' \in Flip(\{B \mid (R(\vec{f}) \Leftarrow B) \in S\})\}$$

with $Flip : \mathcal{P}(\mathcal{P}(A)) \to \mathcal{P}(\mathcal{P}(A))$ the set of all possible tuples made by selecting one atom per body:

$$Flip(\{B_1,\dots,B_n\}) = \{\{A_1,\dots,A_n\} \mid (A_1,\dots,A_n) \in B_1 \times \dots \times B_n\}$$

We have that $\forall R \in \mathcal{R}, \mathcal{L}(R,S^c) = \overline{\mathcal{L}(R,S)}$.

Example 5 (Complement). Let S_{sh} be the SHoCs defined in Section 1:

$$R_{sh}(leaf,z) \quad R_{sh}(leaf,s(N)) \quad R_{sh}(node(T_1,E,T_2),s(N)) \Leftarrow R_{sh}(T_1,N) \wedge R_{sh}(T_2,N)$$

The complement S^c_{sh} of S_{sh} is:

$R^c_{sh}(node(T_1,E,T_2),z)$

$R^c_{sh}(node(T_1,E,T_2),s(N)) \Leftarrow R^c_{sh}(T_1,N) \qquad R^c_{sh}(node(T_1,E,T_2,s(N)) \Leftarrow R^c_{sh}(T_2,N)$

With notations from the definition, here are the details of the computation, where $R = R_{sh}$ and $F(R) = \{(leaf,z),(leaf,s),(node,z),(node,s)\}$.

- For $\vec{f} = (node,s)$, we have:
  - Let $E_1 = \{B \mid R(\vec{f}) \Leftarrow B \in S\}$, i.e., $E_1 = \{\{R_{sh}(T_1,N),R_{sh}(T_2,N)\}\}$
  - Then, $Flip(E_1) = \{\{A\} \mid (A) \in B\}$ with $B = \{R_{sh}(T_1,N),R_{sh}(T_2,N)\}$, so $Flip(E_1) = \{\{R_{sh}(T_1,N)\},\{R_{sh}(T_2,N)\}\}$.
  - This yields the two rules whose head is $R^c_{sh}(node(T_1,E,T_2),s(N))$.
- For $\vec{f} = (leaf,z)$ or $\vec{f} = (leaf,s)$, we have:
  - Let $E_2 = \{B \mid R(\vec{f}) \Leftarrow B \in S\}$, i.e., $E_2 = \{\emptyset\}$
  - We have $Flip(E_2) = \emptyset$, so there is no rule with $R_{sh}(leaf,z)$ or $R_{sh}(leaf,s)$ as a head.
- For $\vec{f} = (node,z)$, we have:
  - Let $E_3 = \{B \mid R(\vec{f}) \Leftarrow B \in S\}$, i.e., $E_3 = \emptyset$

- We have $Flip(E_3) = \{\emptyset\}$, because the neutral element of the cartesian product is $\{\emptyset\}$.
- Thus, there is a rule $R^c_{sh}(node(T_1, E, T_2), z)$.

Theorem 1 (SHoCs emptiness problem is undecidable). *For S a SHoCs and R a relation, the SHoCs emptiness problem is to decide whether $\mathcal{L}(R, S){=}\emptyset$. This problem is undecidable.*

The proof uses an encoding of Minsky machines into SHoCs. See [19] for details.

Theorem 2 (SHoCs membership problem is decidable). *For S a SHoCs, $\overrightarrow{t}$ a tuple of terms, and R a relation, the SHoCs membership problem is to decide whether $\overrightarrow{t} \in \mathcal{L}(R, S)$. This problem is decidable.*

This theorem is a corollary of one of the theorems of the next section on the teacher. See [19] for details.

5 Satisfiability as model inference

This section presents a learner-teacher procedure for proving or disproving the satisfiability of a set Γ of constrained clauses. The procedure shows satisfiability of Γ by outputting a SHoCs S such that $S \models \Gamma$. Unsatisfiability is shown by outputting a contradiction in Γ. We have implemented this procedure in a manner reminiscent of an Implication CounterExample (ICE) procedure [9], in which two entities, a *learner* and a *teacher*, communicate SHoCs and counterexamples (ground instances of Γ). The teacher's role is to verify that a given SHoCs S satisfies all formulas of Γ, and if not extract a counterexample that is given to the learner. The learner's role is to propose a new SHoCs inferred from counterexamples that the teacher previously gave it.

The learner-teacher loop starts from Γ and an empty SHoCs $S_0 = \emptyset$. The teacher verifies whether $S_0 \models \Gamma$. If not, it outputs a ground counterexample $\widehat{\varphi}_0$, i.e., a ground instance of some clause in Γ which is not satisfied by S_0. The learner builds a new SHoCs S_1 satisfying $\widehat{\varphi}_0$ and gives it back to the teacher. If $S_1 \not\models \Gamma$, the teacher outputs a new counterexample $\widehat{\varphi}_2$. The learner builds a new SHoCs S_2 satisfying $\widehat{\varphi}_0$ and $\widehat{\varphi}_1$, etc. This loop may go on forever or stop if (a) the learner fails to build a model because the set of counterexamples is unsatisfiable, i.e., we found a counterexample to the verification problem; (b) the teacher succeeds to prove that the current model satisfies Γ, i.e., we proved that Γ is satisfiable, thus, verification succeeds. We begin by introducing the teacher procedure, then the learner procedure, and finally define the learner-teacher loop. This procedure enjoys refutational completeness and a relative positive completeness (see Section 5.3). All the proofs for this section can be found in [19].

5.1 Teacher

The teacher takes as input a finite set of clauses Γ, representing the program and property to verify, and a SHoCs S, representing a proposed model that is intended to prove satisfiability of Γ. If $S \models \Gamma$ then $\mathit{Teacher}(S, \Gamma)$ returns *None* and the verification algorithm succeeds. If $S \not\models \Gamma$ then $\mathit{Teacher}(S, \Gamma)$ returns $\mathit{Some}(\widehat{\varphi})$ where $\widehat{\varphi}$ is a *counterexample* to $S \models \Gamma$, i.e. a ground instance of some clause $\varphi \in \Gamma$ such that $S \not\models \widehat{\varphi}$. In the following, we use the hat on formulas ($\widehat{\varphi}$) and sets of formulas ($\widehat{\Gamma}$) to signal that their atoms are ground.

The problem that *Teacher* is trying to solve is undecidable. To see this, let S be a SHoCs and R a relation. Then, $\mathit{Teacher}(S, \{\mathit{False} \Leftarrow R(\overrightarrow{X})\}) = \mathit{None} \Leftrightarrow \mathcal{L}(R, S) = \emptyset$, but the emptiness problem of SHoCs is undecidable (Theorem 1). *Teacher* is based on the *Inhabits* procedure which takes as input a SHoCs S and $W = \{R_1(\overrightarrow{p}_1), \ldots, R_n(\overrightarrow{p}_n)\}$ a set of atoms and tries to find a single substitution σ such that every atom of $\sigma(W)$ is true in S. Informally, $\sigma(W)$ is true in S if every ground instance of $\sigma(W)$ belongs to S. Formally, given a set of atoms W, we say that W is true in S, written $S \models W$, if for all σ' s.t. $\mathit{Vars}(\sigma'(W)) = \emptyset$ we have for every atom of the shape $R(\overrightarrow{p})$ in W that $\sigma'(\overrightarrow{p}) \in \mathcal{L}(R, S)$. Note that a substitution σ applied to a set of atom W is also a set of atoms $\sigma(W)$, so the previous definition covers $S \models \sigma(W)$. We first define how to *unfold* an atom with a SHoC, which is a core mechanism of the *Inhabits* procedure.

Definition 12 **(*unfold* an atom along a SHoC).** *Let $A = R(f_1(\overrightarrow{p}_1), \ldots, f_n(\overrightarrow{p}_n))$ be an atom whose root functions are $(f_1, \ldots, f_n)$ and $\varphi = H \Leftarrow B$ a SHoC with $H = R(f_1(\overrightarrow{X}_1), \ldots, f_n(\overrightarrow{X}_n))$. Then $\mathit{unfold}(A, \varphi) = \sigma(B)$ with $\sigma = \{(X_{i,j}, p_{i,j}) \mid i \in [1 \ldots n] \wedge j \in [1 \ldots |\overrightarrow{p}_i|]\}$ a substitution unifying A and H.*

Definition 13 **(*Inhabits*).** *Let S be a SHoCs and W a set of atoms. $\mathit{Inhabits}(W, S)$ is a non-deterministic algorithm defined as:*

1. *Let $\sigma_0 := \{(X, X) \mid X \in \mathit{Vars}(W)\}$, $W_0 := W$, and $i := 0$.*
2. *If $W_i = \emptyset$, then return $\mathit{Some}(\sigma_i)$.*
3. *For each $A \in W_i$, choose one SHoC $\varphi_A = H_A \Leftarrow B_A \in S$. Let $U = \{A \overset{?}{=} H_A \mid A \in W_i\}$. If U is not unifiable, then return None. Otherwise, let σ be a most general unifier of U. Set $W_{i+1} := \bigcup_{A \in W_i} \mathit{unfold}(\sigma(A), \varphi_A)$, $\sigma_{i+1} := \sigma \circ \sigma_i$, $i := i + 1$, and go back to instruction 2.*
   If no such choice can be made, i.e, if $S = \emptyset$, then also return None.

Variables from SHoCs are universally quantified, so we assume no variable collision. In practice, this can be avoided by a renaming. We write $\mathit{Inhabits}^{det}$ for a determinisation of *Inhabits* where the choices are implemented as a breadth-first search until the returned value is a $\mathit{Some}(\sigma)$ or all branches return *None*.

Example 6 (Inhabits). Let $W = \{R_{he}(\mathit{node}(\mathit{leaf}, X, L), s(s(D)))\}$ and let S_3 the SHoCs from section 4.1. The execution of $\mathit{Inhabits}^{det}(W, S_3)$ returns $\mathit{Some}(\sigma)$ with $\sigma(X) = X$, $\sigma(L) = \mathit{node}(\mathit{leaf}, L_2, \mathit{leaf})$, and $\sigma(D) = z$. By correctness of the *Inhabits* procedure (theorem 3), $S \models \sigma(W)$, i.e., for any value x for X and l_2

for L_2, $\big(node(leaf, x, node(leaf, l_2, leaf)),\ s(s(z))\big) \in \mathcal{L}(R_{he}, S_3)$. See the extended version of this paper for a detailed execution [19].

Theorem 3 (*Inhabits* correctness and relative completeness).

- *If $Inhabits(W, S)$ terminates with $Some(\sigma)$, then $S \models \sigma(W)$*
- *If there exists a substitution σ such that $S \models \sigma(W)$, then there exists a terminating execution of $Inhabits(W, S)$ returning $Some(\sigma')$.*

However, there is no termination guarantee when there exists no substitution σ such that $S \models \sigma(W)$.

Teacher's procedure uses $Inhabits^{det}$ to search for a counterexample to the claim $S \models \Gamma$, i.e., it checks if S satisfies the *negation* of any formula $\varphi \in \Gamma$. For instance, if $\varphi = R(\vec{p}) \Leftarrow B$, proving that S satisfy the negation of φ consists in finding a substitution σ such that $S \models \sigma(B)$ and $S \not\models \sigma(R(\vec{p}))$. Using Definition 9, we can build R^c, the complement of R, and this boils down to finding σ such that $S \models \sigma(B)$ and $S \models \sigma(R^c(\vec{p}))$.

Definition 14 (Negating a clause). *Let φ be the clause $H \Leftarrow B$. We write $W_{\overline{\varphi}}$ for the set $\{R^c(\vec{p}) \mid R(\vec{p}) \in H\} \cup B$.*

Proposition 2 (Use *Inhabits* to check a formula w.r.t. a SHoCs).

$$\big(S \not\models \varphi\big) \Leftrightarrow \big(\exists\sigma,\ S \models \sigma(B) \text{ and } S \not\models \sigma(H)\big) \Leftrightarrow \big(Inhabits^{det}(W_{\overline{\varphi}}, S) = Some(\_)\big)$$

Because the substitutions that *Inhabits* returns do not necessarily yields ground counterexamples, we need a way to convert a non-ground formula to a ground formula.

Definition 15 (Smallest grounding). *The smallest grounding of a variable X, written $Grd(X)$, is a substitution σ such that: (a) $\sigma(X)$ is ground (b) for any other substitution σ' such that $\sigma'(X)$ is ground, $ht(\sigma(X)) \leq ht(\sigma'(X))$.*

Note that a term may have several smallest grounding. We exploit the minimality of groundings in the proof of Lemma 2. Smallest grounding extends to pattern, tuple, atom, set of atom, clause, and is possible because we only consider inhabited algebraic data types.

Definition 16 (Teacher). *Let S be a SHoCs and Γ be a finite set of clauses. $Teacher(S, \Gamma)$ is defined as*

1. *In parallel, run one instance $Inhabits^{det}(W_{\overline{\varphi}}, S)$ for each formula $\varphi \in \Gamma$ and enforce the depth of recursive calls (the value i) to be the same among instances of $Inhabits^{det}$ that have not yet terminated.*
2. *If some instance $Inhabits^{det}(W_{\overline{\varphi}}, S)$ returns $Some(\sigma)$, then let $\sigma' = Grd(\sigma(W_{\overline{\varphi}}))$ and $\widehat{\varphi} = \sigma'(\sigma(\varphi))$, and return $Some(\widehat{\varphi})$.*
3. *If all instances have returned None, then return None.*

Theorem 4 **(*Teacher* correctness and relative completeness).**

- *If $Teacher(S, \Gamma)$ terminates with $Some(\widehat{\varphi})$, then $\widehat{\varphi}$ is a ground instance of some formula $\varphi \in \Gamma$ and $S \not\models \widehat{\varphi}$, so $S \not\models \Gamma$.*
- *If $S \not\models \Gamma$ then there exists $\widehat{\varphi}$ such that $Teacher(S, \Gamma)$ terminates with $Some(\widehat{\varphi})$.*

Lemma 1 **(*Inhabits* height boundedness).** *If $Inhabits(W, S)$ terminates at step i with $Some(\sigma)$, then $i \leq ht(\sigma(W)) \leq ht(W) + i$.*
Moreover, if there exists a substitution σ such that $Vars(\sigma(W)) = \emptyset$ and $S \models \sigma(W)$, then there exists an execution of $Inhabits(W, S)$ which stops in at most $ht(\sigma(W))$ steps by returning $Some(\_)$.

Lemma 2 **(*Teacher* height boundedness).** *If $Teacher(S, \Gamma) = Some(\widehat{\varphi})$ then for any other counterexample $\widehat{\varphi}'$ of $S \models \Gamma$, $ht(\widehat{\varphi}) \leq ht(\widehat{\varphi}') + dh$ with dh only depending on Γ.*

5.2 Learner

The learner receives a finite set of ground clauses $\widehat{\Gamma}$ from the teacher. This set is a counterexample to every previous SHoCs proposed by the learner. Starting from $\widehat{\Gamma}$, the objective of the learner is to build a *new* SHoCs satisfying $\widehat{\Gamma}$. If $\widehat{\Gamma}$ is contradictory, then the teacher has found a real counterexample to the property and $Learner(\widehat{\Gamma})$ returns *None*. If $\widehat{\Gamma}$ is satisfiable, then $Learner(\widehat{\Gamma}) = Some(S)$ with S a SHoCs such that $S \models \widehat{\Gamma}$. We will furthermore ensure that S is minimal w.r.t the number of relations defined in S among all possible SHoCs that satisfy $\widehat{\Gamma}$.

Finding a new SHoCs S is implemented as a constraint solving problem in the *Answer Set Programming* (ASP) paradigm with support for so-called *choice rules*, of which *Clingo* [10] is the solver we use. From $\widehat{\Gamma}$, the learner builds a set of constraints $C(\widehat{\Gamma})$ defining all the possible choices to build a valid SHoCs satisfying $\widehat{\Gamma}$. Then, *all the constraints* of $C(\widehat{\Gamma})$ are given to the ASP solver which returns *one* solution, if it exists. In our definition of $C(\widehat{\Gamma})$, we use the **choose-one** keyword to ask the ASP solver to pick exactly one atom in a set and check that it is not in contradiction with the other constraints of $C(\widehat{\Gamma})$. For instance, to satisfy the two constraints **choose-one**$\{x > 1, x < 1\}$ and **choose-one**$\{x > 5, x = 5\}$, the solver can pick $x > 1$ from the first constraint and either $x > 5$ or $x = 5$ from the second. *A contrario*, picking $x < 1$ from the first constraint leads to a contradiction with any of the two possibilities for the second constraint.

To guarantee (relative) completeness of the learner-teacher loop, solving $C(\widehat{\Gamma})$ is done incrementally by considering first the smallest solutions w.r.t their number of relation symbols. We write i_{step} for the number of *new* relation symbols in the inferred SHoCs, i.e., relation symbols that are not already present in $\widehat{\Gamma}$. The constraints $C(\widehat{\Gamma})$ are checked for satisfiability, starting with $i_{step} = 0$, and i_{step} is incremented as long as the constraints are unsatisfiable. This process can be bounded by an upper-bound on i_{step}. Our upper-bound for i_{step} is the number of subterms appearing in $\widehat{\Gamma}$. Using this upper-bound is sound because, with one relation symbol to recognize exactly one subterm, it becomes immediate to create a SHoCs recognizing all positive atoms occuring in the input set of (ground) counterexamples. If a solution for the constraints have not been found within this upper-bound, then they are unsatisfiable.

Definition 17 (Constraints generated by the learner $C(\widehat{\Gamma})$). *Let $\widehat{\Gamma}$ be a finite set of ground Horn clauses. $C(\widehat{\Gamma})$ is the set of constraints generated by the following rules:*

*(a) For each i in $[1 \dots i_{step}]$, **choose-one** type for the relation R_i^{new} among the tuples of types of sub-terms of $\widehat{\Gamma}$.*

*(b) For each $\widehat{\varphi} \in \widehat{\Gamma}$ with $\widehat{\varphi} = R_1(\overrightarrow{t}_1) \vee \dots \vee R_n(\overrightarrow{t}_n) \vee \neg R'_1(\overrightarrow{t}'_1) \dots\dots \vee \neg R'_k(\overrightarrow{t}_k)$, **choose-one** literal that needs to be true, i.e. choose one $R'_i(\overrightarrow{t}'_i)$ to be false or one $R_i(\overrightarrow{t}_i)$ to be true.*

(c) For each $R(\overrightarrow{t})$ that must be true a new SHoC is be created. If $\overrightarrow{t}$ is of the form $(f_1(\overrightarrow{t}_1), \dots, f_p(\overrightarrow{t}_p))$, the head of the new SHoC will be $R(f_1(\overrightarrow{X}_1), \dots, f_p(\overrightarrow{X}_p))$.

*(d) For each new SHoC $\varphi = R(\overrightarrow{p}) \Leftarrow B$ that is being created, construct its body B, i.e., choose which atoms $R'(\overrightarrow{X})$ are to be in B. For any relation R' and type-compatible variables $\overrightarrow{X}$ among $Vars(\overrightarrow{p})$, the following constraint is generated: **choose-one** $\{R(\overrightarrow{X}) \in B,\ R(\overrightarrow{X}) \notin B\}$, therefore choosing a subset of possible atoms to constitue the body B.*

(e) For each SHoC $\varphi = R(\overrightarrow{p}) \Leftarrow B$ that has been created because some atom $R(\overrightarrow{t})$ needed to be true, add the constraint that every atom of $unfold(R(\overrightarrow{t}), \varphi)$ must also be true.

(f) Add the constraint that no atom $R'(\overrightarrow{t}')$ that must be false is made true by some SHoC.

The following example illustrates the main steps of the learning algorithm.

Example 7 (Learning the leq relation). Let $i_{step} = 0$ and let $\widehat{\Gamma}$ be the set of the following 5 ground examples:

(1) $R_{leq}(z,z)$ (2) $R_{leq}(z,s(z))$ (3) $R_{leq}(s(z),s(z)) \Leftarrow R_{leq}(z,z)$
(4) $R_{leq}(s(z),z) \Leftarrow R_{leq}(s(s(z)),s(z))$ (5) $false \Leftarrow R_{leq}(s(z),z)$

Since $i_{step} = 0$, Rule (a) does not apply. Rule (b) divides all the atoms of the 5 ground examples into two sets: those who should be true and those who should be false. For (1) and (2), there is a unique possibility, i.e., $R_{leq}(z,z)$ and $R_{leq}(z,s(z))$ need to be true. For (3), the constraint to add to $C(\widehat{\Gamma})$ is **choose-one**$\{R_{leq}(s(z),s(z)), \neg R_{leq}(z,z)\}$. Later, when solved by the ASP solver, this choice operator will have only one possible solution since $R_{leq}(z,z)$ is already true. In the same way, solving the additional **choose-one** constraints for (4) and (5) has only one solution and yields the positive atoms: $R_{leq}(z,z)$, $R_{leq}(z,s(z))$, $R_{leq}(s(z),s(z))$, and the negative ones: $R_{leq}(s(z),z)$, $R_{leq}(s(s(z)),s(z))$. Rule (c) generates one SHoC for every atom that must be true, by replacing sub-terms with variables and introducing bodies B_1, B_2, B_3 to be determined:

$\varphi_1 = R_{leq}(z,z) \Leftarrow B_1$; $\varphi_2 = R_{leq}(z,s(X)) \Leftarrow B_2$; $\varphi_3 = R_{leq}(s(X_1),s(X_2)) \Leftarrow B_3$

Rule (d) completes $C(\widehat{\Gamma})$ with constraints on all B_i. For B_1 this constraint is **choose-one**$\{\emptyset\}$. For B_2, this constraint is **choose-one**$\{\emptyset, \{R_{leq}(X, X)\}\}$. For B_3, four **choose-one** constraints are generated, one for each atom of $R_{leq}(X_1, X_1)$, $R_{leq}(X_1, X_2)$, $R_{leq}(X_2, X_1)$, and $R_{leq}(X_2, X_2)$, which yields 16 possible different B_3, including $\emptyset$, $\{R_{leq}(X_1, X_2)\}$, $\{R_{leq}(X_1, X_2), R_{leq}(X_2, X_1)\}$, ... When solving these constraints, the ASP solver can choose $B_1 = B_2 = B_3 = \emptyset$. With this choice, constraints added to $C(\widehat{\Gamma})$ by step 5 are trivially satisfied because no new atom is forced to be true. However, the ASP solver cannot choose $B_3 = \emptyset$, because constraints added by step 6 yields a contradiction, i.e., $R_{leq}(s(s(z)), s(z))$ is made true by φ_3 but it belongs to the set of false atoms. Hence, the ASP solver must change some of its choices. One possible change is to define $B_3 = \{R_{leq}(X_1, X_2)\}$. The clause φ_3 becomes $R_{leq}(s(X_1), s(X_2)) \Leftarrow R_{leq}(X_1, X_2)$. Since it was created to recognize $R_{leq}(s(z), s(z))$, this forces $R_{leq}(z, z)$ to be in the set of atoms that need to be true but, since it is already there, nothing changes. With this last SHoC S, every atom that must be true is still made true by some clause, and no atom that must be false is now made true by any SHoC. Thus, the learner has found a model for $\widehat{\Gamma}$ so it returns $Some(\{\varphi_1, \varphi_2, \varphi_3\})$. Note that in this simple example the learner converged in one step, without having to increment $i_{step} = 0$. Having $i_{step} > 0$ would result in introducing i_{step} additional relation symbols.

Lemma 3. *The learner always terminates.*

Theorem 5 (Learner correctness, completeness, minimality). *Let $\widehat{\Gamma}$ be a finite set of counterexamples. If Γ is satisfiable, then $Learner(\widehat{\Gamma}) = Some(S)$ with $S \models \widehat{\Gamma}$ and such that S is minimal among SHoCs that satisfy $\widehat{\Gamma}$ w.r.t the number of relations. If Γ is unsatisfiable, then $Learner(\widehat{\Gamma}) = None$.*

Proof. A satisfiable, finite, ground set of formulas $\widehat{\Gamma}$ has a first-order Herbrand structure with a finite number of true atoms. Any first-order Herbrand structure with a finite number of true atoms can be represented by a SHoCs by introducing additional intermediate relation symbols to ensure that all clauses are shallow (and becomes a SHoC). Hence, if the set $\widehat{\Gamma}$ is satisfiable then there exists a SHoCs satisfying it such that its number of relations in no more than the number of subterms appearing in Γ, which is the bound used for i_{step}.

To satisfy $\widehat{\Gamma}$, at least one literal of each clause $\varphi \in \widehat{\Gamma}$ must be true. Choosing such literals is the role of constraint (b). Having the generated SHoCs to satisfy every chosen positive literal in ensured by constraints (c) and (e). Constraint (c) creates a SHoC for each chosen positive literal, so we know that each atom that must be true leads to the creation of a SHoC to recognize it. Ensuring that the created SHoC indeed recognizes the atom is the role of constraint (e). Having the generated SHoCs to not satisfy any chosen negative literal is directly ensured by constraint (f). Therefore if $Learner(\widehat{\Gamma}) = Some(S)$, we have $S \models \widehat{\Gamma}$. The minimality is a consequence of the incremental solving. Conversely, if $\widehat{\Gamma} = None$, then no matter the choice of literal, no SHoCs can verify $\widehat{\Gamma}$, so $\widehat{\Gamma}$ is unsatisfiable.

5.3 Satisfiability process

Definition 18 (The satisfiability procedure $Sat(\Gamma)$).
Given Γ a finite set of constrained clauses, the loop $Sat(\Gamma)$ proceeds as:

0. *Let $\widehat{\Gamma}_0 := \emptyset$ and $i := 0$.*
1. *If $Learner(\widehat{\Gamma}_i) = None$, then return "Disproved: $\widehat{\Gamma}_i$".*
   If $Learner(\widehat{\Gamma}_i) = Some(S_i)$, then go to step 2.
2. *If $Teacher(S_i, \Gamma) = None$, then return "Proved: S_i".*
   If $Teacher(S_i, \Gamma) = Some(\widehat{\varphi}_i)$, let $\widehat{\Gamma}_{i+1} := \widehat{\Gamma}_i \cup \{\widehat{\varphi}_i\}$, $i := i + 1$, and go to step 1.

This procedure does not always finish, as its goal is undecidable, but it enjoys refutational completeness and a relative positive completeness.

Lemma 4 (Progress). *During the execution $Sat(\Gamma)$:*

– *The learner never outputs the same model twice, as the model is either correct or the learner received a counterexample to it.*
– *The teacher never outputs the same ground constraint twice, as every model the learner outputs satisfies every ground constraint the teacher sent.*

Theorem 6 (Refutational completeness). *Let Γ a finite set of contradictory clauses. Then $Sat(\Gamma)$ terminates with "Disproved".*

Proof. Let $Grd(\Gamma)$ be the set of ground instances of Γ. By definition of Herbrand models (every element of the domain can be expressed as a term) and because Γ is contrady, $Grd(\Gamma)$ is contradictory too. By first-order logic compactness theorem, we know that a set of formulas is contradictory iff there exists a finite subset of contradictory formulas, so let $F \subseteq Grd(\Gamma)$ be such a set.
Let $H = \max_{\widehat{\varphi} \in F}(ht(\widehat{\varphi}))$ be the height of a highest clause in F. Let dh be the value defined from Γ as mentioned in Lemma 2. Let $Cl(F) = \{\widehat{\varphi} \mid \widehat{\varphi} \in Grd(\Gamma) \wedge ht(\widehat{\varphi}) \leq H + dh\}$. $Cl(F)$ is finite and also contradictory, as $F \subseteq Cl(F)$.
For any step i of $Sat(\Gamma)$ such that $Learner(\Gamma_i) = Some(S_i)$, we have $S_i \not\models F$ and so let h' be the height of a smallest counterexample of $S_i \models F$. By Theorem 5, $Teacher(S_i, \Gamma) = Some(\widehat{\varphi})$ with $\widehat{\varphi}$ a ground instance of some formula $\varphi \in \Gamma$ such that $ht(\widehat{\varphi}) \leq h' + dh$. Thus $\widehat{\varphi}_i \in Cl(F)$.
Because every counterexample $\widehat{\varphi}_i$ that the teacher outputs is contained in $Cl(F)$ and of Lemma 4, the ground clauses $\widehat{\Gamma}_{|Cl(F)|}$ accumulated by the learner at step $|Cl(F)|$ would necessarily be such that $F \subseteq \widehat{\Gamma}_{|Cl(F)|}$. Therefore there exists a step $i_{unsat} \in [0 \ldots |Cl(F)|]$ such that $\widehat{\Gamma}_{i_{unsat}}$ is unsatisfiable, and the $Sat(\Gamma)$ procedure stops at this step with "Disproved".

Satisfiability of Γ is undecidable and $Sat(\Gamma)$ may not terminate for two reasons. The first is that Γ may be satisfiable but not admit a SHoCs, in which case Sat will run indefinitely. The second is that even if a satisfying S_i is proposed by the learner, the call to $Teacher(S_i, \Gamma)$ may not terminate.

Theorem 7 (Relative completeness). *Let Γ be a finite set of clauses. If Γ admits a SHoCs satisfying it, then there exists a step $i \in \mathbb{N}$ in which S_i, the learner's SHoCs output at this step, is such that $S_i \models \Gamma$.*

Proof. In this proof, we call *size* of a SHoCs the number of relations that it defines. Suppose that Γ admits a SHoCs. Let N be the size of a smallest SHoCs S such that $S \models \Gamma$. Any first-order model of Γ is also a model of $Grd(\Gamma)$ and of every subset of it, so, for any step i of the *Sat* procedure, $S \models \widehat{\Gamma}_i$. Therefore any smallest SHoCs of a subset $\widehat{\Gamma}_i \subseteq Grd(\Gamma)$ is smaller or equal than N. Because of learner minimality (Theorem 5), any SHoCs the learner proposes is of size smaller or equal to N.
Because the number of SHoCs of any fixed size is finite (modulo relation and variable names), we know that the number of SHoCs (that do not satisfy Γ and) that are smaller or equal than N is finite.
Because of lemma 4, we know that the learner never outputs twice the same SHoCs. We know that the learner always terminates (Lemma 3) and, as long as the learner proposes SHoCs that do not satisfy Γ, the teacher terminates too (Theorem 4).
Therefore the learner will end up proposing a (smallest) SHoCs satisfying Γ.

6 Implementation and Experiments

We implemented the verification algorithm in Ocaml. The implementation is available at `https://gitlab.inria.fr/tlosekoo/auto-forestation.git`. This includes an implementation of SHoCs, model checking and model-inference procedure. The *teacher* closely follows the breadth-first search of the procedure in Section 5. The *learner* delegates learning of SHoCs to the finite-model finder *Clingo* [10].

6.1 Approximations

Since the programs we verify are deterministic and terminating, their clausal representation has only one possible model. However, this model may not be representable using a SHoCs. Thus, trying to verify a property using an exact model of the relation will fail on such programs. We circumvent this problem by approximating relations, following the approach proposed in [18]. Assume that we have a relation $plus(N, M, U)$ relating N and M with their sum U and a relation $le(N, M)$ relating N and M iff $N < M$. Let ϕ_4 be the property

$$\phi_4 \overset{def}{=} plus(N, M, U) \wedge le(z, M) \Rightarrow le(N, U)$$

which cannot be proved straightforwardly using SHoCs because there is no SHoCs representing *exactly* the relation *plus*. We can prove this property with an over-approximation of the relation *plus*, say $plus^+$, such that $plus^+(N, M, U) \wedge le(z, M) \Rightarrow le(N, U)$ is true. This is the `plus_le.smt2` example of our benchmark. Inspection of the SHoCs inferred by our tool over-approximates the *plus* relation by $\{(s^n(z), z, s^n(z)) \mid 0 \leq n\} \cup \{(s^n(z), s^k(z), s^m(z)) \mid 0 \leq n < m$ and $0 < k\}$ which is enough to carry out the proof.

Predicates on which using an approximation is safe are selected w.r.t. the properties to prove and the relations definition. If the property is implicative and if a relation occurs on the body (resp. head) side of the implication then its relation can *not* be under- (resp. over-) approximated. If it occurs on both sides its relation cannot be approximated. Moreover, because relations are defined using other relations, then approximating a relation must account for these connections. More precisely, a relation R which cannot be over-approximated and which contains a defining clause whose body (resp. head) contains another relation R' forbids R' to be over- (resp. under-) approximated. The same is true for under-approximations. E.g., for the property ϕ_4, we can over-approximate *plus* but we cannot approximate *le* since it occurs on both sides of the implication. For the property ϕ_3 of Section 2, *height* can be over-approximated but *leq* cannot be under-approximated, even only appearing on the head-side of the property, because *leq* also appears in the body of a clause defining *height*, so under-approximating *leq* would propagate and cause an under-approximation of *height*.

Our current implementation focuses on over-approximations but cannot, yet, produce under-approximations. Over-approximations are computed by forgetting about functionality clauses (e.g. the clause $N = M \Leftarrow heightRB(T, N) \wedge heightRB(T, M)$ for the *height* relation). This allows any over-approximation for relations coming from a function in the program. We do not yet have any satisfying way to over-approximate other relations or to under-approximate them.

6.2 Benchmarks

For the experiments, we use examples coming from [13], where regular sets are sufficient to carry out the proof, examples coming from [6,7,18] where convoluted tree automata are necessary, and add some new examples which are out of the scope of [18], such as the property ϕ_3 in Section 2.

All of our experimental results are available at `http://people.irisa.fr/Thomas.Genet/AutoForestation/shocs/benchmarks/index.html`. All the programs are defined as SMTLIB functions and are tranformed into constrained clauses by our tool. On each example, the result of this translation as well as the found SHoCs models can be consulted in the answer log of the tool. On a total of 174 examples, our solver proves 106, disproves 32, and timeouts on 36 after 60s. Further increasing the timeout does not have much of an impact. The 88 positive and negative examples verified by [18] are also verified by our solver, except that some of them need more time. Our solver succeeds on 23 out of the 79 first-order Isaplanner examples in less than 5s. Our approach and [22] seems to be complementary as they succeed on different sets of examples. This can be observed on the IsaPlanner benchmark where our technique fails on most of examples that [22] handles (i.e. 4, 5, 29, 30, 39, 42, 50, 62, 67, 71, 86) and succeeds on examples on which they do not report any success (i.e. 10, 11, 17, 18, 20, 21, 22, 23, 24, 25, 31, 32, 33, 34, 45, 65, 68, 69).

Spacer [12], Eldarica [14], and RInGen [21] are powerful general purpose SMT solvers with some ADT support. However, this support for ADTs does not cover relational properties and, unsurprisingly, those solvers do not perform well on positive examples of this benchmark. We used a 60s timeout for Spacer, Eldarica (with a RInGen preprocessing), and RInGen (with CVC4 finite model finding). The following table sums up the results for positive and negative examples.

	Our solver	[18]	Spacer	Eldarica	RInGen
Positive (143)	106	78	15	9	36
Negative (32)	32	29	4	31	31

Since RInGen is based on regular model inference [16], it succeeds on benchmarks having a regular model but fails on examples needing regular relations. Here are a few examples of properties that are automatically proven using our solver and out of the scope of [18,22] and aforementionned solvers.

1. $\forall(L_1, L_2 : \mathit{nat\ list}).\ \mathit{length}(L_1) \leq \mathit{length}(\mathit{append}(L_1, L_2))$ (link)
2. $\forall(T : \mathit{nat\ tree}).\ \mathit{height}(T) \leq \mathit{numnodes}(L)$ (link)
3. $\forall(T : \mathit{nat\ tree}).\ \mathit{heightRB}(T) \leq \mathit{height}(T)$ (link)
4. $\forall(T_1, T_2, T_3 : \mathit{nat\ tree}).\ \mathit{subtree}(T_1, T_2) \wedge \mathit{subtree}(T_2, T_3) \Rightarrow \mathit{subtree}(T_1, T_3)$ (link)
5. $\forall(L : \mathit{nat\ list}).\ \mathit{length}(L) = \mathit{length}(\mathit{insertionSort}(L))$ (link)

Although the property 1 is in the scope of convoluted tree automata, the solver of [18] fails to prove it on *nat* lists but succeeds on lists of $a's$ and $b's$. As mentioned in Section 4.2, models represented with SHoCs are generic. In the case of this property, the SHoCs proving the property for *nat* list and for lists of $a's$ and $b's$ are the same. The property 2 is out of the scope of [18] and also requires to take advantage of the genericity of SHoCs. The property 3 is the property ϕ_3 from Section 2. Interestingly, the solver builds an over-approximation of the *height* relation which is sufficient to prove ϕ_3. When functions are translated into relations, ϕ_3 becomes $\mathit{heightRB}(T, N) \wedge \mathit{height}(T, M) \Rightarrow \mathit{leq}(N, M)$, where *heightRB* and *height* occurs only on the left-hand side of the implication. Thus, using over-approximations for these relations for proving this property is safe. Properties 4 and 5 are other challenging properties. In particular, since *insertionSort* is defined using an intermediate function, proving this property using, e.g., automatic induction would require to guess and prove non-trivial intermediate lemmas. Finally, those experiments also reveal that the learner may find a correct SHoCs that our (incomplete) teacher may fail to accept. This can be observed in the log of the example `tree_height_max_node.smt2` where the last model proposed by the teacher for *height* is exactly the SHoCs of Section 4.1 but it is not accepted by the teacher. This property can be proven by splitting the function *height* into two parts (`tree_shallow_taller_node.smt2`). Finally, on examples coming from [13] where using a non-relational model suffices to prove the property, our solving technique succeeds except on some examples such as `timbuk_delete_not_member.smt2`.

7 Related work

Our solver automatically proves properties on programs manipulating ADTs. Proving the property on the program boils down to checking satisfiability of a set of CHCs. Satisfiability is shown by constructing a SHoCs recognizing the Herbrand model of the set of CHCs. This kind of technique is closely related to [13, 16, 20] where the model is regular and represented by a tree automaton. We have shown that SHoCs generalize tree automata and that, in practice, they succeed to perform the same proofs. Solvers of [13, 20] are complete w.r.t. regular models, i.e., if a regular model exists then it will eventually be found. Since model-checking of SHoCs is undecidable in general, the learner-teacher loop is not guaranteed to find a SHoCs. However, we have shown relative completeness of the loop. Besides, we conjecture that our algorithm for SHoCs solving should also be complete if we restrict SHoCs to recognize only regular sets.

Using SHoCs instead of (convoluted) tree automata significantly improves the efficiency of the solver. First, genericity of SHoCs allows to forget about information which is not relevant for the proof, e.g., forget about the values contained in a tree. This permits to carry out proofs which were out of grasp for [11, 13, 18] not because of expressivity but because of efficiency reasons. Second, expressivity of SHoCs permits to forget about the fixed convolution strategy of [18]. This allows our solver to cover many examples where there is no best convolution strategy, and thus no solution, for [18].

Among clause-based formalism used to represent languages, we mentioned *CS-programs* [17] but there are more recent extensions like *SCF-programs* [5]. We shown in Section 4.1 that expressivity of CS-programs and SHoCs overlap but none is included in the other. SCF-programs are more expressive that CS-programs and SHoCs presented here. In particular, SCF-programs can represent languages of the form $\{(g^n(a), g^n(b)) \mid n \in \mathbb{N}\}$ (like SHoCs) and also languages of the form $\{g^n(h(g^n(a))) \mid n \in \mathbb{N}\}$ (out of the scope of SHoCs). However, to the best of our knowledge, there exists no inference procedure for those formalisms such as the one presented in this paper which targets automatic program verification.

8 Conclusion and future work

This paper defines a class of Horn clauses, called Shallow Horn clauses or SHoCs, and shows that this class of Horn clauses is particularly useful for automated verification of relational properties of functional programs. SHoCs are strictly more expressive than both ordinary and convoluted tree automata, while at the same time offering a more compact representation compared to tree automata. They constitute a new element in the family of restricted Horn clauses, incomparable to other existing classes such as CS programs. We show how program verification using SHoCs amounts to satisfiability checking of a set of constrained clauses that encode the program and the property to verify. Satisfiability amounts to inferring a model which is done using a learner-teacher algorithm. Experiments show that the SHoCs-based approach leads to an efficient program verifier which can verify strictly more programs than previous tree automata-based algorithms.

Currently, the examples we experiment with are about trees and ADT-like structures. Representing *graphs* using SHoCs would be challenging but not impossible. Representing a graph using an adjacency list (a list of pair of nodes) would be of little help since the structure of the graph is not reflected in the list. However, if we fix the number of nodes in the graph, then one line of the adjacency-matrix may be represented as $(A_1, ..., A_n)$ with A_i a boolean representing adjacency of a given node with node i. This is a bit more structured and should be exploitable. The greatest fixed-point semantics can also be considered for SHoCs, thus allowing them to represent infinite trees.

Further work will consider how the framework can be extended to encompass both relations on algebraic data types and on numeric values, building upon [3, 15]. Secondly, it would be useful to add under-approximations for proving implicational properties where the right-hand side of the implication is a relation which cannot be modelled by a SHoCs, e.g., the *plus* relation. More generally, it would be useful to extend the approximation framework so as to consider the two kinds of approximations for a relation at the same time. For instance, this would permit to prove $plus(X, s(z), Y) \Rightarrow plus(s(z), X, Y)$ by proving $plus^+(X, s(z), Y) \Rightarrow plus^-(s(z), X, Y)$.

Finally, on a practical point of view, we would like to apply our solver on CHCs generated from functional programs. There exists tools to transform functional programs into CHCs and we would like to combine them with our solver to perform automatic verification. We believe that this could also be used to design new tactics in proof assistants for ADT-related properties.

Acknowledgements We would like to thank Benoît Montagu, Pierre Lermusiaux and the anonymous referees for all their comments and suggestions.

References

1. Barrett, C., Fontaine, P., Tinelli, C.: The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org` (2016)
2. Barrett, C., Shikanian, I., Tinelli, C.: An abstract decision procedure for a theory of inductive data types. Journal on Satisfiability, Boolean Modeling and Computation **3**(1-2), 21–46 (2007)
3. Bautista, S., Jensen, T., Montagu, B.: Lifting Numeric Relational Domains to Algebraic Data Types. In: SAS'22. LNCS, vol. 13790, pp. 104–134. Springer (2022)
4. Bjørner, N., Gurfinkel, A., McMillan, K., Rybalchenko, A.: Horn clause solvers for program verification. In: Fields of Logic and Computation II, pp. 24–51. Springer (2015)
5. Boichut, Y., Chabin, J., Réty, P.: Towards more precise rewriting approximations. J. Comput. Syst. Sci. **104**, 131–148 (2019)
6. Claessen, K., Johansson, M., Rosén, D., Smallbone, N.: Tip and isaplanner benchmarks (2015), `https://tip-org.github.io/`

7. Dixon, L., Fleuriot, J.: Isaplanner: A prototype proof planner in isabelle. In: CADE'03. vol. 2741, pp. 279–283. Springer (2003)
8. van Emden, M.H., Kowalski, R.A.: The semantics of predicate logic as a programming language. J. ACM **23**(4), 733–742 (1976)
9. Garg, P., Löding, C., Madhusudan, P., Neider, D.: Ice: A robust framework for learning invariants. In: International Conference on Computer Aided Verification. pp. 69–87. Springer (2014)
10. Gebser, M., Kaminski, R., Kaufmann, B., Schaub, T.: Answer Set Solving in Practice. Synthesis Lectures on Artificial Intelligence and Machine Learning, Morgan & Claypool Publishers (2012)
11. Genet, T., Haudebourg, T., Jensen, T.: Verifying Higher-Order Functions with Tree Automata. In: International Conference on Foundations of Software Science and Computation Structures. pp. 565–582. Springer (2018)
12. Gurfinkel, A.: Program Verification with Constrained Horn Clauses. In: CAV'22. LNCS, vol. 13371, pp. 19–29. Springer (2022)
13. Haudebourg, T., Genet, T., Jensen, T.: Regular Language Type Inference with Term Rewriting. Proceedings of the ACM on Programming Languages **4**(ICFP), 1–29 (2020)
14. Hojjat, H., Rümmer, P.: The ELDARICA horn solver. In: FMCAD'18. pp. 1–7. IEEE (2018), `https://github.com/uuverifiers/eldarica/`
15. Journault, M., Miné, A., Ouadjaout, A.: An Abstract Domain for Trees with Numeric Relations. In: ESOP'19. LNCS, vol. 11423, pp. 724–751. Springer (2019)
16. Kostyukov, Y., Mordvinov, D., Fedyukovich, G.: Beyond the elementary representations of program invariants over algebraic data types. In: Freund, S.N., Yahav, E. (eds.) PLDI'21. pp. 451–465. ACM (2021)
17. Limet, S., Salzer, G.: Tree tuple languages from the logic programming point of view. Journal of Automated Reasoning **37**, 323–349 (2006)
18. Losekoot, T., Genet, T., Jensen, T.: Automata-based Verification of Relational Properties of Functions over Data Structures. In: FSCD'23. vol. 260. LIPIcs (2023)
19. Losekoot, T., Genet, T., Jensen, T.: Verification of Programs with ADTs using Shallow Horn Clauses – extended version. Tech. rep., Univ Rennes, Inria, IRISA (2024), `https://hal.inria.fr/hal-04669706`
20. Matsumoto, Y., Kobayashi, N., Unno, H.: Automata-based abstraction for automated verification of higher-order tree-processing programs. In: Asian Symposium on Programming Languages and Systems. pp. 295–312. Springer (2015)
21. Regular Invariant Generator (2021), `https://github.com/Columpio/RInGen`
22. Shimoda, T., Kobayashi, N., Sakayori, K., Sato, R.: Symbolic automatic relations and their applications to SMT and CHC solving. In: International Static Analysis Symposium. pp. 405–428. Springer (2021)
23. Unno, H., Torii, S., Sakamoto, H.: Automating induction for solving horn clauses. In: International Conference on Computer Aided Verification. pp. 571–591. Springer (2017)