COLLEGE DOCTORAL BRETAGNE | MATHS, TELECOMS INFORMATIQUE, SIGNAL SYSTEMES, ELECTRONIQUE

Université de Rennes

Thèse de doctorat de

L'Université de Rennes

École Doctorale N° 601
Mathématiques, Télécommunications, Informatique, Signal, Systèmes, Électronique
Spécialité : *Informatique*

Par
Théo LOSEKOOT

Automatic Program Verification by Inference of Relational Models

Thèse présentée et soutenue à Rennes, le 2024-12-17
Unité de recherche : IRISA

Rapporteurs avant soutenance :

Naoki KOBAYASHI   Professor - The university of Tokyo
Mihaela SIGHIREANU   Professeur des universités - ENS Paris-Saclay

Composition du Jury :

Président :		
Examinateurs :	David BAELDE	Professeur des universités - ENS Rennes
	Naoki KOBAYASHI	Professor - The university of Tokyo
	Mihaela SIGHIREANU	Professeur des universités - ENS Paris-Saclay
	Jean-Marc TALBOT	Professeur des universités - Aix Marseille Université
Dir. de thèse :	Thomas GENET	Professeur des universités - Université de Rennes
Co-dir. de thèse :	Thomas JENSEN	Directeur de recherche - INRIA Rennes

Résumé en Français

This document starts by a summary written in French but is then written in English.

La preuve formelle d'un programme consiste en la vérification de son comportement lors de *toutes* ses possibles exécutions. Cela permet d'augmenter la confiance que l'on porte en le code et peut s'utiliser en plus des tests classiques. Cependant, la vérification formelle des programmes est un problème difficile, ce qui en a fait un sujet de recherche intéressant qui remonte au moins aux années 1980 et qui a gagné en popularité au cours de ces dernières années. Nous présentons dans cette thèse de nouvelles méthodes faisant progresser la preuve de programmes. Plus précisément, nous nous concentrons sur la recherche automatique de représentations plus simples des programmes pour réaliser la preuve.

Types de propriétés De nombreux types de propriétés peuvent être envisagés en fonction du contexte dans lequel nous nous trouvons. Par exemple, on peut vouloir affirmer des propriétés temporelles, comme le fait qu'un distributeur automatique de boissons ne reste jamais bloqué après le paiement, ou que la commande d'activer les freins d'un train les active effectivement rapidement. Dans le contexte d'un protocole de communication, nous pouvons vouloir prouver que les messages échangés entre deux entités ne peuvent pas être lus ou modifiés par une troisième entité. Nous pouvons également vouloir prouver qu'un certain bloc de construction de programme se comporte fonctionnellement comme prévu, par exemple en vérifiant qu'une implémentation donnée d'une fonction de tri est effectivement capable de trier. Nous nous concentrons sur ce dernier type de vérification, où la fonction exécutée par le programme importe mais pas son temps d'exécution, ni sa mémoire, ni d'autres méta-propriétés. Plus précisement, cette thèse est à propos de la preuve automatique de programmes fonctionnels du premier-ordre manipulant des types algébriques de données. Les propriétés ciblées sont celles décrivant le rapport entre les entrée et les sorties du programme, c'est à dire les propriétés *relationelles*.

Assistants de preuve et preuve automatique Étant donné un programme, une personne experte peut écrire une preuve papier qu'il fonctionne comme prévu, c'est-à-dire qu'il satisfait les propriétés souhaitées. Cependant, les preuves papier étant difficiles à croire et à mettre à jour, les assistants de preuve (ou prouveur de théorèmes interactif) ont commencé à être utilisés à cette fin. Un assistant de preuve a pour rôle principal de vérifier que les preuves données par l'humain soient correctes. Pour cela, une description formelle du langage de programmation doit être fournie à l'assistant de preuve, ainsi que le programme et les propriétés à vérifier, ainsi que la preuve de ces propriétés. Ce processus de preuve formelle d'un programme est fastidieux car chaque détail doit être traité. Par exemple, le micro-noyau seL4 [1] a été prouvé formellement [25] en utilisant Isabelle/HOL, ce qui a duré environ 11 personnes*années. Heureusement, les assistants de preuve offrent une certaine automatisation, permettant ainsi de prouver des lemmes automatiquement, sans assistance humaine. Cependant, prouver automatiquement des propriétés complexes sur un large programme est trop compliqué pour les techniques actuelles, mais des techniques spécialisées permettent de se concentrer sur certains sous-ensembles de propriétés sur lesquels elles peuvent être efficaces. Ces techniques peuvent être mises en œuvre sous la forme de SMT-solvers (Satisfiability Modulo Theory), qui répondent à la question de savoir si un ensemble de formules est satisfaisable ou non dans la logique du premier ordre avec une théorie donnée. Parmi les théories courantes, on trouve la théorie des entiers, qui permet de raisonner sur des programmes manipulant des entiers, ou la théorie des types de données algébriques (ADT), sur laquelle nous nous concentrons dans cette thèse.

Prouver automatiquement une propriété sur un programme fonctionnel manipulant des données algébriques Prouver automatiquement une propriété sur le comportement entrée-sortie des programmes fonctionnels manipulant des types de données algébriques (ou arbres) est un sujet de recherche actif depuis les années 1980, parmi lesquels on remarque les travaux de Jones [21]. Ce sujet s'est popularisé au cours des dernières années et une vue d'ensemble de certains de ces travaux plus récents peut être trouvée dans le chapitre 3.

Comme les programmes de manipulation d'arbres sont définis syntaxiquement, une approche tentante pour prouver des propriétés est de transformer le programme en axiomes et d'utiliser ensuite les techniques de preuve syntaxique standard de la lo-

1. https ://github.com/seL4/seL4

gique formelle. Dans ces approches, une des difficultés consiste à adapter les systèmes de preuves et la technique de recherche de preuve associée pour qu'ils soient efficaces dans ce contexte de preuve sur programme, notamment en guidant la recherche pour trouver les lemmes intermédiaires pertinents. Une autre difficulté consiste à utiliser efficacement la nature (souvent) inductive des programmes et à permettre au système de preuve de générer des preuves inductives. Unno *et al.* ont proposé un tel système de preuve [36], qui est discuté dans la section 3.1, et Tsukada *et al.* [35] a écrit une vue d'ensemble claire et relativement complète du model-checking en tant que méthode de recherche de preuves syntaxiques.
Une autre technique consiste à représenter le programme ou une approximation du programme dans un formalisme plus faible, c'est-à-dire un formalisme moins expressif que les langages de programmation mais dont les problèmes de décision sont plus simples et ayant de meilleures propriétés de clôtures, puis à vérifier les propriétés sur cette représentation approximative. Un cas particulier de cette idée consiste en le calcul de l'ensemble des configurations que le programme peut atteindre (les configurations accessibles) et ensuite vérifier que son intersection avec les mauvaises configurations (représentant la négation des propriétés) est vide. Cette technique se décline selon le formalisme utilisé pour représenter l'ensemble des termes accessibles, que l'on retrouve sous forme de grammaires dans les travaux de Jones *et al.* [21, 22], d'automates d'arbres dans les travaux de Genet *et al.* [14, 15], ou sous une forme plus générique dans les travaux de Tsukada *et al.* [35]. La plupart de ces travaux calculent les termes accessibles sans se préoccuper des lemmes intermédiaires ni de la relation entre les entrées et les sorties, ce qui permet d'utiliser des formalismes simples (tels que les langages réguliers) mais limite la précision et donc les propriétés qui peuvent être prouvées. L'utilisation de langages réguliers ne permet de prouver que les propriétés qui ne nécessitent pas de relation précise entre les entrées et les sorties, ce que l'on appelons des propriétés non relationnelles. Par exemple, la propriété énonçant qu'une liste non vide a une longueur strictement positive, $len\ (x :: l) > 0$, est non relationnelle et peut être prouvée par ces méthodes. En revanche, la propriété énonçant qu'une liste à laquelle on ajoute un élément voit sa longueur augmenter, $len\ (x :: l) > len\ l$, est relationnelle (puisque la variable l est présente deux fois) et nécessite une modélisation plus précise de *len*.

Cette thèse Cette thèse se concentre sur les techniques qui représentent les programmes en utilisant un formalisme plus simple que le language dans lequel ils sont écrits, les abstrayant et simplifiant dans le processus, afin de prouver des propriétés sur ces programme. Cette représentation simplifiée est appelée *modèle* et nous appelons ces techniques de recherche de modèles des techniques *sémantiques*, en contraste avec les techniques de preuves syntaxiques. Un modèle de programme est une représentation de la relation entrée/sortie du programme. Nous sommes particulièrement intéressés par des formalismes plus expressifs que les langages réguliers, afin de pouvoir prouver des propriétés relationnelles, ce qui est également la direction prise par des travaux récemment publiés [18, 34]. Les formalismes pour représenter de tels modèles sont un point central dans ce travail, ce qui est reflété par leur importance dans cette thèse ; les automates d'arbres convolués sont définis et étendus dans le chapitre 5 et les Shallow Horn Clauses sont présentés dans le chapitre 7.

Contributions Cette thèse s'appuie sur les résultats préliminaires obtenus par Timothée Haudebourg lors de son séjour dans le groupe de Naoki Kobayashi à l'Université de Tokyo. Le formalisme de représentation des relations entrées-sortie sur les termes est un point central de notre travail, car il conditionne l'expressivité des relations représentées (et donc directement les propriétés qui peuvent être théoriquement prouvées). De plus, ce choix de formalisme influence également la manière dont un modèle doit être appris et la manière dont les clauses peuvent être vérifiées dans ces modèles. Nous utilisons un cadre générique Learner/Teacher pour prouver les propriétés, qui est décrit plus en détail dans le chapitre 4. Nos contributions sont les suivantes :

- Re-formalisation et simplification les automates d'arbres convolués ;
- Formalisation du Learner et du Teacher pour les automates d'arbres convolués ;
- Amélioration du Teacher, augmentant ainsi son efficacité sur les instances positives et négatives ;
- Preuve de la correction et complétude relative de cette approche ;
- Développement d'un nouveau formalisme pour représenter les langages de tuple d'arbres, appelé Shallow Horn Clauses et abrégé SHoCs, et preuve de certaines de ses propriétés de clôture et de décision ;
- Définition de la procédure Learner/Teacher en utilisant les SHoCs au lieu des

automates d’arbres convolués et preuve des propriétés de correction et de complétude relative ;

- Définition d’un cadre formel permettant d’approximer les clauses de Horn représentant le programme tout en conservant la correction de l’approche ;
- Implémentation du Learner/Teacher pour les automates d’arbres convolués et pour les SHoCs, ainsi que leurs améliorations ;
- Analyse comparative de ces approches.

ACKNOWLEDGEMENT

Thank you to the two reviewers of this thesis, Naoki KOBAYASHI and Mihaela SIGHIREANU, for their very thorough, useful, and in-depth comments. Thank you to all members of the jury for your time and positive feedback. Merci à mes encadrants, Thomas GENET et Thomas JENSEN, pour leur supervision hebdomadaire, leur aide, et leur confiance pour ces trois (et quelques) années. Merci à l'équipe Celtique/Épicure pour l'accueil et pour avoir répondu à mes nombreuses questions, quelque soit leur pertinence (tout particulièrement à Benoît à ce propos). Merci aux enseignants de l'université avec qui j'ai pu collaborer pour donner cours, ça a toujours été un plaisir. Merci à Lydie pour s'être administrativement battue et pour avoir gagné. Merci à Simon pour les ajouts de cohérence et de vie au laboratoire. Merci à Alan pour avoir organisé tous ces midi jeux ; j'espère un jour avoir accès à tes statistiques. Merci au bureau F213, et plus particulièrement aux personnes l'ayant occupé, pour avoir formé cette plaisante zone de travail allégé. Merci à Jean-loup pour avoir été un superbe camarade de bureau, malicieux et patient, et vive la science qu'on a pu faire ensemble. Merci à Nicolas qui, bien que le couloir nous séparait, a été un autre pilier du fun. Merci à Alice et Léo pour avoir accepté des informaticiens dans votre groupe de matheux et augmenté ma confiance badmintonistique. Merci à Mathieu pour m'avoir appris autant d'informatique (entre autres mais c'est le thème ici) que le reste des cours. Merci à Thibaut pour son aide logique et nos pauses catégoriques. Merci à Jules pour m'avoir distrait et pour avoir regardé mon code. Merci à mes amis de plus longue date qui ont eu la gentillesse de venir à la soutenance et de dire comprendre. Enfin, merci à ma famille qui a toujours été là.

TABLE OF CONTENTS

1 Introduction **11**

1.1 Overview of models formalisms to represent programs 13

1.1.1 Proof using Tree automata 14

1.1.2 Proof using Convoluted tree automata - first contribution 15

1.1.3 Proof using Shallow Horn Clauses - second contribution 17

1.2 Contribution 17

1.3 Outline 18

2 Prerequisites **20**

2.1 Term algebra and Herbrand model 20

2.2 Term manipulation 22

2.3 First-order clauses 25

2.4 Tree automata 26

3 State of the art **30**

3.1 Syntactic methods 30

3.2 Semantic methods 32

3.2.1 Automatic verification with regular language 33

3.2.2 Automatic verification with relational formalisms 39

4 General approach: from programs and properties to proofs **43**

4.1 From SMTLIB to clauses 44

4.2 Approximation of clauses 49

4.3 Model search for clauses: a generic Learner/Teacher procedure 54

5 Convoluted tree automata **58**

5.1 Convolution with padding 58

5.1.1 Standard left convolution 59

5.1.2 Right and complete convolution 63

5.1.3 Generalizing convolutions 65

5.2 Convolution without padding 68

6 Learner/Teacher for convoluted automata 74

6.1 Teacher 74

6.1.1 The $Inhabits_{\mathcal{A}}$ procedure 75

6.1.2 *Teacher* definition and theorems 86

6.2 Learner 88

6.3 Assembling the learner and teacher: *Sat* theorems 93

7 Shallow Horn Clauses (SHoCs) 96

7.1 SHoCs definition 96

7.2 ϵ-clauses and their elimination 98

7.3 Closure properties and decision procedures of SHoCs 101

7.4 Expressivity of SHoCs 106

7.4.1 SHoCs and convoluted automata 107

7.4.2 SHoCs and CS-programs 109

7.4.3 SHoCs and relational alternating automata 110

8 Learner/Teacher for SHoCs 112

8.1 Teacher 112

8.2 Learner 113

9 Implementation and experiments 118

9.1 Implementation 118

9.1.1 Teacher 119

9.1.2 Learner 120

9.2 Benchmarks 121

9.2.1 Zoom in on some benchmarks 129

10 Conclusion and perspectives 133

10.1 Conclusion 133

10.2 Perspectives 134

Bibliography 137

A Appendix **141**
A.1 Omitted proofs for the Teacher 141
A.2 Example of relations defined using first-order Horn clauses 147
A.3 Undecidability of the teacher procedure and of the emptiness of SHoCs 148

CHAPTER 1

INTRODUCTION

Formally proving that a program behaves as expected allows to statically verify *all* of its behaviors, to have more confidence in the code, and is expected to be realized alongside testing. However, the formal verification of programs is a hard problem, which made it a research subject that can be traced back at least to the 1980s and that have been gaining popularity in the recent years. The work compiled in this thesis progresses in this direction of proving programs. More precisely, we focus on automatically exhibiting simpler representations of programs to carry out the proof.

Types of properties Many types of properties can be considered, depending on the setting in which we find ourselves. For example, we may want to assert temporal properties, such as that a snack vending machine never gets stuck in the checkout process, or that sending the command to activate a train's brakes actually activates them quickly. In the context of a communication protocol, we may want to prove that messages exchanged between two entities cannot be read or altered by a third entity. Alternatively, we may want to prove that a certain program building block functionally behaves as planned, such as verifying that a given implementation of a sort function is indeed able to sort its input data. We focus on this last type of verification, where the function performed by the program matters but not its execution time, nor memory, nor other meta-properties.

Proof assistants and automatic proving Given a program, an expert can write a pen-and-paper proof that it works as expected, i.e. that it satisfies some desired properties. However, pen-and-paper proofs being hard to trust and update, proof assistants (or interactive theorem provers) have started to be used for such a purpose. For it, a formal description of the programming language must be provided to the proof assistant, together with the program and properties we want to verify on it. This process of formally proving a program is very tedious, as every detail must be laid out. For example,

the micro-kernel seL4 [1] has been formally proved [25] using Isabelle/HOL, which took around 11 person*years. Fortunately, proof assistants provide some automation, thus allowing to prove lemmas without human assistance. However, automatically proving intricate properties on a large project is too complicated for current techniques, but specialized techniques allow to focus on certain subsets of properties on which they can be efficient. These techniques can be implemented as Satisfiability Modulo Theory (SMT) solvers, which answer to whether a set of formulas is satisfiable or not in first-order logic with a given theory. Among the common theories dealt by SMT solvers are the theory of integers, allowing to reason about programs manipulating integers, and the theory of Algebraic Data Types (ADT), which we focus on in this thesis.

Automatically proving a property on a tree-manipulating functional program Automatically proving a property about the input-output behaviour of ADT-manipulating (or tree-manipulating) functional programs is an active research subject since the 1980s, with for example the work of Jones [21]. The popularity of this subject has increased over the recent years, and an overview of some of these more recent work can be found in Chapter 3.
Since tree-manipulating programs are syntactically defined, a tempting approach to prove properties on them is to transform a program into axioms and then use standard syntactic proof techniques from formal logic. In these approaches, one difficulty is to adapt a proof system to be efficient for this purpose and guide the search to find relevant intermediate lemmas. Another difficulty is to proficiently make use of the (often) inductive nature of programs and allow the proof system to generate inductive proofs. Unno *et al.* proposed such a proof system [36], which is discussed in Section 3.1, and Tsukada *et al.* [35] wrote a clear and comprehensive overview of model-checking as syntactic proof searching.
One particular technique is to represent (an approximation of) the program in a weaker formalism, i.e. a formalism that is less expressive than programming languages but that has easier decision and closure properties, and then check the properties on this approximated representation. This idea can also be used to compute the set of configurations that the program might reach (the accessible configurations) and then check that its intersection with bad configurations (representing the negation of properties) is empty. This technique is declined depending on the formalism used to represent the set

1. https://github.com/seL4/seL4

of accessible terms, which can be found in the form of grammars in the work of Jones *et al.* [21, 22], tree automata in the work of Genet *et al.* [14, 15], or under more generic form in the work of Tsukada *et al.* [35]. Most of this line of work computes the accessible terms without worrying about intermediate lemmas nor about the relation between inputs and outputs. This allows to use simple formalisms (such as regular languages), but limits the precision and therefore also the properties that can be proved. Using regular languages only allows to prove properties that do not need a precise input/output relation, which we could call non-relational properties. For example, the property stating that a non-empty list has a positive length, $len\ (x :: l) > 0$, is non-relational and can be proven with these methods. On the other hand, the property stating that a list to which an element is added increases its length, $len\ (x :: l) > len\ l$, is relational since the variable l is used twice. This relational property requires a more precise modeling of len.

This thesis This thesis focuses on techniques that abstract and simplify the program to prove properties on it, that we call semantic techniques in Chapter 3. In particular, we focus on the exhibition of a simpler representation of the program (a model) allowing to directly check the desired properties. A program model is any representation of the input/output relation of the program. We are particularly interested in formalisms that are more expressive than regular languages, in order to be able to prove relational properties, which is also the direction taken by recently published works [18, 34]. The formalisms to represent such models are a pivotal point in this work, which is reflected by their importance in this thesis ; convoluted tree automata are reworked in Chapter 5 and Shallow Horn Clauses are presented in Chapter 7. Here is an overview of some properties that can or cannot be proved depending on the expressivity of the formalisms for the models. This overview covers three formalisms: tree automata, convoluted tree automata, and Shallow Horn Clauses (SHoCs).

1.1 Overview of models formalisms to represent programs

Let $nat ::= z \mid s(nat)$ and $natTree ::= leaf \mid node(natTree, nat, natTree)$ be the ADTs of natural numbers and binary trees of nats. Horn clauses are a widespread formalism to represent programs and properties [5]. In this clausal formalism, we represent boolean n-ary functions as n-ary relations and non-boolean n-ary functions as $n+1$-ary

relations.
For example, the *height* function that could be written as `let height T = match T with | leaf -> z | node(l, _, r)-> s (max (height l)(height r))` in OCaml-like syntax is encoded as a binary relation Height. That is, first-order clauses are defined in order to denote a single Herbrand model $\mathcal{H}$ in which Height denotes the height relation $\{(leaf, z), (node(leaf, z, leaf), s(z)), \dots\}$. The predicates Leq (less than or equal), IsEmpty (is a tree empty), All0 (is a tree full of zeros), No0 (is a tree free of zeros), Height (the height of a binary tree), and HeightRB (the height of the rightmost branch of a binary tree) can be uniquely defined by a set of first-order Horn clauses using standard semantics. See Appendix A.2 for their precise definition. Let P denote the set of clauses that defines the program on which we want to prove properties.
To prove a property φ on P we have to prove $P \models \varphi$. Since there is only one model $\mathcal{H}$ of P, this reduces to checking that $\mathcal{H} \models \varphi$. We now define three properties on P:

$$\begin{aligned}
\varphi_1 &\stackrel{def}{=} \quad \text{IsEmpty}(T) \Leftarrow \text{All0}(T) \wedge \text{No0}(T) \\
\varphi_2 &\stackrel{def}{=} \quad \text{Leq}(N, M) \Leftarrow \text{HeightRB}(T_2, N) \wedge \text{HeightRB}(node(T_1, E, T_2), M) \\
\varphi_3 &\stackrel{def}{=} \quad \text{Leq}(N, M) \Leftarrow \text{HeightRB}(T, N) \wedge \text{Height}(T, M)
\end{aligned}$$

Property φ_1 can be proven by representing $\mathcal{H}$ with a tree automaton. Property φ_2 cannot be proven with tree automata but can be proven using convoluted tree automata. Finally, property φ_3 cannot be proven using convoluted tree automata but can be with SHoCs, as the Height relation requires a more expressive recursion scheme.

1.1.1 Proof using Tree automata

First, we focus on proving that $\mathcal{H} \models \varphi_1$. Recall that the Herbrand model $\mathcal{H}$ is the set of atoms satisfying P. Let $\mathcal{H}_{\text{IsEmpty}} = \{\text{IsEmpty}(t) \mid \text{IsEmpty}(t) \in \mathcal{H}\}$, $\mathcal{H}_{\text{All0}} = \{\text{All0}(t) \mid \text{All0}(t) \in \mathcal{H}\}$, and $\mathcal{H}_{\text{No0}} = \{\text{No0}(t) \mid \text{No0}(t) \in \mathcal{H}\}$ be sets of atoms and subsets of $\mathcal{H}$. Thus, proving φ_1 is equivalent to showing that, for any term t,
(1) if $\mathcal{H} \models \text{All0}(t) \wedge \text{No0}(t)$ then $\mathcal{H} \models \text{IsEmpty}(t)$, or equivalently

(2) if $\text{All0}(t) \in \mathcal{H}_{\text{All0}} \wedge \text{No0}(t) \in \mathcal{H}_{\text{No0}}$ then $\text{IsEmpty}(t) \in \mathcal{H}_{\text{IsEmpty}}$
To prove (2), we need to pick *all* terms t such that $\text{All0}(t)$ belongs to $\mathcal{H}_{\text{All0}}$, $\text{No0}(t)$ belongs to $\mathcal{H}_{\text{No0}}$ and check that the term $\text{IsEmpty}(t)$ (for the same t) belongs to $\mathcal{H}_{\text{IsEmpty}}$.
For proving this automatically, we need a finite representation of $\mathcal{H}_{\text{All0}}$, $\mathcal{H}_{\text{No0}}$, and $\mathcal{H}_{\text{IsEmpty}}$. Those three sets are regular, meaning they can be represented by a tree au-

tomaton. These automata can be can be automatically inferred [30, 15, 19, 27], which is also done in the Example 4.14. Tree automata rewrite terms to states. If a term t rewrites to a state q then it is said to be *recognized* in q. Here is a possible tree automaton recognizing terms t such that $\text{IsEmpty}(t)$ belong to $\mathcal{H}_{\text{IsEmpty}}$: $\mathit{leaf} \to q_{\text{IsEmpty}}$. This automaton only recognizes the empty tree *leaf*. We can do the same for No0 with the following tree automaton:

$$\mathit{leaf} \to q_{\text{No0}} \qquad z \to q_0 \quad s(q_s) \to q_s$$
$$\mathit{node}(q_{\text{No0}}, q_s, q_{\text{No0}}) \to q_{\text{No0}} \quad s(q_0) \to q_s$$

This automaton recognizes z in q_0 and recognizes all natural numbers greater than 0 in q_s, e.g., $s(z) \to s(q_0) \to q_s$. It also recognizes all trees of natural numbers greater than 0 in state q_{No0}. Thus, the language recognized by q_{No0} is the set of terms t such that $\text{No0}(t)$ belongs to $\mathcal{H}_{\text{No0}}$. Finally, we can do the same for All0 with the following tree automaton where state q_{All0} recognizes terms t such that $\text{All0}(t)$ belong to $\mathcal{H}_{\text{All0}}$:

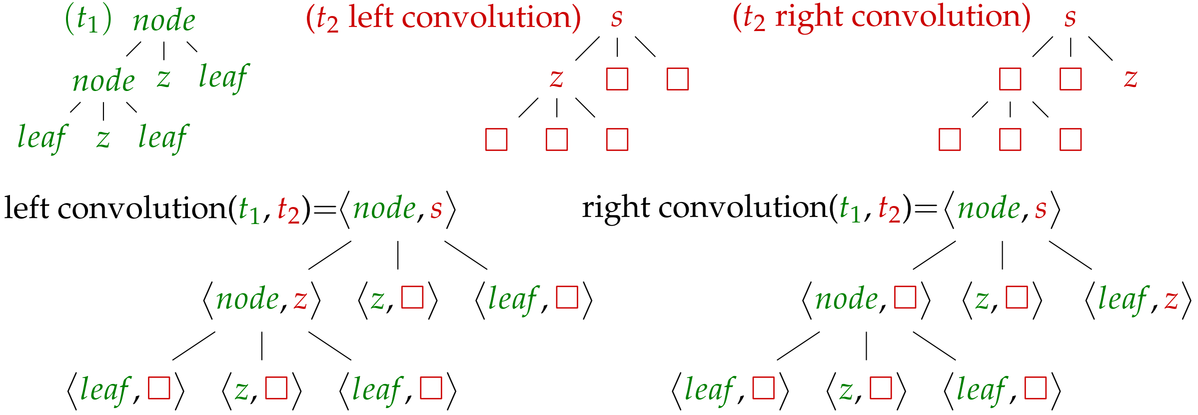
$$\mathit{leaf} \to q_{\text{All0}} \quad z \to q_0$$
$$\mathit{node}(q_{\text{All0}}, q_0, q_{\text{All0}}) \to q_{\text{All0}}$$

Thus, to prove (2) it is enough to prove that all terms recognized by both q_{All0} and q_{No0} are also recognized by q_{IsEmpty}, i.e., prove that the intersection between the languages of q_{All0} and q_{No0} is included in the language denoted by q_{IsEmpty}, which is decidable on tree automata.

1.1.2 Proof using Convoluted tree automata - first contribution

Models that can be expressed using regular languages with tree automata have limitations. To prove the property φ_2, the model needs to preserve the relation that exists between the rightmost branch of $\mathit{node}(T_1, E, T_2)$ and its height M. Verifying φ_2 cannot follow the previous approach. In this thesis, precisely in Chapters 5 and 6, we use an extension of tree automata with convolutions [7] to recognize regular languages of tuples to prove such properties. A convolution transforms a n-tuple of terms into a term built on n-tuples of symbols. To recognize the *binary* relation HeightRB, a convoluted tree automaton recognizes the overlaying of terms t_1 and t_2 for all (t_1, t_2) belonging to the relation (i.e. such that $\text{HeightRB}(t_1, t_2) \in \mathcal{H}$). For instance, to overlay $t_1 = \mathit{node}(\mathit{node}(\mathit{leaf}, z, \mathit{leaf}), z, \mathit{leaf})$ and $t_2 = s(z)$, one needs to define a new symbol $\langle \mathit{node}, s \rangle$ for the convolution of the top symbols of the two terms and then overlay subterms of t_1 and t_2. Since *node* and s have a different numbers of children, a direct overlaying is impossible. Convoluted tree automata solve this by extending the arity

of symbols so that they are all the same, replacing non-existent subterms by a padding symbol $\square$. However, when using such a representation, overlaying t_1 and t_2 leads to many possible trees representing t_2. As a result, convoluted tree automata are parameterized by a *fixed* overlaying strategy, e.g., the left (resp. right) convolution strategy overlays the leftmost (resp. rightmost) branches of the two terms together.



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

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

In particular, this automaton recognizes the right convolution of t_1 and t_2, that is $\langle node, s\rangle(\langle node, \square\rangle(\langle leaf, \square\rangle, \langle z, \square\rangle, \langle leaf, \square\rangle), \langle z, \square\rangle, \langle leaf, z\rangle)$ (depicted above), in the state q_{hRB}. Similarly, it is possible to build a convoluted automaton recognizing the Leq relation, which has the following transitions: $\langle z, z\rangle \to q_{\text{Leq}}$, $\langle \square, z\rangle \to q_{\text{Leq}}$, $\langle \square, s\rangle(q_{\text{Leq}}) \to q_{\text{Leq}}$, $\langle z, s\rangle(q_{\text{Leq}}) \to q_{\text{Leq}}$, $\langle s, s\rangle(q_{\text{Leq}}) \to q_{\text{Leq}}$. Then, using these two automata, it is possible to prove φ_2: we have to prove that for all t_1, t_2, e, n, m such that $(node(t_1, e, t_2), m)$ and (t_2, n) belong to the language recognized by q_{hRB}, then the pair (n, m) is recognized by q_{Leq}. This can easily be proven using standard algorithms on tree automata. Convoluted tree automata successfully model relations for which it is enough to relate a single *fixed* branch of a term (e.g. leftmost or rightmost branch) with another *fixed* branch in another term. Note that for another function, say HeightLB computing the height of the *leftmost* branch of a binary tree, we would need to choose a different strategy for convolution than the right convolution.

1.1.3 Proof using Shallow Horn Clauses - second contribution

Using the *fixed-branch* restriction enables efficient verification of many list processing programs [29], but it prevents convolution from modelling relations that depend on unpredictable choice of branches, such as the relation Height between a binary tree and its height. *E.g.*, it is impossible to have a representation for $\mathcal{H}$ (the model of P) sufficiently precise for proving $\mathcal{H} \models \varphi_3$ when using convoluted tree automata. To overcome this limitation, we define in this thesis a restriction of Horn clauses called Shallow Horn Clauses (SHoCs) to represent such models. In a set of SHoCs, the predicate symbols replace states of tree automata and the language recognized by a predicate R is the set of tuples of terms $(t_1, \ldots, t_n)$ such that $R(t_1, \ldots, t_n)$ belongs to the smallest Herbrand model of the SHoCs (equivalently to the least fixpoint of the SHoCs). For instance, the SHoCs recognizing the HeightRB relation (and equivalent to the previous convoluted automaton) consists of the following two clauses:

$$\text{HeightRB}(leaf, z) \qquad \text{HeightRB}(node(T_1, E, T_2), s(N)) \Leftarrow \text{HeightRB}(T_2, N).$$

Restricting Horn clauses to SHoCs yields decidability of boolean operations like intersection, union, and complement, some of which are necessary for our verification. Besides, since SHoCs do not rely on a *fixed* convolution strategy they can precisely model the *height* function and prove property φ_3. SHoCs are slightly more expressive than convoluted automata, but more importantly they are less rigid and easier to manipulate.

1.2 Contribution

This thesis builds upon the preliminary results [18] obtained by Timothée Haudebourg during his stay in Naoki Kobayashi's group in The University of Tokyo. The formalism for representing the input-output relations on terms is a central point of our work, as it conditions the expressiveness of the relations represented (and therefore directly which properties can theoretically be proven). Moreover, it also influences the way in which a model must be learned and the way in which the clauses can be verified in these models. We use a generic Learner/Teacher framework for proving the properties, which is described in more details in Chapter 4.
Our contributions are:

- Re-formalizing and simplifying the convoluted tree automata formalism ;

- Formalizing the Teacher and the Learner for convoluted tree automata ;
- Improving the Teacher, therefore increasing its efficiency on both positive and negative instances ;
- Proving correctness and relative completeness of this approach ;
- Developing a novel formalism for representing tree tuple languages called Shallow Horn Clauses and proving some of its properties ;
- Defining the Learner/Teacher procedure using SHoCs instead of convoluted tree automata and proving the same correctness and relative completeness properties ;
- Defining a formal framework allowing to safely approximate Horn clauses ;
- Implementing the Learner/Teacher for both convoluted tree automata and SHoCs, together with their improvements ;
- Benchmarking these approaches.

1.3 Outline

Chapter 2 goes over necessary prerequisites to understand the rest of this thesis. These prerequisites include typed alphabets, terms, patterns, algebraic datatypes, substitutions, unification, and tree automata.
Chapter 3 presents some of the state-of-the-art methods for proving properties on functional programs, especially similar technique as those defined in this thesis or techniques proving similar properties.
Chapter 4 presents the overview of the verification technique, going from the program and properties to the proof that they are satisfied or not. To this end are defined the translation from program to clauses, the clauses approximation procedure, and the generic (w.r.t. models) Learner/Teacher procedure for checking satisfiability of the clauses, which reflects whether the properties are true.
Chapter 5 present a re-formalization of convoluted tree automata that unifies the definitions of left, right, complete, and new convolutions. Moreover, a simplification that allows to remove the padding is presented, which allows more compact automata. The chapter starts by the standard left-convolution with padding and progresses to more general convolutions.

Chapter 6 is about the Learner/Teacher procedure using these convoluted tree automata as well as proofs of its correctness and partial completeness. Improvements of the Teacher are also presented.
Chapter 7 presents Shallow Horn Clauses, a new formalism for representing relations of terms based on a syntactic restriction of Horn clauses, as a fairly direct extension of convoluted automata. Most of their closure properties and decision problems are also addressed, together with a comparison with existing tree-tuple formalisms.
Chapter 8 is about the Learner/Teacher procedure using shallow Horn clauses. The Teacher is almost immediate by similarity with the Teacher for convoluted automata, while the Learner must be completely redefined.
Chapter 9 discusses the implementation of this procedure, some practical enhancements that have been implemented, and experimental results.
Finally, Chapter 10 concludes and proposes further work.

CHAPTER 2

PREREQUISITES

This chapter introduces the basic definitions needed to understand this thesis, including that of alphabet, term, pattern, algebraic datatype, substitution, unification, and tree automaton. We use a simply-typed, also called multi-sorted, version of these concepts. We start by defining the term algebra and Herbrand model in Section 2.1, then functions on terms in Section 2.2, then first-order clauses in Section 2.3, and finally tree automata in Section 2.4.

2.1 Term algebra and Herbrand model

Terms are written using symbols, or functions, from an alphabet.

Definition 2.1 (Typed alphabet)**.** w A typed alphabet (Σ, τ, Γ) is a set of symbols Σ, a set of types Γ, and a typing function τ which assigns to each symbol f a type $\tau(f) = (\tau_1, \dots, \tau_n) \to \tau_0$ with $\forall i \in [0 \dots n], \tau_i \in \Gamma$ and $n \in \mathbb{N}$ varying for each symbol f. When $n = 0$, the symbol is a constant and does not take input. For $f \in \Sigma$ and $\tau(f) = (\tau_1, \dots, \tau_n) \to \tau_0$, we say that f is of arity n, written $|f| = n$, and that τ_0 is the *output type* of f, written $\tau_{out}(f) = \tau_0$. When clear from context, we identify the tuple (Σ, τ, Γ) with Σ.

Definition 2.2 (Term)**.** A (typed) term t over an alphabet Σ is the data of a symbol $f \in \Sigma$, called the *root symbol* of t and written $Root(t)$, together with a list $t_1, \dots, t_{|f|}$ of $|f|$ terms, called *children* of t, such that their type is compatible, i.e. such that $\tau(f) = \big(\tau_{out}(Root(t_1)), \dots, \tau_{out}(Root(t_{|f|}))\big) \to \tau_{out}(f)$. Such a term t is written $f(t_1, \dots, t_{|f|})$. We overload τ, the typing function, with $\tau(t) = \tau_{out}(Root(t))$. The set of terms over an alphabet Σ is written $\mathcal{T}(\Sigma)$, and the set of terms of a certain type τ_0 is written $\mathcal{T}_{\tau_0}(\Sigma) = \{t \mid t \in \mathcal{T}(\Sigma) \wedge \tau(t) = \tau_0\}$.

Definition 2.3 (Pattern)**.** A pattern p over an alphabet Σ and a set of (typed) variables $\mathcal{X}$ is a term over the alphabet $\Sigma \cup \mathcal{X}$, with each variable X considered as a 0-ary symbol of

type $\tau(X)$. The set of patterns over the alphabet Σ and variables $\mathcal{X}$ is written $\mathcal{T}(\Sigma, \mathcal{X})$. A pattern that does not contain any variable is also called a term, or even a *ground* term to emphasize.

Set of terms, initial algebra, Herbrand structure The initial algebra of a typed alphabet is the set of terms that can be defined on it. This initial algebra, associating the application of a function f on (type-compatible) terms $t_1, \ldots, t_n$ to the term $f(t_1, \ldots, t_n)$, is also called a *Herbrand structure*.

Example 2.4. Let Σ be a typed alphabet defining natural numbers and lists of natural numbers as follows: $\Sigma = \{z, s, nil, cons\}$ with $\tau(z) = nat$, $\tau(s) = (nat) \to nat$, $\tau(nil) = natlist$, and $\tau(cons) = (nat, natlist) \to natlist$.
The set of terms $\mathcal{T}(\Sigma)$ contains, among others, z and $cons(s(z), nil)$ but not $s(nil)$ because of typing constraints. With $X \in \mathcal{X}$ a variable of type *nat*, $s(X) \in \mathcal{T}(\Sigma, \mathcal{X})$.

An algebraic datatype is a way of describing a typed alphabet that is usually used in functional programming.

Definition 2.5 (Algebraic datatype)**.** An algebraic data type is a (possibly recursive) sum type of product types. In a definition

$$\tau_0 = (\tau_1^1 \times \ldots \times \tau_{n_1}^1) + \ldots + (\tau_1^k \times \ldots \times \tau_{n_k}^k)$$

each (non-necessarily different) product $(\tau_1^i \times \ldots \times \tau_{n_i}^i)$ of the disjoint sum is usually given a symbol f_i whose type is $\tau(f_i) = \tau_1^i \times \ldots \times \tau_{n_i}^i \to \tau_0$.

Example 2.6 (nat and natlist)**.** The natural numbers and natural number lists can be defined as algebraic datatypes by $nat = () + (nat)$ and $natlist = () + (nat \times natlist)$. In programming languages, these are usually written in a grammar-like syntax: $nat = z \mid s(nat)$ and $natlist = nil \mid cons(nat, natlist)$, which defines the typed alphabet of Example 2.4.

In this thesis, every function of the program will be represented as a *relation* on the domain of terms, i.e. as a function returning a boolean. To this end, we introduce the notion of atom and Herbrand model.

Definition 2.7 (Relation symbol and atom)**.** A relation symbol is a function symbol R with $\tau(R) = \tau_1 \times \ldots \times \tau_n \to bool$, with *bool* a special datatype and for all $i \in [1 \ldots n]$, τ_i an algebraic datatype. This relation symbol will denote a subset of $\tau_1 \times \ldots \times \tau_n$.

An atom $R(p_1, \ldots, p_n)$ is a pattern of type bool, i.e. a relation symbol R applied to patterns $p_1, \ldots, p_n$.

Definition 2.8 (Herbrand model)**.** A Herbrand model $\mathcal{H}$ (or Herbrand interpretation or Herbrand base) over an alphabet Σ is a set A of ground atoms over Σ. This set A represents the atoms that are true.
Equivalently, a Herbrand model (over an alphabet Σ) is a first-order model $\mathcal{H}$ that extends the Herbrand structure of Σ with the interpretation of the relations.

In this thesis, the only first-order models we are interested in are the Herbrand models. In first-order logic, this corresponds to using the theory of recursive algebraic datatypes.

Definition 2.9 (Theory of recursive algebraic datatypes)**.** The theory of algebraic datatypes restricts the possible first-order models to those whose domain and function interpretation are a Herbrand structure, i.e., to Herbrand models. This allows to use first-order function symbols as constructors of algebraic datatypes. See [2] for a detailed explanation.

Note that, to define a Herbrand model $\mathcal{H}$, it is sufficient to define the interpretation of every relation, as the Herbrand structure is fixed. We often explicitly define functions using the $\mapsto$ arrow, with $K \mapsto V$ meaning that K is mapped to V.

Example 2.10 (Herbrand model)**.** Following Example 2.6, the Herbrand model $\mathcal{H}$ for the strictly inferior relation Less : $Nat \times Nat \rightarrow bool$ and the length relation Len : $NatList \times Nat \rightarrow bool$ is given by the following informal interpretation of the two relations:

$$\text{Less} \mapsto \{(n_1, n_2) \mid \tau(n_1) = Nat \wedge \tau(n_2) = Nat \wedge n_1 \text{ is inferior to } n_2\}$$
$$\text{Len} \mapsto \{(l, n) \mid \tau(l) = NatList \wedge \tau(n) = Nat \wedge l \text{ is of length } n\}.$$

2.2 Term manipulation

This section introduces some classic functions and problems on patterns. First, we define a function that fetches the set of variables of an object.

Definition 2.11 (Variables)**.** A function $Vars(\cdot)$ fetches the set of variables in a pattern and is extended to tuples, sets, etc.

With variables come substitutions, which are a way to replace variables in patterns by a pattern.

Definition 2.12 (Substitution)**.** A substitution σ is a finite map between variables and patterns. The application of a substitution σ to a variable X, written $\sigma(X)$, is defined as p if there exists a binding $(X, p) \in \sigma$ and X otherwise. Substitutions are typed, so $(X, p) \in \sigma \Rightarrow \tau(X) = \tau_{out}(p)$. The application of a substitution is generalized to patterns by $\sigma(f(p_1, \ldots, p_n)) = f(\sigma(p_1), \ldots, \sigma(p_n))$. More generally, a substitution can be applied to any mathematical object that contains variables, such as tuples or sets. The composition of substitutions, which first applies σ_1 and then σ_2, is written $\sigma_1;\sigma_2$ or $\sigma_2 \circ \sigma_1$. The domain of a substitution is the set of variables for which a binding is defined and is written $dom(\sigma)$.

Definition 2.13 (Position)**.** A position π is a word over $\mathbb{N}$ used to point at a subterm. We write the concatenation operator $\cdot$ and the empty word ϵ.
The set of positions of a pattern $f(p_1, \ldots, p_n) \in \mathcal{T}(\Sigma, \mathcal{X})$ is inductively defined as:

$$Pos(f(p_1, \ldots, p_n)) = \{\epsilon\} \cup \bigcup_{i \in [1 \ldots n]} \{i \cdot \pi \mid \pi \in Pos(p_i)\}$$

The subterm of a pattern p at position π is defined iff $\pi \in Pos(p)$. In that case, it is inductively defined as:

$$p[\epsilon] = p \qquad \text{and} \qquad f(p_1, \ldots, p_n)[i \cdot \pi] = p_i[\pi]$$

The length of a position π, defined as the length of the string it is represented by, is written $|\pi|$.

Definition 2.14 (Tuple of elements)**.** A tuple of elements $(e_1, \ldots, e_n)$ is also written $\vec{e}$ and $\vec{e}[i]$ means e_i, in accordance with Definition 2.13. For example, $\vec{t}$ represents a tuple of terms.

Definition 2.15 (Typed complement)**.** Given a set E of terms that are all of some type τ_E, its typed complement w.r.t τ_E is the set $\{t \mid \tau(t) = \tau_E \wedge t \notin E\}$ and is simply written $\overline{E}$ because τ_E should be clear from context.

Definition 2.16 (Height)**.** The height $ht(\cdot)$ of a pattern is inductively defined as

$$ht(f(p_1, \ldots, p_n)) = 1 + \max_{i \in [1 \ldots n]} ht(p_i)$$

The height definition immediately extends to atoms $R(\vec{p})$ as the maximum of the height of patterns of $\vec{p}$, and to sets and tuples as the maximum of each individual height of their component.
Syntactic unification consists in making patterns equal by applying a substitution.

Definition 2.17 (Syntactic unification)**.** A (syntactic) unification problem is a set $E = \{p_1 \stackrel{?}{=} p'_1, \dots, p_n \stackrel{?}{=} p'_n\}$ of equalities between patterns. A solution to this unification problem is a substitution σ such that for all $p \stackrel{?}{=} p'$ in E, $\sigma(p) = \sigma(p')$. Solutions can be pre-ordered by precision, with $\sigma_1 \leq \sigma_2 \iff \exists\sigma.\ \sigma_1;\sigma = \sigma_2$. A Most General Unifier (MGU) is a least precise unifier σ, i.e. such that, for any other unifier σ', we have $\sigma \leq \sigma'$.

With syntactic unification, either the patterns are not unifiable, in which case there is no solution, or there exists a unique MGU modulo variable renaming.

Example 2.18 (Height, complement, substitution, unification)**.** Let $p_1 = cons(s(X), L)$, $p_2 = L'$, and $p_3 = s(z)$ be three patterns. Then,

- $Vars(p_1) = \{X, L\}$.
- With $\sigma = \{(X, s(z)), (L', nil)\}$, we have $\sigma(p_1) = cons(s(s(z)), L)$ and $\sigma(p_2) = nil$.
- With $\pi = 1 \cdot 1$, we have $p_1[\pi] = X$ and $p_2[\pi]$ is undefined.
- $ht(p_1) = 3$, $ht(p_2) = 1$, and $ht(p_3) = 2$.
- With $E = \{p_1 \stackrel{?}{=} p_2,\ X \stackrel{?}{=} p_3\}$, we have $MGU(E) = \sigma$ with $\sigma = \{(L',\ cons(s(s(z)), L)), (X,\ s(z))\}$. Then, applying σ to p_1 yields $\sigma(p_1) = cons(s(s(z)), L)$. The set $E = \{p_1 \stackrel{?}{=} L\}$ is non-unifiable using the finite terms we consider.
- Following Example 2.10, the typed complement of $\mathcal{L}(\mathcal{H}, \text{Less},)$ is $\overline{\mathcal{L}(\mathcal{H}, \text{Less})} = \{(n_1, n_2) \mid n_1 \geq n_2\}$.

A popular formalism to represent programs, or more generally term-transforming functions, is that of Term Rewriting System (or TRS)

Definition 2.19 (Term Rewriting System)**.** A term rewriting system $\mathcal{D}$ is a set of rewrite rules. A rewrite rule is a pair of patterns, commonly written as $l \to r$, to indicate that the left-hand side l can be replaced by the right-hand side r. A rule $l \to r$ can be applied to a pattern p if the left term l matches some subterm of p, that is, if there is some substitution σ and a position π such that $p[\pi] = \sigma(l)$. The pattern p' resulting from this rule application is then a modified version of p whose subterm at position π is replaced by $\sigma(r)$. p is then said to be rewritten in one step to p' by the system $\mathcal{D}$ and is written $p \to_{\mathcal{D}} p'$.

Example 2.20 (*double* program as TRS)**.** Let $\mathcal{D}_d$ be the TRS composed from the following two rewriting rules:

$$double(z) \to z \qquad double(s(X)) \to s(s(double(X))).$$

This TRS defines the *double* function in that every term $double(n)$ can be rewritten, in $n+1$ steps, into $2*n$. For example, we have $double(s(s(z))) \to_{\mathcal{D}_d} s(s(double(s(z)))) \to_{\mathcal{D}_d} s(s(s(s(double(z))))) \to_{\mathcal{D}_d} s(s(s(s(z))))$.

2.3 First-order clauses

This section introduces the notion of clauses, Horn clauses, and of minimal Herbrand model.

Definition 2.21 (First-order clause)**.** A clause is a first-order formula of the form

$$\forall \vec{X},\ R_1(\vec{p_1}) \vee \ldots \vee R_k(\vec{p_k}) \vee \neg R_{k+1}(\overrightarrow{p_{k+1}}) \vee \ldots \vee \neg R_n(\vec{p_n})$$

with $\vec{X}$ a tuple of typed variables, $1 \leq k \leq n$, and for all i in $[1 \ldots n]$, R_i is a relation symbol, $\vec{p_i}$ is a tuple of patterns, and $R_i(\vec{p_i})$ is an atom whose variables are all included in $\vec{X}$.

Clauses are also written $R_1(\vec{p_1}) \vee \ldots \vee R_k(\vec{p_k}) \Leftarrow R_{k+1}(\overrightarrow{p_{k+1}}) \wedge \ldots \wedge R_n(\vec{p_n})$, where universal quantification is implicit, $H = R_1(\vec{p_1}) \vee \ldots \vee R_k(\vec{p_k})$ is called the *head* and $B = R_{k+1}(\overrightarrow{p_{k+1}}) \wedge \ldots \wedge R_n(\vec{p_n})$ the *body*. The head and body of a clause can be manipulated as sets of their atoms, since order usually does not matter and duplicates are useless.

Definition 2.22 (Horn clauses and Strict Horn clauses)**.** A clause $H \Leftarrow B$ is called a *Horn clause* if its head contains at most one atom, i.e. $|H| \leq 1$. A Horn clause is called *strict* if $|H| = 1$ and its head does not contain the equality predicate, as it is the only pre-defined relation we consider.

Proposition 2.23 (Minimal Herbrand model)**.** *Let $\mathcal{C}$ be a set of strict Horn clauses. There exists a unique* minimal Herbrand model $\mathcal{H}(\mathcal{C})$ *of $\mathcal{C}$ [11], which corresponds to the inductive definition (least fixed-point) of $\mathcal{C}$.*

Example 2.24. Let $\mathcal{C}$ be the following set of strict Horn clauses:

$$\{R_0(a,b),\ \ R_1(f(X),g(Y)) \Leftarrow R_0(X,Y),\ \ R_0(f(X),g(Y)) \Leftarrow R_1(X,Y)\}$$

The minimal Herbrand model $\mathcal{H}(\mathcal{C})$ is defined by

$$\mathcal{H}(\mathcal{C}) = \{R_0(f^{2n}(a), g^{2n}(b)) \mid n \geq 0\} \cup \{R_1(f^{2n+1}(a), g^{2n+1}(b)) \mid n \geq 0\}.$$

The language of R_0 is $\mathcal{L}(R_0, \mathcal{H}(\mathcal{C})) = \{(f^{2n}(a), g^{2n}(b)) \mid n \geq 0\}$.

We now recall some standard first-order notations.

Definition 2.25 (Notation for some standard first-order concepts)**.** Given a first-order model $\mathcal{M}$, a formula φ, and an assignment of variables λ, we write (i) $\mathcal{M}, \lambda \models \varphi$ to state that formula φ is true under the assignment λ in model $\mathcal{M}$; (ii) $\mathcal{M} \models \varphi$ to state that $\mathcal{M}, \lambda \models \varphi$ is true for any assignment λ ; (iii) $Assigns(\mathcal{M}, \varphi)$ the set of assignments λ such that $\mathcal{M}, \lambda \models \varphi$. A set of formulas Γ is thought as their conjunction, so $\mathcal{M}, \lambda \models \Gamma$ means that, for every formula $\varphi \in \Gamma$, we have $\mathcal{M}, \lambda \models \varphi$.

Note that, in Herbrand models, an assignment is the same thing as a substitution whose codomain only contains ground terms.

2.4 Tree automata

This section introduces the basics of tree automata, which are the tree counterpart to finite automata on string.

Definition 2.26 (Tree automaton)**.** A (bottom-up) tree automaton $\mathcal{A} = (Q, \Delta)$ over an alphabet Σ is given by a finite set of states Q and a set of transitions Δ of the form $f(q_1, \ldots, q_{|f|}) \to q_0$, where $f \in \Sigma$ and $\forall i \in [0 \ldots |f|], q_i \in Q$.

Definition 2.27 (Language recognized by an automaton)**.** The set of terms recognized in a state q of an automaton $\mathcal{A}$ is inductively defined as

$$\mathcal{L}(\mathcal{A}, q) = \{f(t_1, \ldots, t_n) \mid f(q_1, \ldots, q_n) \to q \in \Delta \ \wedge \bigwedge_{i \in [1 \ldots n]} t_i \in \mathcal{L}(\mathcal{A}, q_i)\}.$$

A language that is recognized by (a state of) a (tree) automaton is said to be a *regular* (tree) language. A reader who is familiar with these notions may wonder what happened to the final states. We prefer to not introduce them, as we often use an automaton to represent multiple relations at once, especially in Chapter 5 with convoluted tree automata.

Definition 2.28 (Typed tree automaton)**.** A typed tree automaton is a tree automaton whose states are typed. We write $\tau(q)$ for the type of the state q. Transitions have to be compatible with the types of the symbols, i.e., for any transition $f(q_1, \ldots, q_n) \to q_0 \in \Delta$, $\tau(f) = (\tau(q_1), \ldots, \tau(q_n)) \to \tau(q_0)$.

We also use Q and Δ as accessors, that is, as functions to respectively extract states $Q(\mathcal{A})$ and transitions $\Delta(\mathcal{A})$ from an automaton $\mathcal{A}$. We usually write q for a state, and $\mathcal{A}$ for an automaton. Tree automata do not have a nice graphical representation, contrary to string automata, so we have to explicitly list their transitions and, optionally, their states.

Example 2.29 (Automata for *all*0 and *no*0)**.** Let $\mathcal{A}_{all0}$ be an automaton with states $\{q_0, q_{all0}\}$ and transitions $node(q_{all0}, q_0, q_{all0}) \to q_{all0} \quad leaf() \to q_{all0} \quad z() \to q_0$.
The language of q_{all0} in $\mathcal{A}_{all0}$, written $\mathcal{L}(\mathcal{A}_{all0}, q_{all0})$, is the set of trees whose elements are all z. The type of q_0 is $\tau(q_0) = nat$ and $\tau(q_{all0}) = nattree$.
A very similar automaton $\mathcal{A}_{no0}$ can be defined with states $\{q_s, q_0, q_{no0}\}$ and transitions $node(q_{no0}, q_s, q_{no0}) \to q_{no0} \quad leaf() \to q_{no0} \quad z() \to q_0 \quad s(q_0) \to q_s \quad s(q_s) \to q_s$.
The language of q_{no0} in $\mathcal{A}_{no0}$, written $\mathcal{L}(\mathcal{A}_{no0}, q_{no0})$, is the set of trees whose elements are all different from z.

Example 2.30 (Automaton for *has*0)**.** Here is an other automaton $\mathcal{A}_{has0}$ with four states $\{q_t, q_0, q_{all0}, q_{nat}\}$ and transitions defined as below:

$$\begin{array}{rrr} node(q_t, q_0, q_t) \to q_{has0} & node(q_t, q_{nat}, q_t) \to q_t & z() \to q_{nat} \\ node(q_{has0}, q_{nat}, q_t) \to q_{has0} & leaf() \to q_t & s(q_{nat}) \to q_{nat} \\ node(q_t, q_{nat}, q_{has0}) \to q_{has0} & & z() \to q_0 \end{array}$$

The language of q_{has0} in $\mathcal{A}_{has0}$, written $\mathcal{L}(\mathcal{A}_{has0}, q_{has0})$, is the set of trees which contain the element z in some node. We also have $\mathcal{L}(\mathcal{A}_{has0}, q_{nat}) = \{n \mid \tau(n) = Nat\}$. Note that, contrary to those of Example 2.29, this automaton is non-deterministic, meaning that a term can be recognized by multiple states. For example, *leaf* is both in $\mathcal{L}(\mathcal{A}_{has0}, q_t)$ and in $\mathcal{L}(\mathcal{A}_{has0}, q_{has0})$.

An automaton can have states which serve no purpose, so we usually only focus on states recognizing at least one term, the accessible ones. Accessible states can easily be computed. A state that is not accessible is useless and can be deleted from an automaton, together with any transition that uses it. An automaton whose states are all accessible is called *reduced*.

Definition 2.31 (Accessible states)**.** A state q of an automaton $\mathcal{A}$ is said to be *accessible* if $\mathcal{L}(q, \mathcal{A}) \neq \emptyset$.

Tree automata can have properties that restrict their shape, namely determinism and completeness, that make them easier to use and manipulate in proofs, but also usually make them bigger.

Definition 2.32 (Semantic determinism/completeness)**.** A reduced automaton is said *semantically deterministic (resp. complete)* if, for every term $t \in \mathcal{T}(\Sigma)$, there is at most (resp. least) one state q such that $t \in \mathcal{L}(q, \mathcal{A})$.

Definition 2.33 (Syntactic determinism/completeness)**.** A reduced automaton is said *syntactically deterministic (resp. complete)* if, for every function f and type-compatible states $q_1, \ldots, q_{|f|}$, there is at most (resp. least) one transition $f(q_1, \ldots, q_{|f|}) \to q$ in $\mathcal{A}$.

Proposition 2.34. *The semantic and syntactic definitions of determinism are equivalent.*

Proposition 2.35. *The semantic and syntactic definitions of completeness are equivalent in the case of a deterministic automaton.*

Note that a deterministic and complete tree automaton over Σ can equivalently be seen as a finite algebra over Σ, and vice versa. The tree automata formalism is extended with a new kind of transitions: ϵ-transitions. These transitions allow to not use a function symbol with the transition. They do not augment the expressivity of the formalism but sometimes allow for a simpler representation.

Definition 2.36 (ϵ-transition)**.** An ϵ-transition is a transition of the form $q \to q'$, with $\tau(q) = \tau(q')$. The inductive definition of the language recognized in a state q of a tree automaton $\mathcal{A}$ with transitions Δ is then extended to take them into account:

$$\mathcal{L}(\mathcal{A}, q) = \{f(t_1, \ldots, t_n) \mid f(q_1, \ldots, q_n) \to q \in \Delta \wedge \bigwedge_{i \in [1 \ldots n]} t_i \in \mathcal{L}(\mathcal{A}, q_i)\} \cup \bigcup_{q' \to q \in \Delta} \mathcal{L}(\mathcal{A}, q').$$

Note that, because of the inductive nature of this definition, a chain of ϵ-transitions $q_1 \to \ldots \to q_n$ in an automaton $\mathcal{A}$ does imply $\mathcal{L}(\mathcal{A}, q_1) \subseteq \mathcal{L}(\mathcal{A}, q_n)$.

Definition 2.37 (Representing Herbrand models with automata)**.** A state q of type *bool* in an automaton $\mathcal{A}$ can be associated to the Herbrand interpretation $\mathcal{L}(\mathcal{A}, q)$, i.e., every relation R represents the set of terms $\{\overrightarrow{t} \mid R(\overrightarrow{t}) \in \mathcal{L}(\mathcal{A}, q)\}$.

Example 2.38. Continuing Example 2.30, let q be a new state of type *bool* and consider the following new transitions:

$$\{\text{Has0}(q_{has0}) \to q,\ \ node(q_t, q_{nat}, q_t) \to q_{isNode},\ \ \text{IsNode}(q_{isNode}) \to q\}.$$

This new state q in the automaton extended with the three new transitions defines the Herbrand model where Has0 represents the unary relation of the set of trees that contain z and IsNode the set of non-empty trees.

Tree automata are closed under complement, union, and intersection. We write $\overline{\mathcal{A}}$ for the typed complement of the automaton $\mathcal{A}$, i.e. the automaton such that, for any state q of $\mathcal{A}$, we have $\mathcal{L}(\overline{\mathcal{A}}, q) = \overline{\mathcal{L}(\mathcal{A}, q)} = \{t \mid t \in \mathcal{T}(\Sigma) \wedge \tau(t) = \tau(q) \wedge t \notin \mathcal{L}(\mathcal{A}, q)\}$. We also write q^c to refer to the state q in $\overline{\mathcal{A}}$.

CHAPTER 3

STATE OF THE ART

This chapter presents some of the techniques that are used to automatically prove properties about programs manipulating algebraic datatypes. We can coarsely split these techniques into two categories: those which try to *syntactically* prove that the properties are a consequence of (a logic representation of) the program by a formal proof, and those which try to *semantically* approximate the program by a simpler representation, a model, on which the properties can more easily be checked. These two categories are not based on a formal distinction.

3.1 Syntactic methods

A syntactic method builds a formal proof whose hypothesis are clauses defining the program and whose conclusion is the property we want to prove. In practice, such solvers use SMT-solvers to handle theory-specific properties that are required in the proof, which allows them to more easily prove quite diverse properties. We only expose the method whose verification objectives are closest to ours.

Automating Induction for Solving Horn Clauses, Unno *et al.* [36] This work focuses on solving the satisfiability of Constrained Horn Clauses (Horn clauses over a specific theory) and is intended, and used, as a backend solver of a tool capable of reducing the task of proving properties to the satisfiability of CHCs, such as RCaml[1]. The verification method deployed is rather distant from ours but can also prove many interesting relational properties over algebraic datatypes. In this work, the satisfiability of Constrained Horn Clauses over any SMT-supported theory are reduced to the validity checking of first-order formulas with inductively-defined predicates. They propose a syntactic proof system that is able to automatically use inductive reasoning on these

1. `https://github.com/hiroshi-unno/coar`

predicates by using case disjunction and inductive invariants. An SMT solver is used to check theory-related constraints that appear as subgoals of some proof steps. This inductive theorem prover analyses all the relations together and try to infer invariants using multiple variables if needed.
Here is an example proving that the *double* function (relating n to $n+n$) only returns even numbers. An inductive definition of natural numbers is used for an easier comparison with other techniques, but this solver is also capable of handling native integers.

Example 3.1 (Double function)**.** The (translation into Horn clauses of the) program consists of the following inductively defined relations, where the output of the function (boolean or not) is the last argument of the relation, like it is done in [36]:

$$\begin{array}{ll} double_1: & double(z,z) \\ double_2: & double(s(X),s(s(R))) \Leftarrow double(X,R) \\ even_1: & even(z,\top) \\ even_2: & even(s(z),\bot) \\ even_3: & even(s(s(X)),R) \Leftarrow even(X,R) \end{array}$$

The goal is to prove the property $\bot \Leftarrow double(X,R) \land even(R,\bot)$.
For this, the solver adds $double(X,R)$ and $even(R,\bot)$ as hypotheses and tries to prove $\bot$. The solver also adds the universally quantified property $\bot \Leftarrow double(X',R') \land even(R',\bot)$ as induction hypothesis, which is also the only clause whose head is $\bot$. However, the solver can not immediately apply this induction hypothesis, although it would prove the goal, because it would not be safe. Their induction principle is that the induction hypotheses can only be applied to subderivations of the atoms $double(X,R)$ and $even(R,\bot)$, i.e. to atoms $double(X',R')$ and $even(R',\bot)$ that derive from $double(X,R)$ and $even(R,\bot)$ using their definition. The proof given by the solver is the following, with Unfold being the case disjunction on the definition of an atom.
Unfold of $double(X,R)$:

1. case of $double_1$: In this case, we have $X = z$ and $R = z$.
   Unfold of $even(z,\bot)$:

   (a) case of $even_1$: Unifying $even(z,\bot)$ with $even_1$ means unifying $\bot$ (from the hypothesis atom $even(z,\bot)$) with $\top$ (from the atom $even(z,\top)$ of the defining rule $even_1$), i.e., $\bot = \top$, which gives a proof of $\bot$.

   (b) case of $even_2$: Gives $z = s(z)$, which is unsatisfiable in the theory of algebraic datatypes, so $\bot$ is proven.
   (c) case of $even_3$: Gives $z = s(s(X'))$, so $\bot$ is proven.

2. case of $double_2$: In this case, we have $X = s(X')$ and $R = s(s(R'))$, with $double(X', R')$ as new hypothesis.
   Unfold of $even(s(s(R')), \bot)$:
   (a) case of $even_1$: Gives $s(s(R')) = z$ and $\bot = \top$, so $\bot$ is proven.
   (b) case of $even_2$: Gives $s(s(R')) = s(z)$, so $\bot$ is proven.
   (c) case of $even_3$: In this case, we have $even(R', \bot)$ as new hypothesis. The hypothesis environment therefore contains both atoms $double(X', R')$ and $even(R', \bot)$ that are subderivations of respectively $double(X, R)$ and $even(R, \bot)$. The induction hypothesis can therefore safely be used and adds $\bot$, the goal, as a new hypothesis.

In the above Example 3.1, the solver used an external SMT-solver for checking (un)satisfiability of the (algebraic datatype) theory-related constraints such as $z = s(z)$. An evaluation of this method on a part of our benchmarks can be found in Section 9.2.

3.2 Semantic methods

Semantic methods are trying to infer a simpler representation of the program on which it is easier to check for the desired properties. One common framework is to represent the property as a set of bad program states that should not be reachable by the program, or, equivalently if the formalism for describing this set is closed by complement, a set of good program states from which the program should not deviate. A program state is also called a *configuration* to avoid using the word 'state'. However, the exact set of reachable configurations usually can not be represented in a formalism which allows to simply check for the emptiness of its intersection with the set of bad configurations. The solver therefore computes an over-approximation of the reachable configurations for which the safety, i.e. the emptiness of the intersection with bad configurations, can easily be checked.
When proving properties on functional languages manipulating trees, a configuration can be represented as a term representing a partial execution of the program and the

program as a term rewriting system (TRS). Any set E of terms which contains the initial configurations (usually the main program function applied to any type-compatible terms) and that is closed by rewriting, i.e., such that any rewriting of a pattern that is in E is also in E, is an over-approximation of the reachable configurations.
Usually, the program inputs and the set of bad configurations are represented by simple regular languages, which leads to the following definition of a safety problem:

Definition 3.2 (Safety problem instance)**.** A *safety problem* instance consists of a tree automaton $\mathcal{A}_I$ of input program configurations, a tree automaton $\mathcal{A}_B$ of bad configurations, and a suitable program representation P such as a TRS.

The set of configurations that can be reached from an initial configuration using the program is called the set of *reachable configurations*. The corresponding decision problem is whether the set of reachable configurations intersects the bad configurations. If the set of reachable configurations do not intersect the bad configurations, i.e., the program is safe, then the safety problem instance is said to be a *positive* instance. Otherwise, it is said to be *negative*. For example, here is a positive safety problem stating that the *double* function only returns even numbers.

Example 3.3 (Positive safety problem: *double* returns even numbers)**.** We consider the program *double* written as a TRS $\mathcal{D}_d$ in Example 2.20. Let I be the regular set of initial program configurations: $I = \{double(n) \mid n \in \mathcal{T}_{Nat}(\Sigma)\}$. We want to assert that the output of the *double* function is even, so the set of bad configurations is the set of odd numbers: $B = \{s^{2n+1}(z) \mid n \in \mathbb{N}\}$.

3.2.1 Automatic verification with regular language

This section focuses on techniques that over-approximate the reachable configurations with regular languages. Using regular languages as an over-approximation formalism of reachable configurations can be traced back to the work of Jones *et al.* [23], then extended to higher-order [21], and completed by a more recent and efficient construction [22]. These three papers use *regular tree grammar* (the tree counterpart of regular string grammar, recognizing the same languages as tree automata) for the representation of regular languages. Tree automata are also used for the abstraction of regular languages, but mostly in more recent work [15, 30, 27, 19]. These papers all propose a correct procedure, but with different efficiency and completeness properties.

The subclass of safety problems that can indeed be solved using only regular abstractions is called regular safety problems.

Definition 3.4 (Regular safety problem)**.** A *regular safety problem* is a safety problem that either is a negative instance or that is a positive instance for which there exists a *regular* over-approximation of the reachable configurations that does not intersect the bad configurations.

The safety problem of Example 3.3 is regular. Examples 3.5 and 3.6 both solve it by exhibiting a regular representation of the reachable terms that do not intersect the bad configurations. These methods are all restricted to solving regular safety problems, so a method is said to be *regular-complete* if it can decide any regular safety problem. We now give an overview of the methods described in [22] and [14], one using regular tree grammar and the other regular tree automaton.

Flow analysis of lazy higher-order functional programs, Jones *et al.* [22] Regular model-checking also employs this method in a more general setting [22]. The program is modeled as a (left-linear higher-order) TRS, and its input is modeled by a regular tree grammar. This paper proposes an algorithm that, given a TRS and a grammar describing the program's inputs, computes an over-approximation of the program's output. The main idea is to iteratively complete the input grammar G_0 by adding new symbols and transitions until reaching a fixed-point, meaning that the set of terms represented by the grammar is closed by rewriting. We write G_i the grammar at step i, and G_* the fixed point.
This method always terminates and thus G_* is a (finite) regular grammar that always over-approximates the output, hence the method's correctness. However, it is not regular-complete. That is, there exists a TRS, input grammar, and regular over-approximation of the reachable terms that would allow to conclude that the program is safe, but the procedure only produces approximations that are too coarse to prove it. There are two main reasons for this completeness failure. The first one is that the over-approximation process does not take the property (bad configurations) into account, which could be used to avoid over-approximating something that it should not. The second reason is that there is no way of parameterizing the over-approximation's precision, so if the approximation is too coarse, there is no way to improve it. We now give an example of this procedure successfully proving the (regular) safety problem of Example 3.3.

Example 3.5. This example solves the safety problem from Example 3.3.
The set of bad configurations, odd numbers, can be described by the symbol R_B in the following regular tree grammar G_B, which should be read as "R_B can be rewritten to either $s(z)$ or to $s(s(R_B))$":

$$G_B = \quad R_B \rightarrow s(z) \mid s(s(R_B))$$

The set of initial configurations I can be described by the symbol R_0 in the following regular tree grammar G_0:

$$G_0 = \begin{array}{l} R_0 \rightarrow double(N) \\ N \rightarrow z \mid s(N) \end{array}$$

Then, the grammar G_0 is successively extended using the rewriting system $\mathcal{D}_d$ that represents the program. Because R_0 can derive $double(z)$ but not z, which is one rewriting step away from $double(z)$ in $\mathcal{D}_d$, a symbol R_1 which derives to z is added to the the derivatives of R_0. G_0 can also derive terms of the form $double(s(X))$ which rewrites to $s(s(double(X)))$, for which R_2 is added in G_1. G_1 is then completed into G_2 in the same way.

$$G_1 = \begin{array}{l} R_0 \rightarrow double(N) \mid R_1 \mid R_2 \\ N \rightarrow z \mid s(N) \\ R_1 \rightarrow z \\ R_2 \rightarrow s(s(double(X))) \\ X \rightarrow N \end{array} \qquad G_2 = \begin{array}{l} R_0 \rightarrow double(N) \mid R_1 \mid R_2 \\ N \rightarrow z \mid s(N) \\ R_1 \rightarrow z \\ R_2 \rightarrow s(s(double(X))) \mid s(s(R_1)) \mid s(s(R_2)) \\ X \rightarrow N \end{array}$$

A fixed point is reached with G_2, as $G_3 = G_2$.

Then computing the intersection of the languages denoted by G_2 and G_B, we find that $\mathcal{L}(G_2) \cap \mathcal{L}(G_B) = \emptyset$. Indeed, every natural number generated by the grammar G^* is even, and the property is thus proved. The reachable terms are exactly represented in this example, which is not always the case when using this method.
This idea of extending the current representation of the reachable configurations by one step of the program can be found again in the following paper, which we present to illustrate the use of tree automata.

Termination criteria for tree automata completion, Genet *et al.* [14] In this paper, the author presents the Tree Automaton Completion algorithm (TAC for short), and termination criteria. The TAC algorithm takes as input a tree automaton $\mathcal{A}_0$ representing the program's inputs in a state q_f and a TRS $\mathcal{D}$, and computes the set of reachable terms.

This algorithm is very similar in essence to that of completing a grammar, but differs in details, particularly in variable names handling, which results in differences about how approximations are done. After taking $\mathcal{A}_0$ and $\mathcal{D}$ as input, the core of TAC consists in iteratively computing $\mathcal{A}_1$, $\mathcal{A}_2$, ... to follow rewriting steps of $\mathcal{D}$ until a fixed point, named $\mathcal{A}^*$, is reached.

As previously, we have that at each step i, $\mathcal{A}_i$ satisfies some properties: first, $\mathcal{L}(A_i, q_f) \subseteq \mathcal{L}(A_{i+1}, q_f)$; second, if $t \in \mathcal{L}(\mathcal{A}_i, q_f)$ and t rewrites to t' using one step of $\mathcal{D}$, then $t' \in \mathcal{L}(\mathcal{A}_{i+1}, q_f)$. We therefore have, if a fixed-point $\mathcal{A}^*$ is reached, that it contains at least all the reachable configurations.

Example 3.6. This example also solves the safety problem from Example 3.3.

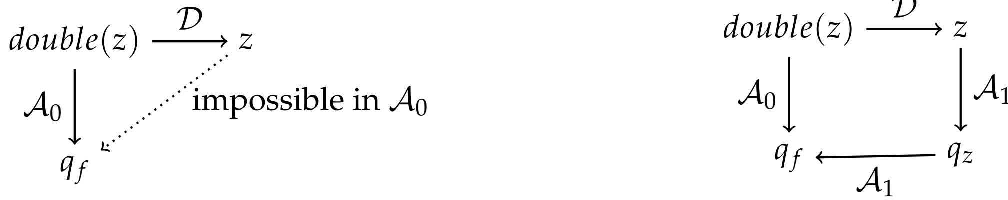
The set of bad configurations can be represented in the state q_o of the following regular tree automaton $\mathcal{A}_B$:

$$z() \to q_e \qquad s(q_e) \to q_o \qquad s(q_o) \to q_e$$

The set of initial configurations I can be represented in the state q_f of the following regular tree automaton $\mathcal{A}_0$:

$$z() \to q_n \qquad s(q_n) \to q_n \qquad double(q_n) \to q_f$$

This automaton $\mathcal{A}_0$ is, in the same way as G_0 from Example 3.5, not closed by rewriting by $\mathcal{D}$. Indeed, $double(z)$ is recognized but z is not, and the same is true for $double(s(z))$ and $s(s(double(z)))$. These are called *critical pairs*, and the goal of the completion is to eliminate them by allowing each term t' that has been rewritten from t to be recognized in the same state as t. This process is depicted in the following figure:



The completion of these two critical pairs yields the following transitions to be added to $\mathcal{A}_0$: $\{z \to q_z,\ \ q_z \to q_f,\ \ \ s(q_o) \to q_e,\ \ s(q_f) \to q_o,\ \ q_e \to q_f\}$.
$\mathcal{A}_1$ has no critical pair, so $\mathcal{A}^* = \mathcal{A}_1$. The intersection between the languages of $\mathcal{A}^*$ and $\mathcal{A}_B$ is empty, which allows to prove the property stating that *double* only returns even numbers.

This short example does not reflect the limitations of the TAC algorithm. One particularity of this algorithm is that the resulting automaton $\mathcal{A}^*$, if any, recognizes exactly the reachable terms $\mathcal{D}^*(\mathcal{I})$ [2]. This is an interesting particularity for some usage, but not for abstraction, as the language $\mathcal{D}^*(\mathcal{I})$ may not be regular and therefore the algorithm would diverge. An instance making the TAC procedure loop can be found in the chapter 3 of [18].
To remedy this problem, the authors proposed to use equations over patterns to merge terms into equivalence classes. For example, $X = s(s(s(X)))$ separates natural numbers into the three classes of natural numbers modulo 3. Using equations makes the abstraction coarser, and therefore allows the algorithm to terminate on instances on which it did not without equations. However, contrary to the method proposed in [22], the abstraction's precision can be controlled by which equations are used. Given the correct equations, the TAC algorithm always finds a sufficient over-approximation, given that there exists one. The new challenge is to find those equations. The authors proposed an algorithm to pick equations from a restrained set of equations, therefore allowing the TAC algorithm to decide many interesting regular safety problems. However, using this restricted set of equations, this procedure is not regular-complete.
This method is efficient on small programs but lacks modularity to be able to treat substantial programs, which is addressed in a more recent work by Haudebourg *et al.* [19].

2. In reality, TAC as shown above computes a *non-controlled* over-approximation, but the input can be transformed so that the output, when terminating, is exactly $\mathcal{D}^*(\mathcal{I})$

We now present two methods that still infer an approximation of reachable configurations but now relying on a Counter-Example Guided Abstraction Refinement (CEGAR), in which a (here regular) abstraction of functions is refined, using examples, until the property is proven true or false.

Automata-based abstraction for automated verification of higher-order tree-processing programs, Matsumoto *et al.* [30] The goal of the paper is to abstract higher-order functional programs into Higher-Order Recursion Schemes (HORS) *on finite types* using predicate abstraction. Each type of the concrete program is abstracted by a state of a tree automaton, where, as before, a state q denotes every term t such that $t \in \mathcal{L}(\mathcal{A}, q)$. Having done so, they can use a *higher-order model checker* to check for a safety property. If the abstract program (the one manipulating automata states) is safe, then the concrete program is safe and the model checking procedure can conclude. If the abstract program has a problematic run, there are two possibilities: First, the concrete program admits the same run, and the property is thus proved to be false. Second, the approximation automaton was too coarse, as the run found can not exist in the concrete program, so the abstracting automaton is refined and the verification procedure continues with this new model. This method is a particular case of Counter-Example Guided Abstraction Refinement (CEGAR) and is based on the work of [26], but uses tree automata as abstraction instead of booleans.
A similar approach is used in *Regular Language Type Inference with Term Rewriting* [19], but the program is represented by a TRS and the abstraction is therefore a transition system manipulating the states of a tree automaton.
These two methods [30] and [19] are both correct and regular-complete. The method from [19] also provides an open-access tool timbuk4 [3].

Comparison summary: Regarding expressivity, [22] did not mention any scope of application for their method, but it seems to solve strictly less problems than the other three. [14] is complete for a (strict) sub-class of regular safety problems that is defined in the paper, and both [30] and [19] are regular-complete.
Regarding speed and memory consumption, [22] did not mention any implementation. The three other methods have comparable speed for small functions. The memory usage for [19] is much more stable than in [14], and does not explode on bigger instances.

3. `https://people.irisa.fr/Thomas.Genet/timbuk/index.html`

Memory usage is not mentioned in [30].

Limitations of regular languages: All of the previously-presented methods use regular approximation of sets of terms. Regular languages are widely used for their good closure properties, decision problem properties, and simplicity. However, their expressivity is quite limited when trying to represent relations. This was not a problem for regular safety problems, as they all can be solved without exhibiting any (infinite) relation between terms. However, many properties require the comparison of elements (of an infinite domain). These kind of properties are not regular and therefore can not be handled by the techniques presented in the previous section.
For example, even the identity relation on natural numbers $\{(n, n) \mid n \in \mathcal{T}_\Sigma(nat)\}$ can not be represented using a regular automaton. Therefore the safety problem with the regular set of inputs $I = \{f(n) \mid n \in \mathcal{T}_\Sigma(Nat)\}$, $B = \{false\}$, and the rewriting system

$$f(N) \to eq(N, N)$$

$$eq(z, z) \to true \qquad eq(s(X), s(Y)) \to eq(X, Y)$$

$$eq(s(X), z) \to false \qquad eq(z, s(Y)) \to false$$

is not regular, i.e., fails to be proved by the techniques presented in the previous section. There exist automata recognizing over-approximations of this *eq* relation, but then not representing the exact relation, which does not allow to conclude. To remedy this problem and prove such *relational* properties, focus was turned to more expressive formalisms to represent the functions of the programs, such as extensions of tree automata called *automata on tuple*, which we also call *convoluted (tree) automata*.

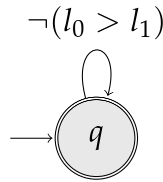
3.2.2 Automatic verification with relational formalisms

We now discuss two papers that prove programs by representing their relations using more expressive formalisms than regular tree automata. This first one combines *symbolic automata* [8] and *automatic relations* and the second one uses and extends *convoluted tree automata*.

Symbolic Automatic Relations and Their Applications to SMT and CHC Solving, Shimoda *et al.* [34] This work defines a new formalism called *symbolic automatic relations* meant to combine *symbolic automata* [8] and *automatic relations* [16]. Symbolic

automatic relations are automata-represented relations between words, hence the "automatic" part of the name [16], which is the vocabulary used when dealing with convolution of words. Moreover, transitions of these automata may use domain-specific operations and predicates, hence the "symbolic" part of the name.

Example 3.7 (Simple automata taken from [34])**.** Here is an automaton $\mathcal{A}$ recognizing the convolution of two words w_0 and w_1 over the domain of natural numbers (every symbol is a number). This automaton recognizes words such that, at every index i, it is not true that $w_0[i]$, the ith element of w_0 if any, is greater that $w_1[i]$. This indirect formulation, using a negation, is necessary for handling words w_0 and w_1 of different length. The transitions of the automata have (as usual) no concept of ith symbol of a word and can only access the current symbol. We write l_0 for the current symbol of the word w_0 and l_1 for that of w_1. This automaton has only one (initial) state q and one transition $q \overset{\neg(l_0 > l_1)}{\rightarrow} q$.



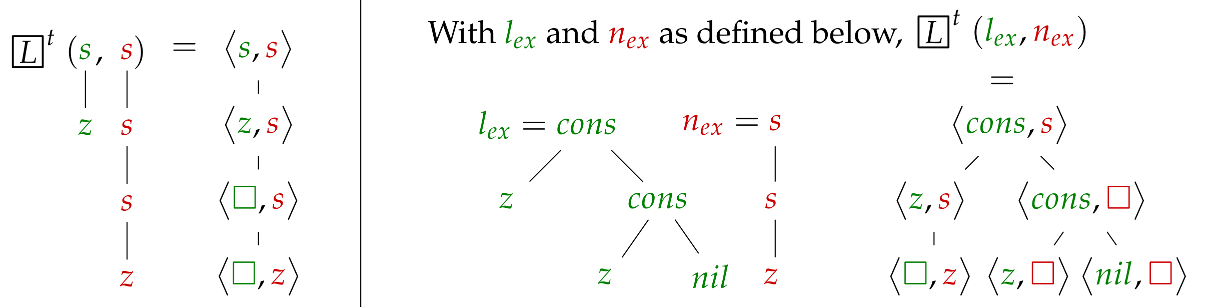
Considering that a word encodes a list, this automaton can then be used to define the *sorted* predicate on lists of natural numbers by $sorted(w) \overset{def}{=} (0 \cdot w, w) \in \mathcal{L}(\mathcal{A}, q)$.

These kind of automata allow to express various relations on a specific domain and can be used to represent the model of a set of formula. Although not detailed in the paper, they extend this formalism from words to trees by using *convolutions*, which are presented in the following paragraph. A procedure for deciding a certain class of first-order formula on these automata is given, thus implementing the teacher part of a Learner/Teacher verification algorithm (presented in Section 4.3). However, every automata has to be given by the user as no procedure for learning an automaton is given, which is left as further work.
We now take a quick look at convoluted tree automata, which are used in the work of Haudebourg *et al.* [18] and formally defined in Chapter 5.

Convoluted tree automata Automata recognizing string relation have been introduced in [4] and their extension on tree relations can be traced back to [9] and [24]. More recently, automata recognizing tree relations can be found in the tree automata

reference [7] or in T. Haudebourg's thesis [18]. To make an automaton recognize a n-ary relation, the main idea is to read the n trees at the same time instead of separately. However, automata classically only read one symbol at a time. To adapt the automata formalism, the idea is to create new symbols representing tuples of symbols, using a *convolution* operation. Terms are then overlayed by following their syntax tree.

Example 3.8 (Convolution)**.** Using $\Sigma = \{z, s, nil, cons\}$, the new alphabet for a binary relation is $\Sigma = \{\langle s,s\rangle, \langle s,z\rangle, \langle s,\square\rangle, \langle z,s\rangle, \langle z,z\rangle, \langle z,\square\rangle, \langle \square,s\rangle, \langle \square,z\rangle, \langle cons,s\rangle, \ldots\}$ with $\square$ a *padding* symbol. The convolution operator is written $\boxed{L}^t$; the convolution between the terms $s(z)$ and $s(s(s(z)))$ and between the list $cons(z, cons(z, nil))$ and its size $s(s(z))$ are depicted in the figure below.



Note that, when overlaying terms of a different shape, every subterm is paired with the subterm that is at the same position, if any, and otherwise with the padding $\square$. This method is simply known as *convolution of terms*, or *left convolution* (hence the L notation).

A state of a convoluted automaton therefore recognizes a tuple $(t_1, \ldots, t_n)$ if it recognizes its convolution $\boxed{L}^t$ $(t_1, \ldots, t_n)$. The automaton recognizing the convolution of the equality relation of natural numbers in a state q_f is then given as the automaton $\mathcal{A}$ with states $\{q_f\}$ and transitions

$$\langle z,z\rangle() \to q_f \qquad \langle s,s\rangle(q_f) \to q_f.$$

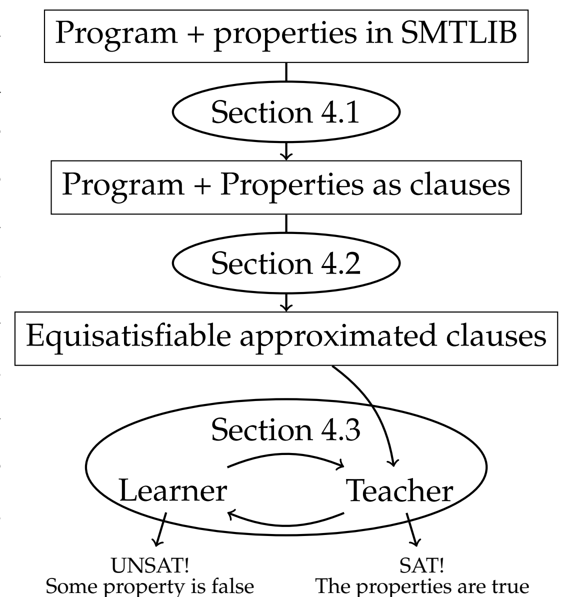
Automatic Verification of Higher-Order Functional Programs using Regular Tree Languages, Haudebourg *et al.* [18] This preliminary work has been started by T. Haudebourg during a stay in N. Kobayashi's group in The University of Tokyo. In it, relations are learned by following a CEGAR loop inspired by Garg *et al.* [12] which is more formally introduced in Section 4.3. It consists of a back and forth between an entity called teacher and an entity called learner, where at each step the learner will

propose an automaton and the teacher will check if it is correct, and if not gives back learning constraints to guide the learner. This technique has been applied in order to prove relational properties such that $length\ (rev\ l) = length\ l$, or $length\ (insert\text{-}sort\ l) = length\ l$.
We directly extend this preliminary work to make it more formal, practical, and proven, which corresponds to Chapters 5 and 6 of this thesis, and then devise a new formalism (Shallow Horn Clauses) to augment the capabilities of this approach, which corresponds to Chapters 7 and 8.

CHAPTER 4

GENERAL APPROACH: FROM PROGRAMS AND PROPERTIES TO PROOFS

This chapter focuses on the whole chain of transformation, going from the functional program and properties written in a "high-level" language SMTLIB to the yes/no answer to whether the properties are true in the program. First, the program and properties are translated from SMTLIB to a set of clauses that represents the same functions in Section 4.1. Then, this set of clauses is modified to allow for approximations in Section 4.2. Finally, the Learner/Teacher procedure that takes this set of clauses as input and checks for satisfiability is presented in Section 4.3.



Section 4.3 gives a generic view of the Learner/Teacher procedure. This generic view describes the roles of the Learner and Teacher and how they interact to define a model-inference procedure. This procedure is generic and will later be instantiated twice, in Chapters 6 and 8. The Learner/Teacher procedure presentation of Section 4.3 is generic in three ways: (a) the inferred models are abstract, no formalism to actually finitely represent them is given; (b) the Learner algorithm, which infers a model from examples, is not given ; (c) the Teacher algorithm, which checks that desired properties are true in a model, is not given either. In this thesis, we re-define and extend the convoluted automata formalism in Chapter 5 and a Learner and Teacher algorithm for those convoluted automata in Chapter 6. Then, we define Shallow Horn Clauses in Chapter 7 and adapt the Learner and Teacher to SHoCs in Chapter 8.

4.1 From SMTLIB to clauses

SMTLIB [1] is a language used for describing satisfiability problems in various theories and that is widely used in the verification community. We parse a restricted subset of SMTLIB 2.6 that is sufficient to describe elementary functional programs and properties. More precisely, we support (i) the definition of monomorphic algebraic datatypes (ii) the definition of (recursive) algebraic functions using simple pattern matching and if-then-else (iii) the definition of universally quantified properties in a first-order clausal syntax.
We did not find an out-of-the-box tool to generate the clauses using the encoding presented in this section, so we had to define one. Definitions of the datatypes do not require any particular processing. The recursive functions and properties, however, are transformed into clauses. The program, i.e. the set of functions that the program defines, is transformed into an equivalent set of clauses $\Gamma_!$ and the properties are transformed into an equivalent set of clauses $\Gamma_?$. Because we only support terminating and deterministic functions, $\Gamma_!$ has exactly one Herbrand model $\mathcal{M}_!$ and the properties are true in the program iff $\Gamma_?$ are true in $\mathcal{M}_!$ (written $\mathcal{M}_! \models \Gamma_?$). Termination and determinism are not checked by our tool but assumed.
During this transformation from the program and properties to clauses, functions are all transformed into relations. Every n-ary boolean function $f : (\tau_1, \ldots, \tau_n) \to Bool$ is transformed into an n-ary relation $R_f : (\tau_1, \ldots, \tau_n) \to Bool$ and every non-boolean n-ary function $f : (\tau_1, \ldots, \tau_n) \to \tau_0$ is transformed into an $n+1$-ary relation $R_f : (\tau_1, \ldots, \tau_n, \tau_0) \to Bool$.

Example 4.1 (All0, No0, IsEmpty)**.** In this example, we define three functions on trees of natural numbers t: (i) All0, returning true iff all the elements of t are equal to z. (ii) No0, returning true iff all the elements of t are different than z. (iii) IsEmpty, returning true iff t is equal to $leaf$. The property we assert is that, for any tree T, IsEmpty$(T) \Leftarrow$ All0$(T) \land$ No0(T). Here is the corresponding SMTLIB file, to show what SMTLIB looks like:

```
(set-logic HORN)

(declare-datatypes ((nat 0) (ntree 0))
        ((      (z)
                (s (pred nat)))
         (      (leaf)
```

```
                (node (hd nat) (l ntree) (r ntree)))))

(define-fun-rec All0 ((t ntree)) Bool
        (match t (
                (leaf true)
                ((node n t1 t2)
                        (ite (= n z) (ite (All0 t1) (All0 t2) false) false
                            )))))

(define-fun-rec No0 ((t ntree)) Bool
        (match t (
                (leaf true)
                ((node n t1 t2)
                        (ite (= n z) false (ite (No0 t1) (No0 t2) false)))
                            )))

(define-fun-rec IsEmpty ((t ntree)) Bool
        (match t (
                (leaf true)
                ((node n t1 t2) false))))

;; property all0_and_no0_is_empty
(assert (forall ((t ntree))
        (=> (and (All0 t) (No0 t))
                (IsEmpty t))))


(check-sat)
```

The clauses extracted from these definitions are the same as those of Section 1.1, which we label here because of their use in Section 4.3.

I_1 : $\text{IsEmpty}(\mathit{leaf})$

I_2 : $\bot \Leftarrow \text{IsEmpty}(\mathit{node}(T_1, E, T_2))$

A_1 : $\text{All0}(\mathit{leaf})$

A_2 : $\text{All0}(\mathit{node}(T_1, z, T_2)) \Leftarrow \text{All0}(T_1) \wedge \text{All0}(T_2)$

A_3 : $\bot \Leftarrow \text{All0}(\mathit{node}(T_1, s(E), T_2))$

A_4 : $\text{All0}(T_1) \Leftarrow \text{All0}(\mathit{node}(T_1, z, T_2))$

A_5 : $\text{All0}(T_2) \Leftarrow \text{All0}(\mathit{node}(T_1, z, T_2))$

N_1 : $\text{No0}(\mathit{leaf})$

N_2 : $\text{No0}(\mathit{node}(T_1, z, T_2)) \Leftarrow \text{No0}(T_1) \wedge \text{No0}(T_2)$

N_3 : $\bot \Leftarrow \text{No0}(\mathit{node}(T_1, s(E), T_2))$

N_4 : $\text{No0}(T_1) \Leftarrow \text{No0}(\mathit{node}(T_1, z, T_2))$

N_5 : $\text{No0}(T_2) \Leftarrow \text{No0}(\mathit{node}(T_1, z, T_2))$

Finally, the clause for the property `all0_and_no0_is_empty` stays the same:

$$\varphi = \text{IsEmpty}(T) \Leftarrow \text{All0}(T) \wedge \text{No0}(T).$$

Note that there is only one Herbrand model $\mathcal{M}_!$ for the set of clauses defining the program. Moreover, in this case, $\mathcal{M}_! \models \varphi$.

Translation of functions into clauses We give here the principle of the translation of functions into clauses. We begin by giving an informal description directed by the syntax of expressions and then show a step-by-step translation of the height definition in Example 4.2.
A function definition in SMTLIB is of the form

```
(define-fun-rec f ((arg_1 type_1) ... (arg_n type_n)) type_0
        expr)
```

The function `f` has n arguments and returns a value of type `type_0`. Each `arg_i` is the ith argument and `type_i` its type. The clauses defined from this function will *all be of the form* $R_f(arg_1, \ldots, arg_n, arg_0) \Leftarrow B$ with the clause's body B defined w.r.t `expr` and possibly containing negated atoms. The procedure is then compositionally defined on the form of `expr` by adding elements to B. A function may yield multiple clauses. This is taken into account due to the translation function being able to, when handling `expr`, fork, which causes the current clause to be duplicated. We consider here that an expression is either (i) a match construction (ii) an if-then-else construction (iii) a value or function application.
The (restricted) SMTLIB match expression is of the form

```
(match x (
        (pattern_1 expr_1)
        ...
        (pattern_k expr_k)
        ))
```

In this case, the translation procedure forks k times, one for each branch. For any branch i, the constraint $x = pattern_i$ is added to their copy of the clause and the translation continues on `expr_i`. An if-then-else expression is of the form

```
(ite condition expr_then expr_else)
```

In this case, the translation function forks twice, one for the `then` branch and one for the `else` branch. First of all, the condition `condition` is translated into a clause $R_c(\ldots) \Leftarrow B_c$ with B_c a set of atoms obtained from `condition` and that may involve other relation symbols. In a model, satisfying the atoms B_c makes the atom $R_c(\ldots)$ equivalent to the condition `condition`. For the `then` branch, the atoms $R_c(\ldots) \wedge B_c$ are thus added to B and the translation continues on `expr_then`. For the `else` branch, the atoms $\neg R_c(\ldots) \wedge B_c$ are added to B and the translation continues on `expr_else`.
When the expression `expr` is a value v, the constraint $arg_0 = v$ is added to B and the translation stops.

Example 4.2 (From the *height* function to the Height relation)**.** Let the definition of *height* be the following:

```
(define-fun-rec height ((T tree)) nat
(match T (
  (leaf z)
  ((node T1 N T2) (s (ite (leq (height T1) (height T2))
      (height T2) (height T1)))))))
```

We define the binary relation Height. The clauses defined will thus be of the form $\text{Height}(T, V) \Leftarrow B$ with the body B of the clause depending on the function's expression. This `height` function is defined as a pattern-matching with two cases, which forks the transformation into two cases:

- The first case forces T to be of the form *leaf*, so the constraint $T = leaf$ is added in B. Moreover, the first case returns z, so we also add the constraint $V = z$ to B. This results in the clause $\text{Height}(T, V) \Leftarrow T = leaf \wedge V = z$.
- The second case forces T to be of the form $node(T1, N, T2)$ and returns `(s (ite (leq (height T1)(height T2))(height T2)(height T1)))`, so the constraint $T = node(T1, N, T2)$ is added to B. Because s, the function applied to the returned value, is simply a constructor, the atom $V = s(V')$ is added to the clause body B and the program expression is now `(ite (leq (height T1)(height T2))(height T2)(height T1))`. When an if-then-else constructor `ite` is encountered, its condition is first translated. The clausal version of the condition is $\text{Leq}(N1, N2) \Leftarrow \text{Height}(T1, N1) \wedge \text{Height}(T2, N2)$. Note that, Leq being already defined (as it is used in the definition of `length`), this clause is a constraint for the Height relation. Then, the if-then-else forks the translation into two branches:

- The first branch has the if-then-else clause's atoms as additional constraints in B and the 'then' branch as current program expression. Concretely, the literals $\text{Height}(T1, N1) \wedge \text{Height}(T2, N2) \wedge \text{Leq}(N1, N2)$ are added to B and the current program's expression is `(height T2)`. This program expression corresponds to adding the atom $\text{Height}(T2, V')$ to B. The resulting clause is $\text{Height}(T, V) \Leftarrow T = node(T1, N, T2) \wedge V = s(V') \wedge \text{Height}(T1, N1) \wedge \text{Height}(T2, N2) \wedge \text{Leq}(N1, N2) \wedge \text{Height}(T2, V')$.
- The second branch has the negation of the condition as additional constraints in B and the 'else' branch as current program expression. Concretely, the literals $\text{Height}(T1, N1) \wedge \text{Height}(T2, N2) \wedge \neg\text{Leq}(N1, N2)$ are added to B and the current program's expression is `(height T1)`. This program expression corresponds to adding the atom $\text{Height}(T1, V')$ to B. The resulting clause is $\text{Height}(T, V) \Leftarrow T = node(T1, N, T2) \wedge V = s(V') \wedge \text{Height}(T1, N1) \wedge \text{Height}(T2, N2) \wedge \neg\text{Leq}(N1, N2) \wedge \text{Height}(T1, V')$.

This translation thus yields three clauses that, once simplified, are:

$\text{Height}(leaf, z)$

$\text{Height}(node(T1, N, T2), s(N2)) \Leftarrow \text{Height}(T1, N1) \wedge \text{Height}(T2, N2) \wedge \text{Leq}(N1, N2)$

$\text{Height}(node(T1, N, T2), s(N1)) \Leftarrow \text{Height}(T1, N1) \wedge \text{Height}(T2, N2) \wedge \neg\text{Leq}(N1, N2)$.

This third clause is, in accordance with Definition 2.21, rather written in the form $\text{Height}(node(T1, N, T2), s(N1)) \vee \text{Leq}(N1, N2) \Leftarrow \text{Height}(T1, N1) \wedge \text{Height}(T2, N2)$, which is the form used in the implementation. However, in this manuscript, we write it $\text{Height}(node(T1, N, T2), s(N1)) \Leftarrow \text{Height}(T1, N1) \wedge \text{Height}(T2, N2) \wedge \text{Leq}(N2, N1)$ to simplify the presentation (by implicitly using $\text{Leq}(N1, N2) \iff \neg\text{Leq}(N2, N1)$). Finally, because a unary function has been transformed into a binary function, one last clause is added to this translation: $N1 = N2 \Leftarrow \text{Height}(T, N1) \wedge \text{Height}(T, N2)$. This last clause is called the *functionality clause* and enforces the definition to have only one Herbrand model, the one representing exactly the height relation.

Here is a collection of definitions we use throughout this manuscript. Note the presence of functionality clauses in the definition of every relation that was (in the program) a non-boolean function and their absence in the case of boolean function, for which the boolean is not reified as the last argument. Non-boolean functions are Max, Plus, Height, HeightRB, and Len. The relation Shal relates trees with natural numbers greater or equal to their height. Recall also that the semantics used for these clauses is not the least fixpoint, but standard first-order semantics (on the theory of ADT).

Definition 4.3 (Clausal definition of Leq, Max, Plus, Len, Height, HeightRB, Shal)**.**

$$\mathrm{Leq}(z,z)$$
$$\mathrm{Leq}(z,s(N))$$
$$\mathrm{Leq}(s(X),s(Y)) \Leftarrow \mathrm{Leq}(X,Y)$$
$$\mathrm{Leq}(X,Y) \Leftarrow \mathrm{Leq}(s(X),s(Y))$$
$$\bot \Leftarrow \mathrm{Leq}(s(X),z)$$

$$\mathrm{Max}(z,z,z)$$
$$\mathrm{Max}(z,N,N)$$
$$\mathrm{Max}(N,z,N)$$
$$\mathrm{Max}(s(N),s(M),s(R)) \Leftarrow \mathrm{Max}(N,M,R)$$
$$R_1 = R_2 \Leftarrow \mathrm{Max}(N,M,R_1) \wedge \mathrm{Max}(N,M,R_2)$$

$$\mathrm{Plus}(z,z,z)$$
$$\mathrm{Plus}(z,N,N)$$
$$\mathrm{Plus}(s(N),M,s(R)) \Leftarrow \mathrm{Plus}(N,M,R)$$
$$R_1 = R_2 \Leftarrow \mathrm{Plus}(N,M,R_1) \wedge \mathrm{Plus}(N,M,R_2)$$

$$\mathrm{Len}(nil,z)$$
$$\mathrm{Len}(cons(X,L),N) \Leftarrow \mathrm{Len}(L,N)$$
$$N_1 = N_2 \Leftarrow \mathrm{Len}(L,N_1) \wedge \mathrm{Len}(L,N_2)$$

$$\mathrm{Height}(leaf,z)$$
$$\mathrm{Height}(node(T_1,E,T_2),s(N_1)) \Leftarrow \mathrm{Height}(T_1,N_1) \wedge \mathrm{Height}(T_2,N_2) \wedge \mathrm{Leq}(N_2,N_1)$$
$$\mathrm{Height}(node(T_1,E,T_2),s(N_2)) \Leftarrow \mathrm{Height}(T_1,N_1) \wedge \mathrm{Height}(T_2,N_2) \wedge \mathrm{Leq}(N_1,N_2)$$
$$N_1 = N_2 \Leftarrow \mathrm{Height}(T,N_1) \wedge \mathrm{Height}(T,N_2)$$

$$\mathrm{HeightRB}(leaf,z)$$
$$\mathrm{HeightRB}(node(T_1,E,T_2),s(N)) \Leftarrow \mathrm{HeightRB}(T_2,N)$$
$$N_1 = N_2 \Leftarrow \mathrm{HeightRB}(T,N_1) \wedge \mathrm{HeightRB}(T,N_2)$$

$$\mathrm{Shal}(leaf,N)$$
$$\mathrm{Shal}(node(T_1,E,T_2),s(N)) \Leftarrow \mathrm{Shal}(T_1,N) \wedge \mathrm{Shal}(T_2,N)$$
$$\mathrm{Shal}(T_1,N) \Leftarrow \mathrm{Shal}(node(T_1,E,T_2),s(N))$$
$$\mathrm{Shal}(T_2,N) \Leftarrow \mathrm{Shal}(node(T_1,E,T_2),s(N))$$
$$\bot \Leftarrow \mathrm{Shal}(node(T_1,E,T_2),z)$$

Once clauses are extracted from the program and properties, we can transform them to approximate the relations and more easily prove or refute the properties.

4.2 Approximation of clauses

Since the programs we verify are deterministic and terminating, their clausal representation has only one possible Herbrand model. However, this model may not be (precisely) representable using the formalism we chose for Herbrand models (first classical tree automata in Example 4.14, then convoluted automata in Chapters 5 and 6, and finally Shallow Horn Clauses in Chapters 7 and 8). Thus, trying to verify a property using an exact model of the relation will fail on such programs. We try to circumvent this

problem by approximating relations. Assume that we have a relation $\text{Plus}(N, M, U)$ relating N and M with their sum U and a relation $\text{Less}(N, M)$ relating N and M iff $N < M$. Let φ be the property

$$\varphi = \text{Less}(N, U) \Leftarrow \text{Plus}(N, M, U) \wedge \text{Less}(z, M)$$

This property φ cannot be proved straightforwardly using any of the above-mentioned formalisms because they cannot represent *exactly* the relation Plus. We can prove this property with an over-approximation of the relation Plus, say Plus^+, that is such that $\text{Less}(N, U) \Leftarrow \text{Plus}^+(N, M, U) \wedge \text{Less}(z, M)$ is true. The model inferred by our tool over-approximates the Plus relation by the set of triples $\{(s^n(z), z, s^n(z)) \mid 0 \leq n\} \cup \{(s^n(z), s^k(z), s^m(z)) \mid 0 \leq n < m \text{ and } 0 < k\}$, which is enough to carry out the proof.
In general, proving a property $R_1(\overrightarrow{p_1}) \vee \ldots \vee R_k(\overrightarrow{p_k}) \Leftarrow R_{k+1}(\overrightarrow{p_{k+1}}) \wedge \ldots \wedge R_n(\overrightarrow{p_n})$ can be done by proving the property $R_1^-(\overrightarrow{p_1}) \vee \ldots \vee R_k^-(\overrightarrow{p_k}) \Leftarrow R_{k+1}^+(\overrightarrow{p_{k+1}}) \wedge \ldots \wedge R_n^+(\overrightarrow{p_n})$ with, for $i \in [1 \ldots k]$, R_i^- any under-approximation of R_i and, for $i \in [k+1 \ldots n]$, R_i^+ any over-approximation of R_i.
However, we make the choice to have a unique representation for any relation, which limits the approximation power and requires to reason about which relation can be over- or under-approximated. For this task, it is more suitable to reason about which relation can *not* be approximated, as we can compute them iteratively.

Definition 4.4 (Approximation lattice)**.** Let $E = \{\emptyset, \{+\}, \{-\}, \{+, -\}\}$ be a 4-element lattice ordered by set inclusion. Its elements characterize relations and are to be read as:

- $\emptyset$: This relation cannot be approximated ;
- $\{+\}$: This relation can be over-approximated but not under-approximated ;
- $\{-\}$: This relation can be under-approximated but not over-approximated ;
- $\{+, -\}$: This relation can be approximated by anything.

Our abstract approximation values are thus functions $\alpha : \mathcal{R} \to E$ that assign to any relation an element of E. We call such a function an *approximation profile*. We use $\sqcap$ for the abstract intersection (defined as the point-by-point intersection of functions) and, with $V \in E$ an abstract value, $R \mapsto V$ is a notation for the function α mapping R to V and any other relation to $\{+, -\}$.

An approximation profile α describes how a model $\mathcal{M}$ can drift from $\mathcal{M}_!$ within safety bounds.

Definition 4.5 (Approximation within bounds)**.** Let $\mathcal{M}_!$ be the unique model of the set of clauses $\Gamma_!$ describing the program and let α be an approximation profile. We say that a model $\mathcal{M}$ *approximates* $\mathcal{M}_!$ *w.r.t* α if, for any relation R defined in $\Gamma_!$,

- $\mathcal{L}(\mathcal{M}, R) \setminus \mathcal{L}(\mathcal{M}_!, R) \neq \emptyset \implies \{+\} \subseteq \alpha(R)$;
- $\mathcal{L}(\mathcal{M}_!, R) \setminus \mathcal{L}(\mathcal{M}, R) \neq \emptyset \implies \{-\} \subseteq \alpha(R)$.

With $\Gamma_?$ the set of formulas defining properties and $\Gamma_!$ the set of formulas defining the program in which we want to prove the properties, the final abstract approximation profile α_f is computed in two steps: computing an initial approximation profile $\alpha_?$ depending on $\Gamma_?$, and then refining it using $\Gamma_!$.

Definition 4.6 (Approximation $\alpha_?$ from $\Gamma_?$)**.** Let $\mathcal{R}_H$ be the set of relation symbols appearing in the head of at least one clause from $\Gamma_?$ and $\mathcal{R}_B$ those appearing in the body of a clause from $\Gamma_?$. Then

$$\alpha_? = [\bigsqcap_{R \in \mathcal{R}_H} R \mapsto \{-\}] \sqcap [\bigsqcap_{R \in \mathcal{R}_B} R \mapsto \{+\}]$$

This $\alpha_?$ forbids to over-approximate any relation appearing in the head of a property and to under-approximate any relation appearing in the body of a property.

Proposition 4.7. *Let* $\mathcal{M}_!$ *be the model of* $\Gamma_!$ *and* $\alpha_?$ *the approximation profile defined from* $\Gamma_?$. *We have* $\mathcal{M}_! \models \Gamma_?$ *iff there exists a Herbrand model* $\mathcal{M}$ *approximating* $\mathcal{M}_!$ *w.r.t* $\alpha_?$ *(Definition 4.5) such that* $\mathcal{M} \models \Gamma_?$.

We now want to relax the clauses of $\Gamma_!$ while keeping that every model satisfying them is an approximation of $\mathcal{M}_!$ w.r.t $\alpha_?$. The clauses $\Gamma_!$ can be partitioned into one set Γ_R per relation R that the program defines. Modifying Γ_R can allow us to over-approximate or under-approximate the relation R. However, we have to be careful, as any set of formulas Γ_R defines exactly R only if the relations used in it are also exact. In other words, relations are defined using (other) relations, thus approximating a relation must account for these connections.
For example, the Height definition presented in Definition 4.3 is composed of the following four clauses:

$$\Gamma_{\text{Height}} = \begin{array}{l} \text{Height}(\mathit{leaf}, z) \\ \text{Height}(\mathit{node}(T_1, E, T_2), s(N_1)) \Leftarrow \text{Height}(T_1, N_1) \land \text{Height}(T_2, N_2) \land \text{Leq}(N_2, N_1) \\ \text{Height}(\mathit{node}(T_1, E, T_2), s(N_2)) \Leftarrow \text{Height}(T_1, N_1) \land \text{Height}(T_2, N_2) \land \text{Leq}(N_1, N_2) \\ N = M \Leftarrow \text{Height}(T, N) \land \text{Height}(T, M) \end{array}$$

When Leq is represented by the exact lesser-or-equal relation, we have that any model $\mathcal{M}$ such that $\mathcal{M} \models \Gamma_{\text{Height}}$ represents Height by the exact height relation. Now, suppose that Leq is allowed to be under-approximated. Then, because $\text{Leq}(N_2, N_1)$ appears in the body of the clause whose head is $\text{Height}(\mathit{node}(T_1, E, T_2), s(N_1))$, it may allow models of the set of clauses Γ_{Height} to under-approximate the Height relation. Similarly but trickier, because this same clause contains an atom $\text{Height}(T_2, N_2)$ in its body besides $\text{Leq}(N_2, N_1)$, under-approximating Leq may also allow models of the set of clauses Γ_{Height} to over-approximate the Height relation (on atoms $\text{Height}(T_2, N_2)$ that keep $\text{Leq}(N_2, N_1)$ false, so as to not make the body of the clause true). For example, if the relation Leq is under-approximated by the empty relation, then Height could be approximated by any relation as long as it contains (leaf, z) and is functional. Therefore, if one wants to represent the exact relation Height using the clauses Γ_{Height}, then Leq must also not be under-approximated. This intuition is formalized by the function $\mathit{restrict}_{\Gamma_!} : (\mathcal{R} \to E) \to (\mathcal{R} \to E)$ that restricts an approximation profile α using $\Gamma_!$.

Definition 4.8 ($\mathit{restrict}_{\Gamma_!}$)**.** Let $\alpha : \mathcal{R} \to E$ be some approximation profile. Let R be a relation symbol and $\Gamma_R \subseteq \Gamma_!$ the set of clauses defining it. Let $\varphi = H \Leftarrow B \in \Gamma_R$ be a clause. Suppose that $R(\overrightarrow{p})$ and $R'(\overrightarrow{p'})$ are two different atoms appearing in φ (so either in H or B). Then

$$\mathit{restrict}_{\Gamma_!}(\alpha) = \alpha \sqcap \begin{cases} R' \mapsto \{-\} & \text{if } R(\overrightarrow{p}) \in H \ \land\ R'(\overrightarrow{p'}) \in H \ \land\ \{-\} \notin \alpha(R) \\ R' \mapsto \{+\} & \text{if } R(\overrightarrow{p}) \in H \ \land\ R'(\overrightarrow{p'}) \in B \ \land\ \{-\} \notin \alpha(R) \\ R' \mapsto \{-\} & \text{if } R(\overrightarrow{p}) \in B \ \land\ R'(\overrightarrow{p'}) \in H \ \land\ \{+\} \notin \alpha(R) \\ R' \mapsto \{+\} & \text{if } R(\overrightarrow{p}) \in B \ \land\ R'(\overrightarrow{p'}) \in B \ \land\ \{+\} \notin \alpha(R) \\ R' \mapsto \{+, -\} & \text{otherwise} \end{cases}$$

To make this definition more intuitive, we expand on the first case: $R' \mapsto \{-\}$ if $R(\overrightarrow{p}) \in H \ \land\ R'(\overrightarrow{p'}) \in H \ \land\ \{-\} \notin \alpha(R)$. This case is to be read as "If R should not be under-approximated and if there is, in the head of a clause defining R, an atom $R(\overrightarrow{p})$ and another atom $R'(\overrightarrow{p'})$, then R' must not be over-approximated". This is because over-approximating R' may compensate for under-approximating R.

The function $restrict_{\Gamma_!}$ can then be applied to an approximation profile α until a fixpoint is reached. The final approximation profile α_f is the fixpoint given by the repeated application of $restrict_{\Gamma_!}$ to the approximation profile $\alpha_?$. We now state that this approximation profile α_f is safe and allows to modify the clauses defining each relation individually.

Proposition 4.9. *Let $\Gamma = \Gamma_! \cup \Gamma_?$ be a set of formula describing both a program and properties. Let $\alpha_f = fixpoint(restrict_{\Gamma_!}, \alpha_?)$ be the final approximation defined by repeated applications of $restrict_{\Gamma_!}$ to $\alpha_?$. For every relation R that the program defines, let Γ'_R be a set of clauses approximating Γ_R w.r.t. α_f, i.e. such that for any model $\mathcal{M}'_R$ of Γ'_R there exists a model $\mathcal{M}_R$ of Γ_R with $\mathcal{M}'_R$ being an approximation of $\mathcal{M}_R$ w.r.t. α_f. We write Γ' for the union of Γ'_R for every relation R of the program. Then, any model $\mathcal{M}'$ satisfying Γ' is an approximation of $\mathcal{M}_!$ w.r.t α_f.*

Now that each relation R has its approximation profile $\alpha_f(R)$, we can relax the clauses using α_f, which corresponds to defining the sets Γ'_R in the above definition. Our current method focuses on over-approximations but cannot, yet, produce under-approximations. Over-approximations are computed by forgetting functionality clauses (e.g. the clause $N = M \Leftarrow \text{Height}(T, N) \wedge \text{Height}(T, M)$ for the Height relation). This allows (some) over-approximations for relations coming from a non-boolean function in the program. We do not yet know any satisfying way to automatically relax the clauses to over-approximate other relations or to under-approximate them. Because our approximation method only consists of removing functionality clauses for overapproximating, then Proposition 4.9 can be used to know that any model $\mathcal{M}'$ of Γ', the approximated set of clauses, is an approximation of $\mathcal{M}_!$ w.r.t α_f.

Finally, note that, when trying to over-/under-approximate a relation, deleting every clause of its definition that contains only negative/positive atoms is not sound. For example, suppose the relation HeightRB of Definition 4.3 is assigned the value $\{-\}$, i.e. we want to under-approximate it without over-approximating it. Then deleting the clause $\text{HeightRB}(leaf, z)$ may seem like a good idea but allows models that contain terms which are not in the exact relation, for example the model that represents HeightRB by the set of tuples $(t, s(n))$ with n the height of the rightmost branch of t, which is not an under-approximation.

Now that clauses extracted from the program have been modified to allow approximations, we define the procedure that checks their satisfiability.

4.3 Model search for clauses: a generic Learner/Teacher procedure

This section presents a generic procedure for proving or disproving the satisfiability of a set Γ of clauses. This model exhibition procedure is very close to the Implication CounterExample (ICE) [12] framework, featuring two entities: a *Learner* and a *Teacher*. The Learner and the Teacher are iteratively communicating candidate models and counterexamples (ground instances of Γ). The Teacher's role is to verify that a given model $\mathcal{M}$ satisfies all formulas of Γ (i.e. that $\mathcal{M} \models \Gamma$), and if not to extract a counterexample to $\mathcal{M} \models \Gamma$ that is given to the Learner. The Learner's role is to propose a new model inferred from the (counter)examples that the Teacher previously answered. The procedure shows satisfiability of Γ by finding a model $\mathcal{M}$ such that $\mathcal{M} \models \Gamma$. Unsatisfiability is shown by finding a contradiction in Γ, i.e., a finite contradictory set of ground instances of Γ. We now define the specification for the Teacher and for the Learner using generic models $\mathcal{M}$.

Definition 4.10 (Teacher specification)**.**

Input: A finite set of clauses Γ and a model $\mathcal{M}$.

Output: *None* if $\mathcal{M} \models \Gamma$ and $Some(\widehat{\varphi})$ with $\mathcal{M} \not\models \widehat{\varphi}$ and $\widehat{\varphi} \in Grd(\Gamma)$ otherwise, with $Grd(\Gamma)$ the set of ground instances of Γ.

Definition 4.11 (Learner specification)**.**

Input: A finite set of ground clauses $\widehat{\Gamma}$.

Output: *None* if $\widehat{\Gamma}$ is contradictory and $Some(\mathcal{M})$ with $\mathcal{M} \models \widehat{\Gamma}$ otherwise.

The whole procedure, $Sat(\Gamma)$, is then designed as a back-and-forth between the Learner and the Teacher. If the Learner cannot find a model of its input examples (because they are contradictory), then Γ is contradictory too. If every formula is satisfied by a model proposed by the Learner, then Γ is shown true.

Definition 4.12 (The satisfiability procedure $Sat(\Gamma)$)**.** Given Γ, a finite set of clauses, the loop $Sat(\Gamma)$ proceeds as:

0. Let $\widehat{\Gamma}_0 := \emptyset$ and $i := 0$.
1. If $Learner(\widehat{\Gamma}_i) = None$, then return "Disproved: $\widehat{\Gamma}_i$".
   If $Learner(\widehat{\Gamma}_i) = Some(\mathcal{M}_i)$, then go to step 2.

2. If $Teacher(\mathcal{M}_i, \Gamma) = None$, then return "Proved: $\mathcal{M}_i$".
   If $Teacher(\mathcal{M}_i, \Gamma) = Some(\widehat{\varphi}_i)$, let $\widehat{\Gamma}_{i+1} := \widehat{\Gamma}_i \cup \{\widehat{\varphi}_i\}$, $i := i + 1$, and go to step 1.

As generic as this procedure currently is, we can already state its progress lemma.

Lemma 4.13 (Progress). *During an execution of* $Sat(\Gamma)$*:*

- *The Learner never outputs the same model twice*
- *The Teacher never outputs the same counterexample twice.*

Proof.

- Any model $\mathcal{M}_i$ that the Learner proposes at a step i is either correct and the procedure stops or the Teacher outputs a counterexample $\widehat{\varphi}_i$ to it that is incorporated into $\widehat{\Gamma}_{i+1}$, therefore ensuring that all next proposed models satisfy $\widehat{\varphi}_i$, what $\mathcal{M}_i$ does not.
- Any proposed model $\mathcal{M}_i$ satisfies every ground constraint $\widehat{\Gamma}_i$ that the Teacher sent from the beginning, so the Teacher cannot find $\widehat{\varphi} \in \widehat{\Gamma}_i$ such that $\mathcal{M} \not\models \widehat{\varphi}$.

□

Using this generic satisfaction procedure requires three elements to be defined:

- the formalism for representing models
- the Learner procedure
- the Teacher procedure.

In order to illustrate our Learner/Teacher procedure, we instantiate it with the formalism of (regular) tree automata. It will later be instantiated with two other formalisms presented in this manuscript: convoluted tree automata in Chapter 5 and then Shallow Horn Clauses in Chapter 7. The Learner and Teacher have to be adapted to the formalism in question, which is the role of Chapters 6 and 8.

Example 4.14 ($Sat(\Gamma)$ with regular automata)**.** Let the formalism for representing models be regular automata, as shown in Definition 2.37. The Learner and Teacher are not given an explicit definition so as to not obfuscate the example, but the Learner will only search for smallest automata w.r.t their number of states and the Teacher will only output counterexamples of minimal height. In the following, we represent models (tree automata) by their transitions only, as there is only one state q_f of type bool and other states can be inferred from the transitions.

Let the input set of clauses Γ be the set of formulas defining the All0, No0, and IsEmpty relations. The property we try to prove is $\text{IsEmpty}(T) \Leftarrow \text{All0}(T) \wedge \text{No0}(T)$. Therefore Γ is the one from Example 4.1.
The final approximation profile is $\alpha_f = \{(\text{IsEmpty}, \{-\}), (\text{All0}, \{+\}), (\text{No0}, \{+\})\}$. However, with the approximation defined in Section 4.2, no clause is modified nor deleted. Note that, in this case, deleting clauses $\{I_1, A_3, N_3\}$ would have been safe.
We give a possible execution of the Learner/Teacher loop $Sat(\Gamma)$ (Definition 4.12) where, at each step i of the execution, is given:

- the model proposed by the Learner $Some(\mathcal{M}_i) = Learner(\widehat{\Gamma}_i)$;
- the Teacher's counterexample $Some(\widehat{\varphi}_i) = Teacher(\mathcal{M}_i)$;
- the clause φ_i of which $\widehat{\varphi}_i$ is an instance.

Note that states have been renamed for clarity. Any state named

- q_0 recognizes only $\{z\}$;
- q_1 recognizes $\{s(z)\}$;
- q_n recognizes any natural number ;
- q_s recognizes any positive natural number ;
- q_l recognizes $leaf$, the empty tree ;
- q_{t_0} recognizes any tree whose elements are only z ;
- q_{t_1} recognizes any tree whose elements are only $s(z)$;
- q_{t_s} recognizes any tree whose elements are positive natural numbers ;
- q_{t_n} recognizes any tree ;
- q_f is the only boolean state and recognizes atoms for relations All0, No0, and IsEmpty.

Note also that the transitions of every model $\mathcal{M}_i$ are ordered, so that the transition presentation keeps the same structure going from step i to step $i+1$. If you prefer not having to understand each automaton but are interested in the relations they denote, you can read only the last line of the transitions (the transitions that rewrite to the state q_f of type $bool$) and use the above description of states based on their name to know which relation they denote.
Finally, note that the relations denoted by the model $\mathcal{M}_i$ at a given step i are not necessarily better than those at a previous step $j < i$, for example at steps 3 and 4. The relation $\mathcal{L}(\mathcal{M}_3, \text{IsEmpty})$ is the correct one, i.e. only contains the empty tree $leaf$, whereas the relation $\mathcal{L}(\mathcal{M}_4, \text{IsEmpty})$ is incorrect, as it is the set of trees whose values are all z.

This is because, in this step 4, having a state recognizing only *leaf* costs one extra state and is not required by the set of examples $\{\widehat{\varphi}_0, \widehat{\varphi}_1, \widehat{\varphi}_2, \widehat{\varphi}_3\}$ used for inferring $\mathcal{M}_4$.

i	$\mathcal{M}_i$	φ_i	$\widehat{\varphi}_i$
0	$\emptyset$	I_1	$\text{IsEmpty}(\mathit{leaf})$
1	$\mathit{leaf}() \to q_l$ $\text{IsEmpty}(q_l) \to q_f$	A_1	$\text{All0}(\mathit{leaf})$
2	$\mathit{leaf}() \to q_l$ $\text{All0}(q_l) \to q_f \quad \text{IsEmpty}(q_l) \to q_f$	N_1	$\text{No0}(\mathit{leaf})$
3	$\mathit{leaf}() \to q_l$ $\text{All0}(q_l) \to q_f \quad \text{IsEmpty}(q_l) \to q_f \quad \text{No0}(q_l) \to q_f$	A_2	$\text{All0}(\mathit{node}(\mathit{leaf}, z, \mathit{leaf}))$ $\Leftarrow \text{All0}(\mathit{leaf})$
4	$\mathit{leaf}() \to q_{t_0} \quad \mathit{node}(q_{t_0}, q_0, q_{t_0}) \to q_{t_0} \quad z() \to q_0$ $\text{All0}(q_{t_0}) \to q_f \quad \text{IsEmpty}(q_{t_0}) \to q_f \quad \text{No0}(q_{t_0}) \to q_f$	I_2	$\bot \Leftarrow$ $\text{IsEmpty}(\mathit{node}(\mathit{leaf}, z, \mathit{leaf}))$
5	$\mathit{leaf}() \to q_{t_0} \quad \mathit{node}(q_{t_0}, q_0, q_{t_0}) \to q_{t_0} \quad z() \to q_0$ $\mathit{leaf}() \to q_l$ $\text{All0}(q_{t_0}) \to q_f \quad \text{IsEmpty}(q_l) \to q_f \quad \text{No0}(q_l) \to q_f$	N_2	$\text{No0}(\mathit{node}(\mathit{leaf}, s(z), \mathit{leaf}))$ $\Leftarrow \text{No0}(\mathit{leaf})$
6	$\mathit{leaf}() \to q_{t_n} \quad \mathit{node}(q_{t_n}, q_n, q_{t_n}) \to q_{t_n} \quad z() \to q_n$ $\mathit{leaf}() \to q_l \quad s(q_n) \to q_n$ $\text{All0}(q_{t_n}) \to q_f \quad \text{IsEmpty}(q_l) \to q_f \quad \text{No0}(q_{t_n}) \to q_f$	N_3	$\bot \Leftarrow$ $\text{No0}(\mathit{node}(\mathit{leaf}, z, \mathit{leaf}))$
7	$\mathit{leaf}() \to q_{t_0} \quad \mathit{node}(q_{t_0}, q_0, q_{t_0}) \to q_{t_0} \quad z() \to q_0$ $\mathit{leaf}() \to q_{t_1} \quad \mathit{node}(q_{t_1}, q_1, q_{t_1}) \to q_{t_1} \quad s(q_0) \to q_1$ $\mathit{leaf}() \to q_l$ $\text{All0}(q_{t_0}) \to q_f \quad \text{IsEmpty}(q_l) \to q_f \quad \text{No0}(q_{t_1}) \to q_f$	N_2	$\text{No0}(\mathit{node}(\mathit{leaf}, s(s(z)), \mathit{leaf}))$ $\Leftarrow \text{No0}(\mathit{leaf})$
8	$\mathit{leaf}() \to q_{t_0} \quad \mathit{node}(q_{t_0}, q_0, q_{t_0}) \to q_{t_0} \quad z() \to q_0$ $\mathit{leaf}() \to q_{t_s} \quad \mathit{node}(q_{t_s}, q_s, q_{t_s}) \to q_{t_s} \quad s(q_0) \to q_s$ $\mathit{leaf}() \to q_l \quad s(q_s) \to q_s$ $\text{All0}(q_{t_0}) \to q_f \quad \text{IsEmpty}(q_l) \to q_f \quad \text{No0}(q_{t_s}) \to q_f$	/	*Proved* : $\mathcal{M}_8$

$I_1 :\ \text{IsEmpty}(\mathit{leaf})$

$I_2 :\ \bot \Leftarrow \text{IsEmpty}(\mathit{node}(T_1, E, T_2))$

$A_1 :\ \text{All0}(\mathit{leaf})$

$A_2 :\ \text{All0}(\mathit{node}(T_1, z, T_2)) \Leftarrow \text{All0}(T_1) \wedge \text{All0}(T_2)$

$A_4 :\ \text{All0}(T_1) \Leftarrow \text{All0}(\mathit{node}(T_1, z, T_2))$

$A_3 :\ \bot \Leftarrow \text{All0}(\mathit{node}(T_1, s(E), T_2))$

$A_5 :\ \text{All0}(T_2) \Leftarrow \text{All0}(\mathit{node}(T_1, z, T_2))$

$N_1 :\ \text{No0}(\mathit{leaf})$

$N_2 :\ \text{No0}(\mathit{node}(T_1, z, T_2)) \Leftarrow \text{No0}(T_1) \wedge \text{No0}(T_2)$

$N_4 :\ \text{No0}(T_1) \Leftarrow \text{No0}(\mathit{node}(T_1, z, T_2))$

$N_3 :\ \bot \Leftarrow \text{No0}(\mathit{node}(T_1, s(E), T_2))$

$N_5 :\ \text{No0}(T_2) \Leftarrow \text{No0}(\mathit{node}(T_1, z, T_2))$

CHAPTER 5

CONVOLUTED TREE AUTOMATA

Automata on convoluted trees is a formalism that has been introduced to generalize regular tree automata in order to define relations on trees. Their standard definition can be found in automata literature [7], among that of other tree-tuple languages such as tuple automata or ground tree transducers. Convoluted (tree) automata are standard tree automata with the exception that they are defined on an alphabet of tuples of n symbols. Reading a term on such an alphabet amounts to reading n terms at the same time, one per projection on the ith symbols. Such an automaton defines a set of tuples of terms, i.e. a relation. The operation of merging those n terms into one is what is called the *convolution*. The standard convolution operator amounts to overlaying the (syntax tree of the) terms, overlaying every symbol that appears at the same position. This can be recursively defined, starting from the root, and adding a padding symbol $\square \notin \Sigma$ where there is an arity mismatch between symbols. This standard convolution is called here the *left convolution*, in order to distinguish it from other convolutions, e.g. the right convolution. See Example 3.8 for a visual representation of the left-convolution.

5.1 Convolution with padding

We first define an operation that is often used when dealing with terms of different arities. It is used for fetching the subterm at a given position only if defined, and otherwise returns a padding.

Definition 5.1. Let p be a pattern and π a position. Then

$$p[\pi]^{\square} = \begin{cases} p[\pi] & \text{if } \pi \in Pos(p) \\ \square & \text{otherwise} \end{cases}$$

5.1.1 Standard left convolution

We then define the left-convolution of a n-tuple of tuples and then use it to define the alphabet of the convolution and the convolution of terms. The left-convolution of a n-tuple of tuples can be thought of as one step of the left-convolution of n terms.

Definition 5.2 (n-ary left-convolution of tuples with padding)**.** The n-ary *left convolution of tuples*, written $\boxed{L}_n$, is defined as:

$$\boxed{L}_n\ ((\vec{e_1}, \ldots, \vec{e_n})) = (\vec{e'_1}, \ldots, \vec{e'_k})$$
$$\text{with } k = \max_{i \in [1 \ldots n]} (|\vec{e_i}|) \quad \text{and} \quad \forall j \in [1 \ldots k], \vec{e'_j} = (\vec{e_i}[j]^{\square})_{i \in [1 \ldots n]}$$

Example 5.3 (n-ary left convolution with padding)**.** Let $\vec{e_1} = (a_1, a_2, a_3)$, $\vec{e_2} = (b_1)$, $\vec{e_3} = (c_1, c_2)$, and $\vec{e_4} = (d_1)$, which can be organised by row in this array disposition:

a_1	a_2	a_3
b_1	$\square$	$\square$
c_1	c_2	$\square$
d_1	$\square$	$\square$

The left convolution $\boxed{L}_4\ (\vec{e_1}, \vec{e_2}, \vec{e_3}, \vec{e_4})$ is equal to $\big((a_1, b_1, c_1, d_1), (a_2, \square, c_2, \square), (a_3, \square, \square, \square)\big)$, which corresponds to the columns of the array.

The n-ary *left-convolution* produces terms on an alphabet of n-ary tuple of symbols, which is defined as follows.

Definition 5.4 (Alphabet for the convoluted terms)**.** Let Σ a typed alphabet. Let $\Sigma_\square = \Sigma \cup \{\square\}$ with $\square$ a new symbol of a new type $\tau_\square$. Then the n-ary left-convoluted alphabet, written $\Sigma^{\boxed{L}_n}$, has for symbols $(\Sigma_\square)^n$, and any (tuple of) symbol $(f_1, \ldots, f_n) \in \Sigma^{\boxed{L}_n}$ has type $\tau((f_1, \ldots, f_n)) = \boxed{L}_n\ (\vec{\tau}_1, \ldots, \vec{\tau}_n) \to (\tau_1, \ldots, \tau_n)$ with $\forall i \in [1 \ldots n], \tau(f_i) = \vec{\tau}_i \to \tau_i$. A symbol $(f_1, \ldots, f_n)$ is sometimes written $\langle f_1, \ldots, f_n \rangle$, or simply $\vec{f}$, for readability.

Definition 5.5 (n-ary left convolution of terms)**.** The n-ary *left-convolution of terms*, written $\boxed{L}^t_n$, takes a tuple of n terms $(t_1, \ldots, t_n)$ on an alphabet $\Sigma_\square$ and returns a term $\boxed{L}^t_n\ (t_1, \ldots, t_n)$ on the convoluted alphabet $\Sigma^{\boxed{L}_n}$. The left-convolution of n terms is

recursively defined as:

$$\boxed{L}^t_n\,(f_1(\vec{t_1}),\ldots,f_n(\vec{t_n})) = \langle f_1,\ldots,f_n\rangle(\boxed{L}^t_n\,(\vec{t'_1}),\ldots,\boxed{L}^t_n\,(\vec{t'_k}))$$
$$\text{with } (\vec{t'_1},\ldots,\vec{t'_k}) = \boxed{L}_n\,(\vec{t_1},\ldots,\vec{t_n})$$

Example 5.6 (n-ary left-convoluted terms)**.** In this example, we use the binary left-convolution of tuples $\boxed{L}_2$ and its extension to terms $\boxed{L}^t_2$. The binary left-convolution alphabet $\Sigma^{\boxed{L}_2}$ contains, among others, the following typed symbols:

- $\tau(\langle\square,z\rangle) = (\tau_\square, nat)$
- $\tau(\langle\square,s\rangle) = ((\tau_\square, nat)) \rightarrow (\tau_\square, nat)$
- $\tau(\langle z,s\rangle) = ((\tau_\square, nat)) \rightarrow (nat, nat)$
- $\tau(\langle s,s\rangle) = ((nat, nat)) \rightarrow (nat, nat)$
- $\tau(\langle cons,s\rangle) = ((nat, nat), (natlist, \tau_\square)) \rightarrow (natlist, nat)$.

Here are two examples of binary left convolution of terms:

$$\boxed{L}^t\,(s(z),\; s(s(z))) = \langle s,s\rangle(\langle z,s\rangle(\langle\square,s\rangle(\langle\square,z\rangle)))$$

With l_{ex} and n_{ex} as defined below, $\boxed{L}^t\,(l_{ex}, n_{ex})$

$$l_{ex} = cons(z, cons(z, nil)) \qquad n_{ex} = s(s(z))$$

$$= \langle cons,s\rangle(\langle z,s\rangle(\langle\square,z\rangle),\ \langle cons,\square\rangle(\langle z,\square\rangle, \langle nil,\square\rangle))$$

Note that, because the alphabet Σ is typed and $\square$ is given a fresh type $\tau_\square$, $\mathcal{T}(\Sigma_\square) = \mathcal{T}(\Sigma) \cup \{\square\}$ and the n-ary left-convolution $\boxed{L}^t_n$ is an isomorphism between $\mathcal{T}(\Sigma_\square)^n$ and $\mathcal{T}(\Sigma^{\boxed{L}_n})$. However, we are especially interested in relations between terms, not in relations between terms with paddings. We say that a type $(\tau_1,\ldots,\tau_n)$ is $\tau_\square$-free if $\forall i \in [i \ldots n], \tau_i \neq \tau_\square$. We have that $\boxed{L}^t_n$ is also an isomorphism between tuples of terms $\mathcal{T}(\Sigma)^n$ and convoluted terms whose type is $\tau_\square$-free $\{t \mid t \in \mathcal{T}(\Sigma^{\boxed{L}_n}) \wedge \tau(t) \text{ is } \tau_\square\text{-free}\}$. For example, $\langle nil, s\rangle(\langle\square, z\rangle)$ has a $\tau_\square$-free type and corresponds to the tuple of terms $(nil, s(z))$ whereas $\langle\square, z\rangle$ has $\langle\tau_\square, Nat\rangle$ as type and corresponds to the tuple of terms with paddings $(\square, z)$. The n-ary left-convolution of terms $\boxed{L}^t_n$ can also be understood in a more denotational manner, which often serves as definition:

Lemma 5.7. *Let* $(t_1,\ldots,t_n) \in \mathcal{T}(\Sigma_\square)^n$*. Then* $t_\square = \boxed{L}^t_n\,(t_1,\ldots,t_n)$ *is the tree such that* $\forall \pi \in \mathbb{N}^*$,

(a) $\pi \in Pos(t_\square) \iff (n = 0 \wedge \pi = \epsilon) \vee \exists i \in [1 \dots n].\ \pi \in Pos(t_i)$

(b) $\pi \in Pos(t_\square) \Rightarrow t_\square[\pi] = \boxed{L}_n^t\ (t_1[\pi]^\square, \dots, t_n[\pi]^\square)$

Proof. By induction on π:

- $\pi = \epsilon$:

  (a): $\epsilon \in Pos(t_\square)$ is always true. Then either $n = 0$ or there exists $i \in [1 \dots n]$ such that t_i exists and then, necessarily, $\epsilon \in Pos(t_i)$.

  (b): $t_\square[\epsilon] = t_\square = \boxed{L}_n^t\ (t_1, \dots, t_n) = \boxed{L}_n^t\ (t_1[\epsilon]^\square, \dots, t_n[\epsilon]^\square)$

- $\pi = \pi' \cdot j$:

  By induction, we have:

  — $\pi' \in Pos(t_\square) \iff (n = 0 \wedge \pi' = \epsilon) \vee \exists i \in [1 \dots n].\ \pi' \in Pos(t_i)$

  — $\pi' \in Pos(t_\square) \Rightarrow t_\square[\pi'] = \boxed{L}_n^t\ (t_1[\pi']^\square, \dots, t_n[\pi']^\square)$

  By case disjunction on $\pi' \in Pos(t_\square)$:

  — Suppose $\pi' \notin Pos(t_\square)$. Then (by inductive property (b)) $\forall i \in [1 \dots n], \pi' \notin Pos(t_i)$. Then $\pi' \cdot j \notin Pos(t_\square)$ and $\forall i \in [1 \dots n], \pi' \cdot j \notin Pos(t_i)$. This concludes for both properties (a) and (b).

  — Suppose $\pi' \in Pos(t_\square)$. By inductive property (b), we have $t_\square[\pi'] = \boxed{L}_n^t\ (t_1[\pi']^\square, \dots, t_n[\pi']^\square)$. For all $i \in [1 \dots n]$, let us write $t_i[\pi']^\square$ as $f_i(\vec{t_i})$ and $(\vec{t'_1}, \dots, \vec{t'_k}) = \boxed{L}_n\ (\vec{t_1}, \dots, \vec{t_n})$. Recall that $t_\square[\pi'] = (f_1, \dots, f_n)(\boxed{L}_n^t\ (\vec{t'_1}), \dots, \boxed{L}_n^t\ (\vec{t'_k}))$.

    By case disjunction on $j \in [1 \dots k]$:

    — Suppose $j \notin [1 \dots k]$. Then $\pi' \cdot j \notin Pos(t_\square)$ and, because $k = \max_{i \in [1 \dots n]}(|f_i|)$, then $\forall i \in [1 \dots n], \pi' \cdot j \notin Pos(t_i)$. This concludes for both properties (a) and (b).

    — Suppose $j \in [1 \dots k]$. Then $t_\square[\pi] = t_\square[\pi' \cdot j] = \boxed{L}_n^t\ (\vec{t'_j})$ by definition of $\boxed{L}_n$. Also $\vec{t'_j} = (\vec{t_1}[j]^\square, \dots, \vec{t_n}[j]^\square) = (t_1[\pi' \cdot j]^\square, \dots, t_n[\pi' \cdot j]^\square)$. Therefore $t_\square[\pi] = \boxed{L}_n^t\ (t_1[\pi]^\square, \dots, t_n[\pi]^\square)$, which concludes for property (b). Moreover, because $j \in [1 \dots k]$, then $k \geq 1$ and there exists $i \in [1 \dots n]$ such that the term $f_i(\vec{t_i}) = t_i[\pi']$ has $|f_i| = k$. So $\pi' \cdot j \in Pos(t_i)$, which concludes for property (a).

$\square$

From the previous lemma follows this intuitive characterisation of left-convoluted trees, in which syntax trees are overlayed where the positions are equal.

Lemma 5.8. *Let* $(t_1, \ldots, t_n) \in \mathcal{T}(\Sigma_\square)^n$*. Then* $t_\square = \boxed{L}_n^t \ (t_1, \ldots, t_n)$ *is such that* $\forall \pi \in Pos(t_\square), Root(t_\square[\pi]) = \vec{f}$ *with* $\vec{f} = (Root(t_i[\pi]^\square))_{i \in [1 \ldots n]}$.

Proof. By lemma 5.7, we have $t_\square[\pi] = \boxed{L}_n^t \ (t_1[\pi]^\square, \ldots, t_n[\pi]^\square)$. It is then immediate that $Root(t_\square[\pi]) = Root(\boxed{L}_n^t \ (t_1[\pi]^\square, \ldots, t_n[\pi]^\square)) = (Root(t_1[\pi]^\square), \ldots, Root(t_n[\pi]^\square))$. □

Tree automata can be defined on a convoluted alphabet in the standard manner (Definitions 2.26, 2.28), and thus recognize convoluted terms. However, convoluted automata are meant to represent relations, not convoluted terms, so we define $\mathcal{R}(\mathcal{A}, q)$ to denote the relation that the state q recognizes in automaton $\mathcal{A}$.

Definition 5.9 (Relation of a convoluted automaton)**.** Let $\mathcal{A}$ be an automaton defined using convolution $\boxed{L}_n$ and q one of its states. Then

$$\mathcal{R}(\mathcal{A}, q) = \{\vec{t} \mid \vec{t} \in \mathcal{T}(\Sigma_\square)^n \wedge \boxed{L}_n^t \ (\vec{t}) \in \mathcal{L}(\mathcal{A}, q)\}$$

We usually are only interested in states whose type is $\tau_\square$-free, i.e. states q such that $\tau(q) = (\tau_1, \ldots, \tau_n)$ implies $\forall i \in [i \ldots n], \tau_i \neq \tau_\square$, as they define an ($n$-ary) relation between terms of $\mathcal{T}(\Sigma)$ (without padding). Other states are only used as a necessary intermediate.

Example 5.10 (Convoluted automata)**.** Let $\mathcal{A}_<$ be the automaton with states $\{q, q_f\}$ and transitions $\{\langle\square, z\rangle() \to q, \ \ \langle\square, s\rangle(q) \to q, \ \ \langle z, s\rangle(q) \to q_f, \ \ \langle s, s\rangle(q_f) \to q_f\}$. $\mathcal{R}(\mathcal{A}_<, q_f)$ is the $<$ relation on Peano numbers and $\tau(q_f) = (nat, nat)$. For example, the binary left-convolution of $s(z)$ and $s(s(s(z)))$ (from Example 5.6) is recognized by this automaton, as shown below. The inductive recognition is shown step by step, replacing each subterm by the state it is recognized by. This presentation comes from the definition of tree automata transitions as rewriting rules.

$$\begin{array}{c}\langle s,s\rangle\\ | \\ \langle z,s\rangle \\ | \\ \langle\square,s\rangle \\ | \\ \langle\square,z\rangle\end{array} \xrightarrow{\langle\square,z\rangle() \to q} \begin{array}{c}\langle s,s\rangle\\ | \\ \langle z,s\rangle \\ | \\ \langle\square,s\rangle \\ | \\ q\end{array} \xrightarrow{\langle\square,s\rangle(q) \to q} \begin{array}{c}\langle s,s\rangle\\ | \\ \langle z,s\rangle \\ | \\ q\end{array} \xrightarrow{\langle z,s\rangle(q) \to q_f} \begin{array}{c}\langle s,s\rangle\\ | \\ q_f\end{array} \xrightarrow{\langle s,s\rangle(q_f) \to q_f} q_f$$

Convolutions and their expressivity Which relations are representable by convoluted tree automata highly depends on the precise datatypes definition. For example, when using the left-convolution, the Len relation can not be represented using standard *nil* and *cons* constructors but it could be if the *cons* constructor had its arguments swapped. This is because left-convoluting a list l and a natural number n, as in Example 5.6, will relate n with the left-most branch of l. An alternative to constructor modification is defining other convolutions.

5.1.2 Right and complete convolution

The *right convolution*, written $\boxed{R}_n$, is defined similarly to $\boxed{L}_n$ but overlays terms from the right branches, therefore adding padding to the left of terms instead of to the right. This right convolution is effective for proving properties relating lists and unary natural numbers, as the recursive argument of the *cons* constructor is overlayed with the recursive argument of the s constructor, which is not the case with the left convolution.

Definition 5.11 (*n*-ary right-convolution with padding)**.** The right-convolution is defined in a very similar fashion as the left one. The differences are written in bold text.

$$\boxed{R}_n\ (\vec{e_1}, \ldots, \vec{e_n}) = (\vec{e'_1}, \ldots, \vec{e'_k})$$
$$\text{with } k = \max_{i \in [1 \ldots n]} (|\vec{e_i}|) \quad \text{and} \quad \forall j \in [1 \ldots k], \vec{e'_j} = (\vec{e_i}[|\boldsymbol{e_i}| - \boldsymbol{k} + j]^{\square})_{i \in [1 \ldots n]}$$

Example 5.12 (Right-convolution)**.** Reusing $\vec{e_1}, \vec{e_2}, \vec{e_3}$, and $\vec{e_4}$ from Example 5.3, they can be organised in the following array, now adding padding to the left

a_1	a_2	a_3
$\square$	$\square$	b_1
$\square$	c_1	c_2
$\square$	$\square$	d_1

The right convolution of these tuples $\boxed{R}_4\ (\vec{e_1}, \vec{e_2}, \vec{e_3}, \vec{e_4})$ again corresponds to the columns of this array and is equal to $\big((a_1, \square, \square, \square), (a_2, \square, c_1, \square), (a_3, b_1, c_2, d_1)\big)$.

The right-convoluted alphabet $\Sigma^{\boxed{R}_n}$ is defined exactly as $\Sigma^{\boxed{L}_n}$, but replacing every $\boxed{L}_n$ by $\boxed{R}_n$. The same is true for the n-ary right convolution of terms $\boxed{R}^t_n$.

Example 5.13 (Right-convoluted terms)**.** Following are two examples of binary right convolution of terms $\boxed{R}^t_2$. Note that, albeit the tuple of symbols of $\Sigma^{\boxed{R}_2}$ are the same as those of $\Sigma^{\boxed{L}_2}$, their type may differ. For example, the symbol $\langle cons, s\rangle \in \Sigma^{\boxed{R}_2}$ has type $\tau(\langle cons, s\rangle) = ((nat, \tau_\Box), (natlist, nat)) \to (natlist, nat)$.

$$\boxed{R}^t_2\,(s(z),\ s(s(s(z)))) = \langle s,s\rangle(\langle z,s\rangle(\langle \Box,s\rangle(\langle \Box,z\rangle)))$$

$$\boxed{R}^t_2\,(l_{ex}, n_{ex}) = \langle cons,s\rangle(\langle z,\Box\rangle,\ \langle cons,s\rangle(\langle z,\Box\rangle,\ \langle nil,z\rangle))$$

The left and right convolution are just two possibilities among many ways of combining a tuple of terms into one term, even when restricting to those being recursively defined from the combination/shuffling of a tuple of tuple such as those defined before. These two convolutions, left and right, both have a common limitation: they do not duplicate nor forget any subterm. That is, in the Definitions 5.2 and 5.11 of those two convolutions, every element of any tuple from the input can be found exactly once in the output. The fact that no element can be duplicated implies that they can only define relations where it is enough to relate a single *fixed* branch of a term (respectively the leftmost and rightmost branch) with another *fixed* fixed branch in another term. This limitation requires to use different convolutions for different relations needing to relate different fixed branches, e.g. *HeightLB* can be represented using left-convolution but not right-convolution, and the opposite is true for *HeightRB*. Some other relations, such as *Height*, cannot be represented using a fixed convolution strategy, as the highest branch is not always the same.

Dropping this constraint, we can define a convolution that is quite expressive, called *complete convolution* in [18]. The complete convolution has the advantage of not depending on the constructor argument's order by being able to duplicate terms, but the drawback of generating (exponentially w.r.t their height) big convoluted terms. This complete convolution relates every combination of different tuple's elements, which results in overlaying every same-depth constructor when convoluting terms.

Definition 5.14 (n-ary complete convolution with padding)**.** The complete-convolution

is defined as:

$$\boxed{C}_n\ (\vec{e_1}, \dots, \vec{e_n}) = (\vec{e'_1}, \dots, \vec{e'_k})$$

$$\text{with } k = \begin{cases} 0 & \text{if } \forall i \in [1 \dots n],\ |\vec{e_i}| = 0 \\ \prod_{i=1}^{n} \max(1, |\vec{e_i}|) & \text{otherwise} \end{cases}$$

and $\forall j \in [1 \dots k], \vec{e'_j} = (\vec{e_1}[\pi_1]^{\square}, \dots, \vec{e_n}[\pi_n]^{\square})$ with $(\pi_1, \dots, \pi_n)$ the jth element of the lexicographically ordered set $\{1, \dots, \max(1, |\vec{e_i}|)\} \times \dots \times \{1, \dots, \max(1, |\vec{e_n}|)\}$.

Once again, the definition of the convolution alphabet $\Sigma^{\boxed{C}_n}$ and of the complete convolution of terms $\boxed{C}^t_n$ follow from the definition of $\boxed{C}_n$.

Example 5.15 (n-ary complete-convoluted terms)**.** Following are two examples of the binary complete convolution of terms using padding, i.e. $\boxed{C}^t_2$. The symbol $\langle cons, s\rangle \in \Sigma^{\boxed{C}_2}$ has type $\tau(\langle cons, s\rangle) = ((nat, nat), (natlist, nat)) \to (natlist, nat)$ and the arity of $\langle cons, cons\rangle$ is $|\langle cons, cons\rangle| = 4$. Note how n_{ex}'s constructors have been duplicated in the complete convolution of l_{ex} and n_{ex}.

$\boxed{C}^t_2\ (s(z), s(s(z))) = \langle s,s\rangle(\langle z,s\rangle(\langle\square,s\rangle(\langle\square,z\rangle)))$

With l_{ex} and n_{ex} as defined below, $\boxed{C}^t_2\ (l_{ex}, n_{ex})$ $=$ $\langle cons,s\rangle(\langle z,s\rangle(\langle\square,z\rangle), \langle cons,s\rangle(\langle z,z\rangle, \langle nil,z\rangle))$

$l_{ex} = cons(z, cons(z, nil))$ $\quad n_{ex} = s(s(z))$

5.1.3 Generalizing convolutions

One other limitation of every previously-defined convolutions ($\boxed{L}_n$, $\boxed{R}_n$, and $\boxed{C}_n$) is that they are all of fixed arity, i.e. every tuple of symbols, and thus tuple of terms, was of a given length n. This prevents to represent relations of different arities by the same automaton.

Example 5.16 (fixed arity limitation)**.** Using (for example) the left convolution, the *max* relation on natural numbers can be represented by the state q_{max} in automaton $\mathcal{A}_{max}$

with states $\{q_{max}, q_l, q_r\}$ and transitions

$$\langle z,z,z\rangle() \to q_{max} \qquad \langle s,s,s\rangle(q_{max}) \to q_{max}$$
$$\langle z,s,s\rangle(q_r) \to q_{max} \qquad \langle s,z,s\rangle(q_l) \to q_{max}$$
$$\langle \square,z,z\rangle() \to q_r \qquad \langle z,\square,z\rangle() \to q_l$$
$$\langle \square,s,s\rangle(q_r) \to q_r \qquad \langle s,\square,s\rangle(q_l) \to q_l$$

The *equality* relation on natural numbers can be represented by the automaton with the following transitions:

$$\langle z,z\rangle() \to q_{eq} \qquad \langle s,s\rangle(q_{eq}) \to q_{eq}.$$

These two relations cannot be defined (in different states) by a single automaton with the convolutions defined up to now, as *max* is of arity 3 and *eq* of arity 2. We can easily augment the convolution formalism to allow for any-arity convolution.

Definition 5.17 (Convolution with various arity)**.** Let $\boxed{\cdot}_n$ be any n-ary convolution among $\boxed{L}_n$, $\boxed{R}_n$, and $\boxed{C}_n$.
The any-ary convolution function $\boxed{\cdot}$ is defined as

$$\boxed{\cdot} = \bigcup_{i \in \mathbb{N}} \boxed{\cdot}_i \,.$$

The alphabet $\Sigma^{\boxed{\cdot}}$ is also defined as

$$\Sigma^{\boxed{\cdot}} = \bigcup_{i \in \mathbb{N}} \Sigma^{\boxed{\cdot}_i}.$$

The recursive extension $\boxed{\cdot}^t$ of $\boxed{\cdot}$ to terms is similarly defined, and is equal to

$$\boxed{\cdot}^t = \bigcup_{i \in \mathbb{N}} \boxed{\cdot}^t_i \,.$$

This defines the three any-ary tuple-convolution operators $\boxed{L}$, $\boxed{R}$, $\boxed{C}$, the three any-ary convoluted alphabets $\Sigma^{\boxed{L}}$, $\Sigma^{\boxed{R}}$, $\Sigma^{\boxed{C}}$, and the three any-ary term-convolution operators $\boxed{L}^t$, $\boxed{R}^t$, $\boxed{C}^t$ for when $\boxed{\cdot}_n$ is equal to, respectively, $\boxed{L}_n$, $\boxed{R}_n$, and $\boxed{C}_n$.

The two automata $\mathcal{A}_{max}$ and $\mathcal{A}_{eq}$ of Example 5.16, defined using $\boxed{L}_3$ and $\boxed{L}_2$, can now

be merged into one automaton $\mathcal{A}_{maxeq}$ defined using $\boxed{L}$ that represents four relations (i.e. has four states) $\{q_{max}, q_l, q_r, q_{eq}\}$ by just taking the set-union of the states and transitions. However, this is not yet very satisfying as the three following relations are very similar but nonetheless needed for the definition of *max* and *eq*:

$$\mathcal{R}(q_{eq}, \mathcal{A}_{maxeq}) = \{(t, t) \mid t \in \mathcal{T}_{nat}(\Sigma)\}$$
$$\mathcal{R}(q_r, \mathcal{A}_{maxeq}) = \{(\square, t, t) \mid t \in \mathcal{T}_{nat}(\Sigma)\}$$
$$\mathcal{R}(q_l, \mathcal{A}_{maxeq}) = \{(t, \square, t) \mid t \in \mathcal{T}_{nat}(\Sigma)\}$$

Padding removing Our solution is to change the convolution method by removing any padding, which were primarily making sense for untyped and fixed-arity convolutions. The only role that the padding had was to fill the gaps of arity mismatch for marking which term was short on subterms. However, because symbols have a type and thus an arity, this information is redundant and can therefore be eliminated.

Example 5.18 (padding-free convoluted automaton $\mathcal{A}_{maxeq}$)**.** The automaton $\mathcal{A}_{maxeq}$ can be rewritten in a padding-free version:

$$\begin{array}{ll}
\langle z, z, z\rangle() \to q_{max} & \langle s, s, s\rangle(q_{max}) \to q_{max} \\
\langle z, s, s\rangle(q_r) \to q_{max} & \langle s, z, s\rangle(q_l) \to q_{max} \\
\langle z, z\rangle() \to q_r & \langle z, z\rangle() \to q_l \\
\langle s, s\rangle(q_r) \to q_r & \langle s, s\rangle(q_l) \to q_l \\
& \\
\langle z, z\rangle() \to q_{eq} & \langle s, s\rangle(q_{eq}) \to q_{eq}
\end{array}$$

Without the use of padding, it can even be simplified to only 6 transitions and 2 states by replacing q_l and q_r by q_{eq}:

$$\begin{array}{ll}
\langle z, z, z\rangle() \to q_{max} & \langle s, s, s\rangle(q_{max}) \to q_{max} \\
\langle z, s, s\rangle(q_{eq}) \to q_{max} & \langle s, z, s\rangle(q_{eq}) \to q_{max} \\
\langle z, z\rangle() \to q_{eq} & \langle s, s\rangle(q_{eq}) \to q_{eq}
\end{array}$$

This formalism is more convenient and allows to factorize states and transitions.

5.2 Convolution without padding

We now formally define the left-convolution without the use of padding. The arity of symbols may change within a tree, so this convolution must be defined on all arities at once.

Definition 5.19 (Left-convolution without padding)**.** The any-ary left convolution, written Ⓛ, is defined as:

$$
\begin{aligned}
&Ⓛ\,((\vec{e_1}, \dots, \vec{e_{\boldsymbol{n}}})) = (\vec{e'_1}, \dots, \vec{e'_k}) \\
&\text{with } k = \max_{i \in [1 \dots n]}(|\vec{e_i}|) \quad \text{and} \quad \forall j \in [1 \dots k], \vec{e'_j} = (\boldsymbol{\vec{e_i}[j]})_{i \in [1 \dots n] \wedge \boldsymbol{j \in Pos(\vec{e_i})}}
\end{aligned}
$$

Once again, the differences with Definition 5.2 are written in bold text. Instead of adding a padding when the subterm $\vec{e_i}[j]$ is not defined, we add the condition $j \in Pos(\vec{e_i})$, making sure of their well-definedness.

Example 5.20 (Left-convolution without padding)**.** Again using $\vec{e_1}$, $\vec{e_2}$, $\vec{e_3}$, and $\vec{e_4}$ from Example 5.3, they can be organised in the following array, now not adding any padding but still aligning them left.

a_1	a_2	a_3
b_1		
c_1	c_2	
d_1		

The left convolution Ⓛ$(\vec{e_1}, \vec{e_2}, \vec{e_3}, \vec{e_4})$ is equal to $\big((a_1, b_1, c_1, d_1), (a_2, c_2), (a_3)\big)$, which corresponds to the columns of the array.

This convolution loses some information: the input tuple from which each output element came. However, knowing the arity of the input tuples is enough to reverse this convolution operation. Therefore, because typed functions f have an arity $|f|$, no information is lost while convoluting terms, as shown by the existence of the function Θ of Definition 5.25 and Lemmas 5.27 and 5.28 which show that removing padding preserves the recognizable relations.
The definitions of the any-ary right-convolution Ⓡ and complete-convolution Ⓒ follow the same spirit as the definition of Ⓛ, i.e. it corresponds to their definition that

uses padding but without it, so we do not explicitly redefine them. We usually write $\odot$ for any of $\textcircled{L}$, $\textcircled{R}$, and $\textcircled{C}$.
We now define convoluted alphabets and convolution of terms. The alphabet for convoluted terms without padding are defined for all arities, so note that tuples of symbols of the alphabet's domain are of all sizes.

Definition 5.21 (Alphabet for the convoluted terms)**.** Let Σ a typed alphabet and $\odot$ a convolution. The convoluted alphabet $\Sigma^{\odot}$ has for domain $\bigcup_{i \in \mathbb{N}} \Sigma^i$ and, for $\langle f_1, \ldots, f_n \rangle$ a (tuple of) symbols, its type is $\tau(\langle f_1, \ldots, f_n \rangle) = \odot(\vec{\tau}_1, \ldots, \vec{\tau}_n) \rightarrow (\tau_1, \ldots, \tau_n)$ with $\forall i \in [1 \ldots n], \tau(f_i) = \vec{\tau}_i \rightarrow \tau_i$.

The convolutions of terms $\textcircled{L}^t$, $\textcircled{R}^t$, $\textcircled{C}^t$ are defined as the ones with padding but using the padding-free convolution $\odot$ instead of $\boxdot_n$.

Definition 5.22 (convolution of terms without paddings)**.** Let $\odot$ be any of $\textcircled{L}$, $\textcircled{R}$, and $\textcircled{C}$. The convolution of terms without padding, written $\odot^t$, takes a tuple of terms $(t_1, \ldots, t_n)$ on an alphabet Σ and returns a term $\odot^t(t_1, \ldots, t_n)$ on the convoluted alphabet $\Sigma^{\odot}$. The convolution of terms is recursively defined as:

$$\begin{aligned} \odot^t\,(f_1(\vec{t_1}), \ldots, f_n(\vec{t_n})) &= \langle f_1, \ldots, f_n \rangle(\odot^t(\vec{t'_1}), \ldots, \odot^t(\vec{t'_k})) \\ \text{with } (\vec{t'_1}, \ldots, \vec{t'_k}) &= \odot(\vec{t_1}, \ldots, \vec{t_n}) \end{aligned}$$

Example 5.23 (Padding-free left-convoluted terms)**.** Here is Example 5.6 with the padding-free left convolution $\textcircled{L}^t$.

```
L^t( s , s )   =   <s,s>
     |   |           |
     z   s         <z,s>
         |           |
         s          <s>
         |           |
         z          <z>
```

With l_{ex} and n_{ex} from Example 5.6, $\textcircled{L}^t(l_{ex}, n_{ex})$

```
            =
       <cons,s>
       /      \
   <z,s>     <cons>
     |       /    \
    <z>    <z>   <nil>
```

The padding-free left-convolution $\textcircled{L}^t$ also has the effect of overlaying same-position subterms, but without the padding symbols from terms that are not defined at a position.

Lemma 5.24. *Let* $(t_1, \ldots, t_n) \in \mathcal{T}(\Sigma)^n$.

Then $t = \textcircled{L}^t((t_1, \ldots, t_n))$ *is the tree such that* $\forall \pi \in \mathbb{N}^*$*:*

(a) $\pi \in Pos(t) \iff (n = 0 \land \pi = \epsilon) \lor \exists i \in [1 \ldots n].\ \pi \in Pos(t_i)$

(b) $\pi \in Pos(t) \Rightarrow t[\pi] = Ⓛ^t((t_i[\pi])_{i \in [1 \ldots n] \land \pi \in Pos(t_i)})$

(c) $\pi \in Pos(t) \Rightarrow Root(t[\pi]) = \vec{f}$ *with* $\vec{f} = (Root(t_i[\pi]))_{i \in [1 \ldots n] \land \pi \in Pos(t_i)}$

Proof. This proof is similar to that of Lemma 5.7 and 5.8 so we elide it. □

The representation of convolutions with or without padding are both equivalent in expressivity. We define a function Θ to go from a padding version to the padding-free version.

Definition 5.25 (Removing padding). Let $\odot$ be any of Ⓛ, Ⓡ, or Ⓒ.

Let $\Theta : (\Sigma_\Box)^n \to \Sigma^{\odot}$ be the function removing paddings from tuples of symbols:

$$\Theta((f_1, \ldots, f_n)) \ = \ (f_i)_{i \in [1 \ldots n] \land f_i \neq \Box}$$

This function is extended to tuples of types:

$$\Theta((\tau_1, \ldots, \tau_n)) \ = \ (\tau_i)_{i \in [1 \ldots n] \land \tau_i \neq \Box}$$

Definition 5.26 (Θ generalization). Θ can be extended to

- terms by the *map* generalisation:

$$\Theta(\vec{f}(\vec{t_1}, \ldots, \vec{t_k})) \ = \ \Theta(\vec{f})(\Theta(\vec{t_1}), \ldots, \Theta(\vec{t_k}))$$

- states by being applied to their type:

$$\Theta(q) \text{ with } (\tau(q) = \tau_q) \quad \text{is} \quad q \text{ with } (\tau(q) = \Theta(\tau_q))$$

- transitions by being applied to their symbol and states:

$$\Theta(f(q_1, \ldots, q_k) \to q) \ = \ \Theta(\vec{f})(\Theta(q_1), \ldots, \Theta(q_k)) \to \Theta(q)$$

- automata by being applied to their set of states and transitions:

$$\Theta(Q, \Delta) = (\Theta(Q), \Theta(\Delta))$$

This operation allows to transform a convoluted term that uses padding into a convoluted term that does not and that represents the same tuple of terms.

Lemma 5.27. *Let* $\vec{t} \in \mathcal{T}(\Sigma)^n$*. Then* $\Theta(\boxed{L}^t(\vec{t})) = \textcircled{L}^t(\vec{t})$*.*

Proof. Let $t_\square = \boxed{L}^t(\vec{t})$ and $t = \textcircled{L}^t(\vec{t})$.

- By Lemma 5.7 and 5.24, we have that $Pos(t_\square) = Pos(t)$.
- $\forall \pi \in Pos(t_\square)$,

$$
\begin{array}{lll}
 & & Root(\Theta(t_\square)[\pi]) \\
= & \text{(by immediate induction on } \pi\text{)} & \Theta(Root(t_\square[\pi])) \\
= & \text{(by Lemma 5.8)} & \Theta((Root(t_i[\pi]^\square))_{i\in[1\ldots n]}) \\
= & \text{(by definition of } \Theta\text{)} & (Root(t_i[\pi]))_{i\in[1\ldots n]\wedge\pi\in Pos(t_i)} \\
= & \text{(by Lemma 5.24.(c))} & Root(t[\pi])
\end{array}
$$

Therefore $\Theta(t_\square) = t$. □

Not using padding preserves the expressivity of automata. We prove that, for every automaton defined using the left-convolution with padding, an equivalent automaton using left-convolution without padding can be defined, i.e. with the same states recognizing the same relation. The other direction, asserting that removing paddings does not augment the expressivity, is more intuitive but not proven here, as the construction may require to duplicate states (for type-related reasons) and is therefore more tedious.

Theorem 5.28 (Θ preserves relations)**.** *Let* $\mathcal{A}$ *be an automaton on alphabet* $\Sigma^{\boxed{L}}$ *and* q *be one of its states. Then* $\mathcal{R}(\mathcal{A}, q) = \mathcal{R}(\Theta(\mathcal{A}), \Theta(q))$ *with* $\Theta(\mathcal{A})$ *an automaton on alphabet* $\Sigma^{\textcircled{L}}$*.*

Proof. First, we have

$$
\begin{array}{l}
\mathcal{R}(\mathcal{A}, q) = \mathcal{R}(\Theta(\mathcal{A}), \Theta(q)) \\
\quad \iff \text{(By definition of } \mathcal{R}(\cdot,\cdot)\text{)} \\
\forall \vec{t} \in \mathcal{T}(\Sigma)^n,\ [\boxed{L}^t(\vec{t}) \in \mathcal{L}(\mathcal{A}, q) \iff \textcircled{L}^t(\vec{t}) \in \mathcal{L}(\Theta(\mathcal{A}), \Theta(q))] \\
\quad \Leftarrow \text{(By Lemma 5.27)} \\
\quad \forall t_\square \in \mathcal{T}_{\tau(q)}(\Sigma^{\boxed{L}}),\ [t_\square \in \mathcal{L}(\mathcal{A}, q) \iff \Theta(t_\square) \in \mathcal{L}(\Theta(\mathcal{A}), \Theta(q))]
\end{array}
$$

Let $t_\square \in \mathcal{T}_{\tau(q)}(\Sigma^{\boxed{L}})$, thus $t_\square = \vec{f}(t_{\square 1}, \ldots, t_{\square n})$ for some symbol $\vec{f} \in \Sigma^{\boxed{L}}$ and $\forall i \in [1 \ldots n], t_{\square i} \in \mathcal{T}(\Sigma^{\boxed{L}})$ with $\tau(t_\square) = \tau(q)$. Recall that $\Theta(t_\square) = \Theta(\vec{f})(\Theta(t_{\square 1}), \ldots, \Theta(t_{\square n}))$. Let us prove $t_\square \in \mathcal{L}(\mathcal{A}, q) \iff \Theta(t_\square) \in \mathcal{L}(\Theta(\mathcal{A}), \Theta(q))$ by double implication.

- $\Rightarrow$: Suppose $t_\square \in \mathcal{L}(\mathcal{A}, q)$. Let us prove $\Theta(t_\square) \in \mathcal{L}(\Theta(\mathcal{A}), \Theta q)$ by induction on the hypothesis $t_\square \in \mathcal{L}(\mathcal{A}, q)$.
  There exists a transition $\overrightarrow{f}(q_1, \ldots, q_n) \to q \in \Delta(\mathcal{A})$ such that $\forall i \in [1 \ldots n], t_{\square i} \in \mathcal{L}(\mathcal{A}, q_i)$. By induction, $\forall i \in [1 \ldots n], \Theta(t_{\square i}) \in \mathcal{L}(\Theta(\mathcal{A}), \Theta(q_i))$.
  The translation of the transition $\Theta(\overrightarrow{f}(q_1, \ldots, q_n) \to q)$ is $\Theta(\overrightarrow{f})(\Theta(q_1), \ldots, \Theta(q_n)) \to \Theta(q)$ and is a transition of $\Theta(\mathcal{A})$. Therefore $\Theta(\overrightarrow{f})(\Theta(t_{\square 1}), \ldots, \Theta(t_{\square n})) \in \mathcal{L}(\Theta(\mathcal{A}), \Theta(q))$, which concludes $\Theta(t_\square) \in \mathcal{L}(\Theta(\mathcal{A}), \Theta(q))$.
- $\Leftarrow$: Suppose $\Theta(t_\square) \in \mathcal{L}(\Theta(\mathcal{A}), \Theta(q))$. Let us prove $t_\square \in \mathcal{L}(\mathcal{A}, q)$ by induction on the hypothesis $\Theta(t_\square) \in \mathcal{L}(\Theta(\mathcal{A}), \Theta(q))$.
  There exists a transition $\overrightarrow{f}'(q_1, \ldots, q_n) \to q \in \Delta(\mathcal{A})$ such that $\forall i \in [1 \ldots n]$, $\Theta(t_{\square i}) \in \mathcal{L}(\Theta(\mathcal{A}), \Theta(q_i))$ and with $\Theta(\overrightarrow{f}) = \Theta(\overrightarrow{f}')$. By induction, $\forall i \in [1 \ldots n], t_{\square i} \in \mathcal{L}(\mathcal{A}, q_i)$.
  Because $\tau(t_\square) = \tau(q)$, we have $\tau_{out}(\overrightarrow{f}) = \tau(q)$. Because the transition $\overrightarrow{f}'(q_1, \ldots, q_n) \to q$ is well-typed, we also have $\tau_{out}(\overrightarrow{f}') = \tau(q)$, so $\tau_{out}(\overrightarrow{f}) = \tau_{out}(\overrightarrow{f}')$. Moreover, $\Theta(\overrightarrow{f}) = \Theta(\overrightarrow{f}')$, so $\overrightarrow{f} = \overrightarrow{f}'$.
  Therefore the transition $\overrightarrow{f}'(q_1, \ldots, q_n) \to q$ can be rewritten as $\overrightarrow{f}(q_1, \ldots, q_n) \to q$ and, using induction hypotheses, be used to conclude that $t_\square \in \mathcal{L}(\mathcal{A}, q)$.

$\square$

Definition 2.37 explains how to see a classical tree automaton as a Herbrand model. However, this is not interesting for convoluted automata, as making the relation symbols appear as term constructors prevents from convoluting its arguments in a satisfying way because it makes all arguments appear as subterms of the same function. Moreover, having a special state of type bool is of no use, as each state already represents a relation.

Definition 5.29 (Representing a Herbrand model with a convoluted automaton)**.** A convoluted automaton $\mathcal{A}$ with states Q represents one relation per state q of Q, which can thus be associated with the Herbrand model which maps each state to its denoted relation: $q_i \mapsto \mathcal{R}(\mathcal{A}, q_i)$.
However, we often rather need to represent a Herbrand model *for some relation symbols* $\{R_1, \ldots, R_n\}$ by a convoluted automaton. In that case, each relation symbol R_i is simply assigned a state q_i that denotes its relation, i.e. $\mathcal{A}$ denotes the Herbrand model $R_i \mapsto \mathcal{R}(\mathcal{A}, q_i)$. This mapping between relation symbols and states is often, in the following, made implicit by (re)naming the state assigned to a relation symbol R by q_R.

By convention, we say that $\mathcal{L}(\mathcal{A}, q) = \mathcal{R}(\mathcal{A}, q) = \emptyset$ for any state q that is not in the set of states $Q(\mathcal{A})$. With $\mathcal{H}$ a Herbrand model denoted by an automaton $\mathcal{A}$, we overload the notation $\mathcal{A} \models \varphi$ to mean $\mathcal{H} \models \varphi$.
Note that we may want to represent a relation by multiple states instead of only one (e.g. in the case of a *deterministic* convoluted automaton). Then, a relation between relation symbols and states $\mu \subseteq \mathcal{R} \times Q$ can be defined, and the corresponding Herbrand model $\mathcal{H}$ is defined as $R \mapsto \bigcup_{q \in \mu(R)} \mathcal{R}(\mathcal{A}, q)$. In that case, we explicitly write $(\mathcal{A}, \mu) \models \varphi$ for $\mathcal{H} \models \varphi$.

Example 5.30 (Length model)**.** The length relation Len can be represented using the right convolution Ⓡ in the automaton $\mathcal{A}$ with states $\{q_n, q_{\text{Len}}\}$ and transitions

$$\langle z \rangle() \to q_n \qquad \langle s \rangle(q_n) \to q_n \qquad \langle nil, z \rangle() \to q_{\text{Len}} \qquad \langle cons, s \rangle(q_n, q_{\text{Len}}) \to q_{\text{Len}}.$$

With $\varphi = \text{Len}(cons(X, L), s(N)) \Leftarrow \text{Len}(L, N)$, we have $\mathcal{A} \models \varphi$.

Finally, note that convolutions can also be seen as a shuffling (with duplication for the complete convolution) of positions.

Lemma 5.31 (Convolution and positions)**.** *Let $\odot$ be any of Ⓛ, Ⓡ, Ⓒ. Let $\vec{e} = (\vec{e_1}, \dots, \vec{e_n})$ with each $\vec{e_i}$ a tuple. Let, for all $i \in [1 \dots n]$, $\vec{\pi_i} = (i \cdot 1, \dots, i \cdot |\vec{e_i}|)$ be the positions pointing to each element of $\vec{e_i}$ from $\vec{e}$. Let $(\vec{e'_1}, \dots, \vec{e'_k}) = \odot(\vec{e_1}, \dots, \vec{e_n})$ and $(\vec{\pi'_1}, \dots, \vec{\pi'_k}) = \odot(\vec{\pi_1}, \dots, \vec{\pi_n})$.*
Then, for all $j \in [1 \dots k]$, writing $\vec{\pi'_j}$ as $(\pi_1 \dots, \pi_m)$, we have $\vec{e'_j} = (\vec{e}[\pi_1], \dots, \vec{e}[\pi_m])$.

Proof. For any $i \in [1 \dots n]$ and $j \in [1 \dots |\vec{e_i}|]$, we have $\vec{e_i}[j] = \vec{e}[i \cdot j] = \vec{e}[\vec{\pi_i}[j]]$. Because $\odot$ is polymorphic w.r.t the tuples element, we immediately have $\forall j \in [1 \dots k], \forall i \in [1 \dots |\vec{e'_j}|], \vec{e'_j}[i] = \vec{e}[\vec{\pi'_j}[i]]$, which concludes. □

CHAPTER 6

LEARNER/TEACHER FOR CONVOLUTED AUTOMATA

This chapter is dedicated to the Learner and Teacher procedures using convoluted tree automata as models. The generic procedure for satisfiability of a set of clauses (for which the learner and teacher are defined) is presented in Section 4.3. Using convoluted automata, the teacher's role is to verify, given a convoluted tree automaton $\mathcal{A}$ and a set of clauses Γ, whether $\mathcal{A} \models \Gamma$. The learner's role is to infer, given a set of ground clauses $\widehat{\Gamma}$, a new automaton $\mathcal{A}$ such that $\mathcal{A} \models \widehat{\Gamma}$, and hopefully $\mathcal{A} \models \Gamma$.
We begin by defining the Teacher procedure in Section 6.1, then define the Learner procedure in Section 6.2, and finally prove two theorems for the whole learner/teacher *Sat* procedure in Section 6.3.

6.1 Teacher

In this section, we define the *Teacher* procedure. The *Teacher* takes as input a finite set of clauses Γ, representing (an approximation of) the program and property to verify, and a convoluted automaton $\mathcal{A}$, representing a proposed model that is intended to prove satisfiability of Γ. If $\mathcal{A} \models \Gamma$ then $\mathit{Teacher}(\mathcal{A}, \Gamma)$ should return *None* and the verification algorithm succeeds. If $\mathcal{A} \not\models \Gamma$ then $\mathit{Teacher}(\mathcal{A}, \Gamma)$ should return $\mathit{Some}(\widehat{\varphi})$ where $\widehat{\varphi}$ is a *counterexample* to $\mathcal{A} \models \Gamma$, i.e. a ground instance of some clause $\varphi \in \Gamma$ such that $\mathcal{A} \not\models \widehat{\varphi}$. However, this problem (that *Teacher* is addressing) is undecidable in general, as stated in Theorem A.10. Therefore the procedure given here, being correct, is incomplete, i.e., it may diverge.
Teacher is based on the $\mathit{Inhabits}_{\mathcal{A}}$ procedure which allows the checking $\mathcal{A} \models \varphi$ of a single formula $\varphi \in \Gamma$. $\mathit{Teacher}(\mathcal{A}, \Gamma)$ is then defined by using one instance of the $\mathit{Inhabits}_{\mathcal{A}}$ procedure per formula $\varphi \in \Gamma$. Every definition throughout this chapter is valid for any convolution $\odot$ and right convolution Ⓡ is used in all the examples. This abstraction

will allow us to easily adapt the whole procedure to SHoCs in Section 8.1. Observe how the precise structure of transitions from tree automata is only used in Definition 6.7 and is afterwards abstracted. This allows us to only redefine one function in order to adapt the whole procedure to SHoCs, as discussed in Section 8.1. Every Lemma and Theorem whose proof is omitted in this section can be found in Appendix A.1.

6.1.1 The $Inhabits_{\mathcal{A}}$ procedure

The goal of this section is to devise an algorithm $Inhabits_{\mathcal{A}}$ to search for counterexamples of a claim $\mathcal{A} \models \varphi$. The algorithm we define is guided by the transitions of the automaton $\mathcal{A}$.
Note that, because every state of a convoluted tree automaton can be seen as the relation symbol of the relation it denotes (Definition 5.29), we may as well use them as relation symbols in first-order atoms.

Definition 6.1 (Atom with a state)**.** An *atom* is the data of a state q and a tuple of pattern $(p_1, \ldots, p_n)$ of compatible type, i.e. $\tau(q) = (\tau(p_1), \ldots, \tau(p_n))$. We write such an atom as a first-order logic atom $q(p_1, \ldots, p_n)$. An atom is usually written ω and a set of atoms Ω.

Notations from first-order logic can thus be used on atoms $\omega = q(p_1, \ldots, p_n)$ and sets of atoms Ω. That is, for $\mathcal{A}$ an automaton, we can write $\mathcal{A} \models \omega$ and $\mathcal{A} \models \Omega$ with the same meaning as in Definition 2.25.
We now make more formal the idea that a counterexample to φ is a substitution that makes the body of the clause true but the head false. Recall that q_R is the state that denotes relation R (Definition 5.29) and that q^c is the state denoting the complement relation of the state q.

Definition 6.2 (Negating a clause)**.** Let φ be the clause $H \Leftarrow B$. We write $\Omega_{\overline{\varphi}}$ for the set $\{q^c_R(\vec{p}) \mid R(\vec{p}) \in H\} \cup \{q_R(\vec{p}) \mid R(\vec{p}) \in B\}$.

Proposition 6.3 (Model checking as substitution existence)**.** *For any automaton $\mathcal{A}$ and clause φ,*

$$\mathcal{A} \not\models \varphi \iff \text{There exists a substitution } \sigma \text{ such that } \mathcal{A} \models \sigma(\Omega_{\overline{\varphi}})$$

Example 6.4 (Verifying that all lists are of even length)**.** Let φ be $\text{Even}(N) \Leftarrow \text{Len}(L, N)$, a formula stating that all lists are of even length. Let the right-convoluted Ⓡ automaton $\mathcal{A}_{le}$ with states $\{q_{\text{Even}}, q_{\text{Len}}, q_n, q^c_{\text{Even}}\}$ and transitions

(A) :	$\langle z \rangle() \rightarrow q_n$	(1)	$\langle z \rangle() \rightarrow q_{\text{Even}}$
(B) :	$\langle s \rangle(q_n) \rightarrow q_n$	(2) :	$\langle s \rangle(q^c_{\text{Even}}) \rightarrow q_{\text{Even}}$
(C) :	$\langle cons, s \rangle(q_n, q_{\text{Len}}) \rightarrow q_{\text{Len}}$	(3) :	$\langle s \rangle(q_{\text{Even}}) \rightarrow q^c_{\text{Even}}$
(D) :	$\langle nil, z \rangle() \rightarrow q_{\text{Len}}$		

Note that the state q_{Len} denotes the relation Len, which related lists and their length ; q_{Even} denotes the relation Even, the set of all even natural numbers ; q^c_{Even} denotes the complement of the relation Even, the set of all odd natural numbers. Even appears in the head of the property φ, which is the reason we need its complement encoded in $\mathcal{A}$ (in q^c_{Even}).
To check whether $\mathcal{A}_{le} \not\models \varphi$, we first translate φ into the set of atoms representing its negation $\Omega_{\overline{\varphi}} = \{q_{\text{Len}}(L, N), q^c_{\text{Even}}(N)\}$. By proposition 6.3, $\mathcal{A}_{le} \not\models \varphi$ iff there exists a substitution σ such that $\mathcal{A}_{le} \models \sigma(\Omega_{\overline{\varphi}})$.

For $\mathcal{A}$ a convoluted automaton and φ a property, the procedure $Inhabits_{\mathcal{A}}(\varphi)$, trying to check whether $\mathcal{A} \models \varphi$ or $\mathcal{A} \not\models \varphi$, will therefore search for a substitution σ such that $\mathcal{A} \models \sigma(\Omega_{\overline{\varphi}})$. We now define this procedure.

Inhabits

Intuitively, any solution σ for an atom $\omega = q(p_1, \ldots, p_n)$, i.e. substitution σ such that $\sigma(p_1, \ldots, p_n) \in \mathcal{R}(\mathcal{A}, q)$, can be found by following transitions of the automaton $\mathcal{A}$. A transition $\langle f_1, \ldots, f_n \rangle(q_1, \ldots, q_k) \rightarrow q$ of $\mathcal{A}$ can act as a typing rule whose application to the atom $q(p_1, \ldots, p_n)$ generates k new atoms (one for each sub-state q_j of the transition) and n new algebraic datatype constraints, the ith stating that pattern p_i must be of the form $f_i(\overrightarrow{X_i})$ with $\overrightarrow{X_i}$ some fresh variables. This kind of reasoning leads to the definition of the *unfolding* functions, which are then used to define $Inhabits_{\mathcal{A}}(\Omega)$.
We first define $compatibility(\overrightarrow{p}, \overrightarrow{f})$, a function that synthesizes under what condition a tuple of patterns can be of the shape of a tuple of function symbols.

Definition 6.5 (Compatibility of patterns with symbols)**.** Let $(p_1, \ldots, p_n)$ be a tuple of patterns and $\langle f_1, \ldots, f_n \rangle$ a tuple of symbols. Then $compatibility((p_1, \ldots, p_n), (f_1, \ldots, f_n))$

is the syntactic unification problem $\{p_i \stackrel{?}{=} f_i(\overrightarrow{X_i}) \mid i \in [1 \dots n]\}$ with, for all $i \in [1 \dots n]$, $\overrightarrow{X_i}$ some fresh variables.

Example 6.6 ($compatibility(\overrightarrow{p}, \overrightarrow{f})$)**.** Let $\overrightarrow{f} = \langle cons, s\rangle$. Let $\overrightarrow{p_1} = (cons(N, L), N)$ and $\overrightarrow{p_2} = (cons(N, L), z)$. Then $compatibility(\overrightarrow{p_1}, \overrightarrow{f}) = \{cons(N, L) \stackrel{?}{=} cons(X_1, X_2), N \stackrel{?}{=} s(X_3)\}$ is unifiable by the MGU $\sigma = \{L \mapsto X_2,\ N \mapsto s(X_3),\ X_1 \mapsto s(X_3)\}$. However, $compatibility(\overrightarrow{p_2}, \overrightarrow{f}) = \{cons(N, L) \stackrel{?}{=} cons(X_1, X_2), z \stackrel{?}{=} s(X_3)\}$ is not unifiable, as z cannot be made equal to $s(X_3)$.

With some patterns $\overrightarrow{p}$ that are trivially compatible with the head symbols $\overrightarrow{f}$ of a transition $r = \overrightarrow{f}(\overrightarrow{q}) \rightarrow q$, we can use the transition to start recognizing $\overrightarrow{p}$ in the state q. That is, in order to satisfy the atom $q(\overrightarrow{p})$, it is sufficient to satisfy the set of atoms $unfoldOne(\overrightarrow{p}, r)$.

Definition 6.7 ($unfoldOne$: generation of new atoms)**.** Let $\overrightarrow{p} = (f_1(\overrightarrow{p_1}), \dots, f_n(\overrightarrow{p_n}))$ be a tuple of patterns and $r = \langle f_1, \dots, f_n\rangle(q_1, \dots, q_k) \rightarrow q$ a transition of some automaton defined using convolution $\bigodot$. Then $unfoldOne(\overrightarrow{p}, r) = \{q_j(\overrightarrow{p'_j}) \mid j \in [1 \dots k]\}$ with $(\overrightarrow{p'_1}, \dots, \overrightarrow{p'_k}) = \bigodot(\overrightarrow{p_1}, \dots, \overrightarrow{p_n})$.

Example 6.8 ($unfoldOne$)**.** Let $\overrightarrow{p} = (cons(X, nil), s(N))$ be a tuple of patterns and $r = \langle cons, s\rangle(q_n, q_{\text{Len}}) \rightarrow q_{\text{Len}}$ be a transition. Then $unfoldOne(\overrightarrow{p}, r) = \{q_n(X), q_{\text{Len}}(nil, N)\}$. In more details, we have, reusing notations from the definition, that $n = k = 2$, $f_1 = cons$, $f_2 = s$, $\overrightarrow{p_1} = (X, nil)$, $\overrightarrow{p_2} = (N)$, $q_1 = q_n$, $q_2 = q_{\text{Len}}$. Note also that, because $((X), (nil, N)) = \circledR(((X, nil)), (N))$, we have $\overrightarrow{p'_1} = (X)$ and $\overrightarrow{p'_2} = (nil, N)$.

Reasoning about the unfolding of a single atom does not take into account that there may be shared variables in a set of atoms Ω. Therefore, we define the conjoint unfolding of atoms by a transition each. Note that in the following definition, contrary to definition 6.7, the patterns may not be able to match the transition's function symbols shape, so it returns an optional value.

Definition 6.9 ($unfolds$: Unfolding a set of atoms with fixed transitions)**.**
Let $T = \{(\omega_1, r_1), \dots, (\omega_n, r_n)\}$ be a set of pairs of an atom and a transition. We write, for every $i \in [1 \dots n]$, the atom ω_i as $q_i(\overrightarrow{p_i})$, the transition r_i as $\overrightarrow{f_i}(\overrightarrow{q_i}) \rightarrow q'_i$, and let the

unification problem $E_i = compatibility(\overrightarrow{p_i}, \overrightarrow{f_i})$. Then, with $E = \bigcup_{i\in[1\ldots n]} E_i$, we define

$$unfolds(T) = \begin{cases} Some(\bigcup_{i\in[1\ldots n]} \Omega_i,\ \sigma_\delta) & \text{if } \sigma_\delta = MGU(E) \text{ exists and, } \forall i \in [1\ldots n], \\ & \Omega_i = unfoldOne(\sigma_\delta(\overrightarrow{p_i}), r_i) \text{ and } q_i = q'_i\,; \\ None & \text{otherwise.} \end{cases}$$

Example 6.10 ($unfolds$)**.** Let

- $\omega_1 = q_{\text{Len}}(L, N)$ and $r_1 = \langle cons, s\rangle(q_n, q_{\text{Len}}) \to q_{\text{Len}}$;
- $\omega_2 = q_{\text{Len}}(L, N)$ and $r_2 = \langle nil, z\rangle() \to q_{\text{Len}}$;
- $\omega_3 = q^c_{\text{Even}}(N)$ and $r_3 = \langle s\rangle(q_{\text{Even}}) \to q^c_{\text{Even}}$;
- $T_{\{1,3\}} = \{(\omega_1, r_1), (\omega_3, r_3)\}$ and $T_{\{2,3\}} = \{(\omega_2, r_2), (\omega_3, r_3)\}$.

Then

- $unfolds(T_{\{1,3\}}) = Some(\Omega_{\{1,3\}}, \sigma_{\{1,3\}})$ with $\sigma_{\{1,3\}} = \{L \mapsto cons(L_1, L_2),\ N \mapsto s(N_1)\}$ and $\Omega_{\{1,3\}} = \{q_n(L_1), q_{\text{Len}}(L_2, N_1), q_{\text{Even}}(N_1)\}$.

  In more details, we have $E_1 = compatibility((L, N), r_1) = \{L \stackrel{?}{=} cons(L_1, L_2), N \stackrel{?}{=} s(N_1)\}$, $E_3 = compatibility((N), r_3) = \{N \stackrel{?}{=} s(N_3)\}$ and $MGU(E_1 \cup E_3) = \sigma_{\{1,3\}}$. Also, $\Omega_1 = unfoldOne(\sigma_{\{1,3\}}(\omega_1), r_1) = \{q_n(L_1), q_{\text{Len}}(L_2, N_1)\}$ and $\Omega_3 = unfoldOne(\sigma_{\{1,3\}}(\omega_3), r_3) = \{q_{\text{Even}}(N_1)\}$, both constituting $\Omega_{\{1,3\}}$.
- $unfolds(T_{\{2,3\}}) = None$ because using transition r_2 on atom ω_2 forces N to be of the shape z and using transition r_3 on atom ω_3 forces N to be of the shape $s(N_1)$, so no MGU exists.

Lemma 6.11 ($unfolds$)**.** *Let $\Omega = \{\omega_1, \ldots, \omega_n\}$ be a set of atoms, $T = \{(\omega_1, r_1), \ldots, (\omega_n, r_n)\}$ be a set of pairs of an atom and a transition of some automaton $\mathcal{A}$, assigning a transition to each atom ω_i of Ω. Suppose $unfolds(T) = Some(\Omega', \sigma_\delta)$. Then for all substitution σ such that $\mathcal{A} \models \sigma(\Omega')$, we have $\mathcal{A} \models \sigma(\sigma_\delta(\Omega))$.*

Now that unfolding a set of atoms that had a transition assigned to each is defined ($unfolds$), we can define the set of possible unfoldings of a set of atoms in an automaton.

Definition 6.12 ($unfolds_{\mathcal{A}}$: All unfoldings of a set of atoms in $\mathcal{A}$)**.** Let $\Omega = \{\omega_1, \ldots, \omega_n\}$ be a set of atoms and $\mathcal{A}$ be an automaton. Then

$$unfolds_{\mathcal{A}}(\Omega) = \{(\Omega', \sigma_\delta) \mid \quad \forall i \in [1 \dots n], \exists r_i \in \Delta(\mathcal{A}), \\ unfolds(\{(\omega_1, r_1), \dots, (\omega_n, r_n)\}) = Some(\Omega', \sigma_\delta).\}$$

Definition 6.13 ($Inhabits_{\mathcal{A}}$)**.** Let $\mathcal{A}$ be a convoluted automaton and Ω a set of atoms. $Inhabits_{\mathcal{A}}(\Omega)$ is a non-deterministic algorithm defined as:

1. `Let` $\sigma_0 := \emptyset, \Omega_0 := \Omega,$ `and` $i := 0$.
2. `If` $\Omega_i = \emptyset$, `then return` $Some(\sigma_i)$.
3. `CHOOSE` $(\Omega_{i+1}, \sigma_\delta) \in unfolds_{\mathcal{A}}(\Omega_i)$.
   `If no such choice can be made (i.e.` $unfolds_{\mathcal{A}}(\Omega_i) = \emptyset$`), then return` $None$.
   `Otherwise, let` $\sigma_{i+1} := \sigma_\delta \circ \sigma_i,\ i := i+1$, `and go back to instruction` 2.

$Inhabits_{\mathcal{A}}$ is a non-deterministic algorithm. We write $Inhabits^{det}_{\mathcal{A}}$ for a determinisation of $Inhabits_{\mathcal{A}}$ where the choices are implemented as a breadth-first search until the returned value is a $Some(_)$ or all branches return $None$. That is, $Inhabits^{det}_{\mathcal{A}}$ returning $Some(_)$ in i steps implies that the same execution can be done in $Inhabits_{\mathcal{A}}$ and that any other execution of $Inhabits_{\mathcal{A}}$ returning $Some(_)$ in j steps is such that $i \leq j$. Also, if there exists an execution of $Inhabits_{\mathcal{A}}$ that returns $Some(_)$ in i steps, then $Inhabits^{det}_{\mathcal{A}}$ returns $Some(_)$ in $j \leq i$ steps. We call *positive* instance a set of atoms Ω for which there exists an execution of $Inhabits_{\mathcal{A}}(\Omega)$ that returns $Some(_)$, or equivalently for which the execution $Inhabits^{det}_{\mathcal{A}}(\Omega)$ returns $Some(_)$.

Example 6.14. This example is a complete execution of $Inhabits^{det}_{\mathcal{A}_{le}}(\Omega_{\overline{\varphi}})$ with $\mathcal{A}_{le}$ and $\varphi = \text{Even}(N) \Leftarrow \text{Len}(L, N)$ defined in Example 6.4.
The initialisation sets $\Omega_0 = \Omega_{\overline{\varphi}} = \{q_{\text{Len}}(L, N), q^c_{\text{Even}}(N)\}$.
The execution does not stop at instruction 2 because $\Omega_0 \neq \emptyset$, so let us consider $unfolds_{\mathcal{A}_{le}}(\Omega_0)$. This was mostly Example 6.10. Indeed, the choice of transition r_ω for each atom ω can be restricted to those defining the state of the atom ω, as otherwise $unfolds$ will only return $None$. With this in mind, there is one potentially compatible transition for the atom $q^c_{\text{Even}}(N)$ and two for $q_{\text{Len}}(L, N)$. Relying on example 6.10, our only choice is to select transition $\langle cons, s\rangle(q_n, q_{\text{Len}}) \to q_{\text{Len}}$ for $q_{\text{Len}}(L, N)$ and $\langle s\rangle(q_{\text{Even}}) \to q^c_{\text{Even}}$ for $q^c_{\text{Even}}(N)$. Thus $unfolds_{\mathcal{A}_{le}}(\Omega_0) = \{(\Omega_1, \sigma_{0\to 1})\}$ with $\sigma_{0\to 1} = \{L \mapsto cons(L_1, L_2),\ N \mapsto s(N_1)\}$ and $\Omega_1 = \{q_n(L_1), q_{\text{Len}}(L_2, N_1), q_{\text{Even}}(N_1)\}$. σ_1 is defined as $\sigma_{0\to 1} \circ \sigma_0 = \sigma_{0\to 1}$.
We have $\Omega_1 \neq \emptyset$, so the execution does not stop at step 1 (recall that the procedure started at step 0), so let us consider $unfolds_{\mathcal{A}_{le}}(\Omega_1)$.
The only four possibilities of transitions choices are:

a) Applying transition
   - $\langle z\rangle() \to q_n$ on $q_n(L_1)$,
   - $\langle z\rangle() \to q_{\text{Even}}$ on $q_{\text{Even}}(N_1)$, and
   - $\langle nil, z\rangle() \to q_{\text{Len}}$ on $q_{\text{Len}}(L_2, N_1)$.

b) Applying transition
   - $\langle z\rangle() \to q_n$ on $q_n(L_1)$,
   - $\langle s\rangle(q^c_{\text{Even}}) \to q_{\text{Even}}$ on $q_{\text{Even}}(N_1)$, and
   - $\langle cons, s\rangle(q_n, q_{\text{Len}}) \to q_{\text{Len}}$ on $q_{\text{Len}}(L_2, N_1)$

c) Same as a), but apply $\langle s\rangle(q_n) \to q_n$ on $q_n(L_1)$ instead.

d) Same as b), but apply $\langle s\rangle(q_n) \to q_n$ on $q_n(L_1)$ instead.

Thus $unfolds_{\mathcal{A}_{le}}(\Omega_1) = \{(\Omega_2^{a)}, \sigma_{1\to 2}^{a)}), (\Omega_2^{b)}, \sigma_{1\to 2}^{b)}), (\Omega_2^{c)}, \sigma_{1\to 2}^{c)}), (\Omega_2^{d)}, \sigma_{1\to 2}^{d)})\}$ with

a) $\Omega_2^{a)} = \emptyset$ and $\sigma_{1\to 2}^{a)} = \{L_1 \mapsto z,\ L_2 \mapsto nil,\ N_1 \mapsto z\}$

b) $\Omega_2^{b)} = \{q_n(L_{21}),\ q_{\text{Len}}(L_{22}, N_{11}),\ q^c_{\text{Even}}(N_{11})\}$ and
$\sigma_{1\to 2}^{b)} = \{L_1 \mapsto z,\ L_2 \mapsto cons(L_{21}, L_{22}),\ N_1 \mapsto s(N_{11})\}$

c), d) Similar to a) and b).

Because $Inhabits^{det}_{\mathcal{A}_{le}}$ is implemented as a breadth-first search, its execution will try these four combinations until one of them returns $Some(_)$. It first tries the case a), so $\Omega_2 = \Omega_2^{a)} = \emptyset$ and $\sigma_2 = \sigma_{1\to 2}^{a)} \circ \sigma_1 = \{L \mapsto cons(z, nil),\ N \mapsto s(z)\}$. Because the set of atoms Ω_2 is empty, the execution stops and returns σ_2.
Note that the found substitution σ_2 makes sense for disproving the property $\text{Even}(N) \Leftarrow \text{Len}(L, N)$, as it witnesses the existence of a list of non-even length.

Theorem 6.15 ($Inhabits_{\mathcal{A}}$ correctness and relative completeness)**.** *Let $\mathcal{A}$ be an automaton and Ω a set of atoms.*

- ***Correctness:*** *If $Inhabits_{\mathcal{A}}(\Omega)$ terminates with $Some(\sigma)$, then $\mathcal{A} \models \sigma(\Omega)$.*
- ***Relative completeness:*** *If there exists a substitution σ such that $\mathcal{A} \models \sigma(\Omega)$, then there exists a terminating execution of $Inhabits_{\mathcal{A}}(\Omega)$ returning $Some(_)$.*

Note that the completeness is relative: no termination guarantee is given when there does not exists such a substitution, which is the focus of Section 6.1.1.
The substitutions returned by $Inhabits_{\mathcal{A}}$ describe counterexamples whose height can be framed by the step i at which the algorithm stopped, which later helps to prove the relative completeness of the Sat procedure.

Lemma 6.16 ($Inhabits_{\mathcal{A}}$ height boundedness)**.** *If* $Inhabits_{\mathcal{A}}(\Omega)$ *terminates at step i with* $Some(\sigma)$*, then* $i \leq ht(\sigma(\Omega)) \leq i + ht(\Omega)$.

We now present two sound optimizations which significantly improve the proving and disproving power of the proof search procedure. Using those optimizations makes this procedure usable and efficient in practice (see experiments in Section 9.2). The first one is a loop detection, allowing the procedure to prune the search space and stop on more negative instances, and the second one is the splitting of independent atoms from a set of atoms, allowing the procedure to lower its unnecessary combinatorics.

Pruning the search tree

The $Inhabits_{\mathcal{A}}$ procedure is for now very *in*efficient at proving the non-existence of a substitution. This first optimisation consists in *pruning the search tree* in order to avoid the following situation.

Example 6.17 (Pruning is necessary)**.** A very simple unsatisfiable set of atoms is the following, stating that N is both even and odd: $\Omega_0 = \{q_{\text{Even}}(N), q^c_{\text{Even}}(N)\}$.
During the execution of $Inhabits_{\mathcal{A}_{le}}(\Omega_0)$, the only element in $unfolds_{\mathcal{A}_{le}}(\Omega_0)$ is (Ω_1, σ_1) with $\sigma_1 = \{N \mapsto N_1\}$ and $\Omega_1 = \{q_{\text{Even}}(N_1), q^c_{\text{Even}}(N_1)\}$, which corresponds to using transitions $\langle s \rangle(q_{\text{Even}}) \rightarrow q^c_{\text{Even}}$ on atom $q^c_{\text{Even}}(N)$ and $\langle s \rangle(q^c_{\text{Even}}) \rightarrow q_{\text{Even}}$ on atom $q_{\text{Even}}(N)$. We see that Ω_1 is very similar to Ω_0, in fact identical modulo renaming. The execution of $Inhabits^{det}_{\mathcal{A}_{le}}(\Omega_0)$ thus runs forever, with every Ω_i being equivalent.

The search space is, for almost all sets of atoms, infinite. Without pruning, it would be impossible to cover the whole search space, and therefore negative instances would (almost) all never terminate.
Pruning consists in, during an execution $Inhabits_{\mathcal{A}}(\Omega)$, returning *None* as soon as the current set of atoms Ω_i is harder than (or equivalent to) a set of atoms Ω_j with $j < i$. See instructions 2 and 3 of the final $Inhabits_{\mathcal{A}}$ algorithm from Definition 6.29 for a precise formulation of this pruning extension. Pruning the search tree allows, in some cases, to finitely ensure that no satisfying substitution exists.
We begin by formally defining what as easier set of atoms is (Definition 6.18) and a simple characterization of them using substitutions (Lemma 6.19).

Definition 6.18 (Easier set of atoms)**.** A set of atoms Ω_a is said to be easier than Ω_b, written $\Omega_a \leq \Omega_b$, whenever, for any first-order model $\mathcal{M}$,

$$Assigns(\mathcal{M}, \Omega_b) \neq \emptyset \implies Assigns(\mathcal{M}, \Omega_a) \neq \emptyset.$$

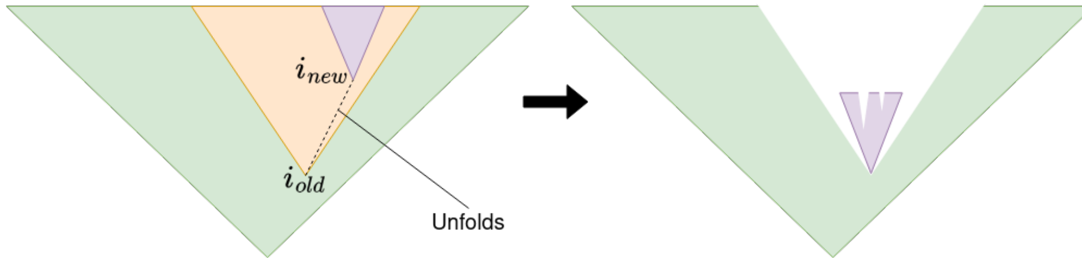
Lemma 6.19 (Characterisation of easier sets of atoms)**.** *Let Ω_a and Ω_b two sets of atoms. Then*

$$\Omega_a \leq \Omega_b \iff \text{there exists a substitution } \sigma \text{ such that } \sigma(\Omega_a) \subseteq \Omega_b.$$

Now, the following lemma and corollaries states that, given two sets of atoms $\Omega_a \leq \Omega_b$, an execution $Inhabits_{\mathcal{A}}(\Omega_b)$ can be simplified into a similar execution of $Inhabits_{\mathcal{A}}(\Omega_a)$.

Lemma 6.20 (Easier relation is preserved by unfolding)**.** *Let some automaton $\mathcal{A}$ and two sets of atoms Ω_a and Ω_b such that $\Omega_a \leq \Omega_b$. Then, for every $(\Omega'_b, \sigma_b) \in unfolds_{\mathcal{A}}(\Omega_b)$, there exists $(\Omega'_a, \sigma_a) \in unfolds_{\mathcal{A}}(\Omega_a)$ such that $\Omega'_a \leq \Omega'_b$.*

Now, we state that pruning branches (returning *None*) in $Inhabits_{\mathcal{A}}(\Omega)$ when having to solve a harder set of atom $\Omega_{i_{new}}$ than an earlier set of atoms $\Omega_{i_{old}}$ is safe, i.e. keeps its minimal positive executions. The following figure illustrates how to shorten an execution in which there exists i_{old} and i_{new} such that $i_{old} < i_{new}$ and $\Omega_{i_{old}} \leq \Omega_{i_{new}}$. The larger (green) triangle represents the whole execution of $Inhabits_{\mathcal{A}}(\Omega)$; the medium (yellow) triangle represents the execution starting from step i_{old} ; the smallest (purple) triangle represents the execution starting from step i_{new}. Shortening the execution consists in replacing the medium triangle by the smallest one. Note that, in the shorten execution, the smallest triangle is not whole. This is because the pruning extension considers *easier* sets of atoms, so $\Omega_{i_{new}}$ may thus have a smaller execution than $\Omega_{i_{old}}$. This corresponds to Lemma 6.21.



Lemma 6.21 (Safety of pruning)**.** *Suppose that an execution of $Inhabits_{\mathcal{A}}(\Omega)$ returns $Some(\sigma)$ in i steps and is such that there exists $i_{old}, i_{new} \in [0 \ldots i-1]$ with $i_{old} < i_{new}$ and $\Omega_{i_{old}} \leq$*

$\Omega_{i_{new}}$. *Then there exists an other execution of* $Inhabits_{\mathcal{A}}(\Omega)$ *that returns* $Some(\sigma')$ *at some step* $j < i$.

Given an execution of $Inhabits_{\mathcal{A}}(\Omega)$ that returns $Some(_)$, applying Lemma 6.21 while possible yields the following corollary.

Corollary 6.22 (Iterative pruning)**.** *Suppose that an execution of* $Inhabits_{\mathcal{A}}(\Omega)$ *returns* $Some(_)$ *in* i *steps. Then there exists an execution of* $Inhabits_{\mathcal{A}}(\Omega)$ *that returns* $Some(_)$ *in* $j \leq i$ *steps such that, for any* i_{old} *and* i_{new} *with* $0 \leq i_{old} < i_{new} \leq j$, *we have* $\Omega_{i_{old}} \not\leq \Omega_{i_{new}}$.

Corollary 6.23. *Extending* $Inhabits_{\mathcal{A}}$ *to return None as soon as* $\Omega_j \leq \Omega_i$ *for some* $j < i$ *does not change the smallest positive executions. In particular, the breadth-first search is unchanged w.r.t positive instances by this pruning extension.*

Example 6.24 (Pruning of the search tree)**.** Getting back to Example 6.17, we have $\Omega_0 = \{q_{\text{Even}}(N), q^c_{\text{Even}}(N)\}$ and $\Omega_1 = \{q_{\text{Even}}(N_1), q^c_{\text{Even}}(N_1)\}$. With $\sigma = \{N \mapsto N_1\}$, we have $\sigma(\Omega_0) \subseteq \Omega_1$, so $\Omega_0 \leq \Omega_1$. That is, this execution (and any other) of $Inhabits_{\mathcal{A}}(\Omega_0)$ would return *None* with the pruning extension.
As for the execution of $Inhabits^{det}_{\mathcal{A}_{le}}(\Omega_0)$ with $\Omega_0 = \{q_{\text{Len}}(L, N), q^c_{\text{Even}}(N)\}$ from Example 6.14, it would not have changed with the pruning. However, if the procedure chose $(\Omega_2^{b)}, \sigma_{1\to 2}^{b)})$ in $unfolds_{\mathcal{A}_{le}}(\Omega_1)$ instead of $(\Omega_2^{a)}, \sigma_{1\to 2}^{a)})$ at step 1, then we would have had $\Omega_2 = \Omega_2^{a)} = \{q_n(L_{21}),\ q_{\text{Len}}(L_{22}, N_{11}),\ q^c_{\text{Even}}(N_{11})\}$. In that case, with $\sigma = \{L \mapsto L_{22},\ N \mapsto N_{11}\}$, we have $\sigma(\Omega_0) \subseteq \Omega_2$, so $\Omega_0 \leq \Omega_2$ and the execution $Inhabits^{det}_{\mathcal{A}_{le}}(\Omega_0)$ would have pruned this branch of the search and got back to choosing differently in $unfolds_{\mathcal{A}_{le}}(\Omega_1)$.

Independent atoms

As can be seen at step $i = 1$ of Example 6.14, there is more entanglement than necessary in the $Inhabits_{\mathcal{A}}$ procedure. Indeed, with $\Omega_1 = \{q_n(L_1), q_{\text{Len}}(L_2, N_1), q_{\text{Even}}(N_1)\}$, the atom $q_n(L_1)$ shares no variable with the others, so its resolution is independent of the resolution of $q_{\text{Len}}(L_2, N_1)$ and $q_{\text{Even}}(N_1)$. These two last atoms, however, cannot be separated, as we need to ensure that the chosen value for N_1 is the same for both atoms. This independence is not yet used in the $Inhabits_{\mathcal{A}}$ algorithm and the resulting unnecessary combinatorics can be seen in the four possible unfoldings $unfolds_{\mathcal{A}_{le}}(\Omega_1) = \{(\Omega_2^{a)}, \sigma_{1\to 2}^{a)}), (\Omega_2^{b)}, \sigma_{1\to 2}^{b)}), (\Omega_2^{c)}, \sigma_{1\to 2}^{c)}), (\Omega_2^{d)}, \sigma_{1\to 2}^{d)})\}$ from Example 6.14 where the atom $q_n(L_1) \in \Omega_1$ shared no variable with the others two and gets needlessly entangled with

their unfolding, thus creating four redundant cases a), b), c), d). To improve on this, we separate Ω into independent sub-problems and solve them individually. We therefore need a notion of independence that would allow for this separate solving,

Definition 6.25 (Independent sets of atoms)**.** Let Ω_a, Ω_b be two sets of atoms. Ω_a and Ω_b are said to be *independent*, written $\Omega_a || \Omega_b$, if for any first-order model $\mathcal{M}$ and assignments $\lambda_a \in Assigns(\mathcal{M}, \Omega_a)$ and $\lambda_b \in Assigns(\mathcal{M}, \Omega_b)$, we have that $\lambda_{ab} = \lambda_a \cup \lambda_b$ is well-defined (i.e. is a function) and that $\lambda_{ab} \in Assigns(\mathcal{M}, \Omega_a \cup \Omega_b)$.

Finding a most precise partitioning of Ω into independent sub-problems is not immediate. Indeed, two sets of atoms Ω_a and Ω_b which share variables may not really depend on each other. E.g. if they share only one variable X such that, for every model $\mathcal{M}$ and assignments $\lambda_a \in Assigns(\mathcal{M}, \Omega_a)$ and $\lambda_b \in Assigns(\mathcal{M}, \Omega_b)$, we have $\lambda_a(X) = \lambda_b(X)$, then $\lambda_{ab} = \lambda_a \cup \lambda_b$ is an assignment and we have $\lambda_{ab} \in Assigns(\mathcal{M}, \Omega_a \cup \Omega_b)$. However, not sharing any variable is a safe, easy-to-compute, and efficient approximation of independence.

Proposition 6.26 (Approximation of independent sets of atoms)**.** *Let Ω_a, Ω_b be two sets of atoms. Then*

$$Vars(\Omega_a) \cap Vars(\Omega_b) = \emptyset \Rightarrow \Omega_a || \Omega_b$$

Definition 6.27 (Split(Ω))**.** Let Ω be a set of atoms. We define a dependency relation $D \subseteq \Omega \times \Omega$ as $D(\omega_1, \omega_2) \doteq Vars(\omega_1) \cap Vars(\omega_2) \neq \emptyset$. Since D is symmetric, its reflexive and transitive closure D^* is an equivalence relation. This equivalence relation is such that two distinct equivalence classes do not share any variable. We define the function Split(Ω) to return the equivalence classes of D^*, which are independent subsets of atoms of Ω.

The $Inhabits_{\mathcal{A}}$ procedure can use this splitting at each step. Each set of atom Ω_i is split into Split(Ω_i) and each subset of Split(Ω_i) is independently solved, in order to lose some unnecessary combinatorics. A precise algorithm is given in Definition 6.29.

Example 6.28 (Split(Ω_1))**.** During the execution of the $Inhabits_{\mathcal{A}_{le}}$ from Example 6.14, we had $\Omega_1 = \{q_n(L_1), q_{\text{Len}}(L_2, N_1), q_{\text{Even}}(N_1)\}$. Splitting Ω_1 yields two independent subsets: $\text{Split}(\Omega_1) = \{\{q_n(L_1)\}, \{q_{\text{Len}}(L_2, N_1), q_{\text{Even}}(N_1)\}\}$. The solving of $\{q_n(L_1)\}$ can now be made independent of that of $\{q_{\text{Len}}(L_2, N_1), q_{\text{Even}}(N_1)\}$.

$Inhabits_{\mathcal{A}}$ final algorithm

Here is the algorithm $Inhabits_{\mathcal{A}}$ that is extended with the pruning of the search space and the splitting of independent atoms. Because independent atoms are split, the non-deterministic algorithm can no longer keep one set of atoms Ω_i and must use sets of sets of atoms. To avoid this additional difficulty, we prefer to write $Inhabits_{\mathcal{A}}$ recursively. An additional argument h (for *h*istory) has been added and corresponds to the list of sets of atoms that led to the current call of $Inhabits_{\mathcal{A}}(\Omega, h)$.

Definition 6.29 ($Inhabits_{\mathcal{A}}(\Omega, h)$)**.** Let $\mathcal{A}$ be a convoluted automaton and Ω a set of atoms. $Inhabits_{\mathcal{A}}(\Omega, h)$ is a non-deterministic recursive algorithm defined as:

1. `If` $\Omega = \emptyset$, `then return` $Some(\emptyset)$.
2. `If there exists` $\Omega_{old} \in h$ `such that` $\Omega_{old} \leq \Omega$ `then return` $None$
3. `Let` $h' = \Omega :: h$
4. `CHOOSE` $(\Omega', \sigma_\delta) \in unfolds_{\mathcal{A}}(\Omega)$
   `If no such choice can be made (when` $unfolds_{\mathcal{A}}(\Omega) = \emptyset$`), then return` $None$
5. `Let` $\{\Omega'_1, \ldots, \Omega'_k\} = \text{Split}(\Omega')$
6. `If there exists` $\Omega'_j \in \text{Split}(\Omega')$ `such that` $Inhabits_{\mathcal{A}}(\Omega'_j, h') = None$, `then return` $None$. `Otherwise, for all` $j \in [1 \ldots k]$, `let` $Some(\sigma_j) = Inhabits_{\mathcal{A}}(\Omega'_j, h')$
7. `Return` $Some(\ (\sigma'_1 \cup \ldots \cup \sigma'_k) \circ \sigma_\delta\ )$.

This algorithm is the same as Definition 6.13 except for

- The recursive formulation, which therefore builds the resulting substitution from the leaves up to the initial call instead of iteratively from the initial call to the leaves. Notations σ_i and Ω_i no longer have meaning with this implementation.
- The splitting of atoms, which leads to having multiple recursive calls to $Inhabits_{\mathcal{A}}$, see instructions 5 and 6 and the join of solutions at instruction 7. This is safe and well-defined, as the splitting of atoms yields independent sets of atoms.
- The history tracking and the branch pruning of instruction 2 and 3, which preserves the refutational completeness and the minimal solutions.

This extended $Inhabits_{\mathcal{A}}$ algorithm, or more precisely its breadth-first search version, is the one the *Teacher* procedure use.

6.1.2 *Teacher* definition and theorems

We now define the *Teacher* procedure. Recall that the *Teacher* takes as input a finite set of clauses Γ and an automaton $\mathcal{A}$. If $\mathcal{A} \models \Gamma$ then $\mathit{Teacher}(\mathcal{A}, \Gamma)$ should return *None*. Otherwise, if $\mathcal{A} \not\models \Gamma$, then $\mathit{Teacher}(\mathcal{A}, \Gamma)$ should return $\mathit{Some}(\widehat{\varphi})$ where $\widehat{\varphi}$ is a *counterexample* to $\mathcal{A} \models \Gamma$, i.e. a ground instance of some clause $\varphi \in \Gamma$ such that $\mathcal{A} \not\models \widehat{\varphi}$. Because the substitutions that $\mathit{Inhabits}_{\mathcal{A}}$ return do not necessarily yield ground counterexamples, we need a way to convert a non-ground formula into a ground formula.

Definition 6.30 (Smallest grounding)**.** The smallest grounding of a variable X, written $\mathit{Grd}(X)$, is a substitution σ such that: (a) $\sigma(X)$ is ground (b) for any other substitution σ' such that $\sigma'(X)$ is ground, $ht(\sigma(X)) \leq ht(\sigma'(X))$.

Note that a term may have several smallest grounding. We exploit the minimality of groundings in the proof of Lemma 6.33. Smallest grounding extends to pattern, tuple, atom, set of atom, clause, and is possible because we only consider inhabited algebraic datatypes.

Definition 6.31 (Teacher)**.** Let $\mathcal{A}$ be an automaton and Γ be a finite set of clauses. $\mathit{Teacher}(\mathcal{A}, \Gamma)$ is defined as

1. In parallel, run one instance $\mathit{Inhabits}^{det}_{\mathcal{A}}(\Omega_{\overline{\varphi}}, [])$ for each formula $\varphi \in \Gamma$ and enforce the depth of recursive calls (the length of the history parameter h) to be the same among instances of $\mathit{Inhabits}^{det}_{\mathcal{A}}$ that have not yet terminated.
2. If some instance $\mathit{Inhabits}^{det}_{\mathcal{A}}(\Omega_{\overline{\varphi}})$ returns $\mathit{Some}(\sigma)$, then let $\sigma' = \mathit{Grd}(\sigma(\Omega_{\overline{\varphi}}))$, $\widehat{\varphi} = \sigma'(\sigma(\varphi))$, and return $\mathit{Some}(\widehat{\varphi})$.

   If all instances have returned *None*, then return *None*.

We now have the following theorem as a rather direct consequence of Theorem 6.15. Note that completeness is still relative, as no termination is claimed in the case of $\mathcal{A} \models \Gamma$.

Theorem 6.32 (*Teacher* correctness and relative completeness)**.** *Let $\mathcal{A}$ be a convoluted automaton and Γ be a set of clauses. Then*

- ***Correctness:*** *If $\mathit{Teacher}(\mathcal{A}, \Gamma)$ terminates with $\mathit{Some}(\widehat{\varphi})$, then $\widehat{\varphi}$ is a ground instance of some formula $\varphi \in \Gamma$ and $\mathcal{A} \not\models \widehat{\varphi}$, so $\mathcal{A} \not\models \Gamma$.*
- ***Relative completeness:*** *If $\mathcal{A} \not\models \Gamma$ then $\mathit{Teacher}(\mathcal{A}, \Gamma)$ terminates with $\mathit{Some}(\widehat{\varphi})$ for some ground formula $\widehat{\varphi}$.*

Proof.

Both properties can be proven by the following equivalence chain:

$Teacher(\mathcal{A}, \Gamma)$ terminates with $Some(_)$
iff (by the *Teacher* procedure)
there exists $\varphi \in \Gamma$ and a substitution σ s.t. $Inhabits^{det}_{\mathcal{A}}(\Omega_{\overline{\varphi}})$ terminates with $Some(\sigma)$
iff (By Theorem 6.15)
there exists $\varphi \in \Gamma$ and a substitution σ such that $\mathcal{A} \models \sigma(\Omega_{\overline{\varphi}})$
iff (By Proposition 6.3)
there exists $\varphi \in \Gamma$ such that $\mathcal{A} \not\models \varphi$
iff
$\mathcal{A} \not\models \Gamma$.

The assertion that if $Teacher(\mathcal{A}, \Gamma) = Some(\widehat{\varphi})$ then $\widehat{\varphi}$ is a ground instance of some formula $\varphi \in \Gamma$ immediately comes from the definition of *Teacher*, and that $\mathcal{A} \not\models \widehat{\varphi}$ from Proposition 6.3. □

The height bound on solutions returned by $Inhabits_{\mathcal{A}}$ (Lemma 6.16) can be lifted to the *Teacher* procedure by the following Lemma 6.33.

Lemma 6.33 (*Teacher* height boundedness)**.** *If $Teacher(\mathcal{A}, \Gamma) = Some(\widehat{\varphi})$, then for any other counterexample $\widehat{\varphi}'$ of $\mathcal{A} \models \Gamma$, $ht(\widehat{\varphi}) \leq ht(\widehat{\varphi}') + dh$ with dh a constant defined from Γ.*

This *Teacher* definition has been inspired by Haudebourg's previous work on a similar Learner/Teacher procedure [18]. The main differences in the *Teacher* procedure are that only our version includes: (a) A more modular definition, allowing to easily extend the procedure to SHoCs instead of tree automata (b) Theorems and proofs for the correctness, relative completeness, and height boundedness (c) The pruning, allowing to terminate on much more instances (d) The splitting into independent atoms, yielding a much faster procedure (e) Defining the teacher as checking every clause in parallel, which is necessary for the relative completeness theorem. (f) Defining the *Teacher* on any form of first-order clauses, not only Horn clauses. (g) The canonization and simplification of sets of atoms, together with the memoisation approach (described in Section 9.1.1).

6.2 Learner

The learner's procedure is responsible for inferring a model, here a convoluted automaton, from examples or finding a contradiction in them. It takes as input a finite set $\widehat{\Gamma}$ of ground clauses and returns *None* if $\widehat{\Gamma}$ is contradictory and $Some(\mathcal{A})$ otherwise, with $\mathcal{A}$ being an automaton satisfying $\widehat{\Gamma}$. This input set of examples is made from one counterexample to each previous automaton proposed by the learner.
The method described here is, this time, very similar to that of Haudebourg [18]. The main differences are that we have proofs for the correctness, completeness, and output's minimality of the procedure. Being defined on padding-free and final-state-free automata do not make much of a theoretical difference, nor does being defined on generic first-order clauses instead of Horn clauses. To learn an automaton, this procedure first generates an *initial automaton* $\mathcal{W}_{\widehat{\Gamma}}$. This automaton recognizes every tuple of terms $\overrightarrow{t}$ for which there exists an atom $R(\overrightarrow{t})$ in the set of examples $\widehat{\Gamma}$ in different states. The procedure then produces a simple set of first-order constraints $E_{\widehat{\Gamma}}$ that are meant to be given to a finite-model finder. These constraints either are unsatisfiable, in which case $\widehat{\Gamma}$ is unsatisfiable too, or admit a model. Finally, in case the constraints admit a (smallest) model, the transitions of the initial automaton $\mathcal{W}_{\widehat{\Gamma}}$ can be rewritten using this model in order to make states recognize more terms, making regularity appear, and thus propose a meaningful automaton $\mathcal{A}$ such that $\mathcal{A} \models \widehat{\Gamma}$ in the hope that $\mathcal{A} \models \Gamma$, with Γ the set of clauses on which the whole learner/teacher procedure was started on.

Definition 6.34 (Initial automaton)**.** The initial automaton $\mathcal{W}_{\widehat{\Gamma}}$ of a given finite set of ground clauses $\widehat{\Gamma}$ is the smallest automaton (up to state renaming) recognizing exactly the terms appearing in $\widehat{\Gamma}$ in a different state for each. That is, for any atom $R(\overrightarrow{t})$ of any $\varphi \in \widehat{\Gamma}$, there exists a state q in $\mathcal{W}_{\widehat{\Gamma}}$ such that $\mathcal{R}(\mathcal{W}_{\widehat{\Gamma}}, q) = \{\overrightarrow{t}\}$.

This initial automaton construction is carried out by classical automata algorithms [7].

Example 6.35 (Initial automaton for Len)**.** Suppose the Learner/Teacher procedure was started on the clauses defining the Len relation (Definition 4.3) using the right-convolution Ⓡ. We observe the procedure after the learner and teacher already had two exchanges, so the learner has accumulated the two previous teacher's output, namely the following examples: $\widehat{\Gamma}_{ex} = \{\text{Len}(nil, z),\ \text{Len}(cons(z, nil), s(z)) \Leftarrow \text{Len}(nil, z)\}$. The corresponding initial automaton is $\mathcal{W}_{\widehat{\Gamma}_{ex}} = (Q, \Delta)$ with $Q = \{q_{l_0}, q_{l_1}, q_z\}$ and $\Delta = \{\langle nil, z\rangle() \to q_{l_0},\ \langle cons, s\rangle(q_z, q_{l_0}) \to q_{l_1},\ \langle z\rangle() \to q_z\}$.

The relation that these states denote are: $\mathcal{R}(\mathcal{W}_{\widehat{\Gamma}_{ex}}, q_{l_0}) = \{(nil, z)\}$, $\mathcal{R}(\mathcal{W}_{\widehat{\Gamma}_{ex}}, q_z) = \{(z)\}$, and $\mathcal{R}(\mathcal{W}_{\widehat{\Gamma}_{ex}}, q_{l_1}) = \{(cons(z, nil), s(z))\}$.
Note that the state q_z recognizes the term $\langle z \rangle$ which does not appear (as root) in $\widehat{\Gamma}$ but is necessary to recognize $(cons(z, nil), s(z))$ in q_{l_1}.

The initial automaton $\mathcal{W}_{\widehat{\Gamma}}$ can then be generalised by merging states into equivalence classes, each class representing a new state. Moreover, each relation that must be learned, i.e. relations appearing in $\widehat{\Gamma}$, must be assigned a subset of these equivalence classes for representing a Herbrand model (see Definition 5.29). Merging states of $\mathcal{W}_{\widehat{\Gamma}}$ leads to additional terms being recognized and makes regularity appear. We search for a merging that minimises the number of states of $\mathcal{W}_{\widehat{\Gamma}}$ while ensuring that the resulting automaton satisfies $\widehat{\Gamma}$.

Definition 6.36 (State merging problem)**.** Given a finite set of ground clauses $\widehat{\Gamma}$, we define a set of constraints on a first-order signature of constants and unary predicates. The first-order signature contains one constant q for each state q of $\mathcal{W}_{\widehat{\Gamma}}$ and one unary predicate R for each relation R that appears in $\widehat{\Gamma}$.
The constraints given to the finite-model finder are $E_{\widehat{\Gamma}} = C_\tau \cup C_f \cup C_{det}$, defined as follows.

- C_τ is a set of disequalities ensuring that merged states all have the same type:
  $C_\tau = \{(q_1 \neq q_2) \mid q_1, q_2 \in Q(\mathcal{W}) \wedge \tau(q_1) \neq \tau(q_2)\}$
- C_f mirrors $\widehat{\Gamma}$ but, instead of relations being applied to a tuple of terms $\overrightarrow{t}$, relations are applied to the state q that recognizes this tuple of terms in $\mathcal{W}_{\widehat{\Gamma}}$.
  $C_f = \{R_1(q_1) \vee \ldots \vee R_k(q_k) \Leftarrow R_{k+1}(q_{k+1}) \wedge \ldots \wedge R_n(q_n) \mid$
  $R_1(\overrightarrow{t_1}) \vee \ldots \vee R_k(\overrightarrow{t_k}) \Leftarrow R_{k+1}(\overrightarrow{t_{k+1}}) \wedge \ldots \wedge R_n(\overrightarrow{t_n}) \in \widehat{\Gamma}\}$
  with, for each $i \in [1 \ldots n]$, q_i is the state of $\mathcal{W}_{\widehat{\Gamma}}$ such that $\mathcal{R}(\mathcal{W}_{\widehat{\Gamma}}, q_i) = \{\overrightarrow{t_i}\}$.
- C_{det} is a set of equality constraints ensuring that merging states still results in a deterministic automaton:
  $C_{det} = \{(q = q') \Leftarrow (q_1 = q'_1) \wedge \ldots \wedge (q_k = q'_k) \mid$
  $\overrightarrow{f}(q_1, \ldots, q_k) \to q \in \Delta(\mathcal{W}_{\widehat{\Gamma}}) \wedge \overrightarrow{f}(q'_1, \ldots, q'_k) \to q' \in \Delta(\mathcal{W}_{\widehat{\Gamma}})\}$

A (minimal) solution to a state merging problem $E_{\widehat{\Gamma}}$ can be computed by a finite model finder and is the data of (i) a finite domain D that represents a new set of states (ii) a function $merge : Q(\mathcal{W}_{\widehat{\Gamma}}) \to D$ that assigns an element of D to each state q of $\mathcal{W}_{\widehat{\Gamma}}$ (iii) a function that assigns a subset of D to each predicate R, i.e. a relation $\mu \subseteq \mathcal{R}(\widehat{\Gamma}) \times$

D. This relation μ serves to indicate which relation is represented by which states, as discussed in Definition 5.29. A solution is written as a triple $(D, merge, \mu)$.
The transitions of the initial automaton can then be rewritten by replacing every state q by $merge(q)$.

Definition 6.37 (Generalisation of the initial automaton)**.** Given a solution $(D, merge, \mu)$ to the state merging problem $E_{\widehat{\Gamma}}$, we generalise the initial automaton $\mathcal{W}_{\widehat{\Gamma}}$ by applying the *merge* function to every one of its states: $merge(\mathcal{W}_{\widehat{\Gamma}}) = (D, \Delta)$ with $\Delta = \{\vec{f}(merge(q_1), \ldots, merge(q_k)) \rightarrow merge(q) \mid \vec{f}(q_1, \ldots, q_k) \rightarrow q \in \Delta(\mathcal{W}_{\widehat{\Gamma}})\}$.
Every relation R appearing in $\widehat{\Gamma}$ is represented by the set of states $\mu(R)$ in $merge(\mathcal{W}_{\widehat{\Gamma}})$.

A given minimisation problem does not necessarily have a solution, as it will be the case for a contradictory set of ground constraints $\widehat{\Gamma}$.

Definition 6.38 (Minimise)**.** Given a finite set of ground clauses $\widehat{\Gamma}$, the *Minimise* function is defined as:

$$Minimise(\widehat{\Gamma}) = \begin{cases} Some(merge(\mathcal{W}_{\widehat{\Gamma}}), \mu) & \text{if } (D, merge, \mu) \text{ is a minimal solution to } E_{\widehat{\Gamma}} \\ & \text{w.r.t the domain size } |D|; \\ None & \text{otherwise (if } E_{\widehat{\Gamma}} \text{ has no model).} \end{cases}$$

Example 6.39 (Generalisation of the initial automaton)**.** Continuing Example 6.35, we now define the minimisation problem $E_{\widehat{\Gamma}_{ex}}$.
The signature has constant symbols $\{q_{l_0},\ q_{l_1},\ q_z\}$ and predicate symbols $\{\text{Len}\}$.
The constraints $C_\tau = \{q_{l_0} \neq q_z,\ q_{l_1} \neq q_z\}$ are stating that q_z cannot be merged with q_{l_0} nor q_{l_1} because they are not of the same type.
The constraints $C_f = \{\text{Len}(q_{l_0}),\ \text{Len}(q_{l_1}) \Leftarrow \text{Len}(q_{l_0})\}$ are stating that Len is denoted by both the state q_{l_0} and, following the implication, q_{l_1}.
The constraints $C_{det} = \emptyset$ are empty, as no two transitions of $\mathcal{W}_{\widehat{\Gamma}_{ex}}$ share the same head symbols.
We have $E_{\widehat{\Gamma}_{ex}} = C_\tau \cup C_f \cup C_{det}$.
The smallest first-order model of these constraints $E_{\widehat{\Gamma}_{ex}}$ is a two-elements set $D = \{q_l, q_n\}$ with $merge(q_{l_0}) = merge(q_{l_1}) = q_l$, $merge(q_z) = q_n$, and $\mu(\text{Len}) = \{q_l\}$. Let $\mathcal{A} = merge(\mathcal{W}_{\widehat{\Gamma}_{ex}})$, so $Minimise(\widehat{\Gamma}_{ex}) = Some(\mathcal{A}, \mu)$.
The generalized automaton $\mathcal{A}$ has two states $\{q_l, q_n\}$ and three transitions $\{\langle nil, z\rangle() \rightarrow q_l,\ \langle z\rangle() \rightarrow q_n,\ \langle cons, s\rangle(q_n, q_l) \rightarrow q_l\}$ and defines (using μ) an almost-correct relation

for Len: the set of pairs (l, n) of a list of zeros together with its size. The only missing rule is $\langle s \rangle(q_n) \to q_n$, which would be added by the learner in the learning step that follows because of the constraint it receives: $\text{Len}(cons(s(z), nil), s(z)) \Leftarrow \text{Len}(nil, z)$.

We now prove the correctness of this approach together with a minimality property. We begin by Lemma 6.40 which asserts a completeness result of this approach. This completeness roughly expresses that any (complete and deterministic) automaton can be obtained by carefully merging the initial automaton.

Lemma 6.40 (State merging completeness)**.** *Let $\widehat{\Gamma}$ be a satisfiable finite set of ground clauses and $\mathcal{A}$ a deterministic and complete automaton with a relation $\mu \subseteq \mathcal{R}(\widehat{\Gamma}) \times Q(\mathcal{A})$ such that $(\mathcal{A}, \mu) \models \widehat{\Gamma}$. Then there exists a function merge $: Q(\mathcal{W}_{\widehat{\Gamma}}) \to Q(\mathcal{A})$ such that $(Q(\mathcal{A}), merge, \mu)$ is a solution to the problem $E_{\widehat{\Gamma}} = C_\tau \cup C_f \cup C_{det}$.*

Proof. Let us first construct the solution $(Q(\mathcal{A}), merge, \mu)$, i.e. define the function *merge*: Let q' be any state of $\mathcal{W}_{\widehat{\Gamma}}$ and let $\overrightarrow{t}$ be the tuple of terms such that $\mathcal{R}(\mathcal{W}_{\widehat{\Gamma}}, q') = \{\overrightarrow{t}\}$. Let q be *the* (by completeness and determinism of $\mathcal{A}$) state such that $\overrightarrow{t} \in \mathcal{R}(\mathcal{A}, q)$. We now define $merge(q')$ to be q.
Let us now prove that $(Q(\mathcal{A}), merge, \mu)$ satisfies $E_{\widehat{\Gamma}}$.

- Only same-type states have been merged, so it immediately satisfies C_τ.
- Let $\varphi_q = R_1(q_1) \vee \ldots \vee R_k(q_k) \Leftarrow R_{k+1}(q_{k+1}) \wedge \ldots \wedge R_n(q_n)$ be a constraint from C_f and let $\varphi = R_1(\overrightarrow{t_1}) \vee \ldots \vee R_k(\overrightarrow{t_k}) \Leftarrow R_{k+1}(\overrightarrow{t_{k+1}}) \wedge \ldots \wedge R_n(\overrightarrow{t_n})$ be the clause that φ_q mirrors.

  Verifying that φ_q is satisfied by $(Q(\mathcal{A}), merge, \mu)$ consists of verifying that $merge(q_1) \in \mu(R_1) \vee \ldots \vee merge(q_k) \in \mu(R_k) \Leftarrow merge(q_{k+1}) \in \mu(R_{k+1}) \wedge \ldots \wedge merge(q_n) \in \mu(R_n)$.

  The construction of the *merge* function gives that, for any $i \in [1 \ldots n]$, $merge(q_i)$ is the one and only (by determinism of $\mathcal{A}$) state such that $\overrightarrow{t_i} \in \mathcal{R}(\mathcal{A}, merge(q_i))$.

  Verifying that φ_q is satisfied by $(Q(\mathcal{A}), merge, \mu)$ is therefore equivalent to checking $(\mathcal{A}, \mu) \models R_1(\overrightarrow{t_1}) \vee \ldots \vee R_k(\overrightarrow{t_k}) \Leftarrow R_{k+1}(\overrightarrow{t_{k+1}}) \wedge \ldots \wedge R_n(\overrightarrow{t_n})$, which is $(\mathcal{A}, \mu) \models \varphi$.

  Because $(\mathcal{A}, \mu) \models \widehat{\Gamma}$, we have $(\mathcal{A}, \mu) \models \varphi$ and thus φ_q is satisfied by $(Q(\mathcal{A}), merge, \mu)$.
- Let $(q = q') \Leftarrow (q_1 = q'_1) \wedge \ldots \wedge (q_k = q'_k)$ be a constraint from C_{det}. This constraint comes from having two transitions $\overrightarrow{f}(q_1, \ldots, q_k) \to q$ and $\overrightarrow{f}(q'_1, \ldots, q'_k) \to q'$ in $\mathcal{W}_{\widehat{\Gamma}}$ and amounts to verify that $merge(q) = merge(q') \Leftarrow merge(q_1) =$

$merge(q'_1) \wedge \ldots \wedge merge(q_k) = merge(q'_k)$. Seeing that $\overrightarrow{f}(merge(q_1), \ldots, merge(q_k)) \rightarrow merge(q)$ and $\overrightarrow{f}(merge(q'_1), \ldots, merge(q'_k)) \rightarrow merge(q')$ are transitions of $\mathcal{A}$, determinism of $\mathcal{A}$ proves the constraint.

□

Theorem 6.41 (Correctness)**.** *Let $\widehat{\Gamma}$ be a finite set of ground clauses. Then $\widehat{\Gamma}$ is contradictory iff $Minimise(\widehat{\Gamma}) = None$. Moreover, if $Minimise(\widehat{\Gamma}) = Some(\mathcal{A}, \mu)$, then $(\mathcal{A}, \mu) \models \widehat{\Gamma}$.*

Proof.

- Suppose $Minimise(\widehat{\Gamma}) = Some(\mathcal{A}, \mu)$, so $\mathcal{A} = merge(\mathcal{W}_{\widehat{\Gamma}})$ for $(D, merge, \mu)$ a minimal solution to the constraints $E_{\widehat{\Gamma}} = C_\tau \cup C_f \cup C_{det}$.

  Let $\varphi = R_1(\overrightarrow{t_1}) \vee \ldots \vee R_k(\overrightarrow{t_k}) \Leftarrow R_{k+1}(\overrightarrow{t_{k+1}}) \wedge \ldots \wedge R_n(\overrightarrow{t_n})$ be a clause of $\widehat{\Gamma}$ and let $\varphi_q = R_1(q_1) \vee \ldots \vee R_k(q_k) \Leftarrow R_{k+1}(q_{k+1}) \wedge \ldots \wedge R_n(q_n)$ be the clause mirroring φ in C_f. The constraint φ_q being satisfied by $(D, merge, \mu)$ gives $merge(q_1) \in \mu(R_1) \vee \ldots \vee merge(q_k) \in \mu(R_k) \Leftarrow merge(q_{k+1}) \in \mu(R_{k+1}) \wedge \ldots \wedge merge(q_n) \in \mu(R_n)$.

  Note that merging states only augments the language of each state, i.e. for any state $q \in Q(\mathcal{W}_{\widehat{\Gamma}})$, $\mathcal{R}(\mathcal{W}_{\widehat{\Gamma}}, q) \subseteq \mathcal{R}(merge(\mathcal{W}_{\widehat{\Gamma}}), merge(q))$. That is, for all $i \in [1 \ldots n]$, $\overrightarrow{t_i} \in \mathcal{R}(\mathcal{A}, merge(q_i))$. Because $\mathcal{A}$ is deterministic and that for all $i \in [1 \ldots n]$, $\overrightarrow{t_i} \in \mathcal{R}(\mathcal{A}, merge(q_i))$, we have that $(\mathcal{A}, \mu) \models R_i(\overrightarrow{t_i}) \iff merge(q_i) \in \mu(R_i)$. Therefore $(\mathcal{A}, \mu) \models \widehat{\Gamma}$.
- Suppose $Minimise(\widehat{\Gamma}) = Some(\mathcal{A}, \mu)$. Because $(\mathcal{A}, \mu) \models \widehat{\Gamma}$, then $\widehat{\Gamma}$ is satisfiable.
- Suppose that $\widehat{\Gamma}$ is satisfiable. Because $\widehat{\Gamma}$ is finite and ground, it admits a Herbrand model whose relations have a finite interpretation. Any finite Herbrand model can be represented by a (complete and deterministic) convoluted automaton $\mathcal{A}$ with some μ to make the link between states and relations, i.e. $(\mathcal{A}, \mu) \models \widehat{\Gamma}$. By Lemma 6.40, there exists a solution $(Q(\mathcal{A}), merge, \mu)$ of the set of constraints $E_{\widehat{\Gamma}}$. Therefore $Minimise(E_{\widehat{\Gamma}})$ returns $Some(\mathcal{A}', \mu)$ for some convoluted automaton $\mathcal{A}'$.

□

Theorem 6.42 (Minimality)**.** *Let $\widehat{\Gamma}$ be a finite set of ground clauses and suppose $Minimise(\widehat{\Gamma}) = Some(\mathcal{A}, \mu)$. Then, for any complete and deterministic automaton $\mathcal{A}'$ such that $\mathcal{A}' \models \widehat{\Gamma}$, we have $|\mathcal{A}| \leq |\mathcal{A}'|$.*

Proof. Let $\mu' = \{(R, q_R) \mid R \in \mathcal{R}(\widehat{\Gamma})\}$ be the implicit link between states and relations in $\mathcal{A}'$, and thus because $\mathcal{A}' \models \widehat{\Gamma}$ we have $(\mathcal{A}', \mu') \models \widehat{\Gamma}$. From Lemma 6.40, it is possible to construct a solution $(Q(\mathcal{A}'), merge, \mu')$ from $(\mathcal{A}', \mu')$ for the set of constraints $E_{\widehat{\Gamma}}$. Because the automaton defined from this solution has states $Q(\mathcal{A}')$, it has the same size as $\mathcal{A}'$. Moreover, *Minimise* returns the smallest solution in the number of states (cardinality of the domain), so $|\mathcal{A}| \leq |\mathcal{A}'|$. □

The output of the *Minimise* function is a tuple of an automaton $\mathcal{A}$ and the link μ between relations and states. In the following, we do not want to have to carry this link μ so as to simplify manipulation, i.e., we want to have exactly one state to represent one relation. From an automaton $\mathcal{A}$ and a function μ, we can complete $\mathcal{A}$ into $\mathcal{A}'$ by adding the following ϵ-transitions: $\Delta^\epsilon = \{q \to q_R \mid R \in dom(\mu) \wedge q \in \mu(R)\}$. These ϵ-transitions are then replaced by non-ϵ-transitions using classical tree automata procedures [7]. We write this extended automaton $incorporate(\mathcal{A}, \mu)$, which denotes the same Herbrand model as $(\mathcal{A}, \mu)$.

Definition 6.43 (Learner)**.** Let $\widehat{\Gamma}$ be a finite set of ground clauses. $Learner(\widehat{\Gamma})$ is defined as:

$$Learner(\widehat{\Gamma}) = \begin{cases} Some(incorporate(\mathcal{A}, \mu)) & \text{if } Minimise(E) = Some(\mathcal{A}, \mu) \\ None & \text{otherwise} \end{cases}$$

Example 6.44 ($Learner(\widehat{\Gamma}_{ex})$)**.** Continuing Example 6.39, the automaton $\mathcal{A}$ is extended with the ϵ-transitions $\{q_l \to q_{\text{Len}}\}$. The resulting automaton, after removing the ϵ-transition, has one new state q_{Len} and two new transitions: $\langle nil, z\rangle() \to q_{\text{Len}}$ and $\langle cons, s\rangle(q_z, q_l) \to q_{\text{Len}}$.

6.3 Assembling the learner and teacher: *Sat* theorems

The Learner/Teacher procedure is generically defined in Section 4.3. With the convoluted automata defined in Chapter 5, the teacher defined in Section 6.1, and the learner defined in Section 6.2, we finally have a complete instantiation of the *Sat* procedure. We now state the two main theorems of *Sat*. This first theorem states that false properties are always (theoretically) found.

Theorem 6.45 (Refutational completeness)**.** *Let* Γ *a finite set of contradictory clauses. Then* $Sat(\Gamma)$ *terminates with "Disproved:* $\widehat{\Gamma}$*" with* $\widehat{\Gamma}$ *a finite subset of the ground instantiations of* Γ.

Proof. Let $Grd(\Gamma)$ be the set of ground instances of Γ. Because we consider the theory of algebraic datatypes and because Γ is contradictory, $Grd(\Gamma)$ is contradictory too. By first-order logic compactness theorem, we know that a set of formulas is contradictory iff there exists a finite subset of contradictory formulas, so let $F \subseteq Grd(\Gamma)$ be such a set. Let $H = \max_{\widehat{\varphi} \in F}(ht(\widehat{\varphi}))$ be the height of a highest clause in F. Let dh be the value defined from Γ as mentioned in the proof of Lemma 6.33. Let $Cl(F) = \{\widehat{\varphi} \mid \widehat{\varphi} \in Grd(\Gamma) \wedge ht(\widehat{\varphi}) \leq H + dh\}$. $Cl(F)$ is finite and also contradictory, as $F \subseteq Cl(F)$.
For any step i of $Sat(\Gamma)$ such that $Learner(\Gamma_i) = Some(\mathcal{A}_i)$, we have $\mathcal{A}_i \not\models F$ and so let h be the height of a smallest counterexample of $\mathcal{A}_i \models F$. By Theorem 8.6, $Teacher(\mathcal{A}_i, \Gamma) = Some(\widehat{\varphi})$ with $\widehat{\varphi}$ a ground instance of some formula $\varphi \in \Gamma$ such that $ht(\widehat{\varphi}) \leq h + dh$. Thus $\widehat{\varphi}_i \in Cl(F)$.
Because every counterexample $\widehat{\varphi}_i$ that the teacher outputs is contained in $Cl(F)$ and of Lemma 4.13, the ground clauses $\widehat{\Gamma}_{|Cl(F)|}$ accumulated by the learner at step $|Cl(F)|$ would necessarily be such that $F \subseteq \widehat{\Gamma}_{|Cl(F)|}$. Therefore there exists a step $i_{unsat} \in [0 \dots |Cl(F)|]$ such that $\widehat{\Gamma}_{i_{unsat}}$ is unsatisfiable, and the $Sat(\Gamma)$ procedure stops at this step with "Disproved: $\widehat{\Gamma}_{i_{unsat}}$". ☐

For this second theorem, recall that satisfiability of Γ is undecidable, so Sat can not be complete. $Sat(\Gamma)$ may not terminate for two reasons. The first is that Γ may be satisfiable but not admit a convoluted automaton, in which case Sat will run indefinitely. The second is that even if a satisfying $\mathcal{A}_i$ is proposed by the learner, the call to $Teacher(\mathcal{A}_i, \Gamma)$ may not terminate. However, we do have a relative completeness theorem.

Theorem 6.46 (Relative completeness)**.** *Let* Γ *be a finite set of clauses. If there exists an automaton* $\mathcal{A}$ *such that* $\mathcal{A} \models \Gamma$*, then there exists a step* $i \in \mathbb{N}$ *in which* $\mathcal{A}_i$*, the learner's automaton output at this step, is such that* $\mathcal{A}_i \models \Gamma$.

Proof. In this proof, we call *size* of a convoluted automaton the number of states that it defines. Suppose that Γ admits an automaton satisfying it, i.e. an automaton $\mathcal{A}$ such that $\mathcal{A} \models \Gamma$. Let N be the size of a *complete and deterministic* automaton $\mathcal{A}$ such that $\mathcal{A} \models \Gamma$. Any first-order model of Γ is also a model of $Grd(\Gamma)$ and of every subset of it, so, for any step i of the Sat procedure, $\mathcal{A} \models \widehat{\Gamma}_i$ (with $\widehat{\Gamma}_i$ the set of examples accumulated

at step i). Therefore any *smallest* deterministic and complete automaton $\mathcal{A}'$ such that $\mathcal{A}' \models \widehat{\Gamma}_i$ is smaller or equal than N. Because of the minimality of the *Minimise* function (Theorem 6.42), any automaton $\mathcal{A}'$ with $(\mathcal{A}', \mu') = \mathit{Minimise}(\widehat{\Gamma}_i)$ is of size smaller or equal than N. Then, with $k = |\mathcal{R}(\Gamma)|$ being the number of relations in Γ, we know that the automaton $\mathcal{A}''$ with $\mathit{Some}(\mathcal{A}'') = \mathit{Learner}(\widehat{\Gamma}_i)$ is of size smaller or equal than $N + k$. Because the number of automata of any fixed size is finite (modulo states names), we know that the number of automata (that do not satisfy Γ and) that are smaller or equal than $N + k$ is finite.

Because of Lemma 4.13, we know that the learner never outputs the same automaton twice. We also know that the learner always terminates and, as long as the learner proposes automata that do not satisfy Γ, the teacher terminates too (Theorem 6.32). Therefore the learner will end up proposing an automaton satisfying Γ. □

CHAPTER 7

SHALLOW HORN CLAUSES (SHOCS)

This section formally defines a SHoC as a syntactic restriction of a strict Horn clause. A set of SHoCs are used to represent a set of relations using inductive semantics, i.e. least fixpoint. The form of SHoCs make them particularly suited for our needs, as they allow to easily represent many relations that are defined in a simple recursive manner. Moreover, they enjoy many closure properties and can be inferred from examples. The restrictions of SHoCs are reminiscent of tree automata and one SHoC can be rather directly compared to one transition of a convoluted tree automaton, which is covered in Section 7.4. We begin by defining them in Section 7.1, introducing ϵ-clauses in Section 7.2, then defining their closure and decision properties in Section 7.3, and finally discussing their expressivity and comparing them with other formalisms in Section 7.4.

7.1 SHoCs definition

We begin by introducing some vocabulary for describing clauses, then define a SHoC and a SHoCs, and finally discuss the motivation for each particular (syntactic) restriction of Horn clauses to obtain SHoCs.

Definition 7.1 (Linearity, flatness, shallowness)**.** A pattern or atom $f(\vec{p})$ is

- *linear* if every variable appears at most once in it ;
- *flat* if $\vec{p}$ is a tuple of variables (note that a constant ($|f| = 0$) is flat) ;
- *shallow* if $\vec{p}$ is a tuple of flat patterns.

Example 7.2. For instance $f(X, g(Y))$ is linear but not flat (because $g(Y)$ is not a variable) nor shallow (because a variable X is not flat), $f(X, X)$ is flat but not linear, and $f(g(X), h(Y, Z))$ is linear and shallow.

Definition 7.3 (Shallow Horn Clause – SHoC)**.** A *Shallow Horn Clause* is a strict Horn clause such that:

(a) the head atom H is linear and shallow,

(b) all atoms of the body B are flat,

(c) the clause has no existential variables, i.e. $Vars(B) \subseteq Vars(H)$.

Definition 7.4 (Shallow Horn Clauses – SHoCs)**.** A SHoCs is a finite set of SHoC.

A SHoCs S define relations inductively using the smallest fixpoint semantics, in accordance with Proposition 2.23. We write $\mathcal{L}(S, R)$ for the relation denoted by R in S and overload the notation $S \models \varphi$ for $\mathcal{H} \models \varphi$ with $\mathcal{H}$ the Herbrand model denoted by (the least fixpoint of) S. For uniformity with convoluted automata, we also sometimes write $\mathcal{R}(S, R)$ for $\mathcal{L}(S, R)$.

Example 7.5 (SHoCs for the relation Len)**.** The Len relation, defined in Definition 4.3 using standard first-order semantics, can be represented by the following SHoCs that contains two clauses:

$$\text{Len}(nil, z) \qquad \text{Len}(cons(E, L), s(N)) \Leftarrow \text{Len}(L, N)$$

SHoC definition motivation The strictness of the SHoCs allows to see each of them as an induction rule, and a SHoCs as an inductively-defined set of relations, see Definition 2.23. Shallow heads and flat bodies in SHoCs work together and greatly limits the expressivity of such clauses ; they only allow relations that do not need to keep the value of some parameter while exploring the others recursively. The absence of existential variables is what allows to have a simple top-down membership procedure, as the procedure does not need to find a value (i.e. prove the *existence*) for any existential variable. However, the shallowness of the head can also yield bulkier SHoCs due to having multiple cases to handle. Note also that the body is not required to be linear. This allows to recognize relations that do not only need to build upon terms but also compare them (e.g. the Shal relation from Definition 4.3). However, this non-linearity may lead to an explosion of the effective complexity for the membership procedure. One last important detail is that there may be variables in the head that do not appear in the body. This allows SHoCs to enjoy a form of genericity which permits to represent relations in a more compact way than tree automata. This is illustrated by the definition of HeightLB by SHoCs in Example 7.28 and by left-convoluted automaton in Example

7.31. The automaton for HeightLB has 6 transitions whereas the SHoCs has 2 clauses. Now, consider the relation $\text{Len}(l, n)$ relating lists with their size. A SHoCs for this relation is given in Example 7.5. Similarly as the SHoCs for HeightLB, this SHoCs does not constrain the structure of elements in the list. Thus, this SHoCs is valid for lists of elements of any type. This results in a more generic and more compact representation than what tree automata can do for regular languages [30, 15, 19, 27] and for regular relations [29], where the complete structure of elements of the list has to be described explicitly by transitions.

Due to the particular form of SHoCs, we can have another notation for them, using positions instead of variables. This notation makes them harder to read but simplifies some definitions, so it is only used in these situations. Because the head of SHoCs is linear and shallow and every variable of the body appears in the head, a variable only indicates which argument of which function of the head it appears in. Each variable may then be replaced by a position $i \cdot j$, with i pointing at the head function and j pointing at the argument within the ith function.

Definition 7.6 (Projector notation for SHoCs). Any SHoC

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

can be equivalently written as

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

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

Example 7.7. The clause $\text{Shal}(node(T_1, E, T_2), s(N)) \Leftarrow \text{Shal}(T_1, N) \wedge \text{Shal}(T_2, N)$ can be written as $\text{Shal}(node, s) \Leftarrow \text{Shal}(1 \cdot 1, 2 \cdot 1) \wedge \text{Shal}(1 \cdot 3, 2 \cdot 1)$.

We write $\mathcal{R}(S) = \{R \mid \exists [R(\vec{p}) \Leftarrow B] \in S\}$ for the set of relation symbols that a SHoCs S defines.

7.2 ϵ-clauses and their elimination

In order to ease the definition of operations such as union or intersection of SHoCs, we introduce a new type of clauses, called ϵ-clauses. The ϵ-clauses are similar to ϵ-transitions of automata that define transitions between states without recognizing any

symbol. However, though ϵ-clauses simplify the definition of some operations on SHoCs, they are not valid SHoCs because their heads do not contain any function symbols, *i.e.*, they are flat. Thus, we need a procedure to generate an equivalent SHoCs without ϵ-clauses, similar to the removal of ϵ-transition in tree automata. For SHoCs, we call this procedure *Extend* and it is defined below. The *Extend* procedure is close to the *unfolding* rule from [28].

Definition 7.8 (ϵ-clause and ϵ-definition)**.** An ϵ-clause is a strict Horn clause $H \Leftarrow B$ such that a) H and B are flat and linear, and b) $Vars(H) = Vars(B)$.
Given a relation symbol R and a SHoCs S, an ϵ-definition (of R) for S is a set $\mathcal{C}_R^\epsilon$ of ϵ-clauses of the form $R(\overrightarrow{X}) \Leftarrow B$ and such that R does not appear in S nor in B.

Definition 7.9 ($Extend(S, \mathcal{C}_R^\epsilon)$)**.** Let S be a SHoCs and suppose that no two clauses in S share variables (or else they can be renamed, as they are bound to a (universal) quantifier). Let $\mathcal{C}_R^\epsilon$ be an ϵ-definition for S. Then

$$Extend(S, \mathcal{C}_R^\epsilon) = S \cup \bigcup_{\varphi \in \mathcal{C}_R^\epsilon} Ext_\varphi$$

where $Ext_{R(\overrightarrow{X}) \Leftarrow R_1(\overrightarrow{X_1}) \wedge \ldots \wedge R_k(\overrightarrow{X_k})}$ is the following set of SHoCs:

$$\left\{ \sigma(R(\overrightarrow{X}) \Leftarrow B_1 \cup \ldots \cup B_k) \,\middle|\, [\bigwedge_{j \in [1 \ldots k]} (R_j(\overrightarrow{p_j}) \Leftarrow B_j) \in S] \wedge [\sigma = MGU(U)] \right\}$$

with $U = \{\overrightarrow{X_j} \stackrel{?}{=} \overrightarrow{p_j} \mid j \in [1 \ldots k]\}$.

Example 7.10 (ϵ-definition and $Extend(S, \mathcal{C}_R^\epsilon)$)**.** Let S be a SHoCs with clauses:

$$R_1(h(X)) \Leftarrow A(X) \qquad R_1(h(X)) \Leftarrow B(X)$$
$$R_2(f(X), g(Y)) \Leftarrow C(X) \quad R_2(f(X), h(Y)) \Leftarrow D(X)$$

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

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

Lemma 7.11 (Elimination of ϵ-clauses preserves language)**.** *Let S be a SHoCs and $\mathcal{C}_R^\epsilon$ be an ϵ-definition for S. Let $S' = Extend(S, \mathcal{C}_R^\epsilon)$. Then*

$$[\mathcal{L}(S \cup \mathcal{C}_R^\epsilon, R) = \mathcal{L}(S', R)] \wedge [\forall R' \in \mathcal{R}(S),\ \mathcal{L}(S, R') = \mathcal{L}(S', R')].$$

Proof. Let $\vec{t} = (t_1, \ldots, t_n) \in \mathcal{T}(\Sigma)^n$ a tuple of n terms
First, note that $\mathcal{L}(S \cup \mathcal{C}^{\epsilon}_R, R)$ is well-defined, as every clause of $\mathcal{C}^{\epsilon}_R$ is a strict Horn clause. This corresponds to the intuitive notion of an ϵ-definition, with $(t_1, \ldots, t_n) \in \mathcal{L}(S \cup \mathcal{C}^{\epsilon}_R, R) \iff$ there exists $(R(X_1, \ldots, X_n) \Leftarrow R_1(\overrightarrow{X_1}) \wedge \ldots \wedge R_k(\overrightarrow{X_k})) \in \mathcal{C}^{\epsilon}_R$ with $\forall j \in [1 \ldots k], \widehat{\sigma}(\overrightarrow{X_j}) \in \mathcal{L}(S, R_j)$ with $\widehat{\sigma} = \{X_i \mapsto t_i \mid i \in [1 \ldots n]\}$.
Also $\forall R' \in \mathcal{R}(S),\ \mathcal{L}(S, R') = \mathcal{L}(S', R')$ is immediate, as every clause coming from $\mathcal{C}^{\epsilon}_R$ has R as head relation with $R \notin \mathcal{R}(S)$.

Now, we prove $\mathcal{L}(S \cup \mathcal{C}^{\epsilon}_R, R) = \mathcal{L}(S', R)$.

- $\mathcal{L}(S \cup \mathcal{C}^{\epsilon}_R, R) \subseteq \mathcal{L}(S', R)$: Suppose $\vec{t} \in \mathcal{L}(S \cup \mathcal{C}^{\epsilon}_R, R)$. Because R does not appear in S, $\vec{t}$ is recognized by a clause of $\mathcal{C}^{\epsilon}_R$. More formally, there exists a clause $R(X_1, \ldots, X_n) \Leftarrow R_1(\overrightarrow{X_1}) \wedge \ldots \wedge R_k(\overrightarrow{X_k})$ in $\mathcal{C}^{\epsilon}_R$ such that, when writing $\widehat{\sigma} = \{X_i \mapsto t_i \mid i \in [1 \ldots n]\}$, we have $\forall j \in [1 \ldots k], S \models \widehat{\sigma}(R_j(\overrightarrow{X_j}))$, i.e. $\widehat{\sigma}(\overrightarrow{X_j}) \in \mathcal{L}(S, R_j)$. Note that $\widehat{\sigma}((X_1, \ldots, X_n)) = (t_1, \ldots, t_n)$.

  For any $j \in [1 \ldots k]$, because $\widehat{\sigma}(\overrightarrow{X_j}) \in \mathcal{L}(S, R_j)$, there exists a clause $\varphi_j = R_j(\overrightarrow{p_j}) \Leftarrow B_j$ such that $\widehat{\sigma}(\overrightarrow{X_j})$ is of the shape $\overrightarrow{p_j}$ and, with $\sigma_j = MGU(\widehat{\sigma}(\overrightarrow{X_j}) \stackrel{?}{=} \overrightarrow{p_j})$, $S \models \sigma_j(B_j)$. Because the variables from the clauses are universally quantified, we can assume them different from any other. Therefore there exists a substitution $\sigma' = MGU(\{\widehat{\sigma}(\overrightarrow{X_j}) \stackrel{?}{=} \overrightarrow{p_j} \mid j \in [1 \ldots k]\})$ such that for all $j \in [1 \ldots k]$, $S \models \sigma'(B_j)$.

  From the existence of σ', there exists $\sigma = MGU(\{\overrightarrow{X_j} \stackrel{?}{=} \overrightarrow{p_j} \mid j \in [1 \ldots k]\})$ a substitution that can be extended into $\widehat{\sigma}$ by letting $\widehat{\sigma} = \sigma; \sigma'$. We now have, for all $j \in [1 \ldots k]$, that $S \models \sigma'(\sigma(B_j))$.

  By definition we have $\sigma(R(\vec{X}) \Leftarrow B_1 \cup \ldots \cup B_k) \in Ext_{R(\vec{X}) \Leftarrow R_1(\overrightarrow{X_1}) \wedge \ldots \wedge R_k(\overrightarrow{X_k})}$. Because for all $j \in [1 \ldots k]$ we have $S \models \sigma'(\sigma(B_j))$ and $\sigma'(\sigma(\vec{X})) = \vec{t}$, we have $\widehat{\sigma}(\vec{X}) = \vec{t} \in \mathcal{L}(S', R)$.

  Moreover, because every tuple of patterns $\overrightarrow{p_j}$ from the head of SHoCs is shallow, we have, for any $(X, p) \in \sigma$, p is shallow. Moreover, because of constraint $Vars(H) = Vars(B)$ of Definition 7.8, we have $dom(\sigma) = Vars(\vec{X})$. Therefore $\sigma(R(\vec{X}) \Leftarrow B_1 \cup \ldots \cup B_k) \in Ext_{R(\vec{X}) \Leftarrow R_1(\overrightarrow{X_1}) \wedge \ldots \wedge R_k(\overrightarrow{X_k})}$ is a SHoC.
- $\mathcal{L}(S \cup \mathcal{C}^{\epsilon}_R, R) \supseteq \mathcal{L}(S', R)$: Suppose $\vec{t} \in \mathcal{L}(S', R)$. Because R does not appear in S, $\vec{t}$ is recognized by (at least) a clause $\varphi \in Ext_{R(\vec{X}) \Leftarrow R_1(\overrightarrow{X_1}) \wedge \ldots \wedge R_k(\overrightarrow{X_k})}$ for some ϵ-clause $R(\vec{X}) \Leftarrow R_1(\overrightarrow{X_1}) \wedge \ldots \wedge R_k(\overrightarrow{X_k}) \in \mathcal{C}^{\epsilon}_R$.

  By definition we have $\varphi = \sigma(R(\vec{X}) \Leftarrow B_1 \cup \ldots \cup B_k)$ with, for all $j \in [1 \ldots k]$, $(R_j(\overrightarrow{p_j}) \Leftarrow B_j) \in S$ and $\sigma = MGU(\{\overrightarrow{X_j} \stackrel{?}{=} \overrightarrow{p_j} \mid j \in [1 \ldots k]\})$.

Because φ recognizes $\vec{t}$ in R, we have that $\vec{t} = \widehat{\sigma}(\vec{X})$ is of the shape $\sigma(\vec{X})$, i.e. σ can be extended into $\widehat{\sigma}$ using σ' by $\widehat{\sigma} = \sigma;\sigma'$. Moreover, for all $j \in [1 \ldots k]$, $S \models \widehat{\sigma}(B_j)$. For every $j \in [1 \ldots k]$, because there exists a clause $(R_j(\vec{p_j}) \Leftarrow B_j) \in S$, we have $\widehat{\sigma}(\vec{X_j}) \in \mathcal{L}(S, R_j)$. Therefore $\vec{t} \in \mathcal{L}(S \cup \mathcal{C}^{\epsilon}_R, R)$.

□

Finally, we extend ϵ-clauses so that using them becomes more practical.

Definition 7.12 (More expressive ϵ-clauses)**.** ϵ-clauses are made more expressive by loosening the constraint $Vars(H) = Vars(B)$ into $Vars(H) \supseteq Vars(B)$ for any ϵ-clause $H \Leftarrow B$. The extension is immediate as, for any type τ, SHoCs can encode the predicate Any_τ that recognizes any term of type τ, defined as: $\{Any_\tau(f(\vec{X})) \Leftarrow \top \mid f \in \Sigma \wedge \tau_{out}(f) = \tau \wedge \vec{X}$ are fresh variables$\}$.

7.3 Closure properties and decision procedures of SHoCs

This section defines the intersection, union, complement, and cylindrification of SHoCs that reflect the associated language operations. We show that SHoCs are not closed by projection. We also show that the membership problem is decidable but emptiness is not.

First, note that a set $\{S_1, \ldots, S_n\}$ of SHoCs can always be combined into one by taking the set-union $S_1 \cup \ldots \cup S_n$ of the clauses composing them, possibly with some renaming if two SHoCs define different relations with the same name.

Definition 7.13 (Set-union of clauses of SHoCs)**.** Let S_1 and S_2 two SHoCs. Then, if $\mathcal{R}(S_1) \cap \mathcal{R}(S_2) = \emptyset$, then $S = S_1 \cup S_2$ is a SHoCs such that $\mathcal{R}(S) = \mathcal{R}(S_1) \cup \mathcal{R}(S_2)$ and $\forall i \in \{1, 2\}, \forall R \in \mathcal{R}(S_i),\ \mathcal{L}(S, R) = \mathcal{L}(S_i, R)$.

We therefore focus on defining operations w.r.t a single SHoCs.

Definition 7.14 (Closure by intersection of relations)**.** Let S be a SHoCs and $\mathcal{R}' \subseteq \mathcal{R}(S)$ a subset of same-type relations. The intersection of $\mathcal{R}'$ in a fresh relation symbol R is a SHoCs $S_{\cap_{\mathcal{R}'}}$ defined as

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

Definition 7.15 (Closure by union of relations)**.** Let S be a SHoCs and $\mathcal{R}' \subseteq \mathcal{R}(S)$ a subset of same-type relations. The union of $\mathcal{R}'$ in a fresh relation symbol R is a SHoCs $S_{\cup_{\mathcal{R}'}}$ defined as

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

Theorem 7.16 (Intersection and union)**.** *Let S be a SHoCs with $\mathcal{R}' \subseteq \mathcal{R}(S)$ a subset of same-type relations. We have $\mathcal{L}(R, S_{\cap_{\mathcal{R}'}}) = \bigcap_{R' \in \mathcal{R}'} \mathcal{L}(R', S)$ and $\mathcal{L}(R, S_{\cup_{\mathcal{R}'}}) = \bigcup_{R' \in \mathcal{R}'} \mathcal{L}(R', S)$.*

Proof. Immediate from the intersection and union definitions and Lemma 7.11. □

Definition 7.17 (Complement specification)**.** Let S be a SHoCs defining the set of relations $\mathcal{R}$. The complement of S is a SHoCs S^c such that $\forall R \in \mathcal{R}, \mathcal{L}(R, S^c) = \overline{\mathcal{L}(R, S)}$.

When S is clear from context, we may simply write R^c to refer to R in S^c.

Definition 7.18 (Complement construction)**.** Let S be a SHoCs. For a given relation R of type $(\tau_1, \ldots, \tau_n)$, let $F(R) = \{(f_1, \ldots, f_n) \mid \forall i \in [1 \ldots n], f_i \in \Sigma \wedge \tau_{out}(f_i) = \tau_i\}$ be the set of tuples of constructors $(f_1, \ldots, f_n)$ whose output type is $(\tau_1, \ldots, \tau_n)$. Then, the complement of S is defined as

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

with $Flip : \mathcal{P}(\mathcal{P}(A)) \to \mathcal{P}(\mathcal{P}(A))$, with A the set of all atoms, the set of all possible sets made by selecting one atom per body (or unordered cartesian product):

$$Flip(\{B_1, \ldots, B_k\}) = \{\{\omega_1, \ldots, \omega_k\} \mid (\omega_1, \ldots, \omega_k) \in B_1 \times \ldots \times B_k\}$$

Theorem 7.19. *We have that $\forall R \in \mathcal{R}, \mathcal{L}(R, S^c) = \overline{\mathcal{L}(R, S)}$.*

Proof. Let us show that $\forall \overrightarrow{t}, \overrightarrow{t} \in \mathcal{L}(R, S^c) \iff \overrightarrow{t} \notin \mathcal{L}(R, S)$ by induction on $ht(\overrightarrow{t})$. Let $\overrightarrow{t} \in \mathcal{T}(\Sigma)^n$ a tuple of terms. We write $\overrightarrow{t} = (f_1(\overrightarrow{t_1}), \ldots, f_n(\overrightarrow{t_n}))$ and $\overrightarrow{f} =$

$(f_1, \dots, f_n)$.

$\vec{t} \in \mathcal{L}(R, S)$

$\iff$ By definition of $\mathcal{L}(\cdot, \cdot)$

There exists a clause $R(\vec{f}) \Leftarrow B \in S$ (using the projector notation) such that for all atoms of the shape $R'(\pi_1, \dots, \pi_k)$ in B we have $(\vec{t}[\pi_1], \dots, \vec{t}[\pi_k]) \in \mathcal{L}(R', S)$.

$\iff$ By definition of $Flip$

For any $B' \in Flip(\{B \mid (R(\vec{f}) \Leftarrow B) \in S\})$, there exists an atom $R'(\pi_1, \dots, \pi_k)$ in B' such that $(\vec{t}[\pi_1], \dots, \vec{t}[\pi_k]) \in \mathcal{L}(R', S)$.

$\iff$ By induction

For any $B' \in Flip(\{B \mid (R(\vec{f}) \Leftarrow B) \in S\})$, there exists an atom $R'(\pi_1, \dots, \pi_k)$ in B' such that $(\vec{t}[\pi_1], \dots, \vec{t}[\pi_k]) \notin \mathcal{L}(R', S^c)$.

$\iff$ By definition of S^c

There exists no clause $R(\vec{f}) \Leftarrow B \in S^c$ such that for all atoms of the shape $R'(\pi_1, \dots, \pi_k)$ in B we have $(\vec{t}[\pi_1], \dots, \vec{t}[\pi_k]) \in \mathcal{L}(R', S^c)$.

$\iff$ By definition of $\mathcal{L}(\cdot, \cdot)$

$\vec{t} \notin \mathcal{L}(R, S^c)$

□

Example 7.20 (Complement)**.** Let S_{sh} be the SHoCs

$$\mathrm{Shal}(\mathit{leaf}, z) \quad \mathrm{Shal}(\mathit{leaf}, s(N)) \quad \mathrm{Shal}(\mathit{node}(T_1, E, T_2), s(N)) \Leftarrow \mathrm{Shal}(T_1, N) \wedge \mathrm{Shal}(T_2, N)$$

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

$\mathrm{Shal}^c(\mathit{node}(T_1, E, T_2), z)$

$\mathrm{Shal}^c(\mathit{node}(T_1, E, T_2), s(N)) \Leftarrow \mathrm{Shal}^c(T_1, N) \quad \mathrm{Shal}^c(\mathit{node}(T_1, E, T_2), s(N)) \Leftarrow \mathrm{Shal}^c(T_2, N)$

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

- For $\vec{f} = (\mathit{node}, s)$, we have:
  - — Let $E_1 = \{B \mid R(\vec{f}) \Leftarrow B \in S\}$, i.e., $E_1 = \{\{\mathrm{Shal}(T_1, N), \mathrm{Shal}(T_2, N)\}\}$
  - — Then, $Flip(E_1) = \{\{\omega\} \mid (\omega) \in B\}$ with $B = \{\mathrm{Shal}(T_1, N), \mathrm{Shal}(T_2, N)\}$, so $Flip(E_1) = \{\{\mathrm{Shal}(T_1, N)\}, \{\mathrm{Shal}(T_2, N)\}\}$.

— This yields the two clauses whose head is $\text{Shal}^c(node(T_1, E, T_2), s(N))$.

- For $\overrightarrow{f} = (leaf, z)$ or $\overrightarrow{f} = (leaf, s)$, we have:
  — Let $E_2 = \{B \mid R(\overrightarrow{f}) \Leftarrow B \in S\}$, i.e., $E_2 = \{\emptyset\}$
  — We have $Flip(E_2) = \emptyset$, so there is no clause with $\text{Shal}(leaf, z)$ nor $\text{Shal}(leaf, s)$ as a head.
- For $\overrightarrow{f} = (node, z)$, we have:
  — Let $E_3 = \{B \mid R(\overrightarrow{f}) \Leftarrow B \in S\}$, i.e., $E_3 = \emptyset$
  — We have $Flip(E_3) = \{\emptyset\}$ because the neutral element of the cartesian product is $\{\emptyset\}$.
  — Thus, there is a clause $\text{Shal}^c(node(T_1, E, T_2), z)$.

Lemma 7.21 (Removing 0-ary atoms)**.** *Let S be a SHoCs. Then there exists an equivalent SHoCs (defining the same relations) S' such that, for any clause $H \Leftarrow B$ in S', there exists no atom whose relation is of arity 0, i.e. of the form $R()$, in B.*

Proof. Given a SHoCs S, it is simple to determine whether a given 0-ary relation symbol R has an empty language. Indeed, because of the constraints regarding the definition of a SHoC, any clause whose head's relation is R is of the form $R() \Leftarrow R_1() \land \ldots \land R_k()$. Therefore, a finite fixed point allows to partition 0-ary relations into those with an empty language and the others (whose language is $\{()\}$). Then, every 0-ary whose language is not empty can be removed from the body of clauses, as they do not add any constraint. On the other hand, every clause that contains a 0-ary relation symbol in their body whose language is empty can be entirely removed, as they can never be satisfied.
The resulting SHoCs defines the exact same relations and meets the no 0-ary relation in bodies criterion. □

Definition 7.22 (Cylindrification of SHoCs)**.** The k-th cylindrification of type τ of a relation R' in a SHoCs S, written $Inj_k^\tau(R', S)$, is defined as a fresh relation R as

$$Extend(S, \mathcal{C}_R^\epsilon) \text{ with } \mathcal{C}_R^\epsilon = \{R(X_1, \ldots, X_{k-1}, X, X_k, \ldots, X_n) \Leftarrow R'(X_1, \ldots, X_n)\}$$

The resulting SHoCs S' is such that

$$\mathcal{L}(S', R) = \{(t_1, \ldots, t_{k-1}, t, t_k, \ldots, t_n) \mid (t_1, \ldots, t_k, \ldots, t_n) \in \mathcal{L}(S, R') \land t \in \mathcal{T}_\tau(\Sigma)\}.$$

Definition 7.23 (Projection of a relation)**.** The ith projection of a relation R is the relation $Proj_i(R) = \{(t_1, \ldots, t_{i-1}, t_{i+1}, \ldots, t_n) \mid (t_1, \ldots, t_i, \ldots, t_n) \in r\}$.

Theorem 7.24 (SHoCs are not closed by projection)**.** *There exists at least one SHoCs S and relation symbol $R \in \mathcal{R}(S)$ such that $Proj_i(\mathcal{L}(S, R))$ cannot be represented by a SHoCs.*

Negative projection closure, by Naoki Kobayashi.
Let us define the tree t_n by $t_0 = e$ and $t_{n+1} = c(a^n b^n e, t_n)$, and the relation $R = \{(a^n b^n e, t_n) \mid n \geq 0\}$. This relation R can be defined by the following SHoCs:
$R(e, e)$
$R(a(X), c(Y, Z) \Leftarrow S(X, Y) \wedge R(Y, Z)$
$S(a(X), a(Y) \Leftarrow S(X, Y)$
$S(b(X), e) \Leftarrow E(X)$
$S(b(X), b(Y)) \Leftarrow S_1(X, Y)$
$S_1(b(X), b(Y)) \Leftarrow S_1(X, Y)$
$S_1(b(X), e) \Leftarrow E(X)$
$E(e)$
Here, the relations $\mathcal{L}_S$, $\mathcal{L}_{S_1}$, $\mathcal{L}_E$, respectively defined by S, S_1, E, are:
$\mathcal{L}_S = \{a^m b^{n+1} e, a^m b^n e \mid m, n \geq 0\}$
$\mathcal{L}_{S_1} = \{b^{n+1} e, b^n e \mid n \geq 0\}$
$\mathcal{L}_E = \{e\}$
However, the projection of R to the first element, $\{a^n b^n e \mid n \geq 0\}$, is not definable by SHoCs, as the unary predicates on unary trees definable by SHoCs must be regular. □

Decision problems

Theorem 7.25 (SHoCs emptiness problem is undecidable)**.** *For S a SHoCs and R a relation, the SHoCs emptiness problem is to decide whether $\mathcal{L}(R, S)$=$\emptyset$. This problem is undecidable.*

The proof uses an encoding of Minsky machines into SHoCs. See Appendix A.3 for details.

Theorem 7.26 (SHoCs membership problem is decidable)**.** *For S a SHoCs, $\vec{t}$ a tuple of terms, and R a relation, the SHoCs membership problem is to decide whether $\vec{t} \in \mathcal{L}(R, S)$. This problem is decidable.*

Proof. Let us consider the $Inhabits_S$ procedure defined for SHoCs from Section 8.1, which is the SHoCs counterpart to $Inhabits_{\mathcal{A}}$. Lemma A.3 is still valid, as any other Lemma on the teacher procedure, when using SHoCs instead of automata. Notice that, for any substitution σ, $\sigma(\vec{t}) = \vec{t}$, as $\vec{t}$ is a tuple of (ground) terms. Lemma A.3 therefore states that, if $Inhabits_{\mathcal{A}}(\{R(\vec{t})\})$ returns $Some(_)$ in i steps, then $i \leq ht(\vec{t})$. Moreover, by Theorem 6.15, we can use $Inhabits_S^{det}$ with an upper-bound of $ht(\vec{t}) + 1$ on the number of steps to decide whether $\vec{t} \in \mathcal{L}(R, S)$.

□

7.4 Expressivity of SHoCs

We illustrate the expressivity of SHoCs with the relations from the examples. The relations IsEmpty, All0, and No0 can be represented by, respectively, $R_{isEmpty}$, R_{all0}, and R_{no0} in the following SHoCs, where R_0 and R_s are additional predicates recognizing, respectively, only z and all positive natural numbers.

Example 7.27 (SHoCs for the relations IsEmpty, All0, and No0)**.**

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

The following set of SHoCs with relations R_{leq}, R_{he}, and R_{heRB} represent, respectively, the models for Leq, Height, and HeightRB to prove property ϕ_3. The additional relation Shal is necessary to express Height as a SHoCs. Recall that the atom $\text{Shal}(t, n)$ is true if tree t is of height smaller or equal to n.

Example 7.28 (SHoCs for the relations R_{leq}, R_{he}, R_{heRB}, and Shal)**.**

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

$$\text{Shal}(leaf, z)$$
$$\text{Shal}(leaf, s(N))$$
$$\text{Shal}(node(T_1, E, T_2), s(N)) \Leftarrow \text{Shal}(T_1, N) \wedge \text{Shal}(T_2, N)$$

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

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

Note that R_{leq} is not used in the definition of R_{len} but Shal is used instead. This is because, in a SHoCs, the variables of the body must be contained in the variables from the head, which forbids to fetch the heights of both subtrees, compare them using R_{leq}, and forget the smallest. Note that all of the above-defined relations can also be defined using a complete-convolution automaton. However, their definition (except for R_{leq}) using complete-convolution automata is bloated with useless states and too big to be practical, which is the reason complete-convolution is not used. There also exists relations that can be defined by SHoCs but not by convoluted automata, e.g. $\{f(t,t) \mid t \in \mathcal{T}(\Sigma)\}$, which is defined using the equality predicate on terms (always definable in SHoCs) and the single clause $f(X,Y) \Leftarrow Eq(X,Y)$.

7.4.1 SHoCs and convoluted automata

SHoCs are more expressive than convoluted tree automata because any convoluted tree automaton has an equivalent SHoCs and some SHoCs-definable relations such as $\{f(t,t) \mid t \in \mathcal{T}(\Sigma)\}$ cannot be represented by a convoluted tree automaton using any of the previously-defined convolution.
Here is the translation of convoluted tree automata into SHoCs. We use the projector notation for SHoCs of Definition 7.6.

Definition 7.29 (Convoluted automata as SHoCs)**.** Let $\odot$ be any of Ⓛ, Ⓡ, Ⓒ. Let $\mathcal{A}$ be a $\odot$-convoluted automaton on alphabet $\Sigma^{\odot}$ with transitions Δ. Then $S_{\mathcal{A}}$ is defined as

$$S_{\mathcal{A}} = \left\{R_q(f_1,\ldots,f_n) \Leftarrow R_{q_1}(\overrightarrow{\pi_1}) \wedge \ldots \wedge R_{q_k}(\overrightarrow{\pi_k}) \mid \langle f_1,\ldots,f_n\rangle(q_1,\ldots,q_k) \to q \in \Delta\right\}$$

$$\text{with } (\overrightarrow{\pi_1},\ldots,\overrightarrow{\pi_k}) = \odot((1\cdot 1,\ldots,1\cdot|f_1|),\ldots,(n\cdot 1,\ldots,n\cdot|f_n|))$$

Note that the tuple $(1\cdot 1,\ldots,1\cdot|f_1|),\ldots,(n\cdot 1,\ldots,n\cdot|f_n|)$ simply corresponds to the positions of subterms for the tuple of functions $(f_1,\ldots,f_n)$.

Lemma 7.30. *Let $\odot$ be any of Ⓛ, Ⓡ, Ⓒ. Let $\mathcal{A}$ be a convoluted automaton on alphabet $\Sigma^{\odot}$ with transitions Δ. Let $S_{\mathcal{A}}$ be the SHoCs defined from $\mathcal{A}$. Let q be a state from $\mathcal{A}$. Then $\mathcal{R}(\mathcal{A},q) = \mathcal{L}(S_{\mathcal{A}},R_q)$.*

Proof. Let $\vec{t} \in \mathcal{T}(\Sigma)^n$ with $\vec{t} = (f_1(\overrightarrow{t_1}),\ldots,f_n(\overrightarrow{t_n}))$. Let us prove by induction on

$ht(\vec{t})$ that $\vec{t} \in \mathcal{R}(\mathcal{A}, q) \iff \vec{t} \in \mathcal{L}(S_\mathcal{A}, R_q)$:

$\vec{t} \in \mathcal{R}(\mathcal{A}, q)$

$\iff$ By definition of $\mathcal{R}(\cdot, \cdot)$

$\bigodot^t (\vec{t}) \in \mathcal{L}(\mathcal{A}, q)$

$\iff$ By definition of $\mathcal{L}(\cdot, \cdot)$

There exists a transition$\langle f_1, \ldots, f_n \rangle (q_1, \ldots, q_k) \to q$ such that, with $(\vec{t'_1}, \ldots, \vec{t'_k}) = \bigodot(\vec{t_1}, \ldots, \vec{t_n}), \forall j \in [1 \ldots k], \vec{t'_j} \in \mathcal{R}(\mathcal{A}, q_j)$

$\iff$ By definition of $S_\mathcal{A}$, induction, and Lemma 5.31

There exists a clause$\langle R_q(f_1, \ldots, f_n) \rangle \Leftarrow R_{q_1}(\vec{\pi'_1}), \ldots, R_{q_k}(\vec{\pi'_k})$ such that, with $(\vec{\pi'_1}, \ldots, \vec{\pi'_k}) = \bigodot((1 \cdot 1, \ldots, 1 \cdot |f_1|), \ldots, (n \cdot 1, \ldots, n \cdot |f_n|))$, $\forall j \in [1 \ldots k], (\vec{t}[\vec{\pi'_j}[1]], \ldots, \vec{t}[\vec{\pi'_j}[|\vec{\pi'_j}|]]) \in \mathcal{R}(S_\mathcal{A}, R_{q_j})$

$\iff$ By definition of $\mathcal{L}(\cdot, \cdot)$ for SHoCs

$\vec{t} \in \mathcal{L}(S_\mathcal{A}, R_q)$

□

Example 7.31. Let $\mathcal{A}$ be the left-convoluted automaton (on $\Sigma^{Ⓛ}$) recognizing the HeightLB relation in a state q_{hlb} with the six following transitions:

$$\begin{array}{rr} \langle leaf, z \rangle () \to q_{hlb} & \langle node, s \rangle (q_{hlb}, q_n, q_t) \to q_{hlb} \\ \langle leaf \rangle () \to q_t & \langle node \rangle (q_t, q_n, q_t) \to q_t \\ \langle z \rangle () \to q_n & \langle s \rangle (q_n) \to q_n \end{array}$$

Its SHoCs translation, using projector notation, is:

$$\begin{array}{rr} R_{hlb}(leaf, z) & R_{hlb}(node, s) \Leftarrow R_{hlb}(1 \cdot 1, 2 \cdot 1) \land R_n(1 \cdot 2) \land R_t(1 \cdot 3) \\ R_t(leaf) & R_t(node) \Leftarrow R_t(1 \cdot 1) \land R_n(1 \cdot 2) \land R_t(1 \cdot 3) \\ R_n(z) & R_n(s) \Leftarrow R_n(1 \cdot 1) \end{array}$$

Using the standard notation, we obtain the following SHoCs:

$$R_{hlb}(leaf, z) \qquad R_{hlb}(node(X_1, X_2, X_3), s(Y_1)) \Leftarrow R_{hlb}(X_1, Y_1) \wedge R_n(X_2) \wedge R_t(X_3)$$
$$R_t(leaf) \qquad R_t(node(X_1, X_2, X_3)) \Leftarrow R_t(X_1) \wedge R_n(X_2) \wedge R_t(X_3)$$
$$R_n(z) \qquad R_n(s(X_1)) \Leftarrow R_n(X_1)$$

This is not the smallest SHoCs that recognizes this relation $heightLB$, relating a tree to the height of its leftmost branch, but this is the one corresponding to the automaton $\mathcal{A}$.

7.4.2 SHoCs and CS-programs

There exists a variety of clause-based formalism to represent languages. The closest one from SHoCs is that of *CS-programs* [28]. A CS-program is a finite set of CS-clauses, which are strict Horn clauses whose body is flat and linear. A CS-program is also interpreted by the least fixed-point semantics. The expressivity of SHoCs and CS-programs are incomparable. The Height and Shal examples cannot be represented using a CS-program because of bodies linearity: CS-programs can only build relations from bottom to top but can not compare two subterms (needed for Height and Shal). On the other hand, since the head of CS-programs is not required to be shallow, it allows them to precisely represent some relations where symbols need to be stacked in the head of the clause. This is the case for the *double* function (Example 7.32) which is out of the scope of SHoCs but in the scope of CS-programs.

Example 7.32. Let $\mathrm{Double}(n, m)$ be the relation s.t. $m = 2 \times n$, defined by the CHCs:

$$\mathrm{Double}(z, z) \qquad \mathrm{Double}(s(N), s(s(M))) \Leftarrow \mathrm{Double}(N, M)$$
$$N = M \Leftarrow \mathrm{Double}(X, N) \wedge \mathrm{Double}(X, M)$$

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

$R(s(X), s(Y)) \Leftarrow R_1(X, Y)$ $R_1(s(X), s(Y)) \Leftarrow R_2(X, Y)$ $R_2(z, s(Y)) \Leftarrow R_1'(Y)$
$R_1'(s(X)) \Leftarrow R_0'(X)$ $R_0'(z)$.
However, representing the Double relation for all natural numbers would require infinitely many predicate symbols and therefore an infinite SHoCs, which is forbidden.

7.4.3 SHoCs and relational alternating automata

SHoCs are equivalent to a direct relational extension of alternating tree automata that is proposed in this subsection. The definition of alternating tree automata and related notions, such as that of an accepting run, can be found in [7]. We restate it here, in a somewhat different vocabulary, for making the difference in the extension clearer.

Definition 7.33 (Alternating tree automata)**.** An alternating tree automaton $\mathcal{A}$ over Σ is the data of a set of states Q and of a mapping Δ from $Q \times \mathbf{\Sigma}$ to $\mathcal{B}^+(Q \times \mathbb{N})$ with $\mathcal{B}(\mathcal{X})$ being the set of positive (using only conjunction and disjunction) propositional formulas on the set of atoms $\mathcal{X}$. Moreover, for any atom (q', j) of $\Delta(q, f)$, we enforce $j \in [1 \dots |f|]$.

Intuitively, the condition for recognizing a term $f(t_1, \dots, t_n)$ in a state q is that the children $t_1, \dots, t_n$, denoted by their index in the tuple, satisfy the formula $\Delta(q, f)$.
Without loss of generality, suppose that every alternating tree automata is in DNF, i.e. such that if $\Delta(q, f) = \varphi$ then φ is in D(isjonctive) N(ormal) F(orm).

Definition 7.34 (Language recognized by an alternating tree automaton)**.** The language recognized by a state q in an alternating automaton $\mathcal{A}$ can be inductively defined as:

$$\mathcal{L}(\mathcal{A}, q) = \bigcup_{f \in \Sigma} \mathcal{L}_f$$

with, when writing $\Delta(q, f)$ as $\bigvee_{j=1}^{m} \bigwedge_{k=1}^{k_j} (q_{(j,k)}, i_{(j,k)})$,

$$\mathcal{L}_f = \bigcup_{j \in [1 \dots m]} \bigcap_{k \in [1 \dots k_j]} \{t \mid Root(t) = f \wedge t[i_{(j,k)}] \in \mathcal{L}(\mathcal{A}, q_{(j,k)})\}$$

Alternating tree automata have the same expressivity as classical tree automata but can be more compact. We now define their direct relational extension. The differences with the non-relational definition is that, in the formulas, instead of pointing to a subterm t_j of a term $f(t_1, \dots, t_n)$ by its index j, it points at a tuple of subterms, each of the form

$t_{(i,j)}$, of a tuple of terms $(f_1(t_{(1,1)}, \ldots, t_{(1,|f_1|)}), \ldots, f_n(t_{(n,1)}, \ldots, t_{(n,|f_n|)}))$ by a tuple of couples of indices (i, j). These differences are written in bold.

Definition 7.35 (Relational Alternating Tree Automata (RATA))**.** A **relational** alternating tree automaton $\mathcal{A}$ over Σ is the data of a set of states Q and of a mapping Δ from $Q \times \boldsymbol{\Sigma^*}$ to $\mathcal{B}^+(Q \times \mathbf{(\mathbb{N} \times \mathbb{N})^*})$ with only a finite number of images that are different from $\bot$. Moreover, for any atom $(q', \boldsymbol{(i, j)})$ of $\Delta(q, \boldsymbol{(f_1, \ldots, f_n)})$, we enforce $\boldsymbol{i \in [1 \ldots n] \wedge j \in [1 \ldots |f_i|]}$.

SHoCs and RATA are equivalent formalisms apart from slight notation differences, the biggest one being that RATA are explicit about every tuple of symbols, while SHoCs are implicit about the ones that are not recognized.

Theorem 7.36. *For any SHoCs S, there exists a RATA $\mathcal{A}_S$ such that, for any relation symbol R, $\mathcal{L}(S, R) = \mathcal{L}(\mathcal{A}_S, R)$. The converse is also true.*

Proof.

- Given a SHoCs S (using the projector notation of Definition 7.6), we can define the following mapping:

$$\Delta_S(R, \vec{f}) = \bigvee_{(R(\vec{f}) \Leftarrow B) \in S} B$$

  This transition function is in DNF and immediately defines a RATA $\mathcal{A}_S$ such that, for any relation symbol R, $\mathcal{L}(S, R) = \mathcal{L}(\mathcal{A}_S, R)$.

- Given a RATA $\mathcal{A}$ with mapping Δ in DNF, we can define the following SHoCs:

$$S_\Delta = \bigcup_{\Delta(R, \vec{f}) = B_1 \vee \ldots \vee B_k} \{R(\vec{f}) \Leftarrow B_j \mid j \in [1 \ldots k]\}$$

  This SHoCs $S_{\mathcal{A}}$ is equivalent to $\mathcal{A}$.

□

CHAPTER 8

LEARNER/TEACHER FOR SHOCS

This chapter is dedicated to the Learner and Teacher procedures using Shallow Horn Clauses as models. The generic procedure for satisfiability of a set of clauses (for which the learner and teacher are defined) is presented in Section 4.3. For the teacher, we reuse quite everything from the teacher defined on convoluted tree automata in Section 6.1. The learner, however, is different, as the procedure defined on automata makes use of the static nature of the convolution, i.e. the initial automaton can be computed and then any (complete and deterministic) automaton can be found by merging its states, which is not the case for SHoCs.
Every proof and theorem of the teacher using convoluted automata (Section 6.1) is preserved when using SHoCs. We do not re-state nor re-prove them, as the idea is the same. The correctness, termination, and minimality properties for the learner on SHoCs are given because the learner definition changes. Finally, the correctness and relative completeness of the whole Learner/Teacher procedure that are stated and proven in Section 6.3 are also preserved, as the theorems for convoluted automata are mirrored in this SHoCs setting.

8.1 Teacher

The teacher procedure *Teacher* that is defined on automata (Chapter 6.1) can easily be used for SHoCs too, as the formalisms are quite close in essence.
Convoluted automata like SHoCs are the data of a set of transitions/clauses (using states/relations). Each of those transitions/clauses are the data of a state/relation symbol, a tuple of function symbols, and of substates/subatoms. In the case of convoluted automata, it is the chosen convolution which dictates how to use those substates. In the case of SHoCs, it is explicitly written in the atoms of the body.
The Section 6.1 can therefore be understood as using SHoCs instead of convoluted automata, with the only difference being the $unfoldOne$ function that needs to be re-

defined. Note that the intention is the same: progress one step into recognizing an atom $R(\vec{p})$ using a compatible transition (now clause) $R(\vec{f}) \Leftarrow B$.

Definition 8.1 (*unfoldOne*: generation of new atoms)**.**
Let $\vec{p} = (f_1(p_{(1,1)}, \ldots, p_{(1,|f_1|)}), \ldots, f_n(p_{(n,1)}, \ldots, p_{(n,|f_n|)}))$ be a tuple of patterns and $\varphi = R(f_1, \ldots, f_n) \Leftarrow B$ a clause of some SHoCs (using the projector notation). Then

$$unfoldOne(\vec{p}, \varphi) = \{R'(p_{(i_1,j_1)}, \ldots, p_{(i_m,j_m)}) \mid R'((i_1, j_1), \ldots, (i_m, j_m)) \in B\}$$

Example 8.2 (*unfoldOne* with SHoCs)**.** Let us consider the SHoCs from Example 7.28 and let $\varphi = \text{Height}(node(T_1, E, T_2), s(N)) \Leftarrow \text{Height}(T_1, N) \wedge \text{Shal}(T_2, N)$ one of its clauses. Let $\vec{p} = (p_1, p_2)$ with $p_1 = node(node(leaf, z, leaf), s(z), leaf)$ and $p_2 = s(s(z))$. Then

$$unfoldOne(\varphi, \vec{p}) = \{\text{Height}(node(leaf, z, leaf), s(z)),\ \text{Shal}(leaf, s(z))\}.$$

8.2 Learner

The learner receives a finite set of ground clauses $\widehat{\Gamma}$ from the teacher. This set is a counterexample to every previous SHoCs proposed by the learner. Starting from $\widehat{\Gamma}$, the objective of the learner is to build a *new* SHoCs satisfying $\widehat{\Gamma}$. If $\widehat{\Gamma}$ is contradictory, then the teacher has found a real counterexample to the property and $Learner(\widehat{\Gamma})$ returns *None*. If $\widehat{\Gamma}$ is satisfiable, then $Learner(\widehat{\Gamma}) = Some(S)$ with S a SHoCs such that $S \models \widehat{\Gamma}$. We will furthermore ensure that S is minimal w.r.t the number of relations defined in S among all possible SHoCs that satisfy $\widehat{\Gamma}$.
Finding a new SHoCs S is implemented as a constraint solving problem in the *Answer Set Programming* (ASP) paradigm with support for so-called *choice rules*. We use the solver *Clingo* [13]. From $\widehat{\Gamma}$, the learner builds a set of constraints $C(\widehat{\Gamma})$ defining all the possible choices to build a valid SHoCs satisfying $\widehat{\Gamma}$. Then, *all the constraints* of $C(\widehat{\Gamma})$ are given to the ASP solver which returns *one* solution, if it exists. In our definition of $C(\widehat{\Gamma})$, we use the **choose-one** keyword to ask the ASP solver to pick exactly one atom in a set and check that it is not in contradiction with the other constraints of $C(\widehat{\Gamma})$. For instance, to satisfy the two constraints **choose-one**$\{x > 1, x < 1\}$ and **choose-one**$\{x > 5, x = 5\}$, the solver can pick $x > 1$ from the first constraint and either $x > 5$

or $x = 5$ from the second. *A contrario*, picking $x < 1$ from the first constraint leads to a contradiction with any of the two possibilities for the second constraint.
To guarantee (relative) completeness of the learner-teacher loop, solving $C(\widehat{\Gamma})$ is done incrementally by considering first the smallest solutions w.r.t their number of relation symbols. We write i_{step} for the number of *new* relation symbols in the inferred SHoCs, i.e., relation symbols that are not already present in $\widehat{\Gamma}$. The constraints $C(\widehat{\Gamma})$ are checked for satisfiability, starting with $i_{step} = 0$, and i_{step} is incremented as long as the constraints are unsatisfiable. This process can be bounded by an upper-bound on i_{step}. Our upper-bound for i_{step} is the number of subterms appearing in $\widehat{\Gamma}$. Using this upper-bound is sound because, with one relation symbol to recognize exactly one subterm, it becomes immediate to create a SHoCs recognizing all positive atoms occurring in the input set of (ground) counterexamples. If a solution for the constraints have not been found within this upper-bound, then they are unsatisfiable.

Definition 8.3 (Constraints generated by the learner $C(\widehat{\Gamma})$)**.** Let $\widehat{\Gamma}$ be a finite set of ground clauses. $C(\widehat{\Gamma})$ is the set of constraints generated by the following rules:

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

(b) For each $\widehat{\varphi} \in \widehat{\Gamma}$ with $\widehat{\varphi} = R_1(\overrightarrow{t_1}) \vee \dots \vee R_n(\overrightarrow{t_n}) \vee \neg R'_1(\overrightarrow{t}'_1) \dots\dots \vee \neg R'_k(\overrightarrow{t_k})$, **choose-one** literal that needs to be true, i.e. choose one $R'_i(\overrightarrow{t}'_i)$ to be false or one $R_i(\overrightarrow{t_i})$ to be true.

(c) For each $R(\overrightarrow{t})$ that must be true a new SHoC is created. If $\overrightarrow{t}$ is of the form $(f_1(\overrightarrow{t_1}), \dots, f_p(\overrightarrow{t_p}))$,the head of the new SHoC will be $R(f_1(\overrightarrow{X_1}), \dots, f_p(\overrightarrow{X_p}))$.

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

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

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

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

Example 8.4 (Learning the Leq relation)**.** Let $i_{step} = 0$ and let $\widehat{\Gamma}$ be the set of the following 5 ground examples:

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

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

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

Rule (d) completes $C(\widehat{\Gamma})$ with constraints on all B_i. For B_1 this constraint is **choose-one**$\{\emptyset\}$. For B_2, this constraint is **choose-one**$\{\emptyset, \{R_{leq}(X,X)\}\}$. For B_3, four **choose-one** constraints are generated, one for each atom of $R_{leq}(X_1,X_1)$, $R_{leq}(X_1,X_2)$, $R_{leq}(X_2,X_1)$, and $R_{leq}(X_2,X_2)$. This yields 16 possible different bodies, including $\emptyset$, $\{R_{leq}(X_1,X_2)\}$, $\{R_{leq}(X_1,X_2), R_{leq}(X_2,X_1)\}$, ... When solving these constraints, the ASP solver can choose $B_1 = B_2 = B_3 = \emptyset$. With this choice, constraints added to $C(\widehat{\Gamma})$ by step 5 are trivially satisfied because no new atom is forced to be true. However, the ASP solver cannot choose $B_3 = \emptyset$, because constraints added by step 6 yields a contradiction, i.e., $R_{leq}(s(s(z)),s(z))$ is made true by φ_3 but it belongs to the set of false atoms. Hence, the ASP solver must change some of its choices. One possible change is to define $B_3 = \{R_{leq}(X_1,X_2)\}$. The clause φ_3 becomes $R_{leq}(s(X_1),s(X_2)) \Leftarrow R_{leq}(X_1,X_2)$. Since it was created to recognize $R_{leq}(s(z),s(z))$, this forces $R_{leq}(z,z)$ to be in the set of atoms that need to be true but, since it is already there, nothing changes. With this last SHoC S, every atom that must be true is still made true by some clause, and no atom that must be false is now made true by any SHoC. Thus, the learner has found

a model for $\widehat{\Gamma}$ so it returns $Some(\{\varphi_1, \varphi_2, \varphi_3\})$. Note that in this simple example the learner converged in one step, without having to increment $i_{step} = 0$. Having $i_{step} > 0$ would result in introducing i_{step} additional relation symbols.

Theorem 8.5. *The learner always terminates.*

Proof. Because we use an upper-bound on i_{step}, the learner procedure terminates if the constraint solving for any fixed i_{step} terminates.
Let some fixed i_{step}. We argue that the number of choices to be made is bounded, and therefore all of the search space can be explored in a finite time.
There are three types of choices in the constraints. Two are clearly finite: (a) The choice of the types of the new relations (constraint (a)). Each relation can take one type from a finite set of possible types. (b) The choice of one literal that needs to be true per example of $\widehat{\Gamma}$ (constraint (b)).
The third type of choice is that of the body of the SHoC created for each atom $R(\overrightarrow{t})$ that needs to be true (constraint (d)). For the body atoms (constraint (d)), there is one binary choice, whether to include the atom or not, per relation and compatible variables. One SHoC creation thus only needs a finite number of choices.
However, note that each SHoC $\varphi \stackrel{def}{=} H \Leftarrow \varphi$ created by the need of an atom $R(\overrightarrow{t})$ to be true leads to the need of every atom from $unfoldOne(R(\overrightarrow{t}), \varphi)$ to be true too (constraint (e)). The procedure still terminates, as $ht(unfoldOne(R(\overrightarrow{t}), \varphi)) < ht(R(\overrightarrow{t}))$, so any atom that is set to be true by the creation of φ is of strictly lower height than $R(\overrightarrow{t})$, so this retroactive "loop" can only go for at most $ht(\widehat{\Gamma})$. □

Theorem 8.6 (Learner correctness, completeness, minimality)**.** *Let $\widehat{\Gamma}$ be a finite set of examples. If Γ is satisfiable, then $Learner(\widehat{\Gamma}) = Some(S)$ with $S \models \widehat{\Gamma}$ and such that S is minimal among SHoCs that satisfy $\widehat{\Gamma}$ w.r.t the number of relations. If Γ is unsatisfiable, then $Learner(\widehat{\Gamma}) = None$.*

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

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

CHAPTER 9

IMPLEMENTATION AND EXPERIMENTS

We implemented the verification procedure, i.e. the model representation, teacher, and learner, for both convoluted tree automata and SHoCs. This implementation is available at `https://gitlab.inria.fr/tlosekoo/auto-forestation.git`. This implementation has been evaluated against examples coming from different sources, as well as some examples that we crafted ourselves. The focus is on algebraic datatypes and recursive functions.

9.1 Implementation

The implementation defines terms, tree automata, convolutions, SHoCs, both learners, the teacher, and the model-inference procedure. The *teacher* follows rather closely the breadth-first search of the procedure from Chapter 6.1. The *learner* delegates learning of SHoCs to the finite-model finder *Clingo* [13], as described in Section 8.2.
The total number of lines of codes is around 15000, mostly in Ocaml but also in Clingo for the Learner part. A significant portion is dedicated to re-implement standard data structures and algorithms. The implementation spans several packages which aim to be reusable:

inout: for simplifying reading and writing from and to system files

misc: the infamous miscellaneous package defining many basic data structures (pairs, extended lists, extended sets, extended maps, ...), list operations, and standard mathematical functions

printing: that defines parameterized printing of many standard data structures

term: defining typed alphabet, terms, patterns, substitutions, unification, and functions manipulating those data structures

clause: defining first-order clauses, their manipulation, and their simplification

program_to_clauses: defining the translation from SMTLIB programs into clauses and the clauses approximation presented in 4.2

io_clingo: defining an API to interact with the Clingo solver

tree_tuple_formalisms: defining both convoluted tree automata and shallow Horn clauses, together with operations to manipulate them

model_checking: defining the teacher and its enhancements described both in Chapter 6 and in Section 9.1.1

model_inference: defining the learner for both convoluted automata and SHoCs (Chapters 6 and 8) and also defines the *Sat* procedure described in Section 4.3. These two modules will be split and the overall structure improved in an incoming version.

9.1.1 Teacher

The teacher, in addition to implementing the pruning and splitting into independent atoms proposed in Section 6.1.1, optimizes the search with a canonization of sets of atoms, the use of memoisation, simplification of sets of atoms, and a few heuristics to choose the order of the breadth-first search. As there is a lot of redundancy in the proof search, memoisation avoids re-computing the unfolding of a set of atoms. However, memoisation alone is not very useful, as even equivalent sets of atoms are often different because of variable names. This is the reason for using canonization, which helps equivalent sets of atoms to have the same internal representation.

Definition 9.1 (Canonization of a set of atoms)**.** Let Ω be a set of atoms. Suppose an order of the alphabet, then terms and tuple of terms can be ordered, so patterns can be preordered as well as sets of atoms. A set of atoms Ω is thus given one linear extension of this preorder. This linear order on Ω yields a linear order on its variables $Vars(\Omega)$ by e.g. a depth-first exploration.
Every variable in Ω is then replaced by a variable named X_i, with i its position in the variable order.

Example 9.2. Let $\Omega = \{\mathrm{Len}(L, s(N)),\ \mathrm{Len}(L, s(N')),\ \mathrm{Even}(N)\}$. The preorder we define on atoms would be $\{\mathrm{Even}(N)\} < \{\mathrm{Len}(L, s(N)), \mathrm{Len}(L, s(N'))\}$. This preorder yields the order on variables $N < L < N'$. The set of atoms Ω is thus canonized as $\{\mathrm{Len}(X_2, s(X_1)),\ \mathrm{Len}(X_2, s(X_3)),\ \mathrm{Even}(X_1)\}$.

Note that this canonization is not perfect, i.e. there exists equivalent sets of atoms that are canonized differently. Literature on graph isomorphism [31] can be of use for improving the canonization.
We now define the simplification of a set of atoms Ω, which allows to remove redundant constraints while preserving and reflecting satisfiability (and the depth of the counterexamples). For this, it is sufficient to search for a substitution σ such that $\sigma(\Omega) \subseteq \Omega$. Indeed, suppose Ω_a and Ω_b two equisatisfiable sets of atoms ($\Omega_a \leq \Omega_b$ and $\Omega_b \leq \Omega_a$ using Definition 6.18). Then, by Lemma 6.19, there exists two substitutions σ_{ab} and σ_{ba} such that $\sigma_{ab}(\Omega_a) \subseteq \Omega_b$ and $\sigma_{ba}(\Omega_b) \subseteq \Omega_a$. By letting $\sigma_{aba} = \sigma_{ba} \circ \sigma_{ab}$, we have $\sigma_{aba}(\Omega_a) \subseteq \Omega_a$.

Definition 9.3 (Simplification of a set of atoms)**.** Let Ω be a set of atoms. A simplification of Ω is a set of atoms $\sigma(\Omega)$ for σ a substitution such that $\sigma(\Omega) \subseteq \Omega$.

We search for a renaming σ yielding a smallest subset $\sigma(\Omega)$. For simplicity, we restrict the search for such a substitution σ to those mapping variables to variables, i.e. a renaming of variables. A precise algorithm is not given here but has been implemented quite efficiently by noticing that, for any solution σ, $\sigma \circ \sigma$ is an at-least-as-good solution.

Example 9.4. Let $\Omega = \{\mathrm{Len}(L, s(N)),\ \mathrm{Len}(L, s(N')),\ \mathrm{Even}(N)\}$. The substitution $\sigma = \{N \mapsto N,\ L \mapsto L,\ N' \mapsto N\}$ is such that $\sigma(\Omega) = \{\mathrm{Len}(L, s(N)), \mathrm{Even}(N)\} \subseteq \Omega$.
Note that, if Ω also contains an atom $R(N')$ for some other relation R, then this same substitution σ would (fortunately) not yield a subset of Ω.

Finally, memoisation is done by keeping track of sets of atoms Ω and their already computed unfoldings $unfolds_{\mathcal{A}}(\Omega)$. More precisely, memoisation links a simplified and canonized set of atom to the splitting into independent atoms of each of its unfoldings. Memoisation is shared by all instances of $Inhabits^{det}_{\mathcal{A}}$ that run in parallel in *Teacher*, as relations are represented by the same model and thus have the same unfoldings.

9.1.2 Learner

The *learner* delegates the model finding to *Clingo* [13], a finite-model finder. For tree automata, this corresponds to the merging of states under constraints, while for SHoCs it completely infers the model.
The learner for tree automata is practically implemented as explained in Section 6.2. However, for SHoCs, we restrict the shape of the SHoCs that the learner can infer in

order to reduce its algorithmic complexity. These syntactic restrictions slightly reduce expressivity of SHoCs but have close to no effect on the procedure as a whole. This immediately has no effect on the teacher procedure, as it is defined and proven for any SHoCs. The Learner theorem 8.6 is preserved, except that inferred SHoCs S are minimal *among those that meet these syntactic restrictions*. The *Sat* procedure theorems (6.46 and 6.45) are not impacted either, except for Theorem 6.46 that undergo the same amendment than Theorem 8.6, namely that the learner's output SHoCs S_i are *among those meeting these syntactic restrictions*.

Definition 9.5 (Syntactic restrictions of SHoCs for faster learning)**.** Let $R(\vec{p}) \Leftarrow B$ be a SHoC generated by the learner.

1. For any $R'(\vec{X}) \in B$, $\vec{X}$ are in the same order as in the depth-first traversal of $\vec{p}$
2. For any distinct pair $R'(\vec{X'}) \in B$ and $R''(\vec{X''}) \in B$, $\vec{X'} \neq \vec{X''}$
3. For any $R'(\vec{X}) \in B$, every $X \in \vec{X}$ comes from a different pattern p of $\vec{p}$.
4. For any $R'(\vec{X}) \in B$, $|R'| \leq |R|$.

Compared to unrestricted SHoCs, some relations may need additional intermediate relations to be represented and some relations simply can not be represented anymore, such as $\{f(t,t) \mid t \in \mathcal{T}(\Sigma)\}$. In practice, however, these four restrictions do not have much of an impact on the inferred SHoCs for our benchmarks, apart from making the procedure quicker.
An other heuristic we use is thanks to the `#minimize` constraint that Clingo proposes. This constraint allows to ask to select, among valid models of minimal size, a model which minimizes a given function. We use this constraint to prioritize SHoCs with fewer clauses and without too much entanglement between relations.
Finally, we have a last easy optimisation. The input set of ground examples $\widehat{\Gamma}$ that the teacher receives may have implication constraints that can be statically resolved e.g. in Example 6.35, where $\widehat{\Gamma}_{ex} = \{\text{Len}(nil, z),\ \ \text{Len}(cons(z, nil), s(z)) \Leftarrow \text{Len}(nil, z)\}$ can be simplified into $\widehat{\Gamma}_{ex} = \{\text{Len}(nil, z),\ \ \text{Len}(cons(z, nil), s(z))\}$.

9.2 Benchmarks

To benchmark the Learner/Teacher procedure using either convoluted tree automata or SHoCs, we use examples coming from [19], where regular sets are sufficient to carry

out the proof, examples coming from [10, 6] where convoluted tree automata are necessary, and add some new examples which are out of the scope of convoluted tree automata, such as the property ϕ_3 from the introductory example of Section 1.1. All of our experimental results are available at `https://tlosekoo.gitlabpages.inria.fr/auto-forestation/index.html` All programs are defined as SMTLIB functions and are transformed into constrained clauses by our tool. On each example, the result of this translation as well as the found models can be consulted in the execution log of the tool.
We present how to read the execution log of our solver on the example *(length_reverse_eq.smt2)*, which asserts that $len\ (rev\ L) = len\ L$.

Example 9.6 (Reading the solver trace)**.**
The first section of the trace recalls the parameters that were used for the model-inference:

```
Inference procedure has parameters:
Timeout: Some(60.) (sec)
Approximation method: remove functionality constraint where possible
```

The second section shows the clauses defining the program and the properties, after presenting the typed alphabet.

```
Learning problem is:

env: {
elt -> {a, b}  ;  eltlist -> {cons, nil}  ;  nat -> {s, z}
}
definition:
{
(append, F:
{
append(nil, l2, l2) <= True
append(cons(h1, t1), l2, cons(h1, _a)) <= append(t1, l2, _a)
}
eq_eltlist(_d, _e) <= append(_b, _c, _d) /\ append(_b, _c, _e)
)
```

```
(reverse, F:
{
reverse(nil, nil) <= True
reverse(cons(h1, t1), _f) <= append(_g, cons(h1, nil), _f) /\ reverse(t1, _g)
}
eq_eltlist(_i, _j) <= reverse(_h, _i) /\ reverse(_h, _j)
)
(length, F:
{
length(nil, z) <= True
length(cons(x, ll), s(_k)) <= length(ll, _k)
}
eq_nat(_m, _n) <= length(_l, _m) /\ length(_l, _n)
)
}

properties:
{
eq_nat(_o, _p) <= length(_q, _p) /\ length(l1, _o) /\ reverse(l1, _q)
}
```

The following section states which relation can be over/under-approximated, and recalls the whole set of clauses from the approximated program and the properties.

```
over-approximation: {append, length, reverse}
under-approximation: {}

Clause system for inference is:

{
append(nil, l2, l2) <= True
length(nil, z) <= True
reverse(nil, nil) <= True
reverse(cons(h1, t1), _f) <= append(_g, cons(h1, nil), _f) /\ reverse(t1, _g)
append(cons(h1, t1), l2, cons(h1, _a)) <= append(t1, l2, _a)
```

```
eq_nat(_o, _p) <= length(_q, _p) /\ length(l1, _o) /\ reverse(l1, _q)
length(cons(x, ll), s(_k)) <= length(ll, _k)
}
```

Finally, the solving result is given by (a) the total time for the proof or counterexample generation ; (b) in the case of a successful proof, the solver provides the final SHoCs (which has been split relation by relation).
Note that SHoCs for equalities are omitted because they are canonical. This is the case here for the equality relation for elt, eltlist, and nat.

```
Solving took 34.501535 seconds.
Yes: |_

_r_1 ->
{
_r_1(cons(x_0_0, x_0_1), cons(x_1_0, x_1_1)) <= _r_1(x_0_1, x_1_1)
_r_1(nil, cons(x_1_0, x_1_1)) <= _r_3(x_1_1)
}
;
_r_2 ->
{
_r_2(cons(x_0_0, x_0_1)) <= True
}
;
_r_3 ->
{
_r_3(nil) <= True
}
;
append ->
{
append(cons(x_0_0, x_0_1), cons(x_1_0, x_1_1), cons(x_2_0, x_2_1)) <= _r_1(x_0_1, x_2_1)
append(cons(x_0_0, x_0_1), cons(x_1_0, x_1_1), cons(x_2_0, x_2_1)) <= _r_2(x_1_1)
append(cons(x_0_0, x_0_1), nil, cons(x_2_0, x_2_1)) <= True
append(nil, cons(x_1_0, x_1_1), cons(x_2_0, x_2_1)) <= reverse(x_1_1, x_2_1)
```

```
append(nil, nil, nil) <= True
}
;
length ->
{
length(cons(x_0_0, x_0_1), s(x_1_0)) <= length(x_0_1, x_1_0)
length(nil, z) <= True
}
;
reverse ->
{
reverse(cons(x_0_0, x_0_1), cons(x_1_0, x_1_1)) <= reverse(x_0_1, x_1_1)
reverse(nil, nil) <= True
}

--
Equality SHoCs are defined for: {elt, eltlist, nat}
_|
```

After giving the final solver's answer, the detail of each step of the Learner/Teacher loop is given, much like the Example 4.14.
The log for our procedure using convoluted automata looks similar. Here is the right-convoluted automaton proving this same property. Please do not try to decipher it too hard, this is just given as a motivation for SHoCs' ease of use.

```
Model:
|_
{
append ->
{{{
Q={q_gen_10, q_gen_102, q_gen_103, q_gen_110, q_gen_111, q_gen_129,
  q_gen_131, q_gen_17, q_gen_22, q_gen_23, q_gen_24},
Q_f={q_gen_10, q_gen_102},
Delta=
{
<cons>(q_gen_23, q_gen_129) -> q_gen_129
```

```
<cons>(q_gen_23, q_gen_22) -> q_gen_129
<nil>() -> q_gen_22
<a>() -> q_gen_23
<b>() -> q_gen_23
<cons, cons>(q_gen_111, q_gen_110) -> q_gen_110
<nil, cons>(q_gen_23, q_gen_22) -> q_gen_110
<a, a>() -> q_gen_111
<a, b>() -> q_gen_111
<b, a>() -> q_gen_111
<b, b>() -> q_gen_111
<cons, cons>(q_gen_111, q_gen_131) -> q_gen_131
<nil, cons>(q_gen_23, q_gen_129) -> q_gen_131
<cons, cons>(q_gen_111, q_gen_17) -> q_gen_17
<nil, nil>() -> q_gen_17
<cons, nil, cons>(q_gen_111, q_gen_131) -> q_gen_10
<cons, nil, cons>(q_gen_111, q_gen_17) -> q_gen_10
<nil, cons, cons>(q_gen_111, q_gen_17) -> q_gen_10
<nil, nil, cons>(q_gen_23, q_gen_129) -> q_gen_10
<nil, nil, nil>() -> q_gen_10
<cons, cons, cons>(q_gen_24, q_gen_102) -> q_gen_102
<cons, cons, cons>(q_gen_24, q_gen_103) -> q_gen_102
<cons, nil, cons>(q_gen_111, q_gen_110) -> q_gen_102
<nil, nil, cons>(q_gen_23, q_gen_22) -> q_gen_102
<cons, cons, cons>(q_gen_24, q_gen_10) -> q_gen_103
<nil, cons, cons>(q_gen_111, q_gen_110) -> q_gen_103
<nil, cons, cons>(q_gen_111, q_gen_131) -> q_gen_103
<a, a, a>() -> q_gen_24
<a, b, a>() -> q_gen_24
<b, a, b>() -> q_gen_24
<b, b, b>() -> q_gen_24
}

Datatype: <eltlist, eltlist, eltlist>
Convolution form: right
}}}
  ;  length ->
```

```
{{{
Q={q_gen_1, q_gen_14},
Q_f={q_gen_1},
Delta=
{
<a>() -> q_gen_14
<b>() -> q_gen_14
<cons, s>(q_gen_14, q_gen_1) -> q_gen_1
<nil, z>() -> q_gen_1
}

Datatype: <eltlist, nat>
Convolution form: right
}}}
  ;  reverse ->
{{{
Q={q_gen_0, q_gen_100, q_gen_12, q_gen_28, q_gen_29},
Q_f={q_gen_0},
Delta=
{
<cons>(q_gen_29, q_gen_28) -> q_gen_28
<nil>() -> q_gen_28
<a>() -> q_gen_29
<b>() -> q_gen_29
<cons, cons>(q_gen_12, q_gen_0) -> q_gen_0
<nil, nil>() -> q_gen_0
<cons, cons>(q_gen_12, q_gen_100) -> q_gen_100
<cons, nil>(q_gen_29, q_gen_28) -> q_gen_100
<nil, cons>(q_gen_29, q_gen_28) -> q_gen_100
<a, a>() -> q_gen_12
<a, b>() -> q_gen_12
<b, a>() -> q_gen_12
<b, b>() -> q_gen_12
}

Datatype: <eltlist, eltlist>
```

```
Convolution form: right
}}}

}
--
Equality automata are defined for: {eq_elt, eq_eltlist, eq_nat}
_|
```

General results

On a total of 174 examples, our solver using convoluted automata proves 78, disproves 30, and timeouts on 66 after 30 seconds. Using SHoCs, it proves 106, disproves 31, and timeouts on 37 after 30 seconds. Further increasing the timeout does not have much of an impact. The 78 positive and negative examples verified by using convoluted automata can also be verified using SHoCs, except that some of them need more time. Our solver, using left-convoluted automata, succeeds on 24 out of the 79 first-order Isaplanner [10] examples in less than 5s. Our approach and [34] are rather complementary on our benchmarks as they succeed on different (not disjoint) sets of examples. This can be observed on the IsaPlanner benchmark where our technique fails on most of examples that [34] handles (i.e. 4, 5, 29, 30, 39, 50, 62, 67, 71, 86) and succeeds on examples on which they do not report any success (i.e. 10, 11, 17, 18, 20, 21, 22, 23, 24, 25, 31, 32, 33, 34, 45, 65, 68, 69).

Spacer [17], Eldarica [20], and RInGen [33] are powerful general purpose SMT solvers with some ADT support. However, this support for ADTs does not cover relational properties and, unsurprisingly, those solvers do not perform well on positive examples of this benchmark. We used a 60s timeout for Spacer, Eldarica (with a RInGen pre-processing), and RInGen (with CVC4 finite model finding). The following table sums up the results for positive and negative examples.

Among syntactic solvers, discussed in Section 3.1, we ran a recent solver that uses automatic induction [32] and that has been implemented in cvc5. This solver also targets (among others) algebraic properties of functional programs. Their solver performs differently than ours, as the approach is significantly different, and succeeds on many interesting relational positive properties.

	SHoCs	Convoluted automata	Spacer	Eldarica	RInGen	[32]
Positive (/143)	106	78	15	9	36	96
Negative (/32)	31	30	4	31	31	3
Unique positive	10	2	0	0	0	22

Since RInGen is based on regular model inference [27], it succeeds on benchmarks having a regular model but fails on examples needing relational information. More precisely, a non-relational solver such as [19] can also handle a restricted form of relations: the finite union of languages $\mathcal{L}_1 \times \ldots \times \mathcal{L}_n$ where $\forall i \in [1 \ldots n]$, $\mathcal{L}_i$ is a regular language. This allows to prove properties with a limited form of relation. For instance, using a non-relational regular solver, it is possible to prove the property $less\ z\ (len\ L_1) \Rightarrow\ less\ z\ (len\ (append\ L_1\ L_2))$. For the tuple of variables (L_1, L_2) to cover all the possible cases, it is enough to consider the two languages $\mathcal{L}_{nil} \times \mathcal{L}_{lists}$ and $\mathcal{L}_{Cons+} \times \mathcal{L}_{lists}$ where $\mathcal{L}_{nil} = \{Nil\}$ and $\mathcal{L}_{Cons+} = \mathcal{L}_{lists} \setminus \mathcal{L}_{nil}$. With the first language, the property is true because the left-hand side of the implication is false. With the second language $\mathcal{L}_{Cons+} \times \mathcal{L}_{list}$, both the left and right-hand side of the implication are true. One of the simplest problem which cannot be proved using a non-relational solver is $cons(N, L) \neq L$, as proving such a property cannot be done using a *finite* union of products of regular languages.
Finally, as discussed in Section 3.1, the syntactic proof system defined by Unno *et al.* [36] also targets (among others) algebraic properties of functional programs. As announced, we compare it with our approach on some examples. Their solver performs succeeds on many interesting relational properties that are both positive and negative. It is harder to compare to ours as it does not take SMTLIB as input format. However, we wrote the examples of the Section 9.2.1 in Caml, which it accepts. On these examples, their solver succeeds on properties 1, 6, and 8, but timeouts on the others. On the other hand, their solver is capable to prove $min\ (plus\ N_1\ N_2)\ N_2 = N_2$ whereas ours can not, as the relation *plus* can not be represented by a SHoCs.

9.2.1 Zoom in on some benchmarks

We first present some properties that can and cannot be proven using convoluted automata and then using SHoCs. The properties that can be proven using convoluted automata can also be proven using SHoCs, but convoluted automata are still surpris-

ingly efficient for some properties.

Convoluted automata

Convoluted automata allow to prove many relational properties that could not be proven by aforementioned non-relational solvers. Because the properties of our benchmarks were mostly either on same-type relations or on lists and natural numbers, the right-convolution was the most efficient convolution. Left-convolution is not adapted for most of the list-based examples and complete-convolution revealed to be too costly in practice though it theoretically allows to prove properties on functions manipulating binary trees. Finally, on examples where using a non-relational model suffices to prove the property, our solving technique is often flexible enough to find such a model, with an efficiency comparable to non-relational solvers.
We present in this section some examples on which our solver, using tree automata, succeeds and fails. For these examples, variables named E represent elements of a two-element domain $\{a, b\}$, L represent lists over this two-element domain, and T binary trees over this two-element domain.
The *Sat* procedure using the right-convolution allows to prove the following positive properties:

- $len\ L = len\ (reverse\ L)$
- $prefix\ L_1\ (append\ L_1\ L_2)$
- $len\ L = len\ (insertionSort\ L)$

It also is able to find a counterexample on the following negative properties:

- $N < (double\ N)$
- $(delete_one\ X\ L) = (delete_all\ X\ L) \Rightarrow (count\ X\ L) = 1$

Finally, here are some examples on which our solver does not terminate due to trying to represent a relation for which a right-convoluted automaton is not expressive enough:

- $N_1 + N_2 = N_2 + N_1$
- $size\ T_1 < size\ (node\ N\ T_1\ T_2)$
- $heightRB\ T \leq height\ T$
- $flip\ (flip\ T) = T$

Note that lists and trees of the above-mentioned properties are over a two-element domain. These positive properties can also theoretically be proven on an infinite domain, such as natural numbers, still by using right-convolution. However, because convoluted automata must consider the terms up to their leaves, this increases the verification cost and makes it unpractical.
The *Sat* procedure does not solve any of the above examples when using left-convolution due to a lack in expressiveness. Using the complete-convolution compensates this lack of expressiveness (except for properties $N_1 + N_2 = N_2 + N_1$ and $size\ T_1 < size\ (node\ N\ T_1\ T_2)$), but still does not allow to prove any of the above positive properties due to generating too-big terms.

Shallow Horn Clauses

In this section, we show some examples on which our solver, using SHoCs, succeeds and fails. For these examples, variables named N represent natural numbers, L represent lists over natural numbers, and T binary trees over natural numbers. For a detailed SMTLIB implementation of functions and properties of examples, see the files linked by their name.

Successful positive examples

1. $len\ L_1 \leq len\ (append\ L_1, L_2)$ *(append_length_leq_nat.smt2)*
2. $height\ T \leq size\ T$ *(tree_nat_depth_leq_size.smt2)*
3. $heightRB\ T \leq height\ T$ *(tree_height_heightRB.smt2)*
4. $subtree\ T_1\ T_2 \wedge subtree\ T_2\ T_3 \Rightarrow subtree\ T_1\ T_3$ *(tree_strict_subtree_trans.smt2)*
5. $len\ L = len\ (insertionSort\ L)$ *(isaplanner_prop20.smt2)*
6. $flip\ (flip\ T) = T$ *(tree_flip_twice.smt2)*
7. $len\ L = len\ (reverse\ L)$ *(length_reverse_eq_nat.smt2)*
8. $prefix\ L_1\ (append\ L_1\ L_2)$ *(prefix_append.smt2)*

Although the property 1 is in the scope of convoluted tree automata, our solver only proves it on lists on natural numbers when using SHoCs. As mentioned in Chapter 7, models represented with SHoCs are generic. In the case of this property, the SHoCs proving it for *nat* list and for lists of $a's$ and $b's$ are the same. The property 2 is out of the scope of convoluted automata with either left- or right-convolution and also

takes advantage of the genericity of SHoCs. Interestingly, the solver builds an over-approximation of the Height relation which is sufficient to prove ϕ_3. When functions are translated into relations, ϕ_3 becomes $\text{HeightRB}(T, N) \land \text{Height}(T, M) \Rightarrow \text{Less}(N, M)$, where HeightRB and Height occurs only on the left-hand side of the implication and using over-approximations for these relations for proving this property is safe. The property 3 is the property ϕ_3 from Section 1.1. Properties 4 and 5 are other challenging properties. In particular, since *insertionSort* is defined using an intermediate function, proving this property using, e.g., automatic induction would require to guess and prove non-trivial intermediate lemmas.

Successful negative examples On the following properties our solver is able to find a counterexample.

- $N < (double\ N)$ *(nat_double_is_le.smt2)*
- $(delete_one\ X\ L) = (delete_all\ X\ L) \Rightarrow (count\ X\ L) = 1$ *(list_delete_all_count.smt2)*

Unsuccessful positive examples On the following properties, our solver does not terminate due to trying to represent a relation for which the SHoCs are not expressive enough.

- $(delete_one\ X\ L) = (delete_all\ X\ L) \Rightarrow (count\ X\ L) \leq 1$ *(list_delete_all_count.smt2)*
- $N_1 + N_2 = N_2 + N_1$ *(plus_commutative.smt2)*
- $mem\ X\ L \Rightarrow mem\ X\ (reverse\ L)$ *(mem_reverse.smt2)*
- $size\ T_1 < size\ (node\ T_1\ N\ T_2)$ *(tree_size_le_node.smt2)*

Finally, those experiments also reveal that the learner may find a correct model that our (incomplete) teacher may fail to accept. This can be observed in the log of the example *(tree_height_max_node.smt2)*, which states that $(height\ (node\ T1\ N\ T2)) = (s\ (max\ (height\ T1)\ (height\ T2)))$, where the last SHoCs proposed by the teacher for *height* is exactly the SHoCs of Section 7.4 but it is not accepted by the teacher. This property can be proven by splitting the *height* function into two parts, see example *(tree_shallow_taller_node.smt2)*.

CHAPTER 10

CONCLUSION AND PERSPECTIVES

10.1 Conclusion

Summary The goal of this thesis was to automatically prove relational properties on tree-manipulating functional programs. We presented two formalisms to represent tree-tuple languages: convoluted tree automata, known in the literature only with padding, and shallow Horn clauses that we proposed in this thesis. The program and its properties are first translated from the high-level language SMTLIB to a set of clauses. This set of clauses is then relaxed using our approximation framework. Convoluted tree automata and SHoCs have then been used to represent models of this set of clauses describing both the (approximated) program and the properties. The generic Learner/Teacher procedure for inferring a model of a set of clauses has been instantiated twice, using convoluted automata and then SHoCs. In both cases, the procedure is correct, and relatively complete. We implemented these procedures, together with practical improvements, and benchmarked them against relational properties on recursively-defined functions. The benchmarks confirm the adequacy of model exhibition as a promising verification technique for a certain type of relational properties and the efficiency of the procedures defined in this thesis. These contributions show that the inference of a finite representation of Herbrand models to prove satisfiability of a set of clauses can be effective compared to state-of-the-art techniques.

Takeaway Convoluted automata can indeed be used to represent recursive functions to aid automated proof and can be practically inferred. With the right-convolution, our implementation successfully represented approximations of functions such as *len*, *append*, *heightRB*, *height*, *insert_sort* that are sufficient to prove some non-trivial relational properties. Shallow Horn clauses are a more generic formalism to represent recursive functions. They are strictly more expressive than tree automata and convoluted tree automata. Moreover, they can also be inferred from examples and are able

to more-precisely represent those relations. Generally, SHoCs are very efficient at representing relations that can be defined by recursive calls to immediate children of its parameters without needing to store data.

Comparison with automatic induction Model exhibition behaves very differently from syntactic proof and the properties that these two paradigms can prove are rather different. The model inference method is often more effective than automatic induction as soon as intermediary lemmas are required to syntactically prove the property. For example, proving that the insertion-sort function preserves the length of its input is hard with automatic induction, as the insertion sort uses intermediate functions and requires intermediate lemmas. Many such properties can be found in the benchmarks or in Section 9.2.1. However, for this model inference method to work, a sufficient approximation of the relation must be representable using the model formalism, and we must be able to validate the model against the properties, which is not always the case. For example, our implementation is powerful enough to prove that flipping a tree twice is the identity, i.e. $flip\ (flip\ (T)) = T$, but not that flipping a tree twice preserves its height, i.e. $height\ (flip\ (flip\ (T))) = height\ T$, which should be an immediate consequence for syntactic proofs.

Comparison with SMT-solvers SMT-solvers that support algebraic datatypes, such as Spacer, Eldarica, or RInGen, perform well on non-relational properties. However, their non-relational model representation fails to prove relational properties. Thus, they fail on most of the benchmarks proposed in this thesis, as shown in Section 9.2.

10.2 Perspectives

Teacher improvement The current teacher can be improved. For example, the property $height\ (flip\ (flip\ (T))) = height\ T$ cannot be automatically proven although an exact representation of $height$ and $flip$ is represented in the final SHoCs that the learner builds (which is necessarily the case because we know that there exists a SHoCs that represents these functions and because of the Learner/Teacher relative completeness Theorem 6.46).
Currently, the teacher searches for a substitution by a breadth-first search approach [1].

1. In practice we implemented a few heuristics but do not deviate much from the breadth-first search

This is convenient for proving its (relative) completeness but not necessary. Similarly to SMT-solvers, a conflict-driven clause-learning algorithm (with, here, a bound on the explored depth) could significantly improve the capabilities of the Teacher. More generally, there may exist more efficient ways of checking the clauses against a convoluted tree automaton or SHoCs.

Clause approximation Our approximation of the set of clauses representing both the program and properties is not satisfying. Whether it be by modification of the set of clauses or by a different interpretation of them, approximation should be improved. Indeed, we analyse the clauses to know which relations can be approximated and how, but our approximation method then only over-approximates relations defined from boolean functions, and can not over-approximate other relations nor under-approximate. Moreover, our method allows some over-approximations but not all of them. For example, the relation $\{(t, n) \mid t \text{ is a tree of height } n\} \cup \{(leaf, s(z))\}$ is a correct over-approximation of the Height relation but cannot be obtained using our clause-modification method. A better approximation framework would allow for any over-approximation and any under-approximation (when it is safe to over-/under- approximate). Moreover, it may be worth trying to have one over-approximation and one under-approximation per relation.

Support for other theories Currently, this method can only handle the theory of recursive algebraic datatypes, which is limiting when trying to prove real-world programs. Support for an other theory, for example computer integers, seems to fit into this framework when using SHoCs. The SHoC formalism would need to allow theory-specific variables in its head and theory-specific predicates in its body. The teacher would need to be extended by using a SMT-solver for the theory when the unfolding of an atom contains such a theory-specific predicate. The learner could use a predefined set of predicates on this theory to be used in the body of its inferred models. Moreover, some theory-specific predicates should also be labeled as "deconstructors", allowing to use variables that were not in the head. Otherwise, the SHoC constraint that every variable from the body appears in the head would be too restricting. Each theory-specific head variable could then be used in exactly one deconstructor, which corresponds precisely to our treatment of algebraic datatypes. For example, the binary -1 relation, relating a known integer n to $n-1$, can be labeled as a deconstructor,

as the constraint $-1(N, N')$ allows to uniquely determine N' from N (and reduces its size). The Double relation could then be realised with the predefined theory-specific predicates and deconstructors *is*0, -1, and -2:

$\text{Double}(X, Y) \Leftarrow is0(X) \wedge is0(Y)$

$\text{Double}(X, Y) \Leftarrow \text{Double}(X', Y') \wedge -1(X, X') \wedge -2(Y, Y')$.

The HeightRB relation can also be written using proper integers by

$\text{HeightRB}(leaf, N) \Leftarrow is0(N)$

$\text{HeightRB}(node(T_1, E, T_2), N) \Leftarrow -1(N, N') \wedge \text{HeightRB}(T_2, N')$.

Polymorphism Polymorphism is currently not handled by this solver. Relying on the whole polymorphic program, a polymorphic function can be instantiated on every datatype it is applied to, thus allowing to prove it using monomorphic techniques, which is not very satisfying. A possible future work is to reuse techniques for reducing the proof of a polymorphic program to that of a monomorphic one [3]. Moreover, SHoCs allow for polymorphism by allowing variables from the head to not appear in the body, as showed in Example 7.5.

Bibliography

[1] Clark Barrett, Pascal Fontaine, and Cesare Tinelli, *The Satisfiability Modulo Theories Library (SMT-LIB)*, `www.SMT-LIB.org`, 2016.

[2] Clark Barrett, Igor Shikanian, and Cesare Tinelli, « An abstract decision procedure for a theory of inductive data types », *in*: *Journal on Satisfiability, Boolean Modeling and Computation* 3.1-2 (2007), pp. 21–46.

[3] Jean-Philippe Bernardy, Patrik Jansson, and Koen Claessen, « Testing polymorphic properties », *in*: *Programming Languages and Systems: 19th European Symposium on Programming, ESOP 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings 19*, Springer, 2010, pp. 125–144.

[4] Jean Berstel, *Transductions and context-free languages*, vol. 38, Teubner Studienbücher : Informatik, Teubner, 1979, ISBN: 3519023407, URL: `https://www.worldcat.org/oclc/06364613`.

[5] Nikolaj Bjørner et al., « Horn clause solvers for program verification », *in*: *Fields of Logic and Computation II*, Springer, 2015, pp. 24–51.

[6] Koen Claessen et al., *TIP and IsaPlanner benchmarks*, `https://tip-org.github.io/`, 2015.

[7] Hubert Comon et al., *Tree Automata Techniques and Applications*, 2008, p. 262, URL: `https://hal.inria.fr/hal-03367725`.

[8] Loris D'Antoni and Margus Veanes, « The power of symbolic automata and transducers », *in*: *Computer Aided Verification: 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I 30*, Springer, 2017, pp. 47–67.

[9] Max Dauchet and Sophie Tison, « The theory of ground rewrite systems is decidable », *in*: *[1990] Proceedings. Fifth Annual IEEE Symposium on Logic in Computer Science*, IEEE, 1990, pp. 242–248.

[10] Lucas Dixon and Jacques Fleuriot, « IsaPlanner: A Prototype Proof Planner in Isabelle », *in*: *CADE'03*, vol. 2741, Springer, 2003, pp. 279–283.

[11] Maarten H. van Emden and Robert A. Kowalski, « The Semantics of Predicate Logic as a Programming Language », *in*: *J. ACM* 23.*4* (1976), pp. 733–742.

[12] Pranav Garg et al., « ICE: A robust framework for learning invariants », *in*: *International Conference on Computer Aided Verification*, Springer, 2014, pp. 69–87.

[13] Martin Gebser et al., *Answer Set Solving in Practice*, Synthesis Lectures on Artificial Intelligence and Machine Learning, Morgan & Claypool Publishers, 2012.

[14] Thomas Genet, « Termination criteria for tree automata completion », *in*: *Journal of Logical and Algebraic Methods in Programming* 85.*1* (2016), pp. 3–33.

[15] Thomas Genet, Timothée Haudebourg, and Thomas Jensen, « Verifying Higher-Order Functions with Tree Automata », *in*: *International Conference on Foundations of Software Science and Computation Structures*, Springer, 2018, pp. 565–582.

[16] Erich Grädel, « Automatic structures: twenty years later », *in*: *Proceedings of the 35th Annual ACM/IEEE Symposium on Logic in Computer Science*, 2020, pp. 21–34.

[17] Arie Gurfinkel, « Program Verification with Constrained Horn Clauses », *in*: *CAV'22*, vol. 13371, LNCS, Springer, 2022, pp. 19–29.

[18] Timothée Haudebourg, « Automatic Verification of Higher-Order Functional Programs using Regular Tree Languages », PhD thesis, Univ. Rennes1, 2020.

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

[20] Hossein Hojjat and Philipp Rümmer, « The ELDARICA Horn Solver », *in*: *FM-CAD'18*, `https://github.com/uuverifiers/eldarica/`, IEEE, 2018, pp. 1–7.

[21] Neil D Jones, « Flow analysis of lazy higher-order functional languages », *in*: *Abstract Interpretation of Declarative Languages*, 1987, pp. 103–122.

[22] Neil D Jones and Nils Andersen, « Flow analysis of lazy higher-order functional programs », *in*: *Theoretical Computer Science* 375.*1-3* (2007), pp. 120–136.

[23] Neil D Jones and Steven S Muchnick, « Flow analysis and optimization of LISP-like structures », *in*: *Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, 1979, pp. 244–256.

[24] Bakhadyr Khoussainov and Anil Nerode, « Automatic Presentations of Structures », *in*: *International Workshop on Logic and Computational Complexity*, Springer, 1994, pp. 367–392.

[25] Gerwin Klein et al., « seL4: Formal verification of an OS kernel », *in*: *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, 2009, pp. 207–220.

[26] Naoki Kobayashi, Ryosuke Sato, and Hiroshi Unno, « Predicate abstraction and CEGAR for higher-order model checking », *in*: *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, 2011, pp. 222–233.

[27] Yurii Kostyukov, Dmitry Mordvinov, and Grigory Fedyukovich, « Beyond the elementary representations of program invariants over algebraic data types », *in*: *PLDI'21*, ed. by Stephen N. Freund and Eran Yahav, ACM, 2021, pp. 451–465.

[28] Sébastien Limet and Gernot Salzer, « Tree tuple languages from the logic programming point of view », *in*: *Journal of Automated Reasoning* 37 (2006), pp. 323–349.

[29] T. Losekoot, T. Genet, and T. Jensen, « Automata-based Verification of Relational Properties of Functions over Data Structures », *in*: *FSCD'23*, vol. 260, LIPIcs, 2023.

[30] Yuma Matsumoto, Naoki Kobayashi, and Hiroshi Unno, « Automata-based abstraction for automated verification of higher-order tree-processing programs », *in*: *Asian Symposium on Programming Languages and Systems*, Springer, 2015, pp. 295–312.

[31] Brendan D McKay et al., « Practical graph isomorphism », *in*: (1981).

[32] Andrew Reynolds and Viktor Kuncak, « Induction for SMT solvers », *in*: *International Workshop on Verification, Model Checking, and Abstract Interpretation*, Springer, 2015, pp. 80–98.

[33] *Regular Invariant Generator*, `https://github.com/Columpio/RInGen`, 2021.

[34] Takumi Shimoda et al., « Symbolic Automatic Relations and Their Applications to SMT and CHC Solving », *in*: *International Static Analysis Symposium*, Springer, 2021, pp. 405–428.

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

[36] Hiroshi Unno, Sho Torii, and Hiroki Sakamoto, « Automating induction for solving horn clauses », *in*: *International Conference on Computer Aided Verification*, Springer, 2017, pp. 571–591.

APPENDIX A

APPENDIX

A.1 Omitted proofs for the Teacher

This section is a collection of lemmas and proofs for the *Teacher* procedure and of definitions used only in them.

Definition A.1 (Set of patterns of atoms)**.** We write $Patterns(q(p_1, \ldots, p_n)) = \{p_1, \ldots, p_n\}$ the set of patterns of an atom $q(p_1, \ldots, p_n)$ and $Patterns(\Omega) = \bigcup_{\omega \in \Omega} Patterns(\omega)$ for the set of patterns that appear in a set of atoms Ω.

The unfolding of a tuple of patterns along a transition only uses direct children. This intuition is transcribed in the following Proposition A.2.

Proposition A.2 ($unfoldOne$ patterns)**.** *Let $\vec{p}$ be a tuple of patterns and r a transition of some automaton such that $unfoldOne(\vec{p}, r)$ is defined. Then*

$$Patterns(unfoldOne(\vec{p}, r)) \subseteq \{p[\pi] \mid p \in \vec{p} \wedge \pi \in Pos(p) \wedge |\pi| = 1\}$$

Lemma 6.11 ($unfolds$)**.** *Let $\Omega = \{\omega_1, \ldots, \omega_n\}$ be a set of atoms, $T = \{(\omega_1, r_1), \ldots, (\omega_n, r_n)\}$ be a set of pairs of an atom and a transition of some automaton $\mathcal{A}$, assigning a transition to each atom ω_i of Ω. Suppose $unfolds(T) = Some(\Omega', \sigma_\delta)$. Then for all substitution σ such that $\mathcal{A} \models \sigma(\Omega')$, we have $\mathcal{A} \models \sigma(\sigma_\delta(\Omega))$.*

Proof. Suppose $\mathcal{A} \models \sigma(\Omega')$ and let $\omega_i = q_i(\vec{p_i}) \in \Omega$. We now prove that $\mathcal{A} \models \sigma(\sigma_\delta(\omega_i))$. Let $r_i = \langle f_1, \ldots, f_n \rangle(\vec{q}) \to q'_i$ be the transition associated to ω_i in T. Because $unfolds(T)$ returned $Some(\Omega', \sigma_\delta)$, we know that that $q'_i = q_i$ and that $\sigma_\delta(\vec{p_i})$ is of the shape $(f_1(\overrightarrow{p'_1}), \ldots, f_n(\overrightarrow{p'_n}))$, as σ_δ is a unifier of $compatibility(\vec{p_i}, (f_1, \ldots, f_n))$. This means that r_i can be applied to recognize $\sigma_\delta(\vec{p_i})$ in q_i or, more precisely, that for any substitution σ, $\mathcal{A} \models \sigma(\sigma_\delta(\omega_i))$ is implied by $\mathcal{A} \models \sigma(unfoldOne(\sigma_\delta(\vec{p_i}), r_i))$. We conclude by noticing $unfoldOne(\sigma_\delta(\vec{p_i}), r_i) \subseteq \Omega'$ and $\mathcal{A} \models \sigma(\Omega')$. □

Lemma A.3 ($Inhabits_{\mathcal{A}}$ ground bounded termination). *Let Ω be a set of atoms and $\mathcal{A}$ a convoluted automaton. If there exists a substitution σ such that $Vars(\sigma(\Omega)) = \emptyset$ and $\mathcal{A} \models \sigma(\Omega)$, then there exists an execution of $Inhabits_{\mathcal{A}}(\Omega)$ which stops in at most $ht(\sigma(\Omega))$ steps by returning $Some(_)$.*

Proof. Let σ be a substitution such that $Vars(\sigma(\Omega)) = \emptyset$ and $\mathcal{A} \models \sigma(\Omega)$.
Let $P(i)$ be the property that there exists an execution of $Inhabits_{\mathcal{A}}(\Omega)$ that either (a) terminated with $Some(_)$ before step i or else (b) reached step i and such that there exists a substitution σ' with $Vars(\sigma'(\Omega_i)) = \emptyset$, $\mathcal{A} \models \sigma'(\Omega_i)$, and $ht(\sigma'(\Omega_i)) \leq ht(\sigma(\Omega)) - i$. Recall that Ω_i and σ_i are the set of atoms and substitution built at step i of the $Inhabits_{\mathcal{A}}$ execution.
Then, $P(i)$ can be proven by induction on i:

- $\boldsymbol{i = 0}$ **:** Immediate by satisfying the (b) condition by taking $\sigma' = \sigma$.
- $\boldsymbol{i > 0}$ **:** Suppose $P(i)$ true.

  If there exists an execution of $Inhabits_{\mathcal{A}}(\Omega)$ that terminates with $Some(_)$ before reaching step i, then it also does so before reaching step $i+1$ so $P(i+1)$ is true.

  Otherwise, suppose that there exists an execution of $Inhabits_{\mathcal{A}}(\Omega)$ that reaches step i and such that there exists a substitution σ' with $Vars(\sigma'(\Omega_i)) = \emptyset$, $\mathcal{A} \models \sigma'(\Omega_i)$, and $ht(\sigma'(\Omega_i)) \leq ht(\sigma(\Omega)) - i$.

  Now, either $\Omega_i = \emptyset$ and then the execution terminates at this step i with $Some(\sigma_i)$ and so $P(i+1)$ is true, or else $\Omega_i \neq \emptyset$. In the case $\Omega_i \neq \emptyset$, using all induction hypotheses, there then exists a transition r_ω for each atom $\omega \in \Omega_i$ such that, with $T = \{(\omega, R_\omega) \mid \omega \in \Omega_i\}$, $unfolds(T) = Some(\Omega_{i+1}, \sigma_\delta)$, with Ω_{i+1} admitting a substitution σ'' such that $\sigma' = \sigma_\delta \circ \sigma''$, $Vars(\sigma''(\Omega_{i+1})) = \emptyset$, $\mathcal{A} \models \sigma''(\Omega_{i+1})$, and $ht(\sigma''(\Omega_{i+1})) < ht(\sigma'(\Omega_i))$. So $P(i+1)$ is true.

Now, let $i = 1 + ht(\sigma(\Omega))$. Because $P(i)$ is true, we know that there exists an execution of $Inhabits_{\mathcal{A}}(\Omega)$ that either returned $Some(_)$ at some step (strictly) before i, so at at most step $i - 1 = ht(\sigma(\Omega))$, or that reached step i and such that there exists a substitution σ' with, among other properties, $ht(\sigma'(\Omega_i)) \leq ht(\sigma(\Omega)) - i$. However, $ht(\sigma(\Omega)) - i < 0$, so no such substitution σ' can exist. Therefore this execution of $Inhabits_{\mathcal{A}}(\Omega)$ returned $Some(_)$ in at most $ht(\sigma(\Omega))$ steps. □

Theorem 6.15 ($Inhabits_{\mathcal{A}}$ correctness and relative completeness). *Let $\mathcal{A}$ be an automaton and Ω a set of atoms.*

- ***Correctness:*** *If $Inhabits_{\mathcal{A}}(\Omega)$ terminates with $Some(\sigma)$, then $\mathcal{A} \models \sigma(\Omega)$.*
- ***Relative completeness:*** *If there exists a substitution σ such that $\mathcal{A} \models \sigma(\Omega)$, then there exists a terminating execution of $Inhabits_{\mathcal{A}}(\Omega)$ returning $Some(_)$.*

Proof. Relative completeness is immediate from Lemma A.3 and the fact that we impose to any datatype to be inhabited .
Intuitively, correctness stems from the fact that each new set of atoms Ω_{i+1} comes from Ω_i by applying one transition to each atom and keeping track of the corresponding substitution σ_{i+1}, so a proof that Ω_{i+1} can be satisfied in $\mathcal{A}$ can be transformed into a proof that Ω_i can be satisfied in $\mathcal{A}$.
More formally, let an execution of $Inhabits_{\mathcal{A}}(\Omega)$ be. For any $i \in \mathbb{N}$, let the property $P(i)$ be "If the execution reached step i, then for any substitution σ, $\mathcal{A} \models \sigma(\Omega_i)$ implies $\mathcal{A} \models \sigma(\sigma_i(\Omega))$". We prove $P(i)$ by induction on the number i of steps taken:

- $\boldsymbol{i = 0}$ **:** $P(0)$ states that for every substitution σ, $\mathcal{A} \models \sigma(\Omega)$ implies $\mathcal{A} \models \sigma(\Omega)$.
- $\boldsymbol{i > 0}$ **:** Suppose the execution reached step i and $P(i)$ is true. Then either it terminates at step i and never reaches step $i+1$, so $P(i+1)$ immediately holds, or else it reaches step $i+1$. Suppose that the execution reached step $i+1$, so we have $Some(\Omega_{i+1}, \sigma_\delta) \in unfolds(T)$ for $T = \{(\omega, r_\omega) \mid \omega \in \Omega_i\}$ with every r_ω some transition of the automaton $\mathcal{A}$, and $\sigma_{i+1} = \sigma_\delta \circ \sigma_i$. By Lemma 6.11, we have that for any substitution σ, $\mathcal{A} \models \sigma(\Omega_{i+1})$ implies $\mathcal{A} \models \sigma(\sigma_\delta(\Omega_i))$. By using the induction hypothesis $P(i)$ on substitution $\sigma \circ \sigma_\delta$, we have $\mathcal{A} \models \sigma(\sigma_\delta(\Omega_i))$ implies $\mathcal{A} \models \sigma(\sigma_\delta(\sigma_i(\Omega)))$. Chaining these two implications yields that for any substitution σ, $\mathcal{A} \models \sigma(\Omega_{i+1})$ implies $\mathcal{A} \models \sigma(\sigma_{i+1}(\Omega))$, so $P(i+1)$ holds.

□

The following Lemma states that patterns that are expected to be found in Ω_i, for i a step of the algorithm, are among the subpatterns at depth i in those of $\sigma_i(\Omega_i)$.

Lemma A.4 ($Inhabits_{\mathcal{A}}$ **patterns).** *Let an execution of $Inhabits_{\mathcal{A}}(\Omega)$. If the execution reached step i, then*

$$Patterns(\Omega_i) \subseteq \{p[\pi] \mid p \in Patterns(\sigma_i(\Omega)) \land \pi \in Pos(p) \land |\pi| = i\}$$

Proof. Let us name this property $P(i)$ and prove it by induction on the step i.

- $\boldsymbol{i = 0}$: $P(0) \iff Patterns(\Omega_0) \subseteq \{p[\epsilon] \mid p \in Patterns(\Omega)\}$

- $\boldsymbol{i > 0}$ **:** Suppose that the execution of $Inhabits_{\mathcal{A}}(\Omega)$ has reached step i and $P(i)$ true. Then either it terminates at step i and never reaches step $i+1$, so $P(i+1)$ immediately holds, or else it reaches step $i+1$. Suppose that the execution reached step $i+1$, so we have $Some(\Omega_{i+1}, \sigma_\delta) \in unfolds(T)$ for $T = \{(\omega, r_\omega) \mid \omega \in \Omega_i\}$ with every r_ω some transition of the automaton $\mathcal{A}$, and $\sigma_{i+1} = \sigma_\delta \circ \sigma_i$.

  Let ω' of the form $q'(\overrightarrow{p'})$ in Ω_{i+1}. By definition of $unfolds(T)$, we have that there exists an atom ω of shape $q(\overrightarrow{p})$ in Ω_i such that $\omega' \in unfoldOne(\sigma_\delta(\omega), r_\omega)$ for r_ω the transition associated to ω in T. By Proposition A.2, we have $Patterns(\omega') \subseteq \{p[\pi] \mid p \in \sigma_\delta(\overrightarrow{p}) \wedge \pi \in Pos(\sigma_\delta(\overrightarrow{p})) \wedge |\pi| = 1\}$. We therefore have $Patterns(\Omega_{i+1}) \subseteq \{p[\pi] \mid p \in \sigma_\delta(Patterns(\Omega_i)) \wedge \pi \in Pos(p) \wedge |\pi| = 1\}$.

  Using the induction hypothesis $P(i)$, we have $Patterns(\Omega_{i+1}) \subseteq \{p[\pi] \mid p' \in Patterns(\sigma_i(\Omega)) \wedge \pi' \in Pos(p') \wedge |\pi'| = i \wedge p \in \sigma_\delta(p'[\pi']) \wedge \pi \in Pos(p) \wedge |\pi| = 1\}$, which simplifies to $Patterns(\Omega_{i+1}) \subseteq \{p[\pi] \mid p \in Patterns(\sigma_\delta(\sigma_i(\Omega))) \wedge \pi \in Pos(p) \wedge |\pi| = i+1\}$. This concludes the proof of $P(i+1)$.

$\square$

Lemma 6.16 ($Inhabits_{\mathcal{A}}$ height boundedness)**.** *If $Inhabits_{\mathcal{A}}(\Omega)$ terminates at step i with $Some(\sigma)$, then $i \leq ht(\sigma(\Omega)) \leq i + ht(\Omega)$.*

Proof. Let an execution of $Inhabits_{\mathcal{A}}(\Omega)$ be.

Let $L(i)$ be the property "If the execution reaches step i, then $i \leq ht(\sigma_i(\Omega))$".

Let $U(i)$ be the property "If the execution reaches step i, then $ht(\sigma_i(\Omega)) \leq i + ht(\Omega)$".

- The property $L(i)$ derives from lemma A.4. Indeed, suppose that the execution reaches step i. If $i = 0$, then $L(0) \iff 0 \leq ht(\sigma_0(\Omega))$ is immediate. Else, let $\omega = q(\overrightarrow{p}) \in \Omega_{i-1}$, as we know $\Omega_{i-1} \neq \emptyset$ for the execution reached step i. Because of Lemma A.4 applied on step $i-1$, we know that any pattern p of $\overrightarrow{p}$ can be written as $p'[\pi]$ for some pattern p' in $Patterns(\sigma_{i-1}(\Omega))$ and $\pi \in Pos(p')$ with $|\pi| = i-1$. Because $p'[\pi]$ is defined, we know that $ht(p') \geq i$. Therefore $ht(\sigma_{i-1}(\Omega)) \geq i$ and $ht(\sigma_i(\Omega)) \geq i$, so $L(i+1)$.
- Let us prove property $U(i)$ by induction on the step i.
  — $\boldsymbol{i = 0}$ **:** $U(0) \iff ht(\Omega) \leq ht(\Omega)$.
  — $\boldsymbol{i > 0}$ **:** Suppose the execution has reached step i and both $L(i)$ and $U(i)$ are true. Then either it terminates at step i and never reaches step $i+1$, so $L(i+$

1) and $U(i+1)$ immediately hold, or else it reaches step $i+1$. Suppose that the execution reached step $i+1$, so we have $Some(\Omega_{i+1}, \sigma_\delta) \in unfolds(T)$ for $T = \{(\omega, r_\omega) \mid \omega \in \Omega_i\}$ with every r_ω some transition of the automaton $\mathcal{A}$, and $\sigma_{i+1} = \sigma_\delta \circ \sigma_i$.

Because of the specific form of the unification problem defined in $unfolds(T)$, we know that for all binding (X, p) in σ_δ, we have $X \in Vars(\Omega_i)$ implies that p is either a variable or of the form $f(\vec{X})$ for f a function and $\vec{X}$ variables. In particular, $ht(\sigma(X)) \leq ht(X) + 1$. Also $Vars(\Omega_i) \subseteq Vars(\sigma_i(\Omega))$. Therefore $ht(\sigma_\delta(\sigma_i(\Omega))) \leq ht(\sigma_i(\Omega)) + 1$. Using $U(i)$, we have $ht(\sigma_{i+1}(\Omega)) \leq i + ht(\Omega) + 1$, so $U(i+1)$ holds.

Combining both the lowerbound given by $L(i)$ and the upperbound given by $U(i)$, we conclude that $i \leq ht(\sigma(\Omega)) \leq i + ht(\Omega)$. □

Lemma 6.19 (Characterisation of easier sets of atoms)**.** *Let Ω_a and Ω_b two sets of atoms. Then*

$$\Omega_a \leq \Omega_b \iff \textit{there exists a substitution } \sigma \textit{ such that } \sigma(\Omega_a) \subseteq \Omega_b.$$

Proof. (by Thibaut Antoine).

- $\Leftarrow$: Let σ be a substitution such that $\sigma(\Omega_a) \subseteq \Omega_b$. Then, for any model $\mathcal{M}$ and assignment $\lambda \in Assigns(\mathcal{M}, \Omega_b)$, we have that $\lambda' = \lambda \circ \sigma = X \mapsto \lambda(\sigma(X))$ is such that $\lambda' \in Assigns(\mathcal{M}, \Omega_a)$. Thus $\Omega_a \leq \Omega_b$.
- $\Rightarrow$: Suppose $\Omega_a \leq \Omega_b$ and let us prove that there exists σ such that $\sigma(\Omega_a) \subseteq \Omega_b$. Let $\mathcal{H}_a$ be the Herbrand model on the domain of patterns and with the interpretation of relations symbols defined as $R \mapsto \{\vec{p} \mid R(\vec{p}) \in \Omega_a\}$. Because assignments in a Herbrand model are substitution, we write them as such. We know that $\sigma_{id} = X \mapsto X$ is such that $\sigma_{id} \in Assigns(\mathcal{H}_a, \Omega_a)$ by definition of $\mathcal{H}_a$. By $\Omega_a \leq \Omega_b$, we know that there exists $\sigma \in Assigns(\mathcal{H}_a, \Omega_b)$. That is, for every atom $\omega_b \in \Omega_b$, we have $\mathcal{H}_a, \sigma \models \omega_b$, i.e. $\mathcal{H}_a \models \sigma(\omega_b)$. By definition of $\mathcal{H}_a$, we thus necessarily have $\sigma(\omega_b) \in \Omega_a$. Therefore $\sigma(\Omega_b) \subseteq \Omega_a$.

□

Lemma 6.20 (Easier relation is preserved by unfolding)**.** *Let some automaton $\mathcal{A}$ and two sets of atoms Ω_a and Ω_b such that $\Omega_a \leq \Omega_b$. Then, for every $(\Omega'_b, \sigma_b) \in unfolds_{\mathcal{A}}(\Omega_b)$, there exists $(\Omega'_a, \sigma_a) \in unfolds_{\mathcal{A}}(\Omega_a)$ such that $\Omega'_a \leq \Omega'_b$.*

Proof. Because $\Omega_a \leq \Omega_b$, let σ be a substitution such that $\sigma(\Omega_a) \subseteq \Omega_b$. Let (Ω'_b, σ_b) an element of $unfolds_{\mathcal{A}}(\Omega_b)$. By definition of $unfolds_{\mathcal{A}}(\Omega_b)$, there exists one transition r_{ω_b} per atom $\omega_b \in \Omega_b$ such that, with $T_b = \{(\omega_b, r_{\omega_b}) \mid \omega_b \in \Omega_b\}$, $unfolds(T_b) = Some(\Omega'_b, \sigma_b)$. Then, because $\sigma(\Omega_a) \subseteq \Omega_b$, $T_a = \{(\omega_a, r_{\sigma(\omega_a)}) \mid \omega_a \in \Omega_a\}$ is such that $unfolds(T_a) = Some(\Omega'_a, \sigma_a)$ with $\Omega'_a \leq \Omega'_b$. □

Lemma 6.21 (Safety of pruning)**.** *Suppose that an execution of* $Inhabits_{\mathcal{A}}(\Omega)$ *returns* $Some(\sigma)$ *in* i *steps and is such that there exists* $i_{old}, i_{new} \in [0 \dots i-1]$ *with* $i_{old} < i_{new}$ *and* $\Omega_{i_{old}} \leq \Omega_{i_{new}}$*. Then there exists an other execution of* $Inhabits_{\mathcal{A}}(\Omega)$ *that returns* $Some(\sigma')$ *at some step* $j < i$*.*

Proof. Let such an execution of $Inhabits_{\mathcal{A}}(\Omega)$ be and let $k = i_{new} - i_{old}$. We build an other execution of $Inhabits_{\mathcal{A}}(\Omega)$ whose sets of atoms and substitutions will be named with a prime to distinguish them from those of the first execution. This other execution can copy $Inhabits_{\mathcal{A}}(\Omega)$ until step i_{old}, i.e. have $\forall j \in [0 \dots i_{old}], \Omega_j = \Omega'_j \wedge \sigma_j = \sigma'_j$. Then, this prime execution can intuitively loosely copy the first execution on steps $[i_{new} \dots i]$. More formally, because $\Omega_{i_{old}} \leq \Omega_{i_{new}}$ and by k applications of Lemma 6.20, we can extend this prime execution until step $i-k$ with the property that $\forall j \in [i_{old} \dots i - k], \Omega'_j \leq \Omega_{j+k}$. In particular $\Omega'_{i-k} \leq \Omega_i$. Because the first execution returned $Some(_)$ at step i, we know that $\Omega_i = \emptyset$, so necessarily $\Omega'_{i-k} = \emptyset$. Also $k = i_{new} - i_{old} > 0$, so this prime execution returns $Some(_)$ in $i-k$ steps. □

Lemma 6.33 (*Teacher* height boundedness)**.** *If* $Teacher(\mathcal{A}, \Gamma) = Some(\widehat{\varphi})$*, then for any other counterexample* $\widehat{\varphi}'$ *of* $\mathcal{A} \models \Gamma$*,* $ht(\widehat{\varphi}) \leq ht(\widehat{\varphi}') + dh$ *with* dh *a constant defined from* Γ*.*

Proof. Let $dh = ht(\Gamma) + d$, with d being the max, among every type, of the height of the smallest term of this type. More precisely, writing T for the set of types, $d = \max_{\tau \in T}(\min_{t \in \mathcal{T}_\tau(\Sigma)}(ht(t)))$.
Suppose $Teacher(\mathcal{A}, \Gamma)$ terminates with $Some(\widehat{\varphi})$. Then there exists $\varphi \in \Gamma$ such that $Inhabits^{det}_{\mathcal{A}}(\Omega_{\overline{\varphi}})$ terminates with $Some(\sigma)$, $\sigma_{Gr} = Grd(\sigma(\Omega_{\overline{\varphi}}))$, and $\widehat{\varphi} = \sigma_{Gr}(\sigma(\varphi))$. Let i be the step at which $Inhabits^{det}_{\mathcal{A}}(\Omega_{\overline{\varphi}})$ terminated. By Lemma 6.16, $ht(\sigma(\Omega_{\overline{\varphi}})) \leq i + ht(\Omega_{\overline{\varphi}})$, so $ht(\sigma(\varphi)) \leq i + ht(\varphi)$.
By the *Teacher* synchronisation of parallel calls to $Inhabits^{det}_{\mathcal{A}}$, for any other formula $\varphi' \in \Gamma$, $Inhabits^{det}_{\mathcal{A}}(\Omega_{\overline{\varphi'}})$ either terminates with *None* or terminates after $Inhabits^{det}_{\mathcal{A}}(\Omega_{\overline{\varphi}})$, i.e., after step i.
Let $\widehat{\varphi}'$ be a smallest ground counterexample to $\mathcal{A} \models \Gamma$. Because $\widehat{\varphi}'$ is ground, $\mathcal{A} \models \Omega_{\overline{\widehat{\varphi}'}}$ means that for every atom of the shape $R(\overrightarrow{t})$ in $\Omega_{\overline{\widehat{\varphi}'}}$ we have $\overrightarrow{t} \in \mathcal{R}(R, \mathcal{A})$. $\widehat{\varphi}'$ can be

written as $\sigma'(\varphi')$ for σ' a substitution and $\varphi' \in \Gamma$. By lemma A.3 and the fact that no $Inhabits^{det}_{\mathcal{A}}$ call, despite being implemented as breadth-first search, has stopped before step i, we have $i \leq ht(\widehat{\varphi}')$.
Combining these two inequalities, we have $ht(\sigma(\varphi)) \leq ht(\widehat{\varphi}') + ht(\varphi)$.
Bounding $ht(\varphi)$ by $ht(\Gamma)$, we find $ht(\sigma(\varphi)) \leq ht(\widehat{\varphi}') + ht(\Gamma)$.
The teacher finishes by returning $\widehat{\varphi} = \sigma_{Gr}(\sigma(\varphi))$. For e a variable, pattern, atom, set of atom, or clause, $ht(Grd(e)(e)) \leq ht(e) + d$, so $ht(\widehat{\varphi}) \leq ht(\widehat{\varphi}') + dh$. □

A.2 Example of relations defined using first-order Horn clauses

The following (Horn) clauses uniquely define the predicates Leq (less than or equal), IsEmpty (is a tree empty), All0 (is a tree full of zeros), No0 (is a tree free of zeros), Height (the height of a binary tree), and HeightRB (the height of the rightmost branch of a binary tree). All variables are written in uppercase and are implicitly universally quantified.

$$
\begin{array}{lll}
\mathrm{Leq}(z, z) & \bot \Leftarrow \mathrm{Leq}(s(X), z) & \mathrm{IsEmpty}(\mathit{leaf}) \\
\mathrm{Leq}(z, s(X)) & \mathrm{Leq}(s(X), s(Y)) \Leftarrow \mathrm{Leq}(X, Y) & \bot \Leftarrow \mathrm{IsEmpty}(\mathit{node}(T_1, E, T_2)) \\
 & \mathrm{Leq}(X, Y) \Leftarrow \mathrm{Leq}(s(X), s(Y)) &
\end{array}
$$

$$
\begin{array}{ll}
\mathrm{All0}(\mathit{leaf}) & \mathrm{All0}(\mathit{node}(T_1, z, T_2)) \Leftarrow \mathrm{All0}(T_1) \wedge \mathrm{All0}(T_2) \\
\bot \Leftarrow \mathrm{All0}(\mathit{node}(T_1, s(E), T_2)) & \mathrm{All0}(T_1) \Leftarrow \mathrm{All0}(\mathit{node}(T_1, z, T_2)) \\
 & \mathrm{All0}(T_2) \Leftarrow \mathrm{All0}(\mathit{node}(T_1, z, T_2))
\end{array}
$$

$$
\begin{array}{ll}
\mathrm{No0}(\mathit{leaf}) & \mathrm{No0}(\mathit{node}(T_1, s(E), T_2)) \Leftarrow \mathrm{No0}(T_1) \wedge \mathrm{No0}(T_2) \\
\bot \Leftarrow \mathrm{No0}(\mathit{node}(T_1, z, T_2)) & \mathrm{No0}(T_1) \Leftarrow \mathrm{No0}(\mathit{node}(T_1, s(E), T_2)) \\
 & \mathrm{No0}(T_2) \Leftarrow \mathrm{No0}(\mathit{node}(T_1, s(E), T_2))
\end{array}
$$

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

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

Note that, because we represent non-boolean n-ary functions by $n+1$-ary relations, we add a last clause to their definition that ensures that the relation is functional. We call these clauses the *functionality* clauses. For example, the functionality clause of the Height definition is $N = M \Leftarrow \text{Height}(T, N) \wedge \text{Height}(T, M)$. Without this functionality clause, a Herbrand model where $\text{Height}(t, n)$ is true for all trees t and all natural numbers n is a correct model for the three first clauses of Height. For a terminating and deterministic function, the added functionality clause ensures, jointly with the others, that the Horn clauses can only be satisfied by a single Herbrand model $\mathcal{H}$ representing the functions.

A.3 Undecidability of the teacher procedure and of the emptiness of SHoCs

This section introduces the material we use for proving that the teacher procedure is undecidable, either using convoluted automata or SHoCs, and that emptiness of SHoCs is undecidable. Not that emptiness of *languages* of convoluted automata is clearly decidable, as a reachability procedure allows to decide whether a state has an empty language or not. However, emptiness of the *relation* denoted by a state of a convoluted automaton may be decidable or not, depending on the convolution. For example, the left- and right-convolution yield decidable emptiness problems of relations, as the reachability procedure on states still applies, whereas it is not straightforward whether the complete convolution yields a decidable emptiness problem.
We begin by introducing (two-counter) Minsky machines, whose halting problem is undecidable. The idea of using the undecidability of the halting problem of two-counters Minsky machines is from private communications with Naoki Kobayashi and [34].

Definition A.5 (Minsky machine)**.**
A Minsky machine is a finite function I that maps a program counter p (a natural number) to its corresponding instruction. An instruction is one of

$$inc_goto(i, p_t) \mid dec_goto(i, p_t, p_e) \mid halt$$

for i a natural number designating a register and p_t, p_e program counters. A configuration is a pair (R, p) with $R = [r_1, \ldots, r_n]$ an array of natural numbers representing the

value of each register and p the program counter. Instruction $inc_goto(i, p_t)$ increments register r_i then changes p to p_t. Instruction $dec_goto(i, p_t, p_e)$ decrements r_i and changes p to p_t if the decrement is successful (i.e. if $r_i > 0$), and else leaves r_i unchanged (at 0) and changes p to p_e. Finally, instruction $halt$ halts the machine.

Example A.6. Here is a simple (non terminating) Minsky machine with 1 register, initial configuration $([0], 0)$, and the following program:

$$\begin{aligned} 0 &: \ inc_goto(1, 1) \\ 1 &: \ inc_goto(1, 2) \\ 2 &: \ dec_goto(1, 0, 3) \\ 3 &: \ halt \end{aligned}$$

A partial run of this machine can be represented by the following sequence on configurations, starting by the initial one:

$$([0], 0) \to ([1], 1) \to ([2], 2) \to ([1], 0) \to ([2], 1) \to ([3], 2) \to ([2], 0) \to \dots$$

An interesting class of Minsky machines are those with only two counters, as Minsky machines with at least two counters are known for having an undecidable halting problem.

Definition A.7 (From a 2-counter Minsky machine to SHoCs)**.**
Let I be a 2-counter Minsky machine. We write S_I the SHoCs that partially encodes I defined by the following 8 fixed clauses and one or two clauses defining R_p for every $p \in dom(I)$:

$$\begin{array}{ll} R_{eq}(z, z) & R_z(z) \\ R_{eq}(s(N), s(M)) \Leftarrow R_{eq}(N, M) & R_{pzz}(pair(N_1, N_2)) \Leftarrow R_z(N_1) \wedge R_z(N_2) \end{array}$$

$$\begin{array}{l} R_{inc}(s(N), s(M)) \Leftarrow R_{inc}(N, M) \\ R_{inc}(z, s(M)) \Leftarrow R_z(M) \end{array}$$

$$\begin{array}{ll} R_{inc_1}(pair(N_1, N_2), pair(M_1, M_2)) & \Leftarrow R_{inc}(N_1, M_1) \wedge R_{eq}(N_2, M_2) \\ R_{inc_2}(pair(N_1, N_2), pair(M_1, M_2)) & \Leftarrow R_{inc}(N_2, M_2) \wedge R_{eq}(N_1, M_1) \end{array}$$

- For every p such that $I(p) = inc_goto(i, p_t)$:
  $R_p(cons(P_1, L_1), cons(P_2, L_2)) \Leftarrow R_{p_t}(L_1, L_2) \wedge R_{inc_i}(P_1, P_2)$
- For every p such that $I(p) = dec_goto(i, p_t, p_e)$:
  $R_p(cons(P_1, L_1), cons(P_2, L_2)) \Leftarrow R_{p_t}(L_1, L_2) \wedge R_{inc_i}(P_2, P_1)$
  $R_p(cons(P_1, L_1), cons(P_2, L_2)) \Leftarrow R_{p_e}(L_1, L_2) \wedge R_{pzz}(P_1)$
- For every p such that $I(p) = halt$:
  $R_p(cons(P_1, L_1), nil)$

Relation R_{eq} defines equality of natural numbers, R_{inc} relates any n with $s(n)$, R_{inc_1} relates pairs of natural numbers such that the second pair is the same as the first except for the first element that is incremented. Relations R_p locally simulate the behavior of the Minsky machine at program counter p.

Proposition A.8. *Let I a two-counter Minsky machine and S_I the associated SHoCs. For any $p \in dom(I)$, lists l and l' of pairs of integers with $l = cons(pair(n_1, n_2), l')$, we have that $(l, l') \in \mathcal{L}(R_i, S_I)$ iff l is the log (list of values that both counters took during the execution) of a terminating run of the Minsky machine I that started on configuration $([n_1, n_2], i)$.*

Corollary A.9. *It follows immediately from proposition A.8 that a Minsky machine I halts from the initial configuration $([0, 0], 0)$ iff there exists a list l of pairs on integers such that $(cons(pair(z, z), l), l) \in \mathcal{L}(R_0, S_I)$.*

Theorem A.10. *The Teacher procedure is undecidable, whether using convoluted automata or SHoCs as models.*

Proof. Then let I be a Minsky machine and $\varphi = \big(\bot \Leftarrow R_0(cons(pair(z, z), L), L)\big)$ a clause. Then, from Corollary A.9 and Theorem 6.32, we have:

$$Teacher(S_I, \{\varphi\}) = Some(_) \iff I \text{ halts}$$

The *Teacher* procedure is therefore undecidable when using SHoCs as models. Moreover, the construction S_I can also be done using a convoluted automaton instead of a SHoCs, so the *Teacher* procedure is undecidable too when using them as models. □

We define the SHoCs S_{shape} to match the shape constraint of the clause, i.e. such that $\mathcal{L}(R_{shape}, S_{shape}) = \{(cons(pair(z, z), t), t) \mid t \text{ is a list of pairs of integers}\}$:

Definition A.11 (S_{shape})**.**

$$R_{shape}(cons(X_1, X_2), nil) \Leftarrow R_{pzz}(X_1) \wedge R_{nil}(X_2)$$
$$R_{shape}(cons(X_1, X_2), cons(Y_1, Y_2)) \Leftarrow R_{pzz}(X_1) \wedge R_{cons_1}(X_2, Y_1) \wedge R_{cons_2}(X_2, Y_2)$$

$$R_{cons_1}(cons(X_1, X_2), z) \Leftarrow R_z(X_1)$$
$$R_{cons_1}(cons(X_1, X_2), s(Y)) \Leftarrow R_s(X_1, Y)$$

$$R_{cons_2}(cons(X_1, X_2), nil) \Leftarrow R_{nil}(X_2)$$
$$R_{cons_2}(cons(X_1, X_2), cons(Y_1, Y_2)) \Leftarrow R_{cons_1}(X_2, Y_1) \wedge R_{cons_2}(X_2, Y_2)$$

$$R_s(s(X), z) \Leftarrow R_z(X)$$
$$R_s(s(X), s(Y)) \Leftarrow R_s(X, Y)$$

$$R_{nil}(nil)$$

$$R_{pzz}(pair(X_1, X_2)) \Leftarrow R_z(X_1) \wedge R_z(X_2)$$
$$R_z(z)$$

Theorem 7.25 (SHoCs emptiness problem is undecidable)**.** *For S a SHoCs and R a relation, the SHoCs emptiness problem is to decide whether $\mathcal{L}(R, S) = \emptyset$. This problem is undecidable.*

Proof. Let I a Minsky machine and S_I the corresponding SHoCs.
Let $S = S_I \cup S_{shape}$ be the set-union of the clauses of S_I and S_{shape} and let S' be the SHoCs resulting from the intersection of relation R_0 and relation R_{shape} of S in a fresh relation R. That is, $\mathcal{L}(S', R) = \mathcal{L}(S, R_0) \cap \mathcal{L}(S, R_{shape}) = \mathcal{L}(S_I, R_0) \cap \mathcal{L}(S_{shape}, R_{shape})$.
From Corollary A.9, I halts iff $\mathcal{L}(R, S') \neq \emptyset$. □

COLLEGE DOCTORAL BRETAGNE | MATHS, TELECOMS INFORMATIQUE, SIGNAL SYSTEMES, ELECTRONIQUE

Université de Rennes

Titre : Vérification automatique de programmes par inférence de modèles relationnels

Mot clés : Vérification formelle, Propriétés relationnelles, Données algébriques, Inférence de modèle

Résumé : Cette thèse porte sur la preuve automatique de propriétés concernant la relation entrée/sortie de programmes fonctionnels manipulant des types de données algébriques (ADT). De récents résultats montrent comment approximer un programme fonctionnel en utilisant un automate d'arbre. Bien qu'expressives, ces techniques ne peuvent pas prouver de propriété reliant l'entrée et la sortie d'une fonction, par exemple qu'inverser une liste préserve sa longueur. Dans cette thèse, nous nous appuyons sur ces résultats et définissons une procédure pour calculer ou sur-approximer une telle relation. Formellement, le problème de la vérification de programmes se réduit à la satisfiabilité de clauses, que nous résolvons en exhibant un modèle. Dans cette thèse, nous proposons deux représentations relationnelles de ces modèles de Herbrand : les automates d'arbres convolués et les *shallow Horn clauses*. Les automates d'arbres convolués généralisent les automates d'arbres et sont généralisés par les shallow Horn clauses. Le problème d'inférence du modèle de Herbrand découlant de la vérification relationnelle étant indécidable, nous proposons une procédure d'inférence incomplète mais correcte. Les expériences montrent que cette procédure est performante en pratique par rapport aux outils actuels, à la fois pour la vérification des propriétés et pour la recherche de contre-exemples.

Title: Automatic Program Verification by Inference of Relational Models

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

Abstract: This thesis 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 thesis, we build upon those results and define a procedure to compute or over-approximate such a relation, thereby allowing to prove properties that require a more precise relational representation. Formally, the program verification problem reduces to satisfiability of clauses over the theory of algebraic data types, which we solve by exhibiting a Herbrand model of the clauses. In this thesis, we propose two relational representations of these Herbrand models: convoluted tree automata and shallow Horn clauses. Convoluted tree automata generalize tree automata and are in turn generalized by shallow Horn clauses. The Herbrand model inference problem arising from relational verification is undecidable, so we propose an incomplete but sound inference procedure. Experiments show that this procedure performs well in practice w.r.t. state of the art tools, both for verifying properties and for finding counterexamples.