Automata-based verification of relational properties of functions over data structures

Théo Losekoot
Université de Rennes, IRISA, France

Thomas Genet
Université de Rennes, IRISA, France

Thomas Jensen
Inria, Université de Rennes, France

Abstract

This paper is concerned with automatically proving properties about the input-output relation of functional programs operating over algebraic data types. Recent results show how to approximate the image of a functional program using a regular tree language. Though expressive, those techniques cannot prove properties relating the input and the output of a function, e.g., proving that the output of a function reversing a list has the same length as the input list. In this paper, we built upon those results and define a procedure to compute or over-approximate such a relation. Instead of representing the image of a function by a regular set of terms, we represent (an approximation of) the input-output relation by a regular set of tuples of terms. Regular languages of tuples of terms are recognized using a tree automaton recognizing convolutions of terms, where a convolution transforms a tuple of terms into a term built on tuples of symbols. Both the program and the properties are transformed into predicates and Constrained Horn clauses (CHCs). Then, using an Implication Counter Example procedure (ICE), we infer a model of the clauses, associating to each predicate a regular relation. In this ICE procedure, checking if a given model satisfies the clauses is undecidable in general. We overcome undecidability by proposing an incomplete but sound inference procedure for such relational regular properties. Though the procedure is incomplete, its implementation performs well on 120 examples. It efficiently proves non-trivial relational properties or finds counter-examples.

2012 ACM Subject Classification Theory of computation → Program verification; Theory of computation → Formal languages and automata theory

Keywords and phrases Formal verification, Tree automata, Constrained Horn Clauses, Model inference, Relational properties, Algebraic datatypes

Digital Object Identifier 10.4230/LIPIcs.FSCD.2023.3

1 Introduction

This paper is concerned with automatically proving properties about the input-output relation of functional programs operating over algebraic datatypes. We explore an approach in which both programs and properties are represented as Constrained Horn Clauses [2], i.e., Horn clauses with additional constraints expressed in an underlying theory. Using such representation, proving a property of a program is reduced to finding a model of the combined set of Horn clauses that represent the program and the property. We illustrate this using an example where we define the type of natural numbers and natural numbers lists, and two recursive functions, *len* computing the length of a list and *less* checking if a natural number is strictly less than another. We aim at (automatically) proving the logical properties $\forall x\ l.\ less\ Z\ (len\ Cons(x,l))$ and $\forall x\ l.\ less\ (len\ l)\ (len\ Cons(x,l))$. Here are the program in Ocaml-like syntax, the logical formulas for properties and their equivalent CHC representation. Note that n-ary functions (like unary *len*) are translated into $n+1$-ary

8th International Conference on Formal Structures for Computation and Deduction (FSCD 2023).
Editors: Marco Gaboardi and Femke van Raamsdonk; Article No. 3; pp. 3:1–3:21
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

relations (like binary Len). Because of this extra argument, we add a functionality constraint (the third clause of Len) for ensuring that the relation represents exactly the function. Without this functionality constraint, we could e.g. have a model where $\text{Len}(Nil, S(Z))$ is true. Arity of predicates, like the binary *less*, do not change: Less is binary. In this case, we cannot use functionality constraint because the result is not reified. Instead, we use bi-implication to exclude all elements which are not in the relation defined by the OCaml function, e.g., exclude $\text{Less}(S(S(Z)), S(Z))$.

```
type nat = Z | S of nat
type natlist = Nil | Cons of nat*natlist
```

```
let rec len (l : natlist) =
match l with
| Nil -> Z
| Cons(h,t) -> S (len t)
```

$\text{Len}(Nil,\ Z).$
$\text{Len}(\underline{l},\ \underline{n}) \Rightarrow \text{Len}(Cons(\underline{x},\underline{l}),\ S(\underline{n})).$
$\text{Len}(\underline{l},\ \underline{n_1}) \wedge \text{Len}(\underline{l},\ \underline{n_2}) \Rightarrow \underline{n_1} = \underline{n_2}.$

```
let rec less (n : nat) (m : nat) =
match (n, m) with
| Z, S(_) -> true
| _, Z -> false
| S(n1), S(m1) -> less n1 m1
```

$\text{Less}(Z,\ S(\underline{m})).$
$\text{Less}(\underline{n},\ Z) \Rightarrow \text{False}.$
$\text{Less}(\underline{n},\ \underline{m}) \iff \text{Less}(S(\underline{n}),\ S(\underline{m})).$

$\forall x\ l.\ less\ Z\ (len\ (Cons(x,l)))$ | $\text{Len}(Cons(\underline{x},\underline{l}),\ \underline{n}) \Rightarrow \text{Less}(Z,\ \underline{n}).$
$\forall x\ l.\ less\ (len\ l)\ (len\ Cons(x,l))$ | $\text{Len}(\underline{l},\ \underline{n}) \wedge \text{Len}(Cons(\underline{x},\underline{l}),\ \underline{n'}) \Rightarrow \text{Less}(\underline{n},\ \underline{n'}).$

Our goal is thus to automatically infer a model of this set of clauses, i.e., solve the satisfiability problem for Constrained Horn Clauses over the theory of inductive datatypes. Tree automata [6] are a well-know formalism to represent, approximate, and infer models on functional programs [11,17] or even on CHCs [16]. In all those works, the inferred model is not relational, i.e., it only consists of a regular set of unrelated terms. For instance, in our example, the first property $\forall x\ l.\ less\ Z\ (len\ (Cons(x,l)))$ is not relational and can thus be proven using regular sets like [11,16,17] do. To perform the proof, the solvers only need to consider two regular languages: $\mathcal{L}_{lists}$ containing all lists of natural numbers and $\mathcal{L}_{Cons+}$ containing all *non-empty* lists of natural numbers. Then, the proof is carried out by showing that if $l \in \mathcal{L}_{lists}$ then, for any natural number x, the term $Cons(x,l)$ belongs to $\mathcal{L}_{Cons+}$. Finally, since any list $l' \in \mathcal{L}_{Cons+}$ have a length strictly greater than 0 then the property is true.

On the opposite, the second property, $\forall x\ l.\ less\ (len\ l)\ (len\ Cons(x,l))$, is relational and, thus, out of the scope of the aforementioned approaches. We still have that if $l \in \mathcal{L}_{lists}$ then $cons(x,l) \in \mathcal{L}_{Cons+}$ but for any $l \in \mathcal{L}_{lists}$ and any $l' \in \mathcal{L}_{Cons+}$ we cannot prove that $less\ (len\ l)\ (len\ l')$. To preserve the relation between the two occurrences of the list l, we use convoluted automata [6] which can represent *regular relations* between terms. We build upon the preliminary results obtained in [12] and propose a sound but incomplete procedure for inferring an automaton that represents a model of the program and the property. This procedure is defined as an Implication Counter Example (ICE) procedure [8].

Contributions:

- Definition of a sound model-checking procedure for CHCs on convoluted tree automata. We propose two sound optimisations of this procedure so as to make it efficient in practice;
- Definition of an ICE procedure for inferring models of CHCs;
- Definition of a specific over-approximation technique enlarging the class of properties which can be proved using regular models on CHCs programs;

- Implementation of the ICE procedure;
- On more than 120 examples, we show that our implementation automatically proves and disproves non-trivial examples.

This paper is organised as follows: In Section 2, we give an overview demonstrating the verification technique presented in this paper. In Section 3, we introduce the notions and notations. In Section 4, we briefly present how to encode functional programs into Horn clauses. In Section 5, we present a transformation from the model-checking procedure for CHCs into a search for a proof in a *proof system* representing the model. In Section 6, we present our use of the proof system for an efficient search. In Section 7, the ICE-procedure for inferring a model is defined. In Section 8, we present our approximation method. In Section 9, we discuss implementation-specific details and experiments. In Section 10, we present related work. Finally, we conclude in Section 11.

2 An overview of the verification procedure on an example

We continue our example of Section 1. We first give more details about the proof of the non-relational property $\forall x\ l.\ less\ Z\ (len\ (Cons(x,l)))$. To represent the set $\mathcal{L}_{lists}$ containing all lists of natural numbers and the set $\mathcal{L}_{Cons+}$ containing all non-empty lists of natural numbers, we use tree automata. Tree automata recognize sets of terms into states using *transitions*. *E.g.*, a tree automaton with states $\{q_{nat}, q_{Nil}, q_{Cons+}\}$ and transitions $\{Z() \to q_{nat},\ S(q_{nat}) \to q_{nat},\ Nil() \to q_{Nil},\ Cons(q_{nat}, q_{Nil}) \to q_{Cons+},\ Cons(q_{nat}, q_{Cons+}) \to q_{Cons+}\}$ recognizes Nil into the state q_{Nil} and any non-empty list of naturals into the state q_{Cons+}. To recognize a term, transitions are used to rewrite the term into a state, e.g, $Nil \to q_{Nil}$, and $Cons(S(Z), Nil) \to^* Cons(S(q_{nat}), q_{Nil}) \to Cons(q_{nat}, q_{Nil}) \to q_{Cons+}$. Similarly $Cons(Z, Cons(S(S(Z)), Nil)) \to^* q_{Cons+}$. To prove the property $\forall x\ l. less\ Z\ (len\ (Cons(x,l)))$ using such an automaton, it is enough to show that if l belongs to $\mathcal{L}_{lists}$ (whose terms are recognized by q_{Nil} or q_{Cons+}), then $Cons(x,l)$ belongs to $\mathcal{L}_{Cons+}$ (whose terms are recognized by q_{Cons+}). Using another automaton for Less, it is possible to show that $(len\ l')$, with l' recognized by q_{Cons+}, belongs to the language $\mathcal{L}_{pos}$ of strictly positive natural numbers, whereas $(len\ Nil)$ belongs to the language $\{Z\}$.

Now, we present a complete overview of our verification procedure for proving the second property $\forall x\ l.\ less\ (len\ l)\ (len\ Cons(x,l))$ which is relational and, thus, out of the scope of solvers like [11, 16, 17]. As shown before, the functions and the property are all translated into a set of CHCs. In the following, we denote by $\mathcal{C}$ this set. Given $\mathcal{C}$, we start the *model inference* phase whose objective is to infer a model of this set, named $\mathcal{M}$ in the following. For each relation R defined by the program, $\mathcal{M}$ contains an automaton $\mathcal{A}_R$ recognizing a language for the relation R. The model inference procedure can either

(i) succeed, i.e. find a model $\mathcal{M}$ satisfying $\mathcal{C}$, and the properties are proved, or
(ii) fail, i.e. find a contradiction, and the properties are disproved, or
(iii) never terminates.

This model inference is implemented as an Implication Counter-Example (ICE) procedure [8] between two entities: a learner and a teacher. The learner's goal is to infer a correct model using only feedback from the teacher. The teacher's goal is to verify if the clauses from $\mathcal{C}$ satisfy $\mathcal{M}$ (the model proposed by the learner) and to give feedback in the form of logical implications which are counter-examples.

Initially, $\mathcal{M}$ associates to each relation symbol an empty relation recognized by an empty automaton, denoted by $\mathcal{A}_\emptyset$. The relation recognized by $\mathcal{A}_\emptyset$, denoted by $\mathcal{R}(\mathcal{A}_\emptyset)$, is the empty relation. On our example, the initial value for $\mathcal{M}$ is thus $\mathcal{M} = \{\mathrm{Len} \mapsto \mathcal{A}_\emptyset, \mathrm{Less} \mapsto \mathcal{A}_\emptyset\}$.

First iteration of the learner-teacher algorithm

The learner proposes the model $\mathcal{M} = \{\text{Len} \mapsto \mathcal{A}_\emptyset, \text{Less} \mapsto \mathcal{A}_\emptyset\}$. The teacher checks if $\mathcal{M}$ satisfies each clause of $\mathcal{C}$, i.e., for each $\varphi \in \mathcal{C}$ it checks if $\mathcal{M} \models \varphi$. This is not true for the clause $\text{Len}(Nil,\ Z)$ which imposes that the pair (Nil, Z) is part of the relation associated with Len. This is not the case here. Thus, the learner provides the ground clause $\text{Len}(Nil,\ Z)$ as a counter-example.

Second iteration of the learner-teacher algorithm

Starting from $\mathcal{M} = \{\text{Len} \mapsto \mathcal{A}_\emptyset, \text{Less} \mapsto \mathcal{A}_\emptyset\}$ and the counter-example $\text{Len}(Nil,\ Z)$, the learner improves $\mathcal{M}$ in order to add the pair (Nil, Z) into the relation associated with Len, i.e., refines the automaton so as to recognize the pair (Nil, Z). For recognizing a relation, we need to extend the tree automaton formalism to recognize regular sets of tuples of terms. A solution proposed in [6] is to use a tree automaton recognizing convolutions of terms. A convolution transforms a tuple of terms into a term built on tuples of symbols. It does so by introducing new *convoluted* symbols which represent tuples of symbols. For example, to recognize the pair (Nil, Z) we define a new symbol $\langle Nil, Z\rangle$ and a tree automaton $\mathcal{A}_1$ with the state q_0 and the unique transition $\langle Nil, Z\rangle() \to q_0$. With such an automaton, the relation recognized by automaton $\mathcal{A}_1$ is $\mathcal{R}(\mathcal{A}_1) = \{(Nil, Z)\}$. Finally, we now have $\mathcal{M} = \{\text{Len} \mapsto \mathcal{A}_1, \text{Less} \mapsto \mathcal{A}_\emptyset\}$. Again, this model is given to the teacher which checks if $\mathcal{M} \models \mathcal{C}$. The teacher finds out that $\mathcal{M} \not\models \text{Len}(\underline{l},\ \underline{n}) \Rightarrow \text{Len}(Cons(\underline{x}, \underline{l}),\ S(\underline{n}))$. Indeed, since $(Nil, Z) \in \mathcal{L}(\mathcal{A}_1)$ we should have $(Cons(i, Nil), S(Z)) \in \mathcal{L}(\mathcal{A}_1)$ for all natural numbers i. The teacher provides a ground instance of this clause as a counter-example, e.g., $\text{Len}(Nil,\ Z) \Rightarrow \text{Len}(Cons(Z, Nil),\ S(Z))$.

Third iteration of the learner-teacher algorithm: Learner part

Starting from $\mathcal{M} = \{\text{Len} \mapsto \mathcal{A}_1, \text{Less} \mapsto \mathcal{A}_\emptyset\}$ and the counter-example obtained from the previous iteration $\text{Len}(Nil,\ Z) \Rightarrow \text{Len}(Cons(Z, Nil),\ S(Z))$, the learner should refine $\mathcal{A}_1$ into $\mathcal{A}_2$ so that it also recognizes the pair $(Cons(Z, Nil), S(Z))$. This time, to build the convolution we have to overlay the terms $Cons(Z, Nil)$ and $S(Z)$. However, because of the different arities of $Cons$ and S, the trees representing those two terms do not perfectly overlap. The convolution adds a padding symbol $\square$ to complement trees in order to have a perfect overlap. Back to our example, with a convolution (known as right-convolution) the tree for $S(Z)$ becomes

$\underset{\square \quad Z}{S}$ and the convolution of $\underset{Z \quad Nil}{Cons}$ and $\underset{\square \quad Z}{S}$ is $\underset{\langle Z, \square\rangle \, \langle Nil, Z\rangle}{\langle Cons, S\rangle}$.

Thus, a refined automaton $\mathcal{A}_2$ recognizing both (Nil, Z) and $(Cons(Z, Nil), S(Z))$ has states $\{q_0, q_1, q_2\}$ and transitions $\{\langle Nil, Z\rangle() \to q_0,\ \langle Z, \square\rangle() \to q_1,\ \langle Cons, S\rangle(q_1, q_0) \to q_2\}$. If we declare states q_0 and q_2 as final (meaning that we ignore the languages recognized by non final states) then $\mathcal{R}(\mathcal{A}_2) = \{(Nil, Z), (Cons(Z, Nil), S(Z))\}$.

A last phase of the ICE learning process is to reduce the number of states of the automaton and, doing so, possibly enlarge the recognized language. Note that this phase was skipped on automaton $\mathcal{A}_1$ because it has only one state. Reducing the number of states consists in finding state merging which are coherent w.r.t. the ground clauses sent by the teacher and coherent w.r.t. types of recognized languages. For instance, on $\mathcal{A}_2$, merging q_0 with q_2 is possible because both recognize pairs of lists and natural numbers. On the opposite, merging q_0 with q_1 is incorrect because q_0 recognize *pairs* of lists and q_1 only recognizes *a unique* natural number (omitting padding). After renaming q_2 to q_0, transitions of the automaton $\mathcal{A}_2$ become $\{\langle Nil, Z\rangle() \to q_0,\ \langle Z, \square\rangle() \to q_1,\ \langle Cons, S\rangle(q_1, q_0) \to q_0\}$. Note that this automaton now recognizes $\{(Nil, Z),\ (Cons(Z, Nil), S(Z)),\ (Cons(Z, Cons(Z, Nil)), S(S(Z))),\ \ldots\}$, i.e., all pairs (l, n) where l is a list of Z whose length is n.

Conclusion of the learner-teacher algorithm

During following iterations, the learner-teacher proceed similarly to infer an automaton for Less and to finish inferring that of Len. Finally, during the 6-th iteration, the learner ends up on the following model $\mathcal{M} = \{\text{Len} \mapsto \mathcal{A}_{\text{Len}}, \text{Less} \mapsto \mathcal{A}_{\text{Less}}\}$ where $\mathcal{A}_{\text{Len}}$ has final states $\{q_0\}$ and the transitions $\{\langle \square, S\rangle(q_1) \to q_1,\ \langle \square, Z\rangle() \to q_1,\ \langle Nil, Z\rangle() \to q_0,\ \langle Cons, S\rangle(q_1, q_0) \to q_0\}$. This automaton is close to automaton $\mathcal{A}_2$ except that it recognizes any natural number in place of Z in the list, i.e., it recognizes all pairs (l, n) where l is a list of natural numbers whose length is n. The automaton $\mathcal{A}_{\text{Less}}$ has the final states $\{q_3\}$ and the transitions $\{\langle \square, Z\rangle() \to q_4,\ \langle \square, S\rangle(q_4) \to q_4,\ \langle Z, S\rangle(q_4) \to q_3,\ \langle S, S\rangle(q_3) \to q_3\}$. This model is given to the teacher which then checks that it satisfies all the clauses of $\mathcal{C}$. This terminates the verification and proves that $\forall x\ l.\ less\ (len\ l)\ (len\ Cons(x, l))$.

3 Prerequisites

3.1 Typed alphabet and term

▶ **Definition 1** (Typed alphabet). *A typed alphabet (Σ, τ, Γ) is a set of symbols Σ, a set of types Γ, and a typing function τ which assigns to each symbol f a type $\tau(f) = \tau_1 \times \ldots \times \tau_n \to \tau_0$ with $\forall i \in [\![0, n]\!], \tau_i \in \Gamma$ and $n \in \mathbb{N}$ varying for each symbol f. When $n = 0$, the symbol is a constant and does not take input. For $f \in \Sigma$ and $\tau(f) = \tau_1 \times \ldots \times \tau_n \to \tau_0$, we say that f is of arity n, written $|f| = n$, and that τ_0 is the* output type *of f, written $\tau_{out}(f) = \tau_0$. When clear from context, we identify the tuple (Σ, τ, Γ) with Σ.*

▶ **Definition 2** (Term). *A (typed) term t over an alphabet Σ is the data of a symbol $f \in \Sigma$, called the* root symbol *of t and written $Root(t)$, together with a list $t_1, \ldots, t_{|f|}$ of $|f|$ terms, called* children *of t, such that their type is compatible, i.e. $\tau(f) = \tau_{out}(Root(t_1)) \times \ldots \times \tau_{out}(Root(t_{|f|})) \to \tau_{out}(f)$. A term t is also written $f(t_1, \ldots, t_{|f|})$. We overload τ with $\tau(t) = \tau_{out}(Root(t))$. The set of terms over an alphabet Σ is written $\mathcal{T}(\Sigma)$.*

▶ **Definition 3** (Substitution). *A substitution σ is a finite map between variables and terms (which may contain variables). The application of a substitution σ to a variable x, written $\sigma(x)$, is defined as t if there exists a binding $(x, t) \in \sigma$ and x otherwise. The application of a substitution is generalized to terms by $\sigma(f(t_1, \ldots, t_n)) = f(\sigma(t_1), \ldots, \sigma(t_n))$. Even more generally, a substitution can be applied to any structure containing variables. The composition of substitution, which first applies σ_1 and then σ_2, is written $\sigma_1; \sigma_2$. The domain of a substitution is the set of variables for which a binding is defined and is written $dom(\sigma)$.*

A function *Vars* is used without definition, if unambiguous, to fetch the set of variables contained in a structure. It can be called, for example, on a term or on a tuple of structures containing variables.

3.2 Tree automaton

▶ **Definition 4** (Tree automaton). *A (bottom-up) tree automaton $\mathcal{A} = (Q, Q_f, \Delta)$ over an alphabet Σ is given by a finite set of states Q, a set of final states $Q_f \subseteq Q$, and a set of transitions (or rules) Δ*

such that transitions are of the form $f(q_1, \ldots, q_{|f|}) \rightarrow q_0$*, where* $f \in \Sigma$ *and* $\forall i \in [\![0, |f|]\!], q_i \in Q$.

▶ **Definition 5** (Language recognized by an automaton)**.** *The set of terms recognized (or accepted) in a state* q *of an automaton* $\mathcal{A}$ *is inductively defined as* $\mathcal{L}(\mathcal{A}, q) = \{f(t_1, \ldots, t_n) \mid f(q_1, \ldots, q_n) \rightarrow q \in \Delta \wedge \bigwedge_{i \in [\![1,n]\!]} t_i \in \mathcal{L}(\mathcal{A}, q_i)\}$*. The language recognized by an automaton is* $\mathcal{L}(\mathcal{A}) = \bigcup_{q_f \in Q_f} \mathcal{L}(\mathcal{A}, q_f)$.

▶ **Definition 6** (Typed tree automaton)**.** *A typed tree automaton is a tree automaton whose states are typed by types of the alphabet. We write* $\tau(q)$ *for the type of the state* q*. Transitions have to be compatible with the types of the symbols, i.e., for any rule* $f(q_1, \ldots, q_n) \rightarrow q_0 \in \Delta$, $\tau(f) = \tau(q_1) \times \ldots \times \tau(q_n) \rightarrow \tau(q_0)$*. All final states must be of the same type. The type of the automaton, written* $\tau(\mathcal{A})$*, is the type of its final states.*

We write $\overline{\mathcal{A}}$ for the complement of the automaton $\mathcal{A}$ w.r.t its type, i.e., $\mathcal{L}(\overline{\mathcal{A}}) = \{t \mid \tau(t) = \tau(\mathcal{A}) \wedge t \notin \mathcal{L}(\mathcal{A})\}$. We also use Q, Q_f, and Δ as accessors, that is, as functions to respectively extract states, final states, and transitions from an automaton. We usually write t or $f(t_1, \ldots, t_n)$ for terms, q for a state, and $\mathcal{A}$ for an automaton. Tuple of elements $(e_1, \ldots, e_n)$ are also written $\vec{e}$ and $\vec{e}[i]$ means e_i.

3.3 Automata recognizing a relation

There exist multiple formalism for representing a relation on terms with an automaton. They differ in their expressive power, closure properties, and decision procedure complexity. The most well known are *tuple automata*, *ground tree transducers*, and *automata on convoluted terms*, all described in [6]. We will pursue an approach based on automata on convoluted terms, or simply convoluted automata.

Convoluted automata are defined w.r.t an operation called *convolution* which transforms an n-tuple of terms into a unique term whose symbols are n-tuple of symbols. Intuitively, an automaton defined on this alphabet of tuple reads n terms at the same time, thereby recognizing a relation. The standard convolution operator amounts to overlaying the (syntax tree of the) terms, starting from the root, and adding a padding symbol $\square \notin \Sigma$ (of type $\tau_\square$) as there is an arity mismatch between symbols. To this end, we extend any alphabet Σ to $\Sigma_\square = \Sigma \cup \{\square\}$. We call this standard convolution the *left convolution*, in order to distinguish it from other convolutions, e.g. the right convolution, that has been used in section 2 and in the rest of the paper. We first define left-convolution of a tuple of tuple, and then use it to define convolution of terms.

▶ **Definition 7** (Left-convolution)**.**

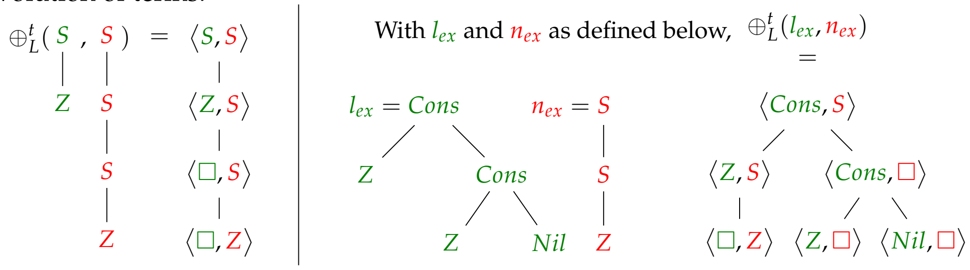
$$\oplus_L \left((e_1^1, \ldots, e_1^{k_1}), \ldots, (e_n^1, \ldots, e_n^{k_n})\right) = \left((\overline{e_1^1}, \ldots, \overline{e_n^1}), \ldots, (\overline{e_1^k}, \ldots, \overline{e_n^k})\right)$$

$$\textit{with } k = \max_{i \in [\![1,n]\!]}(k_i) \quad \textit{and} \quad \forall i \in [\![1,n]\!], \forall j \in [\![1,k]\!],\ \overline{e_i^j} = e_i^j \textit{ if } j \leq k_i \textit{ and } \square \textit{ otherwise}$$

▶ **Definition 8** (Left-convolution of terms)**.** *The* n*-ary* left-convolution*, written* $\oplus_L^t$*, takes* n *terms* $(t_1, \ldots, t_n)$ *on an alphabet* $\Sigma_\square$ *and returns a term* $\oplus_L^t(t_1, \ldots, t_n)$ *on a convoluted alphabet* $\Sigma_{\oplus_L} = {\Sigma_\square}^n$ *whose elements are written* $\langle f_1, \ldots, f_n \rangle$ *or* $\vec{f}$*. The left-convolution of* n *terms is recursively defined as:*

$$\oplus_L^t(f_1(\vec{t}_1), \ldots, f_n(\vec{t}_n)) = \langle f_1, \ldots, f_n \rangle(\oplus_L^t(\vec{t'_1}), \ldots, \oplus_L^t(\vec{t'_k})) \textit{ with } (\vec{t'_1}, \ldots, \vec{t'_k}) = \oplus_L(\vec{t}_1, \ldots, \vec{t}_n)$$

▶ **Example 9** (Left convoluted terms). Let $\Sigma_{ex} = \{Z, S, Nil, Cons\}$, with $\tau(Z) = nat$, $\tau(S) = nat \to nat$, $\tau(Nil) = natlist$, $\tau(Cons) = nat \times natlist \to natlist$, be a typed alphabet for natural numbers and lists of natural numbers. Following are two examples of left convolution of terms.

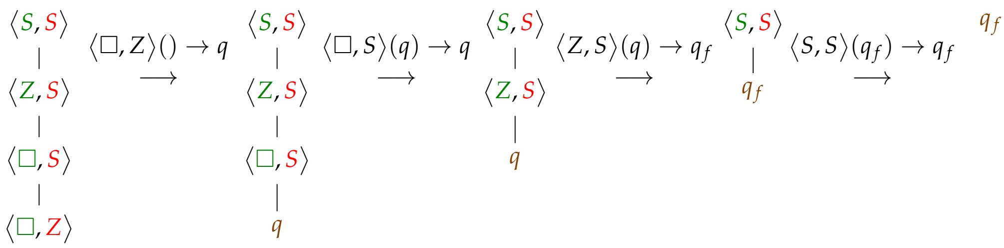


Note that, due to type constraints, $\mathcal{T}(\Sigma_\square) = \mathcal{T}(\Sigma) \cup \{\square\}$. The left-convolution $\oplus_L^t$ of n terms is an isomorphism between $\mathcal{T}(\Sigma_\square)^n$ and $\mathcal{T}(\Sigma_{\oplus_L})$. Automata recognizing convoluted terms thus recognize relations on $\mathcal{T}(\Sigma_\square)^n$.

▶ **Definition 10** (Regular relation). *A relation recognized by a tree automaton is said to be* regular*. The relation recognized by automaton $\mathcal{A}$ is $\mathcal{R}(\mathcal{A}) = {\oplus_L}^{-1}(\mathcal{L}(\mathcal{A})) = \{\vec{t} \mid \oplus_L(\vec{t}) \in \mathcal{L}(\mathcal{A})\}$. Similarly, the relation recognized by state q of $\mathcal{A}$ is $\mathcal{R}(\mathcal{A}, q) = {\oplus_L}^{-1}(\mathcal{L}(\mathcal{A}, q))$.*

We impose that the type of any final state q_f is $\tau_\square$-free, that is, $\tau(q_f) = (\tau_1, \dots, \tau_n)$ with $\forall i \in [\![i, n]\!], \tau_i \neq \tau_\square$. This ensures that an automaton defines a relation between terms of $\mathcal{T}(\Sigma)$, i.e. terms without padding.

▶ **Example 11** (Convoluted automata). Let $\mathcal{A}_<$ be the automaton with states $\{q, q_f\}$, of which q_f is final, and transitions $\{\langle\square, Z\rangle() \to q,\ \langle\square, S\rangle(q) \to q,\ \langle Z, S\rangle(q) \to q_f,\ \langle S, S\rangle(q_f) \to q_f\}$. $\mathcal{R}(\mathcal{A}_<)$ is the $<$ relation on Peano numbers and $\tau(\mathcal{A}_<) = nat \times nat$. For example, the convolution of $S(Z)$ and $S(S(S(Z)))$ is recognized by this automaton, as shown below.

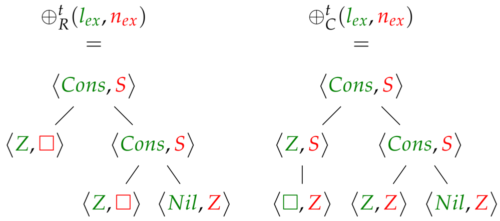


Convolutions and their expressivity

Which relations are representable by convoluted tree automaton highly depends on the precise datatypes definition. For example, when using the left-convolution, the Len relation can only be represented if the Cons constructor had its arguments swapped. This is because left-convoluting a list l and a natural number n will relate n with the left-most branch of l. Instead of modifying constructors, we can define other convolutions. The *right convolution*, written $\oplus_R$, is defined similarly to $\oplus_L$ but adds padding to the left of terms instead of to the right. This right convolution is effective for proving properties relating lists and unary natural numbers. Finally, we define the *complete convolution*, written $\oplus_C$, which is more expressive than both the left and the right convolution. This complete convolution relates every combination of tuple's element, which results in overlaying every same-depth constructor when convoluting terms. The complete convolution has the advantage of not depending on the constructor argument's order and being able to duplicate terms, but the drawback of generating big convoluted terms. Both convolution are extended to terms in the same way $\oplus_L$ was.

▶ **Example 12.**

On the left is depicted the *right* convolution of l_{ex} and n_{ex} (of example 11), and on the right their *complete* convolution. Note how n_{ex}'s constructors have been duplicated in the complete convolution.



Since definitions of this paper hold for any convolution, we write $\bigcirc$ for any of $\oplus_L$, $\oplus_R$, or $\oplus_C$.

4 Functional programs and their logical representation

Regular models of functional programs

We consider first-order monomorphic functional programs. Such programs define a set of functions of the form $f : \tau_1 \to \ldots \to \tau_n$ and of the form $f : \tau_1 \to \ldots \to \tau_n \to bool$, with each τ_i being an algebraic datatype. Each of these can be viewed as a relation on $\tau_1 \times \ldots \times \tau_n$. Formally, these relations constitute a (relational) first-order structure on L, with L being the signature (the set of relation symbols together with their type). In our setting, the structures are typed, i.e. a relation R of type $\tau(R) = \tau_1 \times \ldots \times \tau_n$ only relates terms $t_1, \ldots, t_n$ satisfying $\forall i \in [\![1, n]\!], \tau(t_i) = \tau_i$.

▶ **Definition 13** (Regular model). *A* regular model *is a function $\mathcal{M}$ mapping each relation symbol $R \in L$ to an automaton $\mathcal{A}_R$. $\mathcal{M}$ denotes $\mathcal{S}_\mathcal{M}$, the L-structure where every $R \in L$ is interpreted as $\mathcal{R}(\mathcal{A}_R)$. We naturally extend first-order semantic judgement to write $\mathcal{M} \models \varphi$ for $\mathcal{S}_\mathcal{M} \models \varphi$.*

Regular models are close in essence to *automatic structures*. Automatic structures [10, 14, 15] are a kind of recursive structures [13], which are part of the study of finite representation of structures. Automatic structures have been studied for their decidable first-order theory. We shall use *tree automata* to represent first-order structures that model functional programs. This allows us to use specific and efficient methods for property checking.

We use Constrained Horn Clauses (CHCs) [2] as representation of our programs. CHCs are first-order Horn clauses with additional constraints from a theory T (see example in the Introduction). A CHC on a signature L is a closed formula of the form $\forall \vec{x}, \psi(\vec{x}) \wedge R_1(\vec{x}_1) \wedge \ldots \wedge R_n(\vec{x}_n) \Rightarrow R_0(\vec{x}_0)$, where $\forall i \in [\![0, n]\!], R_i \in L$. The formula $\psi(\vec{x})$ adds theory-related constraints. The semantic judgement $\mathcal{S} \models \varphi$ is standard first-order logic (modulo theory T). We usually leave out the universal quantifiers in front of CHCs: every variable in a formula is implicitly universally quantified. In our setting, we use the theory of inductive datatypes [1] over an alphabet Σ, which means that the value of variables are within $\mathcal{T}(\Sigma)$ and constraints are of the form $x = f(\vec{y})$, where $f \in \Sigma$, x is a variable and $\vec{y}$ is a tuple of variables. For simplicity, we sometimes write $R(t)$ for $x = t \wedge R(x)$. A *ground* CHC is one that has no variables or, in our context, where every variable's value is completely determined by datatypes constraints (for example, $x = Nil \Rightarrow R(x)$ is considered ground).

Our encoding of functional programs into clauses prevents us from using Horn clauses in the translation of the if-then-else construct. For example, the simple translation of `let f x = if p x then e else e'` yields the two clauses $\{\mathrm{P}(x) \Rightarrow \mathrm{F}(x, e), \; \neg\mathrm{P}(x) \Rightarrow \mathrm{F}(x, e')\}$. We therefore use non-Horn constrained clauses for modeling such functions. In the following, we handle a negated literal in the body as a positive head, in disjunction with the other heads. Other work [20] models similar programs with Horn clauses by reifying the truth of a predicate in the terms as its last argument, allowing to negate it in the body of a clause. Both ways of treating negation seems viable for our purpose but we have only experimented with the first one.

5 Model-checking of regular structures

In this section, we present the procedure for checking the truth of a given CHC φ in a model $\mathcal{M}$, i.e., check if $\mathcal{M} \models \varphi$. This model-checking fulfills the *teacher* role of the ICE model inference procedure (See sections 2 and 7). This procedure is devised as a counter-example search. A counter-example is a ground instantiation of each variable of φ, written as a ground substitution σ, that disproves $\mathcal{M} \models \varphi$. This procedure either returns *None* if $\mathcal{M} \models \varphi$, and otherwise $Some(\sigma)$, with σ a counter-example. However, this problem is undecidable in general, as showed in [18]. Therefore the procedure given here is correct but incomplete, that is, it may diverge.

The model checking problem can be seen as a type checking procedure where typing rules correspond to rules of automata.

▶ **Definition 14** (Type checking instance). *A* typing obligation $\omega = [\langle x_1, \ldots, x_n \rangle : (\mathcal{A}, q)]$ *is the data of a tuple* $\langle x_1, \ldots, x_n \rangle$*, with each* x_i *being a variable or* $\square$*, and of a* target type $(\mathcal{A}, q)$*. A* typing problem (E, Ω) *is a set of typing obligations* Ω *together with a set of constraints* E*, each of the form* $x = f(\vec{y})$ *with* f *a symbol of* Σ*. A solution for a typing problem is a substitution* $\sigma : \mathcal{X} \to \mathcal{T}(\Sigma)$ *that satisfies every typing obligation and constraint:*

$$\sigma \models (E, \Omega) \;\doteq\; \sigma \models \Omega \;\wedge\; \sigma \models E \qquad \textit{with}$$

$$\sigma \models \Omega \;\doteq\; \big(\forall [\vec{x} : (\mathcal{A}, q)] \in \Omega,\; \sigma(\vec{x}) \in \mathcal{R}(\mathcal{A}, q)\big) \;\; \textit{and}$$

$$\sigma \models E \;\doteq\; \big(\forall (x = f(\vec{y})) \in E,\; \sigma(x) = f(\sigma(\vec{y}))\big)$$

▶ **Definition 15** (Coherence of a constraint set). *A set of constraints* E *is said to be* coherent *if it admits a syntactic unifier. The most general unifier (MGU) of a coherent set* E *is written* σ_E*.*

Note that, given a typing problem (E, Ω) with a coherent E, any σ such that $\sigma \models (E, \Omega)$ is equivalent to a σ' such that $\sigma_E; \sigma' \models \Omega$ (by characterisation of the MGU).

▶ **Definition 16** (Model checking as type checking).
Let some CHC formula $\varphi = \psi(\vec{x}) \wedge R_1(\vec{x}_1) \wedge \ldots \wedge R_n(\vec{x}_n) \Rightarrow R_0(\vec{x}_0)$ *and model* $\mathcal{M}$*.*
The set of typing problems associated to φ *and* $\mathcal{M}$ *is* $tp(\varphi, \mathcal{M}) = \{(\psi(\vec{x}), \Omega) \mid \Omega \in \Omega_s\}$ *with*

$$\Omega_s = \Big\{ \{[\vec{x}_1 : (\mathcal{A}_1, q_1)], \ldots, [\vec{x}_n : (\mathcal{A}_n, q_n)], [\vec{x}_0 : (\mathcal{A}_0, q_0)]\} \mid$$
$$\mathcal{A}_1 = \mathcal{M}(R_1) \wedge \ldots \wedge \mathcal{A}_n = \mathcal{M}(R_n) \wedge \mathcal{A}_0 = \overline{\mathcal{M}(R_0)} \wedge \forall i \in [\![0, n]\!], q_i \in Q_f(\mathcal{A}_i) \Big\}$$

The set of solutions σ to $tp(\mathcal{M}, \varphi)$ is the same as the set of counter-examples to $\mathcal{M} \models \varphi$. Intuitively, for such a counter-example to exist, it should validate the atoms $R_1(\vec{x}_1), \ldots, R_n(\vec{x}_n)$ (i.e. be recognized by $\mathcal{M}(R_1) \ldots, \mathcal{M}(R_n)$) and invalidate the atom $R_0(\vec{x}_0)$ (i.e. be recognized by $\overline{\mathcal{M}(R_0)}$).

▶ **Theorem 17** (Model checking as type checking).
For each model $\mathcal{M}$ and CHC property φ, $\mathcal{M} \not\models \varphi \iff \exists\sigma, \exists(E,\Omega) \in tp(\mathcal{M},\varphi),\ \sigma \models (E,\Omega)$.

▶ **Example 18** (Model checking a property). Let φ be $\mathrm{Len}(\underline{l},\ \underline{n}) \Rightarrow \mathrm{Even}(\underline{n})$, a formula stating that all lists are of even length. Let $\mathcal{M} = \{\mathrm{Len} \mapsto \mathcal{A}_{\mathrm{Len}},\ \ \mathrm{Even} \mapsto \mathcal{A}_{\mathrm{Even}}\}$ where $\mathcal{A}_{\mathrm{Len}}$ and $\mathcal{A}_{\mathrm{Even}}$ respectively define the length relation on integer lists and the even predicate of integers. $\mathcal{A}_{\mathrm{Len}}$ has states $\{q_f, q\}$, final states $\{q_f\}$, and rules $\{(A) : \langle Z, \square\rangle() \to q,\ \ (B) : \langle S, \square\rangle(q) \to q,\ \ (C) : \langle Cons, S\rangle(q, q_f) \to q_f,\ \ (D) : \langle Nil, Z\rangle() \to q_f\}$. $\mathcal{A}_{\mathrm{Even}}$ has states $\{q_e, q_o\}$, final states $\{q_e\}$, and rules $\{(1) :\ \langle Z\rangle() \to q_e,\ \ (2) :\ \langle S\rangle(q_o) \to q_e,\ \ (3) : \langle S\rangle(q_e) \to q_o\}$.
To check whether $\mathcal{M} \not\models \varphi$, we first translate $(\mathcal{M},\varphi)$ into a typing problem instance. Note that Even appears in the head of the property φ, therefore we will need to complement $\mathcal{A}_{\mathrm{Even}}$. We write its complement $\mathcal{A}_{\mathrm{Odd}}$, which is the same automaton but with final states $\{q_o\}$.

$$tp(\mathcal{M},\varphi) = \{(E_0,\Omega_0)\} \text{ with } E_0 = \emptyset \text{ and } \Omega_0 = \{[\langle l, n\rangle : (\mathcal{A}_{\mathrm{Len}}, q_f)], [\langle n\rangle : (\mathcal{A}_{\mathrm{Odd}}, q_o)]\}$$

In this case, $tp(\mathcal{M},\varphi)$ only contains one element (as each automaton only has one final state), therefore $\mathcal{M} \not\models \varphi \iff \exists\sigma,\ \sigma \models (\emptyset,\Omega_0)$.

5.1 Proof system

A proof obligation is the assertion that some typing problem (E,Ω) admits a solution, which is written as $\vdash (E,\Omega)$. We first define the *unfolding* of typing obligations and then the proof system. Any solution for a typing obligation $\omega = [\langle x_1,\dots,x_n\rangle : (\mathcal{A}, q)]$ can be found by following transitions of the automaton $\mathcal{A}$. A transition $\langle f_1,\dots,f_n\rangle(q_1,\dots,q_k) \to q$ of $\mathcal{A}$ (note that q is the same between the typing obligation and the rule's goal state) can act as a typing rule whose application generates k new typing obligations (one for each sub-state q_j of the rule) and n new algebraic datatype constraints, the i^{th} stating that variable x_i is of the form $f_i(\vec{x}_i)$ with $\vec{x}_i$ some fresh variables. We formally define this step as *unfolding* a typing obligation.

▶ **Definition 19** (Unfolding a typing obligation).
$unfold([\langle x_1,\dots,x_n\rangle : (\mathcal{A},q)]) = \{(E_r,\Omega_r) \mid r \in \Delta(\mathcal{A}) \wedge r = \langle f_1,\dots,f_n\rangle(q_1,\dots,q_k) \to q\}$ *with* $E_r = \{x_i = f_i(\vec{x}_i) \mid i \in [\![1,n]\!]\}$ *and* $\Omega_r = \{[\bigcirc(\vec{x}_1,\dots,\vec{x}_n)[j] : (\mathcal{A}, q_j)] \mid j \in [\![1,k]\!]\}$ *where* $\forall i \in [\![1,n]\!]$, $\vec{x}_i$ *are fresh variables.*

▶ **Example 20** (Unfolding). Continuing with Example 18, we set $\omega_1 = [\langle l,n\rangle : (\mathcal{A}_{\mathrm{Len}}, q_f)]$ and $\omega_0 = [\langle n\rangle : (\mathcal{A}_{\mathrm{Odd}}, q_o)]$. Now, ω_0 can be unfolded by rules $\{(3)\}$ and ω_1 by $\{(C),(D)\}$.

$$unfold(\omega_0) = \{(E_{(3)},\Omega_{(3)})\} \text{ with } E_{(3)} = \{n = S(m)\} \text{ and } \Omega_{(3)} = [\langle m\rangle : (\mathcal{A}_{\mathrm{Odd}}, q_e)].$$

$$unfold(\omega_1) = \{(E_{(D)},\Omega_{(D)}),\ (E_{(C)},\Omega_{(C)})\} \text{ with}$$

$$E_{(D)} = \{l = Nil,\ n = Z\},\ \ \Omega_{(D)} = \emptyset,$$

$$E_{(C)} = \{l = Cons(l_1,l_2),\ n = S(n_1)\},$$

$$\Omega_{(C)} = \{[\langle l_1, \square\rangle : (\mathcal{A}_{\mathrm{Len}}, q_n)],\ \ [\langle l_2, n_1\rangle : (\mathcal{A}_{\mathrm{Len}}, q_f)]\}.$$

We define the unfolding of a set of typing obligations as the (combination of) unfolding of *each* typing obligation at the same time, that is the application of one rule of the automaton to each typing obligation.

▶ **Definition 21** (Unfolding a typing problem).

$$unfolds(\Omega) = \{(\bigcup_{\omega\in\Omega} E_\omega, \bigcup_{\omega\in\Omega} \Omega_\omega,) \mid \forall \omega \in \Omega, (E_\omega, \Omega_\omega) \in unfold(\omega)\}$$

▶ **Example 22.** $unfolds(\{\omega_0, \omega_1\}) = \{(E_{(3)} \cup E_{(D)},\ \Omega_{(3)} \cup \Omega_{(D)}),\ (E_{(3)} \cup E_{(C)},\ \Omega_{(3)} \cup \Omega_{(C)})\}$

Finally, the proof system on typing problems consists of two deduction rules. The rule CONCLUDE concludes a proof when no typing obligation are left and when the algebraic datatype constraints are consistent. The rule STEP applies unfolding of typing problems using rules of the tree automaton.

▶ **Definition 23** (Proof system). *Our proof system contains two rules.*

$$\textsc{Conclude}\ \frac{}{\vdash (E, \varnothing)}\ \text{if } Coherent(E) \qquad \textsc{Step}\ \frac{\vdash (E \cup E', \Omega')}{\vdash (E, \Omega)}\ \text{if } Coherent(E \cup E') \text{ and } (E', \Omega') \in unfolds(\Omega)$$

▶ **Example 24.** Continuing example 20, we build a proof tree of $\vdash (E_0,\ \Omega_0)$. Rule CONCLUDE cannot be immediately applied, so let us consider STEP, and thus $unfolds(\Omega_0)$.

Its element $(E_{(3)} \cup E_{(D)},\ \Omega_{(3)} \cup \Omega_{(D)})$ can be discarded because $E_{(3)} \cup E_{(D)}$ is contradictory, as both constraints $n = Z$ and $n = S(m)$ are present. Its other element, $(E_{(3)} \cup E_{(C)},\ \Omega_{(3)} \cup \Omega_{(C)})$, is coherent, so we can apply the STEP rule. We write it (E_1, Ω_1) where $E_1 = \{l = Cons(l_1, l_2),\ n = S(n_1), n = S(m)\}$ and Ω_1 is the set of typing obligations $\Omega_1 = \{[\langle l_1, \square\rangle : (\mathcal{A}_{\text{Len}}, q_n)],\ [\langle l_2, n_1\rangle : (\mathcal{A}_{\text{Len}}, q_f)],\ [\langle m\rangle : (\mathcal{A}_{\text{Odd}}, q_e)]\}$. We now have the new typing problem $(E_0 \cup E_1, \Omega_1)$. Rule CONCLUDE still cannot be applied. Then, $unfolds(\Omega_1)$ has 8 elements, only 4 of which are coherent. Its four coherent element can be seen as two times the almost-same two elements, the only difference being which rule has been applied to $[\langle l_1, \square\rangle : (\mathcal{A}_{\text{Len}}, q_n)]$. For this example, we only show the two elements that used rule (A), (E_2, Ω_2) and (E'_2, Ω'_2) with

$$E_2 = \{l_1 = Z,\ l_2 = Nil,\ n_1 = Z,\ m = Z\}, \quad \Omega_2 = \varnothing,$$
$$E'_2 = \{l_1 = Z,\ l_2 = Cons(l_{21}, l_{22}),\ n_1 = S(n_{11}),\ m = S(m_1)\},$$
$$\Omega'_2 = \{[\langle l_{21}, \square\rangle : (\mathcal{A}_{\text{Len}}, q_n)], [\langle l_{22}, n_{11}\rangle : (\mathcal{A}_{\text{Len}}, q_f)], [\langle m_1\rangle : (\mathcal{A}_{\text{Odd}}, q_o)]\}$$

Constraints $E_1 \cup E_2$ are coherent and Ω_2 is empty, so rule CONCLUDE can be applied and a solution can be built from $E_0 \cup E_1 \cup E_2$, that is $\{n \mapsto S(Z),\ l \mapsto Cons(Z, Nil)\}$. The final proof tree is depicted on the right. For now, every proof tree is a single line. This will no longer be true with the introduction of the rule SPLIT in section 6.

$$\textsc{Step}\ \cfrac{\textsc{Step}\ \cfrac{\textsc{Conclude}\ \cfrac{}{\vdash (E_1 \cup E_2, \varnothing)}}{\vdash (E_1, \Omega_1)}}{\vdash (\varnothing, \Omega_0)}$$

▶ **Definition 25** (Heights). *We define a useful metric for proofs, the* height*:*

- *The height of a term $t = f(t_1, \ldots, t_n)$ is inductively defined as $h(t) = 1 + \max_{i\in[\![1,n]\!]}(h(t_i))$.*
- *The height of a ground formula $\underline{\varphi}$, written $h(\underline{\varphi})$, is defined as the height of the highest term occurring in it.*
- *The height of a substitution σ together with a typing obligation $\omega = [\langle x_1, \ldots, x_n\rangle : (\mathcal{A}, q)]$ is defined as $h(\sigma, \omega) = \max_{i\in[\![1,n]\!]}(h(\sigma(x_i)))$.*

- *The height of a substitution with a set of typing obligations is* $h(\sigma,\Omega) = \max_{\omega\in\Omega}(h(\sigma,\omega))$.
- *The height of a proof tree* T*, written* $h(T)$*, is defined as the maximal number of occurrences of the* Step *rule on a branch.*

▶ **Theorem 26** (Proof system is correct and complete)**.**
We have $\forall(E,\Omega), (\exists\sigma, \sigma \models (E,\Omega)) \iff \vdash (E,\Omega)$*. More precisely, for any* (E,Ω) *and* $n \in \mathbb{N}$*,*

(A) For any proof tree T *of* $\vdash (E,\Omega)$ *with* $h(T) = n$*, there exists a substitution* σ *such that* $\sigma \models (E,\Omega)$ *and* $h(\sigma,\Omega) = n$*.*
(B) For any substitution σ *such that* $\sigma \models (E,\Omega)$ *and* $h(\sigma,\Omega) = n$*, there exists a proof tree* T *of* $\vdash (E,\Omega)$ *such that* $h(T) = n$*.*

The proof can be found in Appendix A.

▶ **Corollary 27** (Smallest counter-example)**.** *By theorem 26, a breadth-first exploration of proof trees for a given typing problem* (E,Ω) *admitting a solution yields a solution of* minimal height, *that is, a substitution* σ *that has the minimal value* $h(\sigma,\Omega)$*.*

6 Proof search procedure

The search of a proof or the certainty of the absence of proof is implemented as a breadth-first exploration of the above-defined proof trees. This problem is undecidable in general [18], thus this procedure either finds a solution to the typing problem (i.e. a counter-example to $\mathcal{M} \models \varphi$) or tries every possibility and finds no counter-example (meaning that $\mathcal{M} \models \varphi$), or diverges. We present two sound optimizations which significantly improve the proving and disproving power of the proof search procedure. Using those optimizations makes this procedure usable and efficient in practice (see experiments in Section 9).

The first optimisation consists in *splitting independent typing obligations* when they do not depend on each other.

▶ **Definition 28** (Independence)**.** *Let* (E,Ω) *be a typing problem with* E *coherent.* $\Omega_a \subseteq \Omega$ *and* $\Omega_b \subseteq \Omega$ *are said* independent w.r.t. E*, written* $\Omega_a \parallel^E \Omega_b$*, when*

$$\forall\sigma_a,\sigma_b, [\sigma_E;\sigma_a \models \Omega_a \wedge \sigma_E;\sigma_b \models \Omega_b] \Rightarrow [\forall x \in Vars(\sigma_E(\Omega_a)) \cap Vars(\sigma_E(\Omega_b)), \sigma_a(x) = \sigma_b(x)]$$

Therefore, any two solutions σ'_a of (E,Ω_a) and σ'_b of (E,Ω_b) with $\Omega_a \parallel^E \Omega$ can first be factorized by σ_E by letting σ_a and σ_b such that $\sigma'_a = \sigma_E;\sigma_a$ and $\sigma'_b = \sigma_E;\sigma_b$ and then joined into $\sigma_{ab} = \sigma_a \cup \sigma_b$, and we have $\sigma_E;\sigma_{ab} \models (E,\Omega_a \cup \Omega_b)$. Finding a most precise partitioning of (E,Ω) into independent sub-problems is hard, as it may require to examine the shape of automata. We define below a safe and easy-to-compute approximation of these independence classes that splits typing obligations whose variables cannot be related even using the equalities of E.

▶ **Definition 29** (Splitting)**.** *Let* E *be a set of constraints. Let* $V_E([\vec{x} : (\mathcal{A}, q)]) \doteq Vars(\sigma_E(\vec{x}))$*. The set* $V_E([\vec{x} : (\mathcal{A}, q)])$ *is the set of variables remaining in a typing obligation after application of the most general unifier* σ_E *of* E*. Note how* $(\mathcal{A}, q)$ *has not been used. We define* $D_E \subseteq \Omega \times \Omega$ *as* $D_E(\omega_1,\omega_2) \doteq (V_E(\omega_1) \cap V_E(\omega_2) \neq \emptyset)$*. Since* D_E *is symmetric, its reflexive and transitive closure* D^*_E *is an equivalence relation. We define the function* $\mathrm{Split}(E,\Omega)$ *to return the equivalence classes of* D^*_E *defined on* Ω*.*

▶ **Lemma 30.** $\forall\Omega_1,\Omega_2 \in \mathrm{Split}(E,\Omega),\ \Omega_1 \parallel^E \Omega_2$.

Proof. For any $\Omega_1, \Omega_2 \in \text{Split}(E, \Omega)$, $Vars(\sigma_E(\Omega_1)) \cap Vars(\sigma_E(\Omega_2)) = \emptyset$. Therefore $\Omega_1 \parallel^E \Omega_2$. ◀

This separation into independent problems makes the search less combinatorial and give rise to a new rule for our typing system:

$$\textsc{Split}\ \frac{\vdash (E, \Omega_1) \quad \dots \quad \vdash (E, \Omega_n)}{\vdash (E, \Omega)} \quad \text{with } \{\Omega_1, \dots, \Omega_n\} = \text{Split}(E, \Omega)$$

▶ **Example 31** (Splitting (E_1, Ω_1)). In example 24, we had $E_1 = \{l = Cons(l_1, l_2),\ n = S(n_1), n = S(m)\}$ and $\Omega_1 = \{\omega_1, \omega_2, \omega_3\}$ with $\omega_1 = [\langle l_1, \square \rangle : (\mathcal{A}_{\text{Len}}, q_n)]$, with $\omega_2 = [\langle l_2, n_1 \rangle : (\mathcal{A}_{\text{Len}}, q_f)]$, and $\omega_3 = [\langle m \rangle : (\mathcal{A}_{\text{Odd}}, q_e)]$. We have $\sigma_{E_1} = \{l \mapsto Cons(l_1, l_2),\ n \mapsto S(n'),\ n_1 \mapsto n',\ m \mapsto n'\}$, $V_{E_1}(\omega_1) = \{l_1\}$, $V_{E_1}(\omega_2) = \{l_2, n'\}$, and $V_{E_1}(\omega_3) = \{n'\}$. Therefore $\text{Split}(E_1, \Omega_1) = \{\{\omega_1\}, \{\omega_2, \omega_3\}\}$.
Solving ω_1 have no impact on the solving of ω_2 and ω_3 because the values that l_1 can take do not influence the values that l_2, n_1, or m_2 can take. On the other hand, because of E_1, m and n_1 must take the same value, and therefore typing obligations ω_2 and ω_3 cannot be separated. Note that applying this Split rule before the second Step (of example 24) would have separated (E_1, Ω_1) into two independent problems.

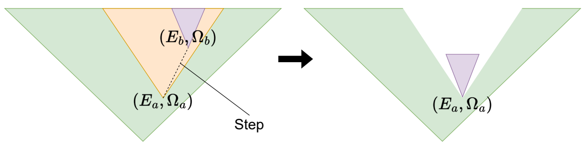
The second optimisation consists in *pruning the search tree*. The search space is, for almost all typing problems, infinite. Without pruning, it would be impossible to cover the whole search space, and therefore negative instances would (almost) all never terminate. Pruning the search tree allows, in some cases, to finitely ensure that no typing proof exists.

▶ **Definition 32** (Pruning). *Let T be a proof tree. A node $\vdash (E_b, \Omega_b)$ that appears in the sub-tree of T whose root is some other node $\vdash (E_a, \Omega_a)$ is* prunable *when both*

(i) At least one Step rule is used on the path between $\vdash (E_a, \Omega_a)$ and $\vdash (E_b, \Omega_b)$;
(ii) $\exists \sigma, \sigma(\sigma_{E_a}(\Omega_a)) \subseteq \sigma_{E_b}(\Omega_b)$.

▶ **Theorem 33** (Safety of pruning). *For any proof tree that contains a prunable node, there exist a strictly smaller (w.r.t the total number of times the Step rule is used) proof tree with the same root.*

The idea of pruning a proof T is to replace the orange proof sub-tree of $\vdash (E_a, \Omega_a)$ with the purple proof tree of $\vdash (E_b, \Omega_b)$ (with minor modifications).



Proof. Let T be a prunable tree, that is such that there exists nodes $\vdash (E_a, \Omega_a)$ and $\vdash (E_b, \Omega_b)$ with respective proof trees T_a and T_b, with T_b a sub-tree of T_a with a Step rule between $\vdash (E_a, \Omega_a)$ and $\vdash (E_b, \Omega_b)$, and σ a substitution such that $\sigma(\sigma_{E_a}(\Omega_a)) \subseteq \sigma_{E_b}(\Omega_b)$.
By theorem 26(A) there exists a substitution σ_b with $\sigma_b \models (E_b, \Omega_b)$ and $h(\sigma_b, \Omega_b) = h(T_b)$. Because σ_{E_b} is the most general unifier of E_b and $\sigma_b \models E_b$, there exists σ' such that $\sigma_b = \sigma_{E_b}; \sigma'$. Therefore the substitution $\sigma_a = \sigma_{E_a}; \sigma; \sigma'$ is such that $\sigma_a(\Omega_a) \subseteq \sigma_b(\Omega_b)$. Because $\sigma_b \models \Omega_b$, we also have $\sigma_a \models \Omega_a$. Because σ_a first applies σ_{E_a}, we have $\sigma_a \models E_a$. Therefore $\sigma_a \models (E_a, \Omega_a)$. Finally, again because $\sigma_a(\Omega_a) \subseteq \sigma_b(\Omega_b)$, we have $h(\sigma_a, \Omega_a) \leq h(\sigma_b, \Omega_b)$. By applying theorem 26(B) there exists a proof T'_a of $\vdash (E_a, \Omega_a)$ with $h(T'_a) = h(\sigma_a, \Omega_a) \leq h(\sigma_b, \Omega_b) = h(T_b)$.

Therefore, the proof tree T whose sub-tree T_a has been replaced by T'_a is valid and smaller. Besides, we know that the sub-tree T'_a is *strictly* smaller than T_a because T_a contains at least one application of the STEP rule between its root and T_b. Therefore, this transformation strictly decreases the size of the proof tree. ◀

▶ **Corollary 34.** *By induction, if there exists a proof tree T of some initial typing problem, then there exists one without any prunable node along the proof tree, and therefore abandoning the search of prunable branches is safe.*

▶ **Example 35** (Pruning of the search tree)**.** During the second STEP application of example 24, the typing problem (E_2', Ω_2') is also in $unfolds(\Omega_1)$. This was no problem, as the algorithm found a solution and stopped. Now, if (for example) automaton $\mathcal{A}_{\text{Len}}$ did not have rule (D), then there would be no solution to the initial typing problem (E_0, Ω_0). The search would never stop, as, after a bit of unification and renaming, (E_0, Ω_0) can be included in $(E_1 \cup E_2', \Omega_2')$. Without pruning, the typing algorithm could therefore loop forever instead of returning *None*. Fortunately, $(E_1 \cup E_2', \Omega_2')$ can be pruned by taking $\sigma = \{l \mapsto l_{22},\ n \mapsto n_{11}\}$, as $\sigma(\sigma_0(\Omega_0)) \subseteq \sigma_2(\Omega_2')$ (with σ_0 and σ_2 being most general unifiers of E_0 and $E_0 \cup E_1 \cup E_2'$, respectively).

7 Regular structure inference

This section presents a procedure for inferring a regular model of a set of CHCs. The input set of CHCs we later use the procedure for is $\mathcal{C} = \Gamma \cup \Gamma'$, with Γ defining a program and Γ' the desired properties. The procedure follows the Implication Counter-Example (ICE) framework [8]. In this framework, the task of inferring a correct model is divided between two entities (or procedures), a *learner* and a *teacher*, working iteratively. There are three possible outcomes for this procedure: either the learner finds a correct model (that the teacher validates), the learner finds a contradiction, or the procedure loops forever with more and more refined models.

The teacher's procedure takes as input a model $\mathcal{M}$ and a CHC system $\mathcal{C}$, and returns an optional ground Horn clause. It returns *None* if $\mathcal{M} \models \mathcal{C}$, and $Some(\sigma(\varphi))$ if $\mathcal{M} \not\models \varphi$ with counter-example σ for some $\varphi \in \mathcal{C}$. With the model checking procedure already defined, a teacher's implementation is only a matter of selecting an order in which to check the formulas. For example, taking as input the problem of example 18, the output would be $\text{Len}(Cons(Z, Nil),\ S(Z)) \Rightarrow \text{Even}(S(Z))$.

The learner's procedure is responsible for inferring a model from examples or finding a contradiction. It takes as input a finite set $\underline{\mathcal{C}}$ of ground CHCs and returns *None* if $\underline{\mathcal{C}}$ is contradictory and $Some(\mathcal{M})$ otherwise, with $\mathcal{M}$ being a smallest model (in the number of states) satisfying $\underline{\mathcal{C}}$. This procedure is divided into two steps, which are the main subject of this section, the *working model generation* and the *working model generalisation*.

▶ **Definition 36** (Working model generation)**.** *The working model $\mathcal{W}$ of a given finite set of ground CHCs $\underline{\mathcal{C}}$ is the smallest model (up to state renaming) recognizing exactly the terms mentioned in $\underline{\mathcal{C}}$ in a different state for each. That is, for any atom $R(\vec{t})$ of any $\varphi \in \underline{\mathcal{C}}$, there exists a state q in $\mathcal{W}(R)$ such that $\mathcal{R}(\mathcal{W}(R), q) = \{\vec{t}\}$.*

This working model construction is carried out by classical automaton algorithms [6]. The model $\mathcal{W}$ can then be generalised by merging states and deciding which equivalence classes are to be considered as final states. Merging states leads to additional terms being recognized and makes regularity appear. We search for a merging that minimises the number of states of $\mathcal{W}$ while ensuring that the resulting model satisfies $\underline{\mathcal{C}}$.

▶ **Definition 37** (State merging problem). *The minimisation problem we define is on the first-order (functional) signature* $S = \{c_q \mid \mathcal{A} \in dom(\mathcal{W}) \wedge q \in Q(\mathcal{A})\} \cup \{Final\}$ *containing only constants, one for each state of every automaton in* $\mathcal{W}$*, and one unary predicate Final. The constraints are* $\mathcal{C}_{ok} \cup \mathcal{C}_f$*. The set* $\mathcal{C}_{ok}$ *represents essential constraints: (i) merged states must belong to the same automaton ; (ii) merged states must be of the same type ; (iii) any final state must be of its automaton's type. The set* $\mathcal{C}_f$ *forces states to be or not to be final, which also have an impact on which states to merge. It is defined from* $\underline{\mathcal{C}}$ *by transforming every clause* $\varphi = R_1(\vec{t}_1) \wedge \ldots \wedge R_n(\vec{t}_n) \Rightarrow R_0(\vec{t}_0)$ *into* $\varphi^q = Final(c_{q_1}) \wedge \ldots \wedge Final(c_{q_n}) \Rightarrow Final(c_{q_0})$*, with each* q_i *being the state of* $\mathcal{W}(R_i)$ *that recognizes exactly* $\vec{t}_i$*. Recall that we use non-Horn clauses, so the head of* φ *could be empty or contain multiple predicates.*

A minimal solution $[\![\cdot]\!]$ to the state merging problem can be computed by a finite model finder. We write $[\![Final]\!]$ for the set of final states of the solution and $[\![c_q]\!]$ for the equivalence class of constant c_q.

▶ **Definition 38** (Generalisation of working model). *Given a solution* $[\![\cdot]\!]$ *to the state merging problem, we generalise the working model* $\mathcal{W}$ *by* $\mathcal{M}$ *with* $\mathcal{M}(R) = (Q, Q_f, \Delta)$ *with* $Q = \{[\![c_q]\!] \mid q \in Q(\mathcal{W}(R))\}$*,* $Q_f = Q \cap [\![Final]\!]$ *and* $\Delta = \{\vec{f}([\![c_{q_1}]\!], \ldots, [\![c_{q_n}]\!]) \to [\![c_{q_0}]\!] \mid \vec{f}(q_1, \ldots, q_n) \to q_0 \in \Delta(\mathcal{W}(R))\}$*.*

▶ **Example 39** (Learner: Model generation). We observe the ICE procedure after learner and teacher already had two exchanges to learn the Len relation defined in Section 2. The learner has accumulated the constraints $\{\text{Len}(Nil, Z),\ \ \text{Len}(Nil, Z) \Rightarrow \text{Len}(Cons(Z, Nil), S(Z))\}$. The generated working model is $\mathcal{W} = \{\text{Len} \mapsto \mathcal{A}\}$ with $\mathcal{A} = (Q, Q_f, \Delta)$, $Q = \{q_{l_0}, q_{l_1}, q_n\}$, $Q_f = \emptyset$, and $\Delta = \{\langle Nil, Z\rangle() \to q_{l_0}\ ;\ \langle Cons, S\rangle(q_n, q_{l_0}) \to q_{l_1}\ ;\ \langle Z, \square\rangle() \to q_n\}$. We have $\mathcal{R}(\mathcal{A}, q_{l_0}) = \{(Nil, Z)\}$, $\mathcal{R}(\mathcal{A}, q_n) = \{(Z, \square)\}$, and $\mathcal{R}(\mathcal{A}, q_{l_1}) = \{(Cons(Z, Nil), S(Z))\}$. Note that state q_n recognizes the term $\langle Z, \square\rangle$ which does not appear in $\underline{\mathcal{C}}$ but is necessary to recognize $(Cons(Z, Nil), S(Z))$.

The minimisation problem is therefore on the signature with unary predicate *Final* and constant symbols $c_{q_{l_0}}$, $c_{q_{l_1}}$, and c_{q_n}. The constraints $\mathcal{C}_{ok}$ are stating that q_n cannot be merged with q_{l_0} nor q_{l_1} because they are not of the same type, and that only q_{l_0} and q_{l_1} can be final, as they are the only states of the automaton's type, *natlist* $\times$ *nat*. The constraints $\mathcal{C}_f$, generated from $\underline{\mathcal{C}}$, are $\{Final(c_{q_{l_0}}),\ Final(c_{q_{l_0}}) \Rightarrow Final(c_{q_{l_1}})\}$. The smallest model is a two-elements set $\{q_l, q_z\}$, with $[\![Final]\!] = \{q_l\}$, $[\![q_{l_0}]\!] = [\![q_{l_1}]\!] = q_l$, and $[\![q_n]\!] = q_z$.

The generalized model is $\mathcal{M} = \{\text{Len} \mapsto \mathcal{A}'\}$ with automaton $\mathcal{A}'$ having states $\{q_l, q_z\}$, final states $\{q_l\}$, and transitions $\{\langle Nil, Z\rangle() \to q_l,\ \langle Cons, S\rangle(q_z, q_l) \to q_l,\ \langle Z, \square\rangle() \to q_z\}$. This automaton recognizes an almost-correct relation: the set of pairs (l, n) of a list of zeros together with its size. The only missing rule is $\langle S, \square\rangle(q_z) \to q_z$, which will be added by the learner in the ICE step that follows.

8 Approximation

As we suppose programs to be deterministic and terminating, the CHC representation of a functional program has only one possible model. For many programs, this model is not regular and cannot be represented using convoluted tree automata. As a result, trying to verify a property using an exact model of the relation will fail on such programs. We circumvent this problem by approximating relations.

Our verification goals are CHCs of the form $\psi(\vec{x}) \wedge R_1(\vec{x}_1) \wedge \ldots \wedge R_n(\vec{x}_n) \Rightarrow R_0(\vec{x}_0)$. Given a relation R we denote by R^+ (resp. R^-) an over-approximation (resp. under-approximation) of R which can also be R itself. A safe way to prove the above implication using approximations is to over-approximate $R_1, \dots, R_n$ and under-approximate R_0. If $\psi(\vec{x}) \wedge R_1^+(\vec{x}_1) \wedge \dots \wedge R_n^+(\vec{x}_n) \Rightarrow R_0^-(\vec{x}_0)$ is true then so is the original CHC. Applying such a reasoning on the CHCs of the verification goal, we can infer which relations can be over or under-approximated. For instance, the functional program computing the sum of two natural numbers is represented by the relation $\mathrm{Plus}(n, m, u)$ associating any two natural numbers n and m with their sum u. This relation is not regular when using unary encoding of numbers. The argument for seeing this is very similar to that of $\{a^n \cdot b^n \mid n \in \mathbb{N}\}$ not being a regular string language. For the string automaton, it would require an unbounded counter for as in order to later exactly match their number with bs. For a convoluted tree automaton to recognize $\mathrm{Plus}(n, m, u)$, the counting is of the depth at which n and m root symbol stop being both S, which later needs to match the number of Ss left on u. However, to prove a property of the form $\mathrm{Plus}(n, m, u) \Rightarrow n \leq u$, we only need a regular over-approximation of the relation Plus, say Plus^+, and an under-approximation of $\leq$, say $\leq^-$, such that $\mathrm{Plus}^+(n, m, u) \Rightarrow n \leq^- u$.

In practice, we focus on over-approximation and do not under-approximate. We thus prove the stronger goal $\mathrm{Plus}^+(n, m, u) \Rightarrow n \leq u$. Here are the clauses defining the Plus relation:

$$\mathrm{Plus}(n, Z, n). \quad \mathrm{Plus}(n, m, u) \Rightarrow \mathrm{Plus}(n, S(m), S(u)). \quad \mathrm{Plus}(v, w, x) \wedge \mathrm{Plus}(v, w, y) \Rightarrow x = y.$$

These clauses form a system where the first clause invalidates under-approximations, the second clause can invalidate both over and under approximations, whereas the third only invalidates over-approximations. We can therefore obtain a safe approximation Plus^+ from Plus by simply removing the third clause. In our example, this suffices to prove $\mathrm{Plus}^+(n, m, u) \Rightarrow n \leq u$ because the approximation Plus^+ we built relates any n, m with all u greater than or equal to n (See the solver result for `isaplanner_prop21.smt2` in `http://people.irisa.fr/Thomas.Genet/AutoForestation/results_right/benchmarks.html`).

Finally, some relations cannot be approximated. If a relation appears on both sides of the verification goal then it cannot be approximated. *E.g.*, to prove $Z < m \wedge \mathrm{Plus}(n, m, u) \Rightarrow n < u$, we can safely use Plus^+. Since $<$ occurs (positively) on the left and right-hand side of the implication, we could use $<^+$ on the left-hand side and $<^-$ on the right-hand side. We could use different approximations for relations appearing at different positions in the formula. However, in our analyser, we choose to use a common approximation for any relation. In our example, we use the intersection between $<^+$ and $<^-$, which is exactly $<$.

9 Implementation and Experiments

We implemented the verification algorithm in Ocaml. It can be found on `https://gitlab.inria.fr/tlosekoo/auto-forestation`. This provides an implementation of terms, tree automata, model checking, model-inference procedure, as well as left, right, and complete convolution.

The *teacher* closely follows the depth-first search of the proof system described in section 5. There is a lot of redundancy in the proof search, so we used canonization and memoisation of typing problems. Memoisation avoids re-computing the unfolding of a typing problem if the search already did. However, memoisation alone is not very useful, as even equivalent typing problems are often different because of variable names. This is the reason for using canonization, which ensures that equivalent typing problems have the same internal representation. The *learner* delegates the merging of states to *Clingo* [9], a finite-model finder.

The solver presented in this paper builds regular relations, as opposed to [11, 16, 17] which only build regular sets of terms. Since regular sets are a particular case of regular relations, our solver should be able to handle the examples covered by those techniques, plus some relational problems. As a result, for the experiments, we choose some examples coming from benchmarks of Timbuk [11], add relational examples taken from the Isaplanner benchmark [4,7] and built relational problems inspired by TIP [4,5]. As shown in Section 2, a typical property which can be automatically proved by those non-relational solvers [11,16,17] is of the form $\forall x\, l.\, less\ Z\ (len\ (Cons(x, l)))$ where l is any list of natural numbers.

Non-relational solvers can also handle a restricted form of relations: the finite union of languages $\mathcal{L}_1 \times \ldots \times \mathcal{L}_n$ where $\forall i \in [\![1, n]\!]$, $\mathcal{L}_i$ is a regular language. This allows to prove properties with a limited form of relation. For instance, using a non-relational regular solver, it is possible to prove the property $\forall l_1\ l_2.\, less\ Z\ (len\ l_1) \Rightarrow\ less\ Z\ (len\ (append\ l_1\ l_2))$ where $append$ is the function concatenating lists and l_1 and l_2 are lists of a. For the tuple of variables (l_1, l_2) to cover all the possible cases, it is enough to consider the two languages $\mathcal{L}_{nil} \times \mathcal{L}_{lists}$ and $\mathcal{L}_{Cons+} \times \mathcal{L}_{lists}$ where $\mathcal{L}_{nil} = \{Nil\}$ and $\mathcal{L}_{Cons+} = \mathcal{L}_{lists} \setminus \mathcal{L}_{nil}$. With the first language, the property is true because the left-hand side of the implication is false. With the second language $\mathcal{L}_{Cons+} \times \mathcal{L}_{list}$, both the left and right-hand side of the implication are true.

One of the simplest problem which cannot be proved using a non-relational "regular" solver is $\forall x\ y.\, Cons(x, y) \neq y$. Proving such a property cannot be done using a finite union of products of regular languages. However, this property can automatically be proven using our relational solver. Additionally to the above examples, we highlight some relational properties which are automatically proven using our solver.

- $\forall (l : ablist).\ (len\ l) = (len\ (reverse\ l))$ `length_reverse_eq.smt2`
- $\forall (l_1 : ablist)\ (l_2 : ablist).\ (prefix\ l_1\ (append\ l_1\ l_2))$ `prefix_append.smt2`
- $\forall (l : ablist).\ (len\ l) = (len\ (sort\ l))$ `sort_length_eq.smt2`
- $\forall (i : nat)(t_1 : natbintree)(t_2 : natbintree).\ t_1 \neq (node\ i\ t_1\ t_2)$ `tree_add_not_eq.smt2`

On the following properties our solver is able to find a counter-example.

- $\forall (n : nat).\ n < (double\ n)$ `nat_double_is_le.smt2`
- $\forall (x : ab)\ (l : ablist).\ (delete\_one\ x\ l) = (delete\_all\ x\ l) \Rightarrow (count\ x\ l) = 1$ `list_delete_all_count.smt2`

On the following properties, our solver does not terminate due to trying to represent a non-regular relation (ICE loops).

- $\forall (x : ab)\ (l : ablist).\ (delete\_one\ x\ l) = (delete\_all\ x\ l) \Rightarrow (count\ x\ l) \leq 1$ `list_delete_all_count.smt2`
- $\forall (n : nat)\ (m : nat).\ n + m = m + n$ `plus_commutative.smt2`

All of our experimental results for all convolution types are available at `http://people.irisa.fr/Thomas.Genet/AutoForestation/`. Because the properties of our database were mostly either on same-type relations or on lists and natural numbers, the right-convolution was the most efficient of convolution type. Left-convolution is not adapted for most of the list-based examples and complete-convolution revealed to be too costly in practice though it should help to prove properties on functions manipulating trees. On a total of 120 examples, our solver (using right-convolution) proves 66, disproves 23, and timeouts on 31 after 60s. Our solver succeeds on 20 out of the 79 first-order Isaplanner examples in less than 60s (and 18 in less than 5s). Our solver reveals to be more efficient on examples where a single level of structure have to be compared, i.e., natural numbers, lists of arbitrary elements, etc. It is generally unsuccessful on examples mixing several layers of structure, e.g., lists of natural numbers, or on examples where a precise counting is required to prove the property. Finally, on examples where using a non-relational model suffices to prove the property, our solving technique is flexible enough to find such a model, with an efficiency comparable with non-relational solvers. For instance, on 11 examples coming from the Timbuk benchmarks, we proved 6 of them (with execution times around 2 seconds), disproved 3, and have a timeout on the 2 last.

10 Related work

Other approaches for automatically proving algebraic and relational properties also rely on a CHC representation. The approach of [20] and [19] aims to solve the satisfiability problem of Horn clauses over any underlying theory supported by an SMT solver. This approach first reduces this problem to validity checking of first-order formulas with inductively-defined predicates. It is then based on syntactic proof, together with calls to the underlying theory solver. They design an inductive proof system tailored to Horn constraint solving. Using the theory of inductive datatypes, their method can reason about, and automatically prove, relational and algebraic properties.

Another approach, which is closer to ours, is that of [18]. This approach aims to check properties on recursive data-structure by using *symbolic automatic relations*, which are (almost) the languages defined by *symbolic synchronous automata* (ss-NFA), the combination of symbolic automata and automatic relations. They devise a sound but (necessarily) incomplete procedure for checking if a given formula admits an assignment of its free variables that makes it true in a given ss-NFA. This procedure corresponds to the *teacher* procedure, but for ss-NFAs. They plan to implement an ICE-based CHC solver, but have left the model discovery (*learner* section) to future work.

By manually writing ss-NFAs, authors of [18] are able to benchmark their verification procedure. Our approach and theirs 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 [18] handles (i.e. 4, 5, 15, 16, 29, 30, 39, 42, 50, 62, 67, 71, 86) and succeeds on examples on which they do not report any success (i.e. 17, 18, 21, 22, 23, 24, 25, 26, 31, 32, 33, 34, 45, 46, 65, 69).

In [3], the authors present an expressive formalism for representing relations between trees called *synchronized context-free programs*. This formalism is more expressive than convoluted tree automata presented here. In particular, it can represent languages of the form $\{(g^n(a), g^n(b)) \mid n \in \mathbb{N}\}$ (like convoluted tree automata) and also languages of the form $\{f(g^n(a), g^n(b)) \mid n \in \mathbb{N}\}$ and $\{g^n(h(g^n(a))) \mid n \in \mathbb{N}\}$ (out of the scope of convoluted tree automata). This formalism is used to precisely approximate the set of outputs of a term rewriting system. However, [3] does not show how to automatically infer such a representation from the term rewriting system.

11 Conclusion and future work

This paper demonstrates that it is possible to use tree automata as a basis for analysing the input-output behaviour of a first-order functional program. This shows that existing automata-based techniques for approximating the set of reachable states of a function can be extended to also compute relations between input and output of a function. Such relational analysis is key to scaling static analyses to larger programs, because it enables a modular, function-by-function analysis technique. The extension to relational analysis is based on the notion of tree automata convolution. We argue that the standard left-convolution can be complemented by other convolution techniques in order to verify more properties of programs. Another technical contribution of the paper is the proof tree pruning used for verifying models of constrained Horn clauses. An efficient implementation of this proof search has been an essential part of the counter-example guided learner-teacher algorithm for inferring models from the CHC representation of the program to be analysed. This is confirmed by the benchmark used to evaluate our implementation of the verifier.

We believe our ICE procedure to be *relatively refutationally complete* and *relatively complete on regular structures*. *Relative* means that we suppose the termination of the model-checking procedure to be able to study the ICE cycle. *Refutationally complete* means that if the set of clauses $\mathcal{C}$ given to the ICE procedure is contradictory, then the procedure eventually finds a contradiction and stops. *Complete on regular structures* means that if the set of clauses $\mathcal{C}$ given to the ICE procedure admits a regular model, then the procedure eventually finds a model of $\mathcal{C}$. This has to be investigated further.

Fixing the convolution to be the either left or right convolution is however insufficient for proving non-trivial properties that would need a different overlay of terms, for example the *height* function on trees. Complete convolution can theoretically overcome this restriction but, as confirmed by our benchmarks, the size explosion of convoluted term makes it unusable in practice. We believe the convolution can and should be non-static, that is, being inferred together with the model.

Moreover, unlike the convolutions presented in this paper, we think that convolution could be lossy. For instance, if a subterm in a relation is not useful to prove a property, we think that we can forget about it in the convolution. Later on, if a new ground counter-example comes to the learner showing that the subterm was, in fact, necessary to prove the property then the convolution needs to be extended for that purpose.

References

1. Clark Barrett, Igor Shikanian, and Cesare Tinelli. An abstract decision procedure for a theory of inductive data types. *Journal on Satisfiability, Boolean Modeling and Computation*, 3(1-2):21–46, 2007.
2. Nikolaj Bjørner, Arie Gurfinkel, Ken McMillan, and Andrey Rybalchenko. Horn clause solvers for program verification. In *Fields of Logic and Computation II*, pages 24–51. Springer, 2015.
3. Yohan Boichut, Jacques Chabin, and Pierre Réty. Towards more precise rewriting approximations. *J. Comput. Syst. Sci.*, 104:131–148, 2019.
4. Koen Claessen, Moa Johansson, Dan Rosén, and Nicholas Smallbone. Tip and isaplanner benchmarks, 2015. `https://tip-org.github.io/`.
5. Koen Claessen, Moa Johansson, Dan Rosén, and Nicholas Smallbone. Tip: tons of inductive problems. In *International Conference on Intelligent Computer Mathematics*, pages 333–337. Springer, 2015.
6. Hubert Comon, Max Dauchet, Rémi Gilleron, Florent Jacquemard, Denis Lugiez, Christof Löding, Sophie Tison, and Marc Tommasi. *Tree Automata Techniques and Applications*. 2008. URL: `https://hal.inria.fr/hal-03367725`.
7. Lucas Dixon and Jacques Fleuriot. Isaplanner: A prototype proof planner in isabelle. In *CADE'03*, volume 2741, pages 279–283. Springer, 2003.
8. Pranav Garg, Christof Löding, P Madhusudan, and Daniel Neider. Ice: A robust framework for learning invariants. In *International Conference on Computer Aided Verification*, pages 69–87. Springer, 2014.
9. Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. *Answer Set Solving in Practice*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2012.
10. Erich Grädel. Automatic structures: twenty years later. In *Proceedings of the 35th Annual ACM/IEEE Symposium on Logic in Computer Science*, pages 21–34, 2020.

11 Timothée Haudebourg, Thomas Genet, and Thomas Jensen. Regular Language Type Inference with Term Rewriting. *Proceedings of the ACM on Programming Languages*, 4(ICFP):1–29, 2020.

12 Timothée Haudebourg. *Automatic Verification of Higher-Order Functional Programs using Regular Tree Languages*. PhD thesis, Univ. Rennes1, 2020.

13 Tirza Hirst and David Harel. More about recursive structures: Descriptive complexity and zero-one laws. In *Proceedings 11th Annual IEEE Symposium on Logic in Computer Science*, pages 334–347. IEEE, 1996.

14 R Hodgson Bernard. *Théories décidables par automate fini (Decidable theories via finite automata)*. PhD thesis, Ph.D. thesis Département de Mathématiques et de Statistique, Université de ..., 1976.

15 Bakhadyr Khoussainov and Anil Nerode. Automatic Presentations of Structures. In *International Workshop on Logic and Computational Complexity*, pages 367–392. Springer, 1994.

16 Yurii Kostyukov, Dmitry Mordvinov, and Grigory Fedyukovich. Beyond the elementary representations of program invariants over algebraic data types. In Stephen N. Freund and Eran Yahav, editors, *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, pages 451–465. ACM, 2021.

17 Yuma Matsumoto, Naoki Kobayashi, and Hiroshi Unno. Automata-based abstraction for automated verification of higher-order tree-processing programs. In *Asian Symposium on Programming Languages and Systems*, pages 295–312. Springer, 2015.

18 Takumi Shimoda, Naoki Kobayashi, Ken Sakayori, and Ryosuke Sato. Symbolic automatic relations and their applications to SMT and CHC solving. In *International Static Analysis Symposium*, pages 405–428. Springer, 2021.

19 Takeshi Tsukada and Hiroshi Unno. Software model-checking as cyclic-proof search. *Proceedings of the ACM on Programming Languages*, 6(POPL):1–29, 2022.

20 Hiroshi Unno, Sho Torii, and Hiroki Sakamoto. Automating induction for solving horn clauses. In *International Conference on Computer Aided Verification*, pages 571–591. Springer, 2017.

A Appendix

Proof of theorem 26

Proof of A. Let suppose that T proves $\vdash (E, \Omega)$ and $h(T) = n$. Let us proceed by induction on the last rule used in T.

- case CONCLUDE:
  By hypothesis, we have that T is of the form $\dfrac{}{\vdash (E, \emptyset)}$ with $Coherent(E)$, and therefore $n = 0$. Take $\sigma = \sigma_E$ a most general unifier of E, which is well-defined, as E is coherent. We have: (i) $\sigma \models E$ is immediate, as σ unifies E; (ii) $\sigma \models \Omega$ is trivial, as $\Omega = \emptyset$; (iii) $h(\Omega, \sigma) = 0 = n$, as Ω is empty.
- case STEP:
  By hypothesis, we have that T is of the form $\textsc{Step}\ \dfrac{T'}{\vdash (E, \Omega)}$ with T' of the form $\dfrac{\cdots}{\vdash (E \cup E', \Omega')}$ and $(E', \Omega') \in unfolds(\Omega)$. By induction, we have that there exists σ' with $\sigma' \models (E \cup E', \Omega')$ and $h(\Omega', \sigma') = h(T')$. We also know that $h(T') = n - 1$. Take $\sigma = \sigma'$. Then:
  + $\boldsymbol{\sigma \models E}$: Immediate by $\sigma' \models E \cup E'$ and monotonicity of first-order logic.
  + $\boldsymbol{\sigma \models \Omega}$: Let $\omega = [\vec{x} : (\mathcal{A}, q)] \in \Omega$. We must prove that $\sigma(\vec{x}) \in \mathcal{R}(\mathcal{A}, q)$. For this, it is sufficient (and necessary) to show that there exists a rule $r = \vec{f}(\vec{q}) \to q$ of $\mathcal{A}$ such that
    * $\forall i \in [\![1, |\vec{f}|]\!], \sigma(x_i) = f_i(\vec{y}_i)$ for some variables $\vec{y}_i$;
    * $\forall j \in [\![1, |\vec{q}|]\!], \sigma \models [\bigcirc(\vec{y}_1, \ldots, \vec{y}_{|\vec{f}|})[j] : (\mathcal{A}, q_j)]$.

Since $(E', \Omega') \in unfolds(\Omega)$, we know that there exists such a rule r with $(E_r, \Omega_r) \in unfold(\omega)$. The first property is immediate from $\sigma \models E'$ and $E_r \subseteq E'$ while the second is immediate from $\sigma \models \Omega'$ and $\Omega_r \subseteq \Omega'$.

+ $\boldsymbol{h(\Omega, \sigma) = n}$: Because $(E', \Omega') \in unfolds(\Omega)$, every variable y in Ω' is such that there exists a variable x in Ω with $\sigma(x) = f(\dots, \sigma(y), \dots)$ for some function f, that is, $h(\sigma, \Omega') < h(\sigma, \Omega)$. Moreover, every variable x in Ω with $h(\sigma(x)) > 1$ yields a least one variable y in Ω' with $h(\sigma(y)) = h(\sigma(x)) - 1$.
Therefore, $h(\sigma, \Omega) = h(\sigma, \Omega') + 1 = h(T') + 1 = n$.

◀

Proof of B.

Let us build a proof tree by induction on $h(\Omega, \sigma)$.

In any case, let suppose that there exists σ such that $\sigma \models (E, \Omega)$ and $h(\Omega, \sigma) = n$. We then construct a proof tree T of $\vdash (E, \Omega)$ such that $h(T) = n$.

- case $h(\Omega, \sigma) = 0$: This is only possible when $\Omega = \emptyset$. Take $T = \dfrac{}{\vdash (E, \Omega)}\,\textsc{Conclude}$. This proof tree T is correct, as $\Omega = \emptyset$ and E is coherent (because $\sigma \models E$). Also $h(T) = 0$.
- case $h(\Omega, \sigma) > 0$:
Because $\sigma \models \Omega$, we have, for each $\omega = [\langle x_1, \dots, x_n \rangle : (\mathcal{A}, q)] \in \Omega$, that there exists an associated rule $r_\omega = \langle f_1, \dots, f_n \rangle (q_1, \dots, q_k) \to q$ such that
  + $\forall i \in [\![1, n]\!], \sigma(x_i) = f_i(\vec{t}_i)$ for some terms $\vec{t}_i$;
  + $\forall j \in [\![1, k]\!], \bigcirc(\vec{t}_1, \dots, \vec{t}_n)[j] \in \mathcal{R}(\mathcal{A}, q_j)$.

  Therefore we can build three functions, F^c, F^t, F^s, which assign to each such typing obligation and rule the following:
  + $F^c(\omega) = \{x_1 = f_1(\vec{x}_1), \dots, x_n = f_n(\vec{x}_n)\}$, with $\forall i \in [\![1, n]\!], \vec{x}_i$ are fresh variables.
  + $F^t(\omega) = \{[\bigcirc(\vec{x}_1, \dots, \vec{x}_n)[j] : (\mathcal{A}, q_j)] \mid j \in [\![1, k]\!]\}$
  + $F^s(\omega) = \{(x_i^j, t_i^j) \mid x_i = f_i(x_i^1, \dots, x_i^m) \in F^c(\omega) \wedge j \in [\![1, m]\!] \wedge \sigma(x_i) = f(t_i^1, \dots, t_i^m)\}$

  Let $E' = \bigcup_{\omega \in \Omega} F^c(\omega)$ and $\Omega' = \bigcup_{\omega \in \Omega} F^t(\omega)$. Note that $(E', \Omega') \in unfolds(\Omega)$.
  Let $\sigma' = \sigma \cup \bigcup_{\omega \in \Omega} F^s(\omega)$. We have:
  + σ' is well-defined: Any binding of σ' which is not in σ is of the form $x_i^j = \sigma(t_i^j)$ for some fresh variable x_i^j. Therefore, as σ is well-defined, so is σ'.
  + $\sigma' \models E \cup E'$: We have $\sigma \subseteq \sigma'$, therefore $\sigma' \models E$. Any constraint of E' is of the form $x_i = f_i(\vec{x}_i)$ with x_i a variable appearing in a node $\omega \in \Omega$, for which we therefore have $\sigma'(x_i) = f_i(\sigma'(\vec{x}_i)) = \sigma'(f_i(\vec{x}_i))$ by definition of $F^s(\omega)$.
  + $\sigma' \models \Omega'$: For any typing obligation $\omega' \in \Omega'$, we have $\omega' \in F^t(\omega)$ for some $\omega \in \Omega$, so $\omega' = [\langle x_1, \dots, x_n \rangle : (\mathcal{A}, q_j)]$ for some $x_1, \dots, x_n$ such that $\langle \sigma'(x_1), \dots, \sigma'(x_n) \rangle \in \mathcal{R}(\mathcal{A}, q_j)$, by definition of $F^t(\omega)$ and $F^s(\omega)$.
  + $h(\Omega', \sigma') = h(\Omega, \sigma) - 1$: For this case, let $\omega = argmax_{\omega \in \Omega}(h(\sigma, \omega))$ and $\omega' = argmax_{\omega' \in \Omega'}(h(\sigma', \omega'))$. By definition of $F^t(\omega)$ and $F^s(\omega)$, we have both $h(\sigma', \Omega') \geq h(\sigma, \omega) - 1$ and $h(\sigma', \Omega') \leq h(\sigma, \omega) - 1$.

  By induction on $\sigma' \models (E \cup E', \Omega')$, we have that there exists a proof tree T' of $\vdash (E \cup E', \Omega')$ such that $h(T') = h(\sigma', \Omega')$.

  Therefore, take $T = \dfrac{T'}{\vdash (E, \Omega)}\,\textsc{Step}$
  We have that T is a valid proof tree and that $h(T) = h(T') + 1 = h(\Omega, \sigma)$.

◀