



**HAL**  
open science

# Automata-Based Verification of Relational Properties of Functions over Algebraic Data Structures

Théo Losekoot, Thomas Genet, Thomas Jensen

► **To cite this version:**

Théo Losekoot, Thomas Genet, Thomas Jensen. Automata-Based Verification of Relational Properties of Functions over Algebraic Data Structures. FSCD 2023 - 8th International Conference on Formal Structures for Computation and Deduction, Jun 2023, Rome, Italy. pp.1-21, 10.4230/LIPIcs.FSCD.2023.7. hal-04216680

**HAL Id: hal-04216680**

**<https://inria.hal.science/hal-04216680>**

Submitted on 25 Sep 2023

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.



Distributed under a Creative Commons Attribution 4.0 International License

# 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

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  $\text{Len}(\text{Nil}, S(Z))$  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  $\text{Less}(S(S(Z)), S(Z))$ .

<pre> 52 type nat = Z   S of nat     type natlist = Nil   Cons of nat*natlist  53 let rec len (l : natlist) =     match l with       Nil -&gt; Z       Cons(h, t) -&gt; S (len t)  54 let rec less (n : nat) (m : nat) =     match (n, m) with       Z, S(_) -&gt; true       _, Z -&gt; false       S(n1), S(m1) -&gt; less n1 m1  55 <math>\forall x l. \text{less } Z (\text{len } (\text{Cons}(x, l)))</math>     <math>\forall x l. \text{less } (\text{len } l) (\text{len } \text{Cons}(x, l))</math> </pre>	<pre> Len(Nil, Z). Len(<math>\underline{l}</math>, <math>\underline{n}</math>) <math>\Rightarrow</math> Len(Cons(<math>\underline{x}</math>, <math>\underline{l}</math>), S(<math>\underline{n}</math>)). Len(<math>\underline{l}</math>, <math>\underline{n}_1</math>) <math>\wedge</math> Len(<math>\underline{l}</math>, <math>\underline{n}_2</math>) <math>\Rightarrow</math> <math>\underline{n}_1 = \underline{n}_2</math>.  Less(Z, S(<math>\underline{m}</math>)). Less(<math>\underline{n}</math>, Z) <math>\Rightarrow</math> False. Less(<math>\underline{n}</math>, <math>\underline{m}</math>) <math>\iff</math> Less(S(<math>\underline{n}</math>), S(<math>\underline{m}</math>)).  Len(Cons(<math>\underline{x}</math>, <math>\underline{l}</math>), <math>\underline{n}</math>) <math>\Rightarrow</math> Less(Z, <math>\underline{n}</math>). Len(<math>\underline{l}</math>, <math>\underline{n}</math>) <math>\wedge</math> Len(Cons(<math>\underline{x}</math>, <math>\underline{l}</math>), <math>\underline{n}'</math>) <math>\Rightarrow</math> Less(<math>\underline{n}</math>, <math>\underline{n}'</math>). </pre>
---	---

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  $\forall x l. \text{less } Z (\text{len } (\text{Cons}(x, l)))$  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  $\text{Cons}(x, l)$  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,  $\forall x l. \text{less } (\text{len } l) (\text{len } \text{Cons}(x, l))$ , is relational and, thus, out of the scope of the aforementioned approaches. We still have that if  $l \in \mathcal{L}_{lists}$  then  $\text{cons}(x, l) \in \mathcal{L}_{Cons+}$  but for any  $l \in \mathcal{L}_{lists}$  and any  $l' \in \mathcal{L}_{Cons+}$  we cannot prove that  $\text{less } (\text{len } l) (\text{len } l')$ . 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;

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

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

## 95 **2 An overview of the verification procedure on an example**

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

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

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

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

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

129 **First iteration of the learner-teacher algorithm**

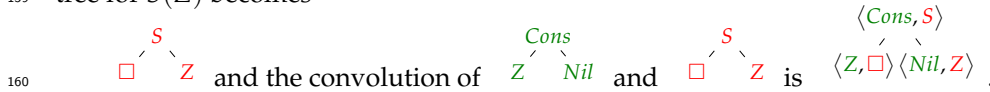
130 The learner proposes the model  $\mathcal{M} = \{\text{Len} \mapsto \mathcal{A}_\emptyset, \text{Less} \mapsto \mathcal{A}_\emptyset\}$ . The teacher checks if  $\mathcal{M}$   
 131 satisfies each clause of  $\mathcal{C}$ , i.e., for each  $\varphi \in \mathcal{C}$  it checks if  $\mathcal{M} \models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

135 **Second iteration of the learner-teacher algorithm**

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

151 **Third iteration of the learner-teacher algorithm: Learner part**

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



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

165 A last phase of the ICE learning process is to reduce the number of states of the automaton  
 166 and, doing so, possibly enlarge the recognized language. Note that this phase was skipped  
 167 on automaton  $\mathcal{A}_1$  because it has only one state. Reducing the number of states consists in  
 168 finding state merging which are coherent w.r.t. the ground clauses sent by the teacher and  
 169 coherent w.r.t. types of recognized languages. For instance, on  $\mathcal{A}_2$ , merging  $q_0$  with  $q_2$  is possi-  
 170 ble because both recognize pairs of lists and natural numbers. On the opposite, merging  $q_0$   
 171 with  $q_1$  is incorrect because  $q_0$  recognize *pairs* of lists and  $q_1$  only recognizes *a unique* natural

172 number (omitting padding). After renaming  $q_2$  to  $q_0$ , transitions of the automaton  $\mathcal{A}_2$  become  
 173  $\{\langle Nil, Z \rangle() \rightarrow q_0, \langle Z, \square \rangle() \rightarrow q_1, \langle Cons, S \rangle(q_1, q_0) \rightarrow q_0\}$ . Note that this automaton now  
 174 recognizes  $\{\langle Nil, Z \rangle, \langle Cons(Z, Nil), S(Z) \rangle, \langle Cons(Z, Cons(Z, Nil)), S(S(Z)) \rangle, \dots\}$ , i.e., all  
 175 pairs  $(l, n)$  where  $l$  is a list of  $Z$  whose length is  $n$ .

### 176 Conclusion of the learner-teacher algorithm

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

## 187 3 Prerequisites

### 188 3.1 Typed alphabet and term

189 ► **Definition 1** (Typed alphabet). A typed alphabet  $(\Sigma, \tau, \Gamma)$  is a set of symbols  $\Sigma$ , a set of types  
 190  $\Gamma$ , and a typing function  $\tau$  which assigns to each symbol  $f$  a type  $\tau(f) = \tau_1 \times \dots \times \tau_n \rightarrow \tau_0$  with  
 191  $\forall i \in \llbracket 0, n \rrbracket, \tau_i \in \Gamma$  and  $n \in \mathbb{N}$  varying for each symbol  $f$ . When  $n = 0$ , the symbol is a constant and  
 192 does not take input. For  $f \in \Sigma$  and  $\tau(f) = \tau_1 \times \dots \times \tau_n \rightarrow \tau_0$ , we say that  $f$  is of arity  $n$ , written  
 193  $|f| = n$ , and that  $\tau_0$  is the output type of  $f$ , written  $\tau_{\text{out}}(f) = \tau_0$ . When clear from context, we  
 194 identify the tuple  $(\Sigma, \tau, \Gamma)$  with  $\Sigma$ .

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

200 ► **Definition 3** (Substitution). A substitution  $\sigma$  is a finite map between variables and terms (which  
 201 may contain variables). The application of a substitution  $\sigma$  to a variable  $x$ , written  $\sigma(x)$ , is defined as  
 202  $t$  if there exists a binding  $(x, t) \in \sigma$  and  $x$  otherwise. The application of a substitution is generalized  
 203 to terms by  $\sigma(f(t_1, \dots, t_n)) = f(\sigma(t_1), \dots, \sigma(t_n))$ . Even more generally, a substitution can be  
 204 applied to any structure containing variables. The composition of substitution, which first applies  $\sigma_1$   
 205 and then  $\sigma_2$ , is written  $\sigma_1; \sigma_2$ . The domain of a substitution is the set of variables for which a binding  
 206 is defined and is written  $\text{dom}(\sigma)$ .

207 A function  $\text{Vars}$  is used without definition, if unambiguous, to fetch the set of variables  
 208 contained in a structure. It can be called, for example, on a term or on a tuple of structures  
 209 containing variables.

### 210 3.2 Tree automaton

211 ► **Definition 4** (Tree automaton). A (bottom-up) tree automaton  $\mathcal{A} = (Q, Q_f, \Delta)$  over an alphabet  
 212  $\Sigma$  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)  $\Delta$



213 such that transitions are of the form  $f(q_1, \dots, q_{|f|}) \rightarrow q_0$ , where  $f \in \Sigma$  and  $\forall i \in \llbracket 0, |f| \rrbracket, q_i \in Q$ .

214 ► **Definition 5** (Language recognized by an automaton). *The set of terms recognized (or ac-*  
 215 *cepted) in a state  $q$  of an automaton  $\mathcal{A}$  is inductively defined as  $\mathcal{L}(\mathcal{A}, q) = \{f(t_1, \dots, t_n) \mid$   
 216  $f(q_1, \dots, q_n) \rightarrow q \in \Delta \wedge \bigwedge_{i \in \llbracket 1, n \rrbracket} t_i \in \mathcal{L}(\mathcal{A}, q_i)\}$ . The language recognized by an automaton is  
 217  $\mathcal{L}(\mathcal{A}) = \bigcup_{q_f \in Q_f} \mathcal{L}(\mathcal{A}, q_f)$ .*

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

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

### 228 3.3 Automata recognizing a relation

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

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

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

$$244 \quad \oplus_L((e_1^1, \dots, e_1^{k_1}), \dots, (e_n^1, \dots, e_n^{k_n})) = ((\overline{e_1^1}, \dots, \overline{e_1^{k_1}}), \dots, (\overline{e_n^1}, \dots, \overline{e_n^{k_n}}))$$

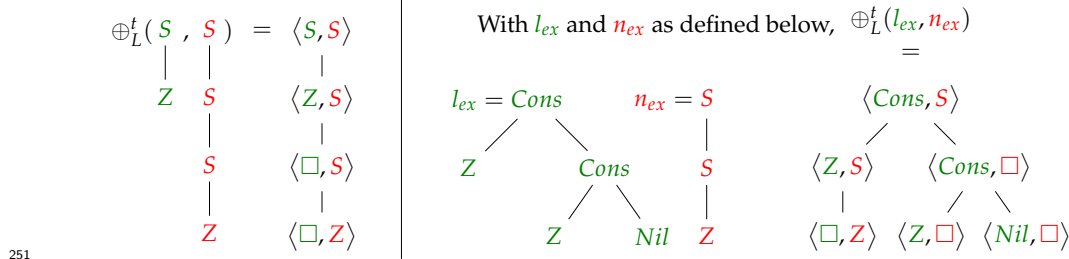
245 with  $k = \max_{i \in \llbracket 1, n \rrbracket} (k_i)$  and  $\forall i \in \llbracket 1, n \rrbracket, \forall j \in \llbracket 1, k \rrbracket, \overline{e_i^j} = e_i^j$  if  $j \leq k_i$  and  $\square$  otherwise

246

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

$$\oplus_L^t(f_1(\vec{t}_1), \dots, f_n(\vec{t}_n)) = \langle f_1, \dots, f_n \rangle (\oplus_L^t(\vec{t}_1), \dots, \oplus_L^t(\vec{t}_k)) \text{ with } (\vec{t}_1, \dots, \vec{t}_k) = \oplus_L(\vec{t}_1, \dots, \vec{t}_n)$$

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

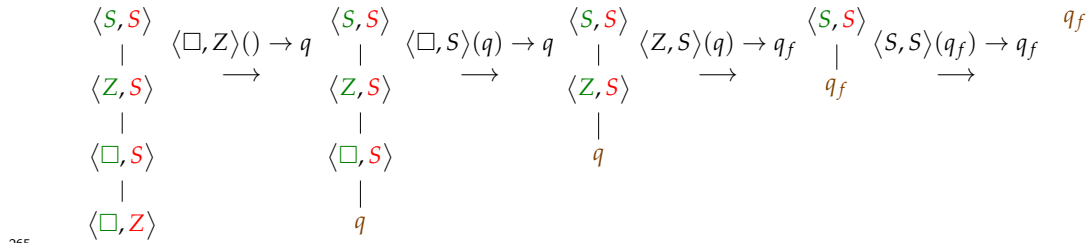


252 Note that, due to type constraints,  $\mathcal{T}(\Sigma_{\square}) = \mathcal{T}(\Sigma) \cup \{\square\}$ . The left-convolution  $\oplus_L^t$  of  $n$   
 253 terms is an isomorphism between  $\mathcal{T}(\Sigma_{\square})^n$  and  $\mathcal{T}(\Sigma_{\oplus_L})$ . Automata recognizing convoluted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

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

258 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  
 259  $\forall i \in \llbracket i, n \rrbracket, \tau_i \neq \tau_{\square}$ . This ensures that an automaton defines a relation between terms of  
 260  $\mathcal{T}(\Sigma)$ , i.e. terms without padding.

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



### 266 Convolutions and their expressivity

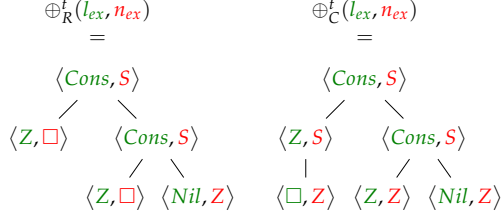
267 Which relations are representable by convoluted tree automaton highly depends on the  
 268 precise datatypes definition. For example, when using the left-convolution, the Len relation  
 269 can only be represented if the Cons constructor had its arguments swapped. This is because  
 270 left-convoluting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



278 depending on the constructor argument's order and being able to duplicate terms, but the  
 279 drawback of generating big convoluted terms. Both convolution are extended to terms in  
 280 the same way  $\oplus_L$  was.

281 ► **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.



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

285 **4 Functional programs and their logical representation**

286 **Regular models of functional programs**

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

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

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

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

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

## 323 5 Model-checking of regular structures

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

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

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

$$\begin{aligned} \sigma \models (E, \Omega) &\doteq \sigma \models \Omega \wedge \sigma \models E \quad \text{with} \\ \sigma \models \Omega &\doteq (\forall [\vec{x} : (\mathcal{A}, q)] \in \Omega, \sigma(\vec{x}) \in \mathcal{R}(\mathcal{A}, q)) \quad \text{and} \\ \sigma \models E &\doteq (\forall (x = f(\vec{y})) \in E, \sigma(x) = f(\sigma(\vec{y}))) \end{aligned}$$

339 ► **Definition 15** (Coherence of a constraint set). A set of constraints  $E$  is said to be coherent if it  
 340 admits a syntactic unifier. The most general unifier (MGU) of a coherent set  $E$  is written  $\sigma_E$ .

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

343 ► **Definition 16** (Model checking as type checking).

344 Let some CHC formula  $\varphi = \psi(\vec{x}) \wedge R_1(\vec{x}_1) \wedge \dots \wedge R_n(\vec{x}_n) \Rightarrow R_0(\vec{x}_0)$  and model  $\mathcal{M}$ .

345 The set of typing problems associated to  $\varphi$  and  $\mathcal{M}$  is  $tp(\varphi, \mathcal{M}) = \{(\psi(\vec{x}), \Omega) \mid \Omega \in \Omega_s\}$  with

$$\begin{aligned} 346 \quad \Omega_s &= \left\{ \{[\vec{x}_1 : (\mathcal{A}_1, q_1)], \dots, [\vec{x}_n : (\mathcal{A}_n, q_n)], [\vec{x}_0 : (\mathcal{A}_0, q_0)]\} \mid \right. \\ 347 \quad &\left. \mathcal{A}_1 = \mathcal{M}(R_1) \wedge \dots \wedge \mathcal{A}_n = \mathcal{M}(R_n) \wedge \mathcal{A}_0 = \overline{\mathcal{M}(R_0)} \wedge \forall i \in \llbracket 0, n \rrbracket, q_i \in Q_f(\mathcal{A}_i) \right\} \\ 348 \end{aligned}$$

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

353 ► **Theorem 17** (Model checking as type checking).

354 For each model  $\mathcal{M}$  and CHC property  $\varphi$ ,  $\mathcal{M} \not\models \varphi \iff \exists \sigma, \exists (E, \Omega) \in tp(\mathcal{M}, \varphi)$ ,  $\sigma \models$   
 355  $(E, \Omega)$ .

356 ► **Example 18** (Model checking a property). Let  $\varphi$  be  $\text{Len}(l, n) \Rightarrow \text{Even}(n)$ , a formula  
 357 stating that all lists are of even length. Let  $\mathcal{M} = \{\text{Len} \mapsto \mathcal{A}_{\text{Len}}, \text{Even} \mapsto \mathcal{A}_{\text{Even}}\}$  where  
 358  $\mathcal{A}_{\text{Len}}$  and  $\mathcal{A}_{\text{Even}}$  respectively define the length relation on integer lists and the even pre-  
 359 dicate of integers.  $\mathcal{A}_{\text{Len}}$  has states  $\{q_f, q\}$ , final states  $\{q_f\}$ , and rules  $\{(A) : \langle Z, \square \rangle () \rightarrow$   
 360  $q, (B) : \langle S, \square \rangle (q) \rightarrow q, (C) : \langle \text{Cons}, S \rangle (q, q_f) \rightarrow q_f, (D) : \langle \text{Nil}, Z \rangle () \rightarrow q_f\}$ .  $\mathcal{A}_{\text{Even}}$   
 361 has states  $\{q_e, q_o\}$ , final states  $\{q_e\}$ , and rules  $\{(1) : \langle Z \rangle () \rightarrow q_e, (2) : \langle S \rangle (q_o) \rightarrow$   
 362  $q_e, (3) : \langle S \rangle (q_e) \rightarrow q_o\}$ .

363 To check whether  $\mathcal{M} \not\models \varphi$ , we first translate  $(\mathcal{M}, \varphi)$  into a typing problem instance. Note  
 364 that Even appears in the head of the property  $\varphi$ , therefore we will need to complement  
 365  $\mathcal{A}_{\text{Even}}$ . We write its complement  $\mathcal{A}_{\text{Odd}}$ , which is the same automaton but with final states  
 366  $\{q_o\}$ .

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

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

## 370 5.1 Proof system

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

380 ► **Definition 19** (Unfolding a typing obligation).

381  $unfold([\langle x_1, \dots, x_n \rangle : (\mathcal{A}, q)]) = \{(E_r, \Omega_r) \mid r \in \Delta(\mathcal{A}) \wedge r = \langle f_1, \dots, f_n \rangle (q_1, \dots, q_k) \rightarrow q\}$   
 382 with  $E_r = \{x_i = f_i(\vec{x}_i) \mid i \in \llbracket 1, n \rrbracket\}$  and  $\Omega_r = \{[\langle \circ(\vec{x}_1, \dots, \vec{x}_n) \rangle [j] : (\mathcal{A}, q_j)] \mid j \in \llbracket 1, k \rrbracket\}$  where  
 383  $\forall i \in \llbracket 1, n \rrbracket$ ,  $\vec{x}_i$  are fresh variables.

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

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

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

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

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

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

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

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

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

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

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

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

$$\begin{array}{c} \text{CONCLUDE} \frac{}{\vdash (E, \emptyset)} \quad \text{STEP} \frac{\vdash (E \cup E', \Omega')}{\vdash (E, \Omega)} \\ \text{402} \quad \text{if Coherent}(E) \quad \text{if Coherent}(E \cup E') \text{ and } (E', \Omega') \in \text{unfolds}(\Omega) \\ \text{403} \\ \text{404} \end{array}$$

405 ► **Example 24.** Continuing example 20, we build a proof tree of  $\vdash (E_0, \Omega_0)$ . Rule CON-  
406 CLUDE cannot be immediately applied, so let us consider STEP, and thus  $\text{unfolds}(\Omega_0)$ .

407 Its element  $(E_{(3)} \cup E_{(D)}, \Omega_{(3)} \cup \Omega_{(D)})$  can be discarded because  $E_{(3)} \cup E_{(D)}$  is con-  
408 tradictory, as both constraints  $n = Z$  and  $n = S(m)$  are present. Its other element,  
409  $(E_{(3)} \cup E_{(C)}, \Omega_{(3)} \cup \Omega_{(C)})$ , is coherent, so we can apply the STEP rule. We write it  $(E_1, \Omega_1)$   
410 where  $E_1 = \{l = \text{Cons}(l_1, l_2), n = S(n_1), n = S(m)\}$  and  $\Omega_1$  is the set of typing oblig-  
411 ations  $\Omega_1 = \{[\langle l_1, \square \rangle : (\mathcal{A}_{\text{Len}}, q_n)], [\langle l_2, n_1 \rangle : (\mathcal{A}_{\text{Len}}, q_f)], [\langle m \rangle : (\mathcal{A}_{\text{Odd}}, q_e)]\}$ . We now  
412 have the new typing problem  $(E_0 \cup E_1, \Omega_1)$ . Rule CONCLUDE still cannot be applied. Then,  
413  $\text{unfolds}(\Omega_1)$  has 8 elements, only 4 of which are coherent. Its four coherent element can be  
414 seen as two times the almost-same two elements, the only difference being which rule has  
415 been applied to  $[\langle l_1, \square \rangle : (\mathcal{A}_{\text{Len}}, q_n)]$ . For this example, we only show the two elements that  
416 used rule (A),  $(E_2, \Omega_2)$  and  $(E'_2, \Omega'_2)$  with

$$\begin{array}{c} \text{417} \quad E_2 = \{l_1 = Z, l_2 = \text{Nil}, n_1 = Z, m = Z\}, \quad \Omega_2 = \emptyset, \\ \text{418} \quad E'_2 = \{l_1 = Z, l_2 = \text{Cons}(l_{21}, l_{22}), n_1 = S(n_{11}), m = S(m_1)\}, \\ \text{419} \quad \Omega'_2 = \{[\langle l_{21}, \square \rangle : (\mathcal{A}_{\text{Len}}, q_n)], [\langle l_{22}, n_{11} \rangle : (\mathcal{A}_{\text{Len}}, q_f)], [\langle m_1 \rangle : (\mathcal{A}_{\text{Odd}}, q_o)]\} \\ \text{420} \end{array}$$

421 Constraints  $E_1 \cup E_2$  are coherent and  $\Omega_2$  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  $\{n \mapsto S(Z), l \mapsto \text{Cons}(Z, \text{Nil})\}$ . 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.

$$\begin{array}{c} \text{CONCLUDE} \frac{}{\vdash (E_1 \cup E_2, \emptyset)} \\ \text{STEP} \frac{}{\vdash (E_1, \Omega_1)} \\ \text{STEP} \frac{}{\vdash (\emptyset, \Omega_0)} \end{array}$$

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

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

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

431 ► **Theorem 26** (Proof system is correct and complete).

432 We have  $\forall (E, \Omega), (\exists \sigma, \sigma \models (E, \Omega)) \iff \vdash (E, \Omega)$ . More precisely, for any  $(E, \Omega)$  and  
 433  $n \in \mathbb{N}$ ,

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

438 The proof can be found in Appendix A.

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

## 442 6 Proof search procedure

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

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

► **Definition 28** (Independence). Let  $(E, \Omega)$  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

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

452 Therefore, any two solutions  $\sigma'_a$  of  $(E, \Omega_a)$  and  $\sigma'_b$  of  $(E, \Omega_b)$  with  $\Omega_a \parallel^E \Omega_b$  can first  
 453 be factorized by  $\sigma_E$  by letting  $\sigma_a$  and  $\sigma_b$  such that  $\sigma'_a = \sigma_E; \sigma_a$  and  $\sigma'_b = \sigma_E; \sigma_b$  and then  
 454 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  
 455 partitioning of  $(E, \Omega)$  into independent sub-problems is hard, as it may require to examine  
 456 the shape of automata. We define below a safe and easy-to-compute approximation of these  
 457 independence classes that splits typing obligations whose variables cannot be related even  
 458 using the equalities of  $E$ .

459 ► **Definition 29** (Splitting). Let  $E$  be a set of constraints. Let  $V_E([\vec{x} : (\mathcal{A}, q)]) \doteq \text{Vars}(\sigma_E(\vec{x}))$ .  
 460 The set  $V_E([\vec{x} : (\mathcal{A}, q)])$  is the set of variables remaining in a typing obligation after application  
 461 of the most general unifier  $\sigma_E$  of  $E$ . Note how  $(\mathcal{A}, q)$  has not been used. We define  $D_E \subseteq \Omega \times \Omega$   
 462 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  
 463 closure  $D_E^*$  is an equivalence relation. We define the function  $\text{Split}(E, \Omega)$  to return the equivalence  
 464 classes of  $D_E^*$  defined on  $\Omega$ .

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

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

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

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

470 **► Example 31** (Splitting  $(E_1, \Omega_1)$ ). In example 24, we had  $E_1 = \{l = \text{Cons}(l_1, l_2), n =$   
 471  $S(n_1), n = S(m)\}$  and  $\Omega_1 = \{\omega_1, \omega_2, \omega_3\}$  with  $\omega_1 = [\langle l_1, \square \rangle : (\mathcal{A}_{\text{Len}}, q_n)]$ , with  $\omega_2 =$   
 472  $[\langle l_2, n_1 \rangle : (\mathcal{A}_{\text{Len}}, q_f)]$ , and  $\omega_3 = [\langle m \rangle : (\mathcal{A}_{\text{Odd}}, q_e)]$ . We have  $\sigma_{E_1} = \{l \mapsto \text{Cons}(l_1, l_2), n \mapsto$   
 473  $S(n'), n_1 \mapsto n', m \mapsto n'\}$ ,  $V_{E_1}(\omega_1) = \{l_1\}$ ,  $V_{E_1}(\omega_2) = \{l_2, n'\}$ , and  $V_{E_1}(\omega_3) = \{n'\}$ .  
 474 Therefore  $\text{Split}(E_1, \Omega_1) = \{\{\omega_1\}, \{\omega_2, \omega_3\}\}$ .

475 Solving  $\omega_1$  have no impact on the solving of  $\omega_2$  and  $\omega_3$  because the values that  $l_1$  can take  
 476 do not influence the values that  $l_2, n_1$ , or  $m_2$  can take. On the other hand, because of  $E_1$ ,  
 477  $m$  and  $n_1$  must take the same value, and therefore typing obligations  $\omega_2$  and  $\omega_3$  cannot be  
 478 separated. Note that applying this SPLIT rule before the second STEP (of example 24) would  
 479 have separated  $(E_1, \Omega_1)$  into two independent problems.

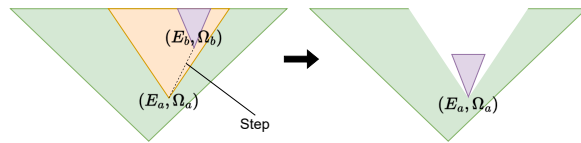
480 The second optimisation consists in *pruning the search tree*. The search space is, for almost  
 481 all typing problems, infinite. Without pruning, it would be impossible to cover the whole  
 482 search space, and therefore negative instances would (almost) all never terminate. Pruning  
 483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

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

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

488 **► Theorem 33** (Safety of pruning). For any proof tree that contains a prunable node, there exist a  
 489 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  $T$   
 490 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).



491 **Proof.** Let  $T$  be a prunable tree, that is such that there exists nodes  $\vdash (E_a, \Omega_a)$  and  $\vdash (E_b, \Omega_b)$   
 492 with respective proof trees  $T_a$  and  $T_b$ , with  $T_b$  a sub-tree of  $T_a$  with a STEP rule between  
 493  $\vdash (E_a, \Omega_a)$  and  $\vdash (E_b, \Omega_b)$ , and  $\sigma$  a substitution such that  $\sigma(\sigma_{E_a}(\Omega_a)) \subseteq \sigma_{E_b}(\Omega_b)$ .

494 By theorem 26(A) there exists a substitution  $\sigma_b$  with  $\sigma_b \models (E_b, \Omega_b)$  and  $h(\sigma_b, \Omega_b) = h(T_b)$ .  
 495 Because  $\sigma_{E_b}$  is the most general unifier of  $E_b$  and  $\sigma_b \models E_b$ , there exists  $\sigma'$  such that  $\sigma_b = \sigma_{E_b}; \sigma'$ .  
 496 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  
 497 also have  $\sigma_a \models \Omega_a$ . Because  $\sigma_a$  first applies  $\sigma_{E_a}$ , we have  $\sigma_a \models E_a$ . Therefore  $\sigma_a \models (E_a, \Omega_a)$ .  
 498 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  
 499 theorem 26(B) there exists a proof  $T'_a$  of  $\vdash (E_a, \Omega_a)$  with  $h(T'_a) = h(\sigma_a, \Omega_a) \leq h(\sigma_b, \Omega_b) =$   
 500  $h(T_b)$ .

501 Therefore, the proof tree  $T$  whose sub-tree  $T_a$  has been replaced by  $T'_a$  is valid and smaller.  
 502 Besides, we know that the sub-tree  $T'_a$  is strictly smaller than  $T_a$  because  $T_a$  contains at least



503 one application of the STEP rule between its root and  $T_b$ . Therefore, this transformation  
 504 strictly decreases the size of the proof tree. ◀

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

508 ▶ **Example 35** (Pruning of the search tree). During the second STEP application of example 24,  
 509 the typing problem  $(E'_2, \Omega'_2)$  is also in *unfolds* $(\Omega_1)$ . This was no problem, as the algorithm  
 510 found a solution and stopped. Now, if (for example) automaton  $\mathcal{A}_{\text{Len}}$  did not have rule  
 511  $(D)$ , then there would be no solution to the initial typing problem  $(E_0, \Omega_0)$ . The search  
 512 would never stop, as, after a bit of unification and renaming,  $(E_0, \Omega_0)$  can be included in  
 513  $(E_1 \cup E'_2, \Omega'_2)$ . Without pruning, the typing algorithm could therefore loop forever instead of  
 514 returning *None*. Fortunately,  $(E_1 \cup E'_2, \Omega'_2)$  can be pruned by taking  $\sigma = \{l \mapsto l_{22}, n \mapsto n_{11}\}$ ,  
 515 as  $\sigma(\sigma_0(\Omega_0)) \subseteq \sigma_2(\Omega'_2)$  (with  $\sigma_0$  and  $\sigma_2$  being most general unifiers of  $E_0$  and  $E_0 \cup E_1 \cup E'_2$ ,  
 516 respectively).

## 517 7 Regular structure inference

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

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

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

537 ▶ **Definition 36** (Working model generation). *The working model  $\mathcal{W}$  of a given finite set of ground*  
 538 *CHCs  $\underline{\mathcal{C}}$  is the smallest model (up to state renaming) recognizing exactly the terms mentioned in  $\underline{\mathcal{C}}$*   
 539 *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  $\mathcal{W}(R)$*   
 540 *such that  $\mathcal{R}(\mathcal{W}(R), q) = \{\vec{t}\}$ .*

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

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

556 A minimal solution  $\llbracket \cdot \rrbracket$  to the state merging problem can be computed by a finite model  
 557 finder. We write  $\llbracket \text{Final} \rrbracket$  for the set of final states of the solution and  $\llbracket c_q \rrbracket$  for the equivalence  
 558 class of constant  $c_q$ .

559 ► **Definition 38** (Generalisation of working model). *Given a solution  $\llbracket \cdot \rrbracket$  to the state merging*  
 560 *problem, we generalise the working model  $\mathcal{W}$  by  $\mathcal{M}$  with  $\mathcal{M}(R) = (Q, Q_f, \Delta)$  with  $Q = \{\llbracket c_q \rrbracket \mid$   
 561  $q \in Q(\mathcal{W}(R))\}$ ,  $Q_f = Q \cap \llbracket \text{Final} \rrbracket$  and  $\Delta = \{\vec{f}(\llbracket c_{q_1} \rrbracket, \dots, \llbracket c_{q_n} \rrbracket) \rightarrow \llbracket c_{q_0} \rrbracket \mid \vec{f}(q_1, \dots, q_n) \rightarrow$   
 562  $q_0 \in \Delta(\mathcal{W}(R))\}$ .*

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

571 The minimisation problem is therefore on the signature with unary predicate  $\text{Final}$  and  
 572 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  
 573 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  
 574 they are the only states of the automaton's type,  $\text{natlist} \times \text{nat}$ . The constraints  $\mathcal{C}_f$ , generated  
 575 from  $\underline{\mathcal{C}}$ , are  $\{\text{Final}(c_{q_{l_0}}), \text{Final}(c_{q_{l_0}}) \Rightarrow \text{Final}(c_{q_{l_1}})\}$ . The smallest model is a two-elements  
 576 set  $\{q_l, q_z\}$ , with  $\llbracket \text{Final} \rrbracket = \{q_l\}$ ,  $\llbracket q_{l_0} \rrbracket = \llbracket q_{l_1} \rrbracket = q_l$ , and  $\llbracket q_n \rrbracket = q_z$ .

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

## 582 8 Approximation

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

588 Our verification goals are CHCs of the form  $\psi(\vec{x}) \wedge R_1(\vec{x}_1) \wedge \dots \wedge R_n(\vec{x}_n) \Rightarrow R_0(\vec{x}_0)$ .  
 589 Given a relation  $R$  we denote by  $R^+$  (resp.  $R^-$ ) an over-approximation (resp. under-  
 590 approximation) of  $R$  which can also be  $R$  itself. A safe way to prove the above implication

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

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

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

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

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

## 623 9 Implementation and Experiments

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

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

635 The solver presented in this paper builds regular relations, as opposed to [11, 16, 17]  
 636 which only build regular sets of terms. Since regular sets are a particular case of regular

637 relations, our solver should be able to handle the examples covered by those techniques,  
 638 plus some relational problems. As a result, for the experiments, we choose some examples  
 639 coming from benchmarks of Timbuk [11], add relational examples taken from the Isaplanner  
 640 benchmark [4,7] and built relational problems inspired by TIP [4,5]. As shown in Section 2, a  
 641 typical property which can be automatically proved by those non-relational solvers [11,16,17]  
 642 is of the form  $\forall x l. \text{less } Z (\text{len } (\text{Cons}(x, l)))$  where  $l$  is any list of natural numbers.

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

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

657 -  $\forall (l : \text{ablist}). (\text{len } l) = (\text{len } (\text{reverse } l))$  length\_reverse\_eq.smt2  
 658 -  $\forall (l_1 : \text{ablist}) (l_2 : \text{ablist}). (\text{prefix } l_1 (\text{append } l_1 l_2))$  prefix\_append.smt2  
 659 -  $\forall (l : \text{ablist}). (\text{len } l) = (\text{len } (\text{sort } l))$  sort\_length\_eq.smt2  
 660 -  $\forall (i : \text{nat}) (t_1 : \text{natbintree}) (t_2 : \text{natbintree}). t_1 \neq (\text{node } i t_1 t_2)$  tree\_add\_not\_eq.smt2

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

662 -  $\forall (n : \text{nat}). n < (\text{double } n)$  nat\_double\_is\_le.smt2  
 663 -  $\forall (x : \text{ab}) (l : \text{ablist}). (\text{delete\_one } x l) = (\text{delete\_all } x l)$  list\_delete\_all\_count.smt2  
 664  $\Rightarrow (\text{count } x l) = 1$

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

667 -  $\forall (x : \text{ab}) (l : \text{ablist}). (\text{delete\_one } x l) = (\text{delete\_all } x l)$  list\_delete\_all\_count.smt2  
 668  $\Rightarrow (\text{count } x l) \leq 1$   
 669 -  $\forall (n : \text{nat}) (m : \text{nat}). n + m = m + n$  plus\_commutative.smt2

670 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  
 671 mostly either on same-type relations or on lists and natural numbers, the right-convolution  
 672 was the most efficient of convolution type. Left-convolution is not adapted for most of the  
 673 list-based examples and complete-convolution revealed to be too costly in practice though it  
 674 should help to prove properties on functions manipulating trees. On a total of 120 examples,  
 675 our solver (using right-convolution) proves 66, disproves 23, and timeouts on 31 after 60s.  
 676 Our solver succeeds on 20 out of the 79 first-order Isaplanner examples in less than 60s (and  
 677 18 in less than 5s). Our solver reveals to be more efficient on examples where a single level  
 678 of structure have to be compared, i.e., natural numbers, lists of arbitrary elements, etc. It is  
 679 generally unsuccessful on examples mixing several layers of structure, e.g., lists of natural  
 680 numbers, or on examples where a precise counting is required to prove the property. Finally,  
 681 on examples where using a non-relational model suffices to prove the property, our solving  
 682

683 technique is flexible enough to find such a model, with an efficiency comparable with  
 684 non-relational solvers. For instance, on 11 examples coming from the Timbuk benchmarks,  
 685 we proved 6 of them (with execution times around 2 seconds), disproved 3, and have a  
 686 timeout on the 2 last.

## 687 **10** Related work

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

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

704 By manually writing ss-NFAs, authors of [18] are able to benchmark their verification  
 705 procedure. Our approach and theirs seems to be complementary as they succeed on different  
 706 sets of examples. This can be observed on the IsaPlanner benchmark where our technique  
 707 fails on most of examples that [18] handles (i.e. 4, 5, 15, 16, 29, 30, 39, 42, 50, 62, 67, 71, 86)  
 708 and succeeds on examples on which they do not report any success (i.e. 17, 18, 21, 22, 23, 24,  
 709 25, 26, 31, 32, 33, 34, 45, 46, 65, 69).

710 In [3], the authors present an expressive formalism for representing relations between  
 711 trees called *synchronized context-free programs*. This formalism is more expressive than  
 712 convoluted tree automata presented here. In particular, it can represent languages of the  
 713 form  $\{(g^n(a), g^n(b)) \mid n \in \mathbb{N}\}$  (like convoluted tree automata) and also languages of the  
 714 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  
 715 tree automata). This formalism is used to precisely approximate the set of outputs of a  
 716 term rewriting system. However, [3] does not show how to automatically infer such a  
 717 representation from the term rewriting system.

## 718 **11** Conclusion and future work

719 This paper demonstrates that it is possible to use tree automata as a basis for analysing  
 720 the input-output behaviour of a first-order functional program. This shows that existing  
 721 automata-based techniques for approximating the set of reachable states of a function can be  
 722 extended to also compute relations between input and output of a function. Such relational  
 723 analysis is key to scaling static analyses to larger programs, because it enables a modular,  
 724 function-by-function analysis technique. The extension to relational analysis is based on  
 725 the notion of tree automata convolution. We argue that the standard left-convolution can  
 726 be complemented by other convolution techniques in order to verify more properties of



727 programs. Another technical contribution of the paper is the proof tree pruning used for  
 728 verifying models of constrained Horn clauses. An efficient implementation of this proof  
 729 search has been an essential part of the counter-example guided learner-teacher algorithm  
 730 for inferring models from the CHC representation of the program to be analysed. This is  
 731 confirmed by the benchmark used to evaluate our implementation of the verifier.

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

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

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

## 750 ——— References ———

- 751 1 Clark Barrett, Igor Shikanian, and Cesare Tinelli. An abstract decision procedure for a theory  
 752 of inductive data types. *Journal on Satisfiability, Boolean Modeling and Computation*, 3(1-2):21–46,  
 753 2007.
- 754 2 Nikolaj Bjørner, Arie Gurfinkel, Ken McMillan, and Andrey Rybalchenko. Horn clause solvers  
 755 for program verification. In *Fields of Logic and Computation II*, pages 24–51. Springer, 2015.
- 756 3 Yohan Boichut, Jacques Chabin, and Pierre Réty. Towards more precise rewriting approximations.  
 757 *J. Comput. Syst. Sci.*, 104:131–148, 2019.
- 758 4 Koen Claessen, Moa Johansson, Dan Rosén, and Nicholas Smallbone. Tip and isaplanner  
 759 benchmarks, 2015. <https://tip-org.github.io/>.
- 760 5 Koen Claessen, Moa Johansson, Dan Rosén, and Nicholas Smallbone. Tip: tons of inductive  
 761 problems. In *International Conference on Intelligent Computer Mathematics*, pages 333–337. Springer,  
 762 2015.
- 763 6 Hubert Comon, Max Dauchet, Rémi Gilleron, Florent Jacquemard, Denis Lugiez, Christof  
 764 Löding, Sophie Tison, and Marc Tommasi. *Tree Automata Techniques and Applications*. 2008. URL:  
 765 <https://hal.inria.fr/hal-03367725>.
- 766 7 Lucas Dixon and Jacques Fleuriot. Isaplanner: A prototype proof planner in isabelle. In *CADE’03*,  
 767 volume 2741, pages 279–283. Springer, 2003.
- 768 8 Pranav Garg, Christof Löding, P Madhusudan, and Daniel Neider. Ice: A robust framework  
 769 for learning invariants. In *International Conference on Computer Aided Verification*, pages 69–87.  
 770 Springer, 2014.
- 771 9 Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. *Answer Set Solving*  
 772 *in Practice*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan &  
 773 Claypool Publishers, 2012.
- 774 10 Erich Grädel. Automatic structures: twenty years later. In *Proceedings of the 35th Annual*  
 775 *ACM/IEEE Symposium on Logic in Computer Science*, pages 21–34, 2020.



- 776 11 Timothée Haudebourg, Thomas Genet, and Thomas Jensen. Regular Language Type Inference  
777 with Term Rewriting. *Proceedings of the ACM on Programming Languages*, 4(ICFP):1–29, 2020.
- 778 12 Timothée Haudebourg. *Automatic Verification of Higher-Order Functional Programs using Regular  
779 Tree Languages*. PhD thesis, Univ. Rennes1, 2020.
- 780 13 Tirza Hirst and David Harel. More about recursive structures: Descriptive complexity and  
781 zero-one laws. In *Proceedings 11th Annual IEEE Symposium on Logic in Computer Science*, pages  
782 334–347. IEEE, 1996.
- 783 14 R Hodgson Bernard. *Théories décidables par automate fini (Decidable theories via finite automata)*.  
784 PhD thesis, Ph.D. thesis Département de Mathématiques et de Statistique, Université de . . . ,  
785 1976.
- 786 15 Bakhadyr Khossainov and Anil Nerode. Automatic Presentations of Structures. In *International  
787 Workshop on Logic and Computational Complexity*, pages 367–392. Springer, 1994.
- 788 16 Yurii Kostyukov, Dmitry Mordvinov, and Grigory Fedyukovich. Beyond the elementary rep-  
789 resentations of program invariants over algebraic data types. In Stephen N. Freund and Eran  
790 Yahav, editors, *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language  
791 Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, pages 451–465. ACM, 2021.
- 792 17 Yuma Matsumoto, Naoki Kobayashi, and Hiroshi Unno. Automata-based abstraction for auto-  
793 mated verification of higher-order tree-processing programs. In *Asian Symposium on Programming  
794 Languages and Systems*, pages 295–312. Springer, 2015.
- 795 18 Takumi Shimoda, Naoki Kobayashi, Ken Sakayori, and Ryosuke Sato. Symbolic automatic  
796 relations and their applications to SMT and CHC solving. In *International Static Analysis  
797 Symposium*, pages 405–428. Springer, 2021.
- 798 19 Takeshi Tsukada and Hiroshi Unno. Software model-checking as cyclic-proof search. *Proceedings  
799 of the ACM on Programming Languages*, 6(POPL):1–29, 2022.
- 800 20 Hiroshi Unno, Sho Torii, and Hiroki Sakamoto. Automating induction for solving horn clauses.  
801 In *International Conference on Computer Aided Verification*, pages 571–591. Springer, 2017.

## 802 **A** Appendix

### 803 Proof of theorem 26

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

806 - case CONCLUDE:

807 By hypothesis, we have that  $T$  is of the form  $\frac{}{\vdash (E, \emptyset)}$  with  $Coherent(E)$ , and therefore  $n =$

808 0. Take  $\sigma = \sigma_E$  a most general unifier of  $E$ , which is well-defined, as  $E$  is coherent. We have:

809 (i)  $\sigma \models E$  is immediate, as  $\sigma$  unifies  $E$ ; (ii)  $\sigma \models \Omega$  is trivial, as  $\Omega = \emptyset$ ; (iii)  $h(\Omega, \sigma) = 0 = n$ ,  
810 as  $\Omega$  is empty.

811 - case STEP:

812 By hypothesis, we have that  $T$  is of the form  $\frac{T'}{\text{STEP } \vdash (E, \Omega)}$  with  $T'$  of the form  $\frac{\dots}{\vdash (E \cup E', \Omega')}$   
813 and  $(E', \Omega') \in unfolds(\Omega)$ . By induction, we have that there exists  $\sigma'$  with  $\sigma' \models (E \cup$   
814  $E', \Omega')$  and  $h(\Omega', \sigma') = h(T')$ . We also know that  $h(T') = n - 1$ . Take  $\sigma = \sigma'$ . Then:

815 +  $\sigma \models E$  : Immediate by  $\sigma' \models E \cup E'$  and monotonicity of first-order logic.

816 +  $\sigma \models \Omega$  : 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  
817 sufficient (and necessary) to show that there exists a rule  $r = \vec{f}(\vec{q}) \rightarrow q$  of  $\mathcal{A}$  such that

818 \*  $\forall i \in [1, |\vec{f}|], \sigma(x_i) = f_i(\vec{y}_i)$  for some variables  $\vec{y}_i$  ;

819 \*  $\forall j \in [1, |\vec{q}|], \sigma \models [\bigcirc(\vec{y}_1, \dots, \vec{y}_{|\vec{q}|})[j] : (\mathcal{A}, q_j)]$ .

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

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

828

### 829 Proof of B.

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

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

833 - case  $h(\Omega, \sigma) = 0$ : This is only possible when  $\Omega = \emptyset$ . Take  $T = \frac{\text{CONCLUDE}}{\vdash (E, \Omega)}$ . This proof  
 834 tree  $T$  is correct, as  $\Omega = \emptyset$  and  $E$  is coherent (because  $\sigma \models E$ ). Also  $h(T) = 0$ .

835 - case  $h(\Omega, \sigma) > 0$ :

836 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  
 837 associated rule  $r_\omega = \langle f_1, \dots, f_n \rangle (q_1, \dots, q_k) \rightarrow q$  such that

838 +  $\forall i \in \llbracket 1, n \rrbracket, \sigma(x_i) = f_i(\vec{t}_i)$  for some terms  $\vec{t}_i$ ;

839 +  $\forall j \in \llbracket 1, k \rrbracket, \bigcirc(\vec{t}_1, \dots, \vec{t}_n)[j] \in \mathcal{R}(\mathcal{A}, q_j)$ .

840 Therefore we can build three functions,  $F^c, F^t, F^s$ , which assign to each such typing  
 841 obligation and rule the following:

842 +  $F^c(\omega) = \{x_1 = f_1(\vec{x}_1), \dots, x_n = f_n(\vec{x}_n)\}$ , with  $\forall i \in \llbracket 1, n \rrbracket, \vec{x}_i$  are fresh variables.

843 +  $F^t(\omega) = \{\bigcirc(\vec{x}_1, \dots, \vec{x}_n)[j] : (\mathcal{A}, q_j) \mid j \in \llbracket 1, k \rrbracket\}$

844 +  $F^s(\omega) = \{(x_i^j, t_i^j) \mid x_i = f_i(x_i^1, \dots, x_i^m) \in F^c(\omega) \wedge j \in \llbracket 1, m \rrbracket \wedge \sigma(x_i) = f(t_i^1, \dots, t_i^m)\}$

845 Let  $E' = \bigcup_{\omega \in \Omega} F^c(\omega)$  and  $\Omega' = \bigcup_{\omega \in \Omega} F^t(\omega)$ . Note that  $(E', \Omega') \in \text{unfolds}(\Omega)$ .

846 Let  $\sigma' = \sigma \cup \bigcup_{\omega \in \Omega} F^s(\omega)$ . We have:

847 +  $\sigma'$  is well-defined: Any binding of  $\sigma'$  which is not in  $\sigma$  is of the form  $x_i^j = \sigma(t_i^j)$  for  
 848 some fresh variable  $x_i^j$ . Therefore, as  $\sigma$  is well-defined, so is  $\sigma'$ .

849 +  $\sigma' \models E \cup E'$ : We have  $\sigma \subseteq \sigma'$ , therefore  $\sigma' \models E$ . Any constraint of  $E'$  is of the form  
 850  $x_i = f_i(\vec{x}_i)$  with  $x_i$  a variable appearing in a node  $\omega \in \Omega$ , for which we therefore have  
 851  $\sigma'(x_i) = f_i(\sigma'(\vec{x}_i)) = \sigma'(f_i(\vec{x}_i))$  by definition of  $F^s(\omega)$ .

852 +  $\sigma' \models \Omega'$ : For any typing obligation  $\omega' \in \Omega'$ , we have  $\omega' \in F^t(\omega)$  for some  $\omega \in \Omega$ ,  
 853 so  $\omega' = [\langle x_1, \dots, x_n \rangle : (\mathcal{A}, q_j)]$  for some  $x_1, \dots, x_n$  such that  $\langle \sigma'(x_1), \dots, \sigma'(x_n) \rangle \in$   
 854  $\mathcal{R}(\mathcal{A}, q_j)$ , by definition of  $F^t(\omega)$  and  $F^s(\omega)$ .

855 +  $h(\Omega', \sigma') = h(\Omega, \sigma) - 1$ : For this case, let  $\omega = \text{argmax}_{\omega \in \Omega} (h(\sigma, \omega))$  and  $\omega' =$   
 856  $\text{argmax}_{\omega' \in \Omega'} (h(\sigma', \omega'))$ . By definition of  $F^t(\omega)$  and  $F^s(\omega)$ , we have both  $h(\sigma', \Omega') \geq$   
 857  $h(\sigma, \omega) - 1$  and  $h(\sigma', \Omega') \leq h(\sigma, \omega) - 1$ .

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

860 Therefore, take  $T = \frac{T'}{\vdash (E, \Omega)}$

861 We have that  $T$  is a valid proof tree and that  $h(T) = h(T') + 1 = h(\Omega, \sigma)$ .

862