Thèse de doctorat de

l'Université de Rennes 1

École doctorale № 601
Mathématiques et Sciences et Technologies de l'Information et de la Communication

présentée par

Timothée Haudebourg

Automatic Verification of Higher-Order Functional Programs using Regular Tree Languages

Thèse présentée et soutenue à Rennes, le 15 Décembre 2020
Unité de recherche: 6074 (IRISA)
Thèse №: 1000

Rapporteurs avant soutenance :

Naoki Kobayashi	Professeur, University of Tokyo
Mihaela Sighireanu	Professeur, ENS Paris-Saclay

Composition du Jury :

Examinateurs :	Jasmin Blanchette	Professeur assistant, Vrije Universiteit Amsterdam
	Sophie Pinchinat	Professeur, Univ Rennes
Dir. de thèse :	Thomas Genet	Maître de conférences, Univ Rennes
Codir. de thèse :	Thomas Jensen	Directeur de recherche, Inria

Contents

Remerciements **7**

Résumé en Français **9**

1 Introduction **15**
1.1 Motivation 16
1.1.1 Testing 16
1.1.2 Proof Assistants 17
1.1.3 Automated Verification Techniques 18
1.1.4 Regular Verification Problems 19
1.2 Our Verification Framework 20
1.2.1 Term Rewriting Systems 20
1.2.2 Tree Automata and Regular Languages 21
1.3 Contributions 21
1.3.1 Equational Abstractions for Higher-Order Programs 22
1.3.2 Regular Type Inference 22
1.3.3 Regular Relations 22
1.3.4 Summary 22

2 Preliminaries **25**
2.1 Trees, Term and Patterns 25
2.2 Rewriting Systems 27
2.2.1 Definitions 27
2.2.2 Properties of TRSs 28
2.2.3 Usage in Functional Program Verification 29
2.3 Tree Languages, Grammars and Automata 32
2.3.1 Regular Tree Languages 33
2.3.2 Bottom-Up Tree Automata 33
2.3.3 Beyond Regularity 35
2.4 Automated Verification and Abstraction 37
2.4.1 Formalization as Model Checking 37
2.4.2 Verification via Abstraction 39

3 State of the Art **43**
3.1 Static Type Systems 43
3.1.1 Intersection and Set Theoretic Types 43
3.1.2 Dependent and Refinement Types 44
3.1.3 Deep Specification 46
3.2 Higher-Order Trees Model Checking 47
3.2.1 Higher Order Recursion Schemes 47

3.2.2 Predicate Abstraction 49
3.3 Regular Verification 50
3.3.1 HORS Extensions 50
3.3.2 Regular Tree Languages based Techniques 53

4 Higher-Order Equational Abstractions 59
4.1 Introduction 59
4.2 Tree Automata Completion Algorithm 60
4.2.1 Core Algorithm 61
4.2.2 Properties of the TAC Algorithm 63
4.2.3 Equations 63
4.2.4 Contracting Equations 65
4.3 Termination Criterion Using Contracting Equations 68
4.3.1 The Role of Contracting Equations 69
4.3.2 The Role of Transitive Equations 70
4.4 A Class of Analyzable Programs 71
4.4.1 Bounded Applications Stacks 71
4.4.2 Type System 72
4.4.3 The $\mathcal{K}$-TRS Class 74
4.5 Verification Procedure 82
4.5.1 Contracting Equations Generation 82
4.5.2 Equations Exploration 84
4.6 Experiments 86
4.6.1 Test Suite 86
4.6.2 Experimental Results 86
4.6.3 Related Work 90
4.7 Conclusion 90

5 Regular Language Type Inference 93
5.1 Introduction 93
5.1.1 Abstraction solution: regular abstract interpretation 93
5.1.2 Modularity solution: regular language types 94
5.1.3 Inference solution: Regular language learning 95
5.2 Regular Abstract Interpretation 96
5.2.1 Regular Abstract Domain 97
5.2.2 Abstract Semantics 97
5.2.3 Abstraction Inference Challenges 100
5.3 Regular Language Types 101
5.3.1 Type Partitions 105
5.3.2 Inference Algorithm 106
5.3.3 Non-Recursive TRS 108
5.3.4 Invariant Learning 112
5.4 Experiments 118
5.4.1 Implementation Details 119
5.4.2 Test Suite 120
5.4.3 Experimental Results 120
5.5 Conclusion 122

6 Regular relations **125**
6.1 Introduction . . . 125
6.2 String Automatic Relations . . . 126
6.3 Tree Automatic Relations . . . 127
6.3.1 Standard Convolution . . . 127
6.3.2 Full Convolution . . . 129
6.4 Relations Inference . . . 132
6.4.1 Constrained Horn Clauses Solving . . . 133
6.4.2 ICE-Based Verification . . . 133
6.4.3 The Teacher . . . 136
6.4.4 The Learner . . . 140
6.4.5 Soundness and Completeness . . . 143
6.5 Experiments . . . 144
6.5.1 Test Suite . . . 144
6.5.2 Experimental Results . . . 145
6.6 Conclusion . . . 146

7 Conclusion and Future Work **147**
7.1 Non-Terminating Programs . . . 148
7.2 Non-Safety Problems . . . 148
7.3 Integration in Higher-Order Functional Languages . . . 151
7.4 Polymorphic Lifting . . . 152
7.5 Regular Relations and Higher-Order . . . 154

Bibliography **163**

Remerciements

Je veux tout d'abord exprimer ma gratitude envers mes deux encadrants Thomas Genet et Thomas Jensen, qui m'ont supportés et guidés tout au long de ces trois années de thèse. Ils ont su me faire confiance dans les longs moments d'incertitudes et j'ai toujours pu trouver la porte de leur bureau ouverte pour y trouver conseil. Je n'aurai pas pu imaginer meilleurs encadrants pour diriger ma thèse.

Je tiens aussi à remercier mes rapporteurs Naoki Kobayashi et Mihaela Sighireanu pour leur intérêt porté à mes travaux ainsi que leur retours de qualité. Je remercie aussi Jasmin Blanchette d'avoir accepté sa place d'examinateur au sein de mon jury de thèse, et Sophie Pinchinat d'avoir présidé ce jury. Merci pour leur intérêt porté à ma présentation, et leur retours.

Merci aussi à David Pichardie et Delphine Demange pour m'avoir ouvert les portes de l'ENS de Rennes et dirigé vers la recherche à l'époque ou je cherchai encore ma voie. J'ai passé au sein de l'ENS, puis de l'IRISA, des année passionnantes.

Je remercie aussi toute l'équipe Celtique qui m'a accueilli pendant ma thèse, en particulier mes différents co-bureaux, Alix Trieux d'abord et Thomas Rubiano enfin mon compagnon d'escalade. Merci aussi à Lydie Mabil puis Stephanie Gosselin Lemaile pour leur aide dans mes démarches administratives.

Merci enfin à ma famille et mes amis qui ont eu la force de rester confinés chez eux pendant la pandémie, qui auraient tant voulu venir m'encourager pendant ma soutenance mais qui ont dû se contenter de m'observer derrière l'écran de leur ordinateur.

Résumé en Français

L'objectif de cette thèse est d'étudier l'usage des langages d'arbres réguliers combinés aux systèmes de réécriture, appliqué à la vérification automatique de propriétés sur des programmes fonctionnels d'ordre supérieur dans le but de faciliter la détection et correction de bugs. Si les bugs sont souvent associés à des épisodes catastrophiques comme la destruction de la fusée Ariane 5 à cause d'un bug dans le programme de guidage, de tels événements ont tendance à cacher la présence massive des bugs dans les programmes communs, où leurs effets sont moins spectaculaires. En 1978, Lientz et al. [LST78] ont montré que les programmeurs concentrent 17% de leur effort de développement à la correction de bugs. Une étude similaire réalisée par Amit et al. en 2020 [AF20] a montré que sur la plateforme de développement partagée GitHub, 20% des changement appliqués aux projets (« commits ») sont dédiés à la correction d'un ou plusieurs bugs. Bien que peu dangereux, il est estimé [RTI02] que ces bugs coûtent en moyenne 59 milliard de dollars par ans à l'économie Étasunienne seule (soit 0.6% de leur PIB), en plus d'être passablement agaçants pour les utilisateurs. De nombreux outils comme Nitpick [BN10] ont été développés pour aider les développeur à tester leur programmes et traquer les bugs. Malheureusement sauf dans de rare cas, le test est insuffisant pour garantir l'absence de bugs. Le reste du temps, une approche plus formelle doit être utilisée pour *prouver* que le programme est correct. En pratique cependant, même en utilisant des assistants de preuve comme Coq [Inr16] ou Isabelle HOL [NPW02], construire la preuve de correction d'un programme demande beaucoup de temps et d'expertise, un investissement difficile à consentir dans l'industrie du développement. Cette thèse s'intéresse à la création de techniques de preuve complètement automatisées permettant de réduire cette investissement. En particulier nous nous intéressons à la vérification automatique de propriétés de sécurité sur des programmes purement fonctionnels d'ordre supérieur, famille de programmes plus naturellement adaptée à la vérification formelle. Plus précisément encore, nous nous intéressons à la famille des propriétés « régulières » (dont la nature est précisée plus bas) pour laquelle nous montrons qu'il est toujours possible de prouver ou d'improuver la propriété complètement automatiquement.

Il existes déjà de nombreuses techniques détaillées dans le Chapitre 3 permettant d'automatiser des parties ou l'entièreté de la vérification, mais sans garantie de résultat sur une famille de propriété particulières. La plupart de ces techniques reposent sur la recherche d'une abstraction de l'exécution du programme permettant de vérifier la propriété donnée. Parmi ces méthodes d'abstraction, Genet et al. [GR10, Gen16] propose d'utiliser des systèmes de réécriture combinés à l'algorithme de complétion d'automate d'arbre pour générer des abstraction régulières. Cette méthode permet ainsi d'envisager le développement d'une technique de vérification complète sur les propriété régulières. C'est l'objectif de cette thèse dans laquelle

- nous étendons la méthode d'abstraction proposée par Genet et al. pour vérifier des propriétés sur les programmes d'ordre supérieur et étudions ses limites,

- nous repoussons ces limites en définissant une nouvelle méthode d'abstraction basée sur l'algorithme de complétion d'automate ainsi qu'une procédure de vérification complète sur les propriété régulières basée sur celle-ci,
- enfin nous étendons notre méthode d'abstraction pour vérifier des propriété relationnelles non-régulières, toujours à l'aide de langages réguliers

Termes, réécriture et vérification régulière

Nous posons ici les outils formels utilisés dans cette thèse pour représenter et analyser les programmes fonctionnels d'ordre supérieur et les propriété régulières que nous vérifions sur ceux-ci. L'exécution d'un programme est communément décrite par la succession des états visités pendant le calcul. Dans notre cas, nous utilisons la théorie des *termes* pour représenter ces états. Un terme est un arbre étiqueté de la forme $f(t_1, \ldots, t_n)$ où f est un symbole et chaque t_i in sous-terme (sous-arbre). On note $\mathcal{T}(\Sigma)$ l'ensemble des termes étiquetés par les symboles inclus dans Σ. Les termes sont utiles pour représenter à la fois les états de l'exécution d'un programme fonctionnel, et les valeurs manipulées. Pour représenter et raisonner sur un programme fonctionnel, nous utilisons les systèmes de réécriture de termes (TRS). Cette représentation est à la fois proche des langages fonctionnels utilisés en pratique, et assez épurée pour simplifier le raisonnement formel. Un TRS, noté R, est composé de règles de réécritures de la forme $g \to d$ où g et d sont des *motifs* : des termes contenant des variables. Si $\mathcal{X}$ est un ensemble de variables, on note $\mathcal{T}(\Sigma, \mathcal{X})$ l'ensemble des motifs étiquetés par les symboles de Σ et contenant des variables de $\mathcal{X}$. Par exemple, considérons le programme OCaml suivant représentant la fonction de tri par insertion :

```
let rec sort = function
        | [] -> []
        | x::l -> insert x (sort l)

and insert x = function
        | [] -> x::[]
        | y::l -> if x y then x::y::l else y::(insert x l)
```

Ce programme peut être représenté par le TRS $\mathcal{R}$ suivant (les variables sont soulignées) :

$$\begin{aligned} sort(nil) &\to nil \\ sort(cons(\underline{x}, \underline{l})) &\to insert(\underline{x}, sort(\underline{l})) \\ insert(\underline{x}, nil) &\to cons(\underline{x}, nil) \\ insert(\underline{x}, cons(\underline{y}, \underline{l})) &\to ite(leq(\underline{x}, \underline{y}), cons(\underline{x}, cons(\underline{y}, \underline{l})), cons(\underline{y}, insert(\underline{x}, \underline{l}))) \end{aligned}$$

Ici l'ensemble des symboles Σ est par convention séparé en deux sous-ensembles $\mathcal{F} = \{sort, insert\}$ contenant les « symboles de fonctions » et $\mathcal{C} = \{0, s, nil, cons\}$ contenant les « symboles constructeurs » permettant de représenter les valeurs du programme.

En complément, let systèmes de réécritures peuvent êtres utilisés pour représenter des ensembles réguliers de termes, appelés « languages réguliers », à l'aide d'*automates d'arbres*. Un automate d'arbre, noté $\mathcal{A}$, est un quadruplet $\langle \Sigma, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$ où Σ est un ensemble de symboles, $\mathcal{Q}$ un ensembles d'états de l'automate, $\mathcal{Q}_f \subseteq \mathcal{Q}$ un sous ensemble d'état finaux et Δ un TRS défini sur les motifs $\mathcal{T}(\Sigma, \mathcal{Q})$ définissant les transitions de l'automate. Le langage reconnu par $\mathcal{A}$, noté $\mathcal{L}(\mathcal{A})$ contient tous les termes $t \in \mathcal{T}(\Sigma)$ tel que $t \to_\Delta^* q$ où q est un étal final de l'automate. Par exemple,

l'automate suivant peut être utilisé pour reconnaitre l'ensemble des listes d'entiers naturels :

$$\begin{aligned} nil &\to q_{list} & cons(q_{\mathbb{N}}, q_{list}) &\to q_{list} \\ 0 &\to q_{\mathbb{N}} & s(q_{\mathbb{N}}) &\to q_{\mathbb{N}} \end{aligned}$$

Ici l'état q_{list} est un état final de l'automate, et $q_{\mathbb{N}}$ un état intermédiaire reconnaissant tous les entiers naturels. Le termes $cons(s(0), nil)$ est reconnu grâce à la séquence de réécriture suivante :

$$cons(s(0), nil) \to cons(s(q_{\mathbb{N}}), nil) \to cons(q_{\mathbb{N}}, nil) \to cons(q_{\mathbb{N}}, q_{list}) \to q_{list}$$

La combinaison des termes de réécritures avec les automates d'arbre nous servent a représenter nos propriétés à vérifier sur les programmes d'ordre supérieur. Par exemple, si nous voulons vérifier la propriété suivante sur le programme de tri par insertion : « pour toute liste l, `sort l` retourne toujours une liste triée ». Cette propriété peut être exprimée comme suit à l'aide du système de réécriture $\mathcal{R}$ défini plus haut :

$$\forall \underline{l}.\ sorted(sort(\underline{l})) \not\to^*_{\mathcal{R}} false$$

où *sorted* est défini à l'aide des nouvelles règles de réécriture suivantes :

$$\begin{aligned} sorted(nil) &\to true \\ sorted(cons(\underline{x}, nil)) &\to true \\ sorted(cons(\underline{x}, cons(\underline{y}, \underline{l}))) &\to and(leq(\underline{x}, \underline{y}), sorted(cons(\underline{y}, \underline{l}))) \end{aligned}$$

Une méthode commune pour vérifier ce type de propriétés dites « de sûreté » est de calculer une abstraction du programme sur-approchant son comportement sur l'ensemble des entrées d'intérêt. Dans notre contexte, l'ensemble des entrées d'intérêt est $I = \{\ sorted(sort(l)) \mid l \in List\ \}$. Notre objectif est donc de calculer un ensemble de termes O tel que

- $\mathcal{R}^*(I) \subseteq O$, où $\mathcal{R}^*(I) = \{\ u \mid s \to^*_{\mathcal{R}} t, s \in I\ \}$; et
- $false \notin O$

Si un tel O existe, celui-ci prouve la validité de notre propriété. Dans le cas des problèmes réguliers dans le quel nous nous plaçons, O est régulier et peut être représenté à l'aide d'un automate d'arbre.

Abstractions équationnelles

Une première idée pour calculer un tel ensemble O est d'utiliser l'Algorithme de Complétion d'Automate [Gen16]. En partant d'un automate $\mathcal{A}^0$ reconnaissant I, cet algorithme est capable de calculer une série d'automates $\mathcal{A}^1, \mathcal{A}^2, \ldots$ convergeant vers $\mathcal{A}^*$ reconnaissant $\mathcal{R}^*(I)$. Tel quel, la terminaison de cet algorithme n'est pas garantie pour tout automate initial $\mathcal{A}^0$ [Gen16]. De plus, l'abstraction obtenue via $\mathcal{A}^*$ peut être trop large pour vérifier une propriété donnée. Par exemple dans notre cas, avoir $false \in \mathcal{L}(\mathcal{A}^*)$ ne permet pas de vérifier notre propriété sur le tri par insertion.

Un solution proposée par Genet [GR10] est de guider l'algorithme de complétion à l'aide d'équations sur les termes. Pour un ensemble d'équation E donné l'idée est la suivante : pour chaque équation $u = v \in E$ (où u et v sont des motifs), si $s = u\sigma$

et $t = v\sigma$ alors s et t seront reconnus par le même état dans $\mathcal{A}^*$. Ce système permet effectivement de contrôler finement la qualité de l'abstraction réalisée. Pour en plus assurer la terminaison de l'algorithme, Genet propose de considérer E composé de trois ensembles d'équations [Gen16] :

- E^c un ensemble d'équations « contractantes » de la forme $t = t|_p$ (où $t|_p$ est le sous-terme de t à la position p). Ces équations permettent de contrôler l'abstraction réalisée par l'algorithme.
- $E_{\mathcal{R}} = \{\ u = v \mid u \to v \in \mathcal{R}\ \}$
- $E_r = \{\ f(x_1, \ldots, x_n) = f(x_1, \ldots, x_n) \mid f \in \Sigma\ \}$ assurant le déterminisme de l'abstraction.

Genet montre que si $\mathcal{T}(\mathcal{C})$ a un nombre fini de formes normales par rapport à $\vec{E^c} = \{\ u \to v \mid u = v \in E^c\ \}$, alors l'Algorithme de Complétion d'Automate termine toujours en utilisant l'ensemble d'équations $E = E^c \cup E_{\mathcal{R}} \cup E_r$. En théorie, résoudre un problème régulier revient alors à trouver le bon ensemble d'équations E^c. En pratique cependant, cette méthode ne marche que dans le cas des programmes de premier ordre. Dans le cas des programmes d'ordre supérieur, les fonctions deviennes des valeurs comme les autres. La séparation entre symboles de fonction ($\mathcal{F}$) et symboles constructeurs ($\mathcal{C}$) n'a plus de sens. Pour garantir la terminaison de la procédure, il est donc nécessaire de trouver un ensemble E^c contractant sur tout $\mathcal{T}(\Sigma)$ et non pas seulement $\mathcal{T}(\mathcal{C})$. Malheureusement chercher un ensemble d'équations contractantes sur un si grand ensemble est trop coûteux en pratique.

Pour cette raison, dans le Chapitre 4 de cette thèse, nous définissons une nouvelle classe de TRS représentant des programmes d'ordre supérieur et continuant d'assurer la terminaison de l'algorithme de complétion avec E^c contractant sur les valeurs non fonctionnelles. En combinaison avec la définition d'une procédure de recherche d'équations implémentée dans Timbuk 3 [Tbk3], nous somme capable de résoudre automatiquement une variété de problèmes réguliers sur des programmes d'ordre supérieur [Exp3]. Un résultat majeur de la formalisation de cette procédure et de nos expérience est la démonstration de l'incomplétude de cette technique sur les problèmes réguliers. La cause de cette incomplétude est l'introduction des ensembles d'équations $E_{\mathcal{R}}$ et E_r. D'un coté E_r limite l'ensemble des abstractions atteignables aux abstractions « functionelles », dans lesquelles chaque terme est abstrait par un unique élément. De l'autre, $E_{\mathcal{R}}$ limite l'ensemble des abstractions atteignables aux abstractions « éfondrantes », dans lesquelles chaque deux termes liés par la relation de réécriture $\to_{\mathcal{R}}$ sont abstraits par le même élément. Nous montrons que parce que les abstractions générées sont à la fois fonctionnelles et éffondrantes, cette procédure ne permet pas de résoudre n'importe quel problème régulier.

Inférence de types réguliers

Pour résoudre les problèmes engendrée par notre première technique, nous introduisons dans le Chapitre 5 une nouvelle technique de vérification automatique, complète sur les problèmes réguliers. Cette nouvelle procédure est fondée sur l'interprétation abstraite du système de réécriture représentant le programme vérifié. Celle-ci est composée de deux éléments :

- Un domaine abstrait $\Lambda = \langle \Sigma^\#, \Delta^\# \rangle$ (un automate) définissant l'abstraction de $\mathcal{T}(\Sigma)$, où $\Sigma^\#$ est un ensemble de valeur abstraites et $\Delta^\#$ un système de réécriture

composé de règles de la forme $f(v_1^\#, \ldots, v_n^\#) \to v^\#$ avec $v, v_1, \ldots, v_n \in \Sigma^\#$. Un terme t de $\mathcal{T}(\Sigma)$ est abstrait par $v^\# \in \Sigma^\#$ ssi $t \to^*_{\Delta^\#} v^\#$.

- Un système de réécriture $\mathcal{R}^\#$ définissant l'abstraction de $\mathcal{R}$, composé de règles de la forme $f(v_1^\#, \ldots, v_n^\#) \to v^\#$ avec $v, v_1, \ldots, v_n \in \Sigma^\#$.

En considérant une nouvelle fois notre propriété sur le tri par insertion :

$$\forall \underline{l}.\ sorted(sort(\underline{l})) \not\to^*_{\mathcal{R}} false$$

Dans notre contexte d'interprétation abstraite, prouver cette propriété revient à trouver Λ et $\mathcal{R}^\#$ tel que :

$$sorted(sort(l^\#)) \not\to^*_{\mathcal{R}^\# \cup \Delta^\#} false^\#$$

où $l^\#$ abstrait toutes les listes et $false^\#$ abstrait la valeur *false*. Cette nouvelle formulation fait disparaitre la quantification universelle. Pour trouver automatiquement Λ et $R^\#$ à partir de cette nouvelle formulation nous proposons une procédure d'inférence de type consistant à typer le terme d'intérêt (ici $sorted(sort(\underline{l}))$) ainsi que le reste du TRS $\mathcal{R}$ avec des types réguliers. Cette procédure d'inférence de type est une analyse arrière, analysant chaque fonction du programme pour lui trouver une signature régulière. Par exemple, notre procédure d'inférence de type affecte la signature suivante aux fonction de notre programme de tri par insertion :

$$\begin{aligned} insert(ab^\#, sorted^\#) &\to sorted^\# \\ sort(l^\#) &\to sorted^\# \\ sorted(sorted^\#) &\to true^\# \end{aligned}$$

où $sorted^\#$ est une nouvelle valeur abstraite (un langage régulier) inférée par notre analyse pour typer les liste triées. Cette valeur, ajoutée dans Λ, est « apprise » en analysant la fonction récursive *sort* avec l'aide de notre nouvelle procédure d'apprentissage d'invariants réguliers : une procédure de raffinement d'abstraction guidée par les contre exemples et assistée par un solveur SMT. Dans notre exemple, les signatures trouvées permettent d'affecter l'annotation de type suivante (et seulement cette annotation) au terme d'intérêt, prouvant ainsi la validité de notre propriété :

$$sorted(sort(\underline{l} : l^\#) : sorted^\#) : true^\#$$

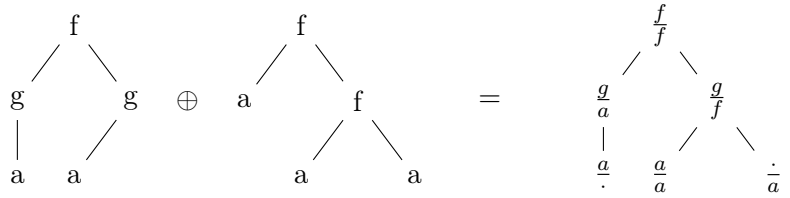
À la fin de l'analyse, les types affectés au TRS constituent une abstraction Λ valide, tandis que les signatures de chaque fonction constituent une abstraction $\mathcal{R}^\#$ du programme validant la propriété. Les abstractions générées par cette technique sont toujours déformantes, mais ne sont pas fonctionnelles contrairement à notre précédente technique, permettant ainsi la résolution de n'importe quel problème régulier.

Relations régulières

Si la technique proposée au Chapitre 5 permet de traiter n'importe quel problème régulier, la portée de ceux-ci ne permet pas de vérifier des propriétés établissant des relations entre les variables du programme. Par exemple, considérons le langage $Eq = \{\ eq(s,t)\ |s \in \mathbb{N}, t \in \mathbb{N}, s = t\ \}$ représentant la relation d'égalité sur les entiers naturels. Ce langage n'est pas régulier : il ne peut pas être représenté par un automate d'arbre. Par conséquent, un problème dont la résolution dépend de la représentation

de ce langage n'est pas un problème régulier. De manière générale, si le problème considéré nécessite la vérification d'une relation inductive entre deux valeurs (ou plus), alors celui n'est pas régulier. Il est donc en dehors du champ des deux techniques de vérifications proposées jusqu'ici.

Dans la Chapitre 6 nous proposons une technique de vérification automatique permettant de contourner cette limitation. Cette technique repose sur l'inférence automatique de « relations régulières » : des langages réguliers encodant des relations à l'aide d'un opérateur de convolution sur les arbres. Dans ce contexte, la relation Eq mentionnée plus haut peut-être indirectement représentée par le langage régulier $Eq_{\oplus} = \{\ s \oplus t \mid eq(s,t) \in Eq\ \}$ où $\oplus$ est un opérateur de convolution sur les arbres [Tata]. Celui-ci est défini comme la « superposition » des arbres s et t illustrée comme suit :



Contrairement aux techniques précédentes, les problèmes considérés sont décrits à l'aide de systèmes de clauses de Horn contraintes (CHC) : un ensemble de clauses de la forme

$$\forall \mathcal{X}.\ \alpha_1 \wedge \cdots \wedge \alpha_n \Rightarrow \alpha_{n+1}$$

où chaque α_i correspond à l'application d'une relation abstraite $p(x_1, \ldots, x_m)$ sur des variables de $\mathcal{X}$, où m est l'arité de la relation p. Résoudre un tel problème revient à trouver un modèle M (une instanciation de chaque relation) tel que M satisfait le système CHC. Dans notre cas, chaque instanciation de relation est donnée sous forme d'une relation régulière tel que défini plus haut. Par exemple, considérons le problème suivant :

$$eq(0,0) \Rightarrow \mathit{true} \qquad \forall \underline{x}.\ eq(s(\underline{x}),0) \Rightarrow \mathit{false}$$
$$\forall \underline{x}, \underline{y}.\ eq(\underline{x},\underline{y}) \Rightarrow eq(s(\underline{x}),s(\underline{y})) \qquad \forall \underline{y}.\ eq(0,s(\underline{y})) \Rightarrow \mathit{false}$$

Ici eq est une relation abstraite. La seule solution à ce problème est le modèle M tel que $M(eq) = Eq_{\oplus}$. Pour inféré automatiquement ce langage nous définissons la première procédure d'inférence de relations régulière, basée sur ICE [GLMN14] : une technique d'apprentissage par implications et contre exemples. Cette procédure est composée de deux algorithmes appelés en boucle jusqu'à ce que la procédure converge vers une solution :

- un « étudiant » chargé de proposer de nouveaux modèles candidats à partir d'un ensembles de contraintes d'apprentissages (initialement vide)
- un « enseignant » chargé d'évaluer les modèles proposés par l'étudiant et, tant que ceux-ci sont invalides, générer de nouvelles contraintes d'apprentissage.

Nous proposons une implémentation pour chacun de ces deux composants. Pour l'étudiant, nous définissons une variation de notre algorithme d'inférence de langage définit dans le Chapitre 5 adapté aux relations régulières. Pour l'enseignant, nous définissons un algorithme de recherche de contre-exemples basé sur la recherche de « chemins synchronisés » dans des automates d'arbres. Nous montrons qu'en posant quelques hypothèses sur le système CHC considéré, cette procédure permet d'inférer n'importe quel modèle régulier.

Chapter 1

Introduction

This thesis studies how *regular languages of trees* combined with *term rewriting systems* can be used to *automatically verify* properties on *higher-order purely functional programs* to facilitate the detection and correction of software bugs. Many tools has been developed to help developers testing their programs and tracking down the bugs. Unfortunately, except in rare cases, testing is not sufficient to guarantee the absence of bugs. In most cases a more formal approach is required to *prove* the correctness of the program. In practice however, using proof assistants such as Coq [Inr16] or Isabelle/HOL [NPW02] to help the developer carry out the proof still requires a lot of time and expertize. Such an investment can be hard to consent in the software development world. Our goal in this thesis is to develop new techniques and tools for the programmers to develop safer programs while reducing the time and expertize needed to verify them. In particular, we focus on the automatic verification of *regular safety properties*, a family of properties for which completely automatic verification can be achieved; and on *higher-order purely functional programs*, a family of programs naturally suited to formal verification.

There already exists many techniques detailed in Chapter 3 whose purpose is to automate parts of the verification process. However most of the time they do not provide any guarantees of completeness on any family of properties. All of these techniques rely on the construction and analysis of an abstraction of the program execution allowing the verification of the given property. Among these abstraction methods, Genet at al. [GR10, Gen16] suggests to use term rewriting systems combined with the Tree Automata Completion algorithm to generate *regular abstractions* of the program execution, which allows us to foresee a complete verification procedure for regular properties. However the proposed method is not yet complete and cannot handle higher-order programs. This is the purpose of this thesis where:

- We extend the abstraction method proposed by Genet et al. in order to verify regular properties on higher-order programs. We then discuss the limits of this technique.
- We go beyonds these limits by developing a new abstraction procedure based on the Tree Automata Completion algorithm and SMT solving. This helps us design a entire verification procedure that is complete on regular safety properties and complete in refutation.
- We then extend this abstraction procedure to go beyond regular properties and verify relational properties, still using regular tree languages.

1.1 Motivation

Bugs are everywhere. They are often associated to catastrophic failures such as the destruction of the Ariane 5 rocket prototype due to a bug in the guidance program, but such major events often hide the massive presence of bugs in non-critical software where their effect is much less spectacular. In 1978, [LST78] showed that programmers focus 17% of development effort in fixing bugs. On the GitHub development platform, 20% of program changes (commits) across projects are dedicated to fixing bugs [AF20]. Even though rarely deadly, it is estimated [RTI02] that software bugs cost $59 billion per year for the U.S. economy alone (0.6% GDP), in addition to being fairly annoying for users. Development methodologies are often used to try to avoid the apparition of bugs, without eradicating them. This motivates the development of methods to detect and fix bugs before they can create damage.

1.1.1 Testing

The most natural way of finding and correcting bugs is by experimenting. The program is verified on a series of inputs, by comparing the resulting output against the expected result given by the programmer. Writing tests is considered as a good practice, and a discipline on its own: all tests are not useful, some are redundant and it is easy to miss the ones that are truly important.

Example 1.1.1. *Consider the following (buggy) program written in the higher-order functional programming language OCaml:*

```
let rec sort leq = function
        | [] -> []
        | x::l -> insert leq x (sort leq l)

and insert leq x = function
        | [] -> x::[]
        | y::l -> if leq x y then x::(insert leq y l) else y::x::l
```

We want to check that the sort function always outputs a sorted list which can be formally stated as:

$$\forall l.\ sorted\ leq\ (sort\ leq\ l) = true$$

where sorted is a predicate function checking that its given list is sorted. To simplify we will consider lists of As and Bs for which the comparison function leq given to sort is defined as follows (where A takes precedence over B):

```
let leq x y = match x, y with
        | B, A -> false
        | _, _ -> true
```

To test this property, one can evaluate sorted leq (sort leq l) with different values of l to ensure that it always return true. However choosing the wrong values for l can be treacherous and give a wrong vision of the behavior of sort. For instance here, every l of length 0, 1 and 2 verifies our property. One could be tempted to generalize this observation and conclude that it is verified for any length.

Many tools have been developed to help programmers in their testing process, proposing new tests and analyzing the relevance of existing tests. One of the goals is generally to ensure that every line of code is tested (visited during the process of at least one test input), and that every edge case is captured. Our previous exemple

shows that it is not enough: the property on *sort* is verified for the edge case (the empty list `[]`), and for `A::B::A::[]` for which every part of the code is visited. The property remains incorrect in general.

Testing can filter common mistake, such as syntax errors, divisions by zero, invalid bounds, etc. However unless the program is tested on every possible inputs (which is usually far from being practicable) this does not give any general warranties on the program's correctness. This is why bugs continue to be spotted even on heavily tested programs. Detecting those bugs requires a more formal approach.

1.1.2 Proof Assistants

The only way to ensure the absence of bugs in a program is to formally prove its correctness. This require being able to formally state the specification the program implements, and also being able to reason about the program itself. If historically such proof have mostly be done by hand on paper, proof assisting tools are now regularly used to help the user through the development of the formal proof. The assisting can take multiple forms, from the verification and diagnostic of program annotations to the verification of complete proof edifices.

Mechanized proof assistants are the most complete form of program verification. An interactive theorem prover such as Coq [Inr16] or Isabelle/HOL [NPW02] can collaborate with the user to build and check complex proof of a wide range of properties. In this settings, the development is usually staged in three phases: the program definition generally as a higher-order functional program, the specification and the proof. For instance, the following code defines the insert-sort algorithm (not buggy this time) in the Coq proof assistant using the Gallina language:

```
Fixpoint insert T (leq : T -> T -> bool) e l :=
        match l with
                | nil => e::nil
                | x::l' => if leq e x
                           then e::x::l'
                           else x::(insert T leq e l')
        end.

Fixpoint sort T (leq : T -> T -> bool) l :=
        match l with
                | nil => nil
                | x::l => insert T leq x (sort T leq l)
                end.
```

Once again we want to specify and prove that the output of the `sort` function is always a list sorted with respect to the input ordering function *leq*. This can be done in Coq by first defining the inductive predicate `sorted` that can then be used in our specification lemma:

```
Inductive sorted T leq : list T -> Prop :=
        | sorted_nil : sorted T leq nil
        | sorted_single x : sorted T leq (x::nil)
        | sorted_more x y l : leq x y = true -> (sorted T leq (y::l))
    -> sorted T leq (x::y::l).
end.

Lemma sort_sorts : forall T leq l, sorted T leq (sort T leq l).
```

The proof of the latter lemma, *sort_sorts*, is then developed in a specific environment, using a variety of proof commands understood by Coq (defined in *The Vernacular* [Inr16]). This includes starting a proof by induction (using the `induction`

command), simplifying the current proof goal (*simpl*) or applying definitions, lemma and theorems (*apply*), etc.

```
Proof.
        intros T leq l. induction l.
                + simpl. apply sorted_nil.
                + simpl. apply insert_preserves_sorting. assumption.
Qed.
```

Proof assistants have been successfully used to develop and prove the correctness of critical softwares such as the certified C compiler CompCert [Inr05] and seL4 micro-kernel [KEH+09].

However powerful, proof assistants completely depend on the user to carry out the proof. A certain degree of automation is possible through the use of *tactics*, but even simple proofs still need to be detailed, in particular to explicit the needed invariants of every loop of the program. For instance the preceding proof requires the definition and proof of the intermediate lemma *insert_preserves_sorting* ensuring the needed invariant on the *insert* function that it preserves "sortedness". The proof of this auxiliary lemma is in fact the main difficulty of the overall proof:

```
Lemma insert_preserves_sorting : forall T leq e l, sorted T leq l ->
    sorted T leq (insert T leq e l).
Proof.
        intros T leq e l l_sorted.
        induction l_sorted.
                + unfold insert. apply sorted_single.
                + unfold insert. destruct (leq e x) eqn:Hleq.
                        * apply sorted_more. auto.
                          apply sorted_single.
                        * apply sorted_more. auto.
                          apply sorted_single.
                + simpl. destruct (leq e x) eqn:Hleq.
                        * apply sorted_more. auto.
                          apply sorted_more. assumption. assumption.
                        * destruct (leq e y) eqn:Hleq'.
                                - apply sorted_more.
                                  auto. apply sorted_more. auto.
                                  assumption.
                                - apply sorted_more. assumption.
                                  simpl in IHl_sorted.
                                  destruct (leq e y).
                                        ** apply diff_true_false
                                           in Hleq'.
                                           contradiction.
                                        ** assumption.
Qed.
```

Overall the high level of expertize and the amount of work needed to design the proofs and to master the interactions with the assistant may discourage users to use such proof assistants to verify non critical softwares. It is not always clear that the cost due to the presence of bugs is worth such an investment.

1.1.3 Automated Verification Techniques

To reduce the cost of the proof, a wide variety of techniques has been developed that introduce degrees of automation in the verification process. Of course, automation has a cost: the less manual work remains for the user, the narrower the range of verifiable properties is. On one side of this spectrum, proof assistants are polyvalent tools that can verify a wide range of properties but with little to no automation. The

user needs to write most of the proof. One the other side of this spectrum, we can put for instance the Hindley-Milner type inference algorithm [Hin69, Mil78]. It is a completely automatic verification algorithm for higher-order functional programs that can verify only one simple property: type safety. Type safety consist in separating the values into exclusive families, types, and verifying that at all points in the programs, the input and output of each function call matches the expected types, so that the program never stops before reaching the result of the computation. Type safety does not verify that the *sort* function above returns a sorted list, but it can at least ensure that the output is a list.

We will see in Chapter 3 that there are numerous techniques developed in between, balancing automation and expressiveness. All of them are based on some abstraction procedure, simplifying the description of the program execution. The Hindley-Milner type inference algorithm itself can be seen as an abstraction procedure where each program execution state is abstracted into a type (the type of the value it returns). Many of the verification techniques such as [MI13, RKJ08] are based on more sophisticated type inference systems where types can describe more complex, non exclusive, family of values. But such techniques require the user to annotate the program in order to carry out the proof.

1.1.4 Regular Verification Problems

A verification problem is the conjunction of a program and a property. Solving a problem consists in proving or disproving that the program verifies the property. It is impossible to define the family of problems that can be automatically verified without requiring some hints of any form from the user. We know that such problem exists, type safety being one of them. In addition, there already exists some verification techniques explored in Chapter 3 such as [Kob09b, KSU11b, SK17] that can already solve problems automatically. Such techniques however focus on relational properties over numerical data-types which is already far more challenging than type safety properties. They are generally not good at verifying *regular problems* that involve non-relational properties on algebraic data-types (trees). In some ways regular problems are simpler as they do not involve any relations between the variables of the program. Instead they can be solved with a regular abstraction of the program, in which each abstract state can be described by a regular language. Type safety is a good example of a regular problem where each state can be abstracted by its type usually defined as a regular language in the program itself. However in general, the abstraction depends on the property to verify and can be far more complex. This makes the verification of regular problems generally far more complicated than simple type safety checking. Our previous *sort* example is a typical regular problem that could easily and automatically be solved by such verification technique. This is because as long as the values in the lists are taken from a finite domain of values (for instance $\{ A, B \}$ in our initial example), there exists a regular abstraction of the program in which *sort* outputs sorted lists. Note that regular problems still represent a small fraction of all the verification problems needed to prove the correctness of a program. The development of automated regular verification techniques does not aim at replacing proof assistants. However it can benefits proof assistants by adding more degrees of automation. In addition the software development process can be sped up by reliably filtering more bugs without spending time writing exhaustive tests.

1.2 Our Verification Framework

In Chapter 3 we see that many theoretical frameworks have been used to tackle the verification of higher-order functional programs with various degrees of automation. Since this thesis focuses on regular problems where execution states are abstracted by regular tree languages, we naturally chose to focus on tree automata [Tata], themselves built on top of term rewriting system [BN98]. Tree automata provide a simple representation of regular tree languages and, thus, of algebraic data types found in functional programs. Meanwhile, term rewriting systems provide a convenient way of representing higher-order functional program execution in interaction with tree automata.

1.2.1 Term Rewriting Systems

TRSs are a widely used theoretical representation of computational systems [BN98] that is especially convenient to model features found in modern functional programming languages, such as pattern matching.

Example 1.2.1 (From OCaml to TRS)**.** *Here is an example of the translation of the insert-sort algorithm written in OCaml (on the top) to a term rewriting system (on the bottom). This time for the sake of readability the comparison function leq is not a parameter of the functions, but is defined elsewhere in the program.*

```
1  let rec sort = function
2  | [] -> []
3  | x::l -> insert x (sort l)
4
5
```

```
6  and insert x = function
7  | [] -> x::[]
8  | y::l -> if leq x y
9            then x::y::l
10           else y::(insert x l)
```

$$
\begin{aligned}
sort(nil) &\to nil \\
sort(cons(\underline{x},\ \underline{l})) &\to insert(\underline{x},\ sort(\underline{l})) \\
insert(\underline{x}) &\to nil \\
insert(cons(\underline{x},\ \underline{l})) &\to ite(leq(\underline{x},\ \underline{y}),\ cons(\underline{x},\ cons(\underline{y},\ \underline{l})),\ cons(\underline{y},\ insert(\underline{x}\ \underline{l})))
\end{aligned}
$$

Each case of each function is translated into one rewriting rule.

In this framework, terms appearing on both sides of each rule represent states of the execution, and rewriting rules represent the logic, or semantics, of the program. If we name $\mathcal{R}$ the TRS modeling the insertion sort algorithm along with the *sorted* predicate function, our previous property on the *sort* function can then be restated as a rewriting problem as follows:

$$\forall \underline{l}.\ sorted(sort(\underline{l})) \not\to^*_{\mathcal{R}} false$$

where $\to^*_{\mathcal{R}}$ is the reflexive, transitive closure of the rewriting relation with regard to $\mathcal{R}$. We want to check that for any list l, the term $sorted(sort(l))$ *never* rewrites to *false* using $\mathcal{R}$ (it is a short way of ensuring it *always* either diverges or rewrites to *true* assuming the program is type safe). Another way to express our problem is by using sets of terms later called *languages* of terms. If we name $\mathcal{L}$ the language of all lists (of As and Bs), and I the language of input states $\{\ sorted(sort(l)) \mid l \in \mathcal{L}\ \}$ then we want to check:

$$false \notin \mathcal{R}^*(I)$$

where $\mathcal{R}^*(I)$ is the set of terms that are *reachable* from I using the TRS $\mathcal{R}$ defined as $\{ u \mid t \in I \wedge t \to_{\mathcal{R}}^* u \}$. Solving this problem can then be done by computing $\mathcal{R}^*(I)$. In our setting, we require I to be a *regular language* which means it can be represented using a *Tree Automaton*.

1.2.2 Tree Automata and Regular Languages

Tree automata [Tata] are a convenient way of representing *regular languages of terms*. A tree automaton is defined as a rewriting system rewriting terms into *states*, representing the nature of the term. We say that a state "recognizes" a term when the term rewrites into this state.

Example 1.2.2. *We previously named $\mathcal{L}$ the language of all lists of As and Bs. Here is a tree automaton recognizing $\mathcal{L}$:*

$$\begin{aligned} A &\to ab & \qquad nil &\to list \\ B &\to ab & cons(ab, list) &\to list \end{aligned}$$

Here ab and $list$ are states of the automaton recognizing the languages $\{A, B\}$ and $\mathcal{L}$ respectively. For instance, the term $cons(A, cons(B, nil))$ of $\mathcal{L}$ is recognized by $list$ using the following rewriting sequence: $cons(A,\ cons(B,\ \underline{nil})) \to cons(A,\ cons(\underline{B},\ list)) \to cons(A,\ \underline{cons(ab,\ list)}) \to cons(\underline{A},\ list) \to \underline{cons(ab,\ list)} \to list$.

By definition, any regular language can be represented using a tree automaton. In addition, the combination of tree automata and rewriting systems has already lead to several algorithms designed for the computation of $\mathcal{R}^*(I)$ when I is regular and $\mathcal{R}$ satisfies some properties. One of them is the Tree Automata Completion (TAC) algorithm [Gen16]. Remember that computing $\mathcal{R}^*(I)$ where I is the set of initial states of a program is the key to the verification of safety properties. If I is recognized by a tree automaton $\mathcal{A}$, then the TAC algorithm aims at computing a new automaton $\mathcal{A}^*$ that recognizes $\mathcal{R}^*(I)$ by completing $\mathcal{A}$ with the right transitions. Of course, computing such $\mathcal{A}^*$ is not possible in general, in particular when $\mathcal{R}^*(I)$ is not regular. In this case, instead the TAC algorithm will compute an *over-approximation* of $\mathcal{R}^*(I)$. In [GR10], this is done by abstracting the program execution using a set of equations over terms. A safety problem is *regular* when there exists a regular over-approximation of $\mathcal{R}^*(I)$ precise enough to solve it. In theory then, given the right set of equations any regular problem can be solved using the TAC algorithm [Gen18]. However there exists no automatic method to find the correct equations given a regular problem. Some leads are given by Genet at al. [Gen16] to facilitate the search of equations for first-order programs using "contracting" equations, but this does not guarantee the termination of the TAC algorithm while analyzing higher-order programs.

1.3 Contributions

In this thesis we explore new automatic verification techniques dedicated to the solving of regular problems on higher-order functional programs, using term rewriting systems and tree automata.

1.3.1 Equational Abstractions for Higher-Order Programs

We pursue the development of the equations-based abstraction procedure defined in [Gen16]. To go beyond the lack of termination guaranties for higher-order functional programs, we state and prove a general termination theorem for the Tree Automata Completion algorithm using contracting equations. From the conditions of the theorem we characterize a class of higher-order functional programs for which the completion algorithm terminates using contracting equations. In addition we formally define and discuss the completeness of the abstraction procedure based on contracting equations and show its *functional collapsing regular completeness* and completeness in refutation. In other words, if there exists a set of contracting equations able to solve the given safety problem, the procedure will eventually find it, and if there exists a counter-example to the verified property, the procedure will eventually find it. Through our OCaml implementation of the procedure in Timbuk 3 [Tbk3], we show how contracting equations allows us to solve regular safety problems even with such a simple procedure. However we also see the limitations of contracting equations, the lack of modularity and their inability to capture many regular abstractions.

1.3.2 Regular Type Inference

To solve these problems we present a new fully automatic verification technique for higher-order functional programs targeting regular safety problems. We solve the incompleteness arising from the use of contracting equations by defining a *regular abstract interpretation framework* where abstractions are directly and more precisely defined as non-deterministic tree automata. We solve the scalability issue by modularizing the analysis of each function of the program. To do that we formulate the abstraction procedure as regular type inference procedure, where each type is a regular language represented by a state of the tree automaton abstraction. Finally we improve the abstraction inference procedure itself by designing an actual regular invariant learning procedure, able to iteratively learn regular languages from examples and counter examples. We show that the resulting abstraction procedure is regularly complete and complete in refutation. We implemented this new verification technique in a new version of Timbuk (4) and compared it against Timbuk 3. The results show that it can handle more regular problems in comparable times while greatly improving the memory usage.

1.3.3 Regular Relations

In a final chapter we show how to extend our abstraction inference procedure to verify non-regular relational properties with regular tree languages. This is done by using tree automatic relations [Tata], a representation of relations where each item of the relation is stored as a convoluted term in a regular language. We propose an extension of the convolution operator to capture more relations, and design a regular Constrained Horn Clauses (CHC) solver able to automatically infer such relations based on the ICE inference procedure [GLMN14]. We provide a Rust implementation of this regular CHC solver tested on several case studies.

1.3.4 Summary

The rest of the thesis is structured as follows. Chapter 2 gives the required definitions of terms, rewriting systems and tree automata used throughout the document. It

also gives an entry point to automated verification techniques from the point of view of model checking, and show how it can be used to formally define different family of problems (such as regular problems) in terms of program abstractions. Chapter 3 gives an overview of the literature focusing on automatic verifications techniques on higher-order functional programs. Chapter 4 explores the use of contracting equations to define regular abstraction while guaranteeing the termination of the Tree Automaton Algorithm. In this chapter we define a general termination criteria for the TAC algorithm and a class of higher-order functional programs for which contracting equations can be used without diverging. In Chapter 5 we tackle the shortcomings encountered at the end of the previous chapter with the definition of a novel verification technique based on the inference of regular languages types. Chapter 6 moves away from regular problems to discuss how regular languages can be used to verify relational properties that are out of the scope of the previously defined techniques. Finally, Chapter 7 concludes this thesis.

Chapter 2

Preliminaries

This chapter introduces general notions of terms, rewriting systems and tree languages that we use throughout this document. We show how terms can be used to model program states, how term rewriting systems can model program semantics, and how tree grammars can model program execution. We describe the automatic verification of higher-functional programs from a model checking point of view as the abstraction of individual states into tree languages.

2.1 Trees, Term and Patterns

Definition 2.1.1 (Tree). *A* finite ordered tree *over a set of labels E is a mapping $P \mapsto E$ where $P \subseteq \mathbb{N}^*$ is a prefix-closed set of* positions *which represent a path from the root of the tree to the target node.*

The rest of the section defines a special family of trees, called *terms*, that are particularly well suited to the modeling of higher-order functional program states, and algebraic data types values. Terms are labeled by ranked alphabets.

Definition 2.1.2 (Ranked Alphabet). *A ranked alphabet Σ is a finite set of symbols (labels) attached to an arity function $ar : \Sigma \to \mathbb{N}$. For simplicity, we write $f \in \Sigma^n$ when $f \in \Sigma$ and $ar(f) = n$, and $\{\ f_1 : n_1, \ldots, f_k : f_k\ \}$ to define the ranked alphabet $\{\ f_1, \ldots, f_k\ \}$ where $ar(f_i) = n_i$ for each i.*

In the context of functional program modeling, symbols of the ranked alphabet are used to represent algebraic data type constructors such as *nil* or *cons*, function names and applications.

Definition 2.1.3 (Term). *Let Σ be a ranked alphabet. A* term *over Σ has the form:*

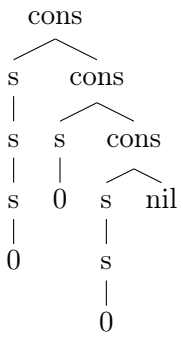
$$f(t_1,\ \ldots,\ t_n)$$

where f is a symbol of Σ^n and for each $i, 1 \leq i \leq n$, t_i is also a (sub)term. The set of terms *over Σ is written $\mathcal{T}(\Sigma)$.*

Every state (expression, values) of higher-order functional programs can be encoded using terms.

Example 2.1.1 (Terms as Values). *Consider the ranked alphabet $\Sigma = \{\ nil : 0,\ cons : 2,\ 0 : 0,\ s : 1\ \}$. This alphabet can be used to build terms representing lists of natural numbers where nil and cons are list constructors, and 0 and s natural number*

constructors using Peano's numbers. For instance, the OCaml list value `3::1::2::[]` *is modeled by the term* $cons(s(s(0))), cons(s(0), cons(s(s(0)), nil)))$. *We sometimes use the following graphical tree representation to represent such terms:*



Example 2.1.2 (Terms as Execution States)**.** *Every state of a higher-order functional program execution is an expression waiting to be reduced into a value. For instance, consider the following OCaml expression* `if a then b else c`. *It can be represented using the term* $ite(a, b, c)$ *where* $ite \in \Sigma$ *is introduced to encode the* `if-then-else` *control flow structure.*

Definition 2.1.4 (Yield)**.** *Let* Σ *be a ranked alphabet. The* yield *of a term* t *of* $\mathcal{T}(\Sigma)$, $Yield(t)$, *is a word of* Σ^* *that is the concatenation of every symbol of the tree from left to right. It can be inductively defined as follows:*

$$Yield(f) = f \qquad \text{if } f \in \Sigma_0$$
$$Yield(f(t_1, \ldots, t_n)) = f \ . \ Yield(t_1) \ . \ \ldots \ . \ Yield(t_n)$$

where "." is the concatenation *operator.*

Definition 2.1.5 (Depth)**.** *The depth of a term* $t \in \mathcal{T}(\Sigma)$, *written* $|t|$ *is inductively defined as*

$$|f| = 0 \text{if } f \in \Sigma_0$$
$$|f(t_1, \ldots, t_n)| = 1 + max(|t_1|, \ldots, |t_n|)$$

Definition 2.1.6 (Position)**.** *A* position *in a term* t *is a prefix-closed word of* $\mathbb{N}^*$ *pointing to a* subterm *of* t. *We write* $t|_p$ *for the subterm of* t *at position* p. *It is defined by:*

$$t|_\lambda = t$$
$$f(t_1, \ \ldots, \ t_n)|_{i.p} = t_i|_p$$

where λ *is the empty word and "." in* $i.p$ *is the* concatenation *operator. We name* $Pos(t)$ *the set of* valid positions p *in* t *for which* $t|_p$ *is defined. A term can then be seen as a tree defined by the mapping* $Pos(t) \mapsto \Sigma$ *giving the head symbol of the tree at a the given position. In addition we write* $t[s]_p$ *for the term* t *where the subterm at position* p *has been replaced by* s.

$$t[s]_\lambda = s$$
$$f(t_1, \ \ldots, \ t_n)[s]_{i.p} = f(t_1, \ \ldots, \ t_i[s]_p, \ \ldots, \ t_n)$$

Example 2.1.3. *Let* $\Sigma = \{cons, \ nil\}$, $t = cons(\underline{x}, \ cons(\underline{y}, \ nil))$ *and* p *the position* $p = 2.1$. *Then,* $t|_\lambda = t$, $t|_p = \underline{y}$ *and* $t[nil]_p = cons(\underline{x}, \ cons(nil, \ nil))$.

Definition 2.1.7 (Subterm Ordering). *We write $s \trianglerighteq t$ or $t \trianglelefteq s$ if there exists a position $p \in Pos(s)$ such that $s|_p = t$. In other words, t is a subterm of s. We write $s \triangleright t$ of $t \triangleright s$ when p is not λ ($s \neq t$). For all set of terms $\mathcal{L} \subseteq \mathcal{T}(\Sigma)$, we write $\mathcal{L}_{\trianglerighteq}$ for the smallest superset of $\mathcal{L}$ closed by subterm.*

Definition 2.1.8 (Pattern). *Let Σ and $\mathcal{X}$ be two disjoint ranked alphabets where $\mathcal{X}$ is a set of constants of arity 0 called* variables. *A term of $\mathcal{T}(\Sigma \cup \mathcal{X})$ is called a* pattern. *We write $Var(p)$ the set of variables occurring in the pattern p ($Var(p) \subseteq \mathcal{X}$). The set of patterns is also written $\mathcal{T}(\Sigma, \mathcal{X})$ to clearly distinguish the symbols from the variables.*

A term without variables is *not* a pattern and is called a *closed* term. To avoid any confusion we avoid the use of "term" to qualify patterns.

Definition 2.1.9 (Linearity). *A pattern p is* linear *if the multiplicity of each variable in p is at most* 1.

Definition 2.1.10 (Substitution). *Let Σ be a ranked alphabet and $\mathcal{X}$ a set of variables. A* substitution *σ is a partial application of $\mathcal{X} \mapsto \mathcal{T}(\Sigma, \mathcal{X})$, mapping variables to terms. We write $Dom(\sigma)$ the set of variables for which $\sigma(x)$ is defined ($Dom(\sigma) \subseteq \mathcal{X}$). We tacitly extend every substitution σ to the endomorphism $\sigma : \mathcal{T}(\Sigma, \mathcal{X}) \mapsto \mathcal{T}(\Sigma, \mathcal{X})$ where $p\sigma$ is the result of the application of the pattern p to the substitution σ.*

Definition 2.1.11 (Context). *Let Σ be a ranked alphabet. A context $C[\,]$ is a term of $\mathcal{T}(\Sigma \cup \{\Box\})$ where $\Box$ is a special "hole" symbol. Here we consider contexts containing a unique symbol $\Box$. We note $C[t] = C[\,][t]_p$ where p is the position such that $C[\,]|_p = \Box$.*

2.2 Rewriting Systems

We have seen that terms are useful to describe execution states of higher-order functional programs. However it does not describe how states are connected one to each other. In this section, we define Term Rewriting Systems, a theoretical framework to describe transition systems between terms using rewriting reduction rules. Most of the definitions in this section are found in *Term Rewriting Systems* [TeReSe] and *Term Rewriting and All That* [BN98] by Baader and Nipkow.

2.2.1 Definitions

Definition 2.2.1 (Rewriting rule). *Let Σ be a ranked alphabet and $\mathcal{X}$ a set of variables. A rewriting rule defined over Σ and $\mathcal{X}$ is a pair $\langle l, r \rangle$ where l and r are patterns of $\mathcal{T}(\Sigma, \mathcal{X})$. We write $l \to r$ for such rewriting rule. The pattern l must not be a variable, and each variable occurring in r must occur in l: $Var(l) \supseteq Var(r)$.*

Definition 2.2.2 (Term Rewriting System). *Let Σ be a ranked alphabet and $\mathcal{X}$ a set of variables. A term rewriting system (TRS) $\mathcal{R}$ over Σ and $\mathcal{X}$ is a set of rewriting rules (over Σ and $\mathcal{X}$). It defines a rewriting relation $\to_{\mathcal{R}}$ in which for all terms $s, t \in \mathcal{T}(\Sigma)$ we have $s \to_{\mathcal{R}} t$ iff there exists a rule $l \to r \in \mathcal{R}$, a substitution $\sigma : \mathcal{X} \to \mathcal{T}(\Sigma)$ and a position $p \in Pos(s)$ such that $s|_p = l\sigma$ and $t = s[r\sigma]_p$. In other words we have $\to_{\mathcal{R}} = \{\ (C[l\sigma], C[r\sigma]) \mid l \to r \in \mathcal{R}\ \}$.*

Definition 2.2.3 (Rewriting path). *Let $\mathcal{R}$ be a TRS defined over Σ and $\mathcal{X}$. For all $n \in \mathbb{N}$ we write $\to_{\mathcal{R}}^n$ the smallest relation such that*

$$\begin{aligned} s \to_{\mathcal{R}}^0 t \quad &\Leftarrow \quad s = t \\ s \to_{\mathcal{R}}^{n+1} t \quad &\Leftarrow \quad s \to_{\mathcal{R}}^n u \wedge u \to_{\mathcal{R}} t \wedge u \neq t \end{aligned}$$

We write $\rightarrow_{\mathcal{R}}^{*}$ for the transitive and reflexive closure of $\rightarrow_{\mathcal{R}}$ such that for any two terms s, t we have $s \rightarrow_{\mathcal{R}}^{*} t$ iff there exists some $n \in \mathbb{N}$ such that $s \rightarrow_{\mathcal{R}}^{n} t$. We say that there exists a rewriting path *between s and t using $\mathcal{R}$ iff $s \rightarrow_{\mathcal{R}}^{*} t$.*

Definition 2.2.4 (Reachable Terms)**.** *If $s \rightarrow_{\mathcal{R}}^{*} t$ we say that t is* reachable *from s. If $\mathcal{L}$ is a set of term (a tree language, as seen in Section 2.3), then $\mathcal{R}^{n}(\mathcal{L})$ is the set of terms reachables from $\mathcal{L}$ in n steps or less: $\mathcal{R}^{n}(\mathcal{L}) = \{\ t \mid s \in \mathcal{L}, t \in \mathcal{T}(\Sigma), k \leq n.\, s \rightarrow_{\mathcal{R}}^{k} t\ \}$. By extension we write $\mathcal{R}^{*}(\mathcal{L})$ for the set of terms reachables from $\mathcal{L}$ in any number of steps: $\mathcal{R}^{*}(\mathcal{L}) = \{\ t \mid s \in \mathcal{L}, t \in \mathcal{T}(\Sigma).\ s \rightarrow_{\mathcal{R}}^{*} t\ \}$.*

Definition 2.2.5 (Irreducible Terms and Normal Forms)**.** *Let $\mathcal{R}$ be a TRS over Σ and $\mathcal{X}$. A term s is* irreducible *w.r.t. $\mathcal{R}$, if it cannot be rewritten: for all term t, if $s \rightarrow_{\mathcal{R}}^{*} t$ then $s = t$. We write $IRR(\mathcal{R})$ the set of terms that are irreducible w.r.t. $\mathcal{R}$. For all terms s, any irreducible term t such that $s \rightarrow_{\mathcal{R}}^{*} t$ is called a* normal form *of s. We then write $s \rightarrow^{!} t$. If t is the unique normal form of s then we write $\overrightarrow{s}^{!} = t$. Finally we say that the term s* diverges *iff it has no normal forms.*

2.2.2 Properties of TRSs

We present here some special properties of term rewriting systems that will be useful throughout the document.

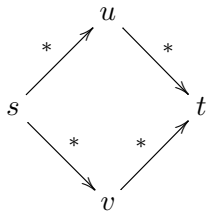
Definition 2.2.6 (Linearity)**.** *A rewriting rule $l \rightarrow r$ is left-linear (resp. right-linear) if l (resp. r) is linear. A TRS $\mathcal{R}$ is left-linear (resp. right-linear) if all its rewriting rules are left-linear (resp. right-linear).*

Left linearity in particular is often found in functional programing languages where it is not possible to use the same name for multiple parameters of the same function (so as to express that they must be the same value) without hiding previous same-name bindings. For instance in the OCaml function definition `let f x x = x`, the first `x` parameter is hidden by the second `x` parameter, they are not the same. It is equivalent to `let f x y = y`. This implies that any TRS encoding an OCaml program will be left-linear.

Definition 2.2.7 (Termination)**.** *A TRS $\mathcal{R}$ is terminating if every term has a normal form with regard to $\mathcal{R}$.*

Definition 2.2.8 (Determinism)**.** *A TRS $\mathcal{R}$ is* deterministic *if every term has at most one normal form with regard to $\mathcal{R}$.*

Definition 2.2.9 (Confluence)**.** *A TRS $\mathcal{R}$ is* confluent *if for each term s and each terms u, v, if $s \rightarrow_{\mathcal{R}}^{*} u$ and $s \rightarrow_{\mathcal{R}}^{*} v$ then there exists a term t such that $u \rightarrow_{\mathcal{R}}^{*} t$ and $v \rightarrow_{\mathcal{R}}^{*} t$:*



It is easy to show that a confluent term rewriting system must also be a deterministic rewriting system since two different normal forms could not be joined by confluence.

Definition 2.2.10 (Orthogonality)**.** *A TRS $\mathcal{R}$ is* orthogonal *if it is left-linear and has no overlapping rules: for every two different rules $l_1 \to r_1, l_2 \to r_2$ of $\mathcal{R}$, there is no position p and substitution $\sigma : \mathcal{X} \mapsto \mathcal{T}(\Sigma, \mathcal{X})$ such that $l_1|_p = l_2\sigma$. In other words, each rule application is independent.*

Any orthogonal TRS is confluent, which in turns means it is deterministic. The study of orthogonal rewriting systems plays a great role in the study of functional programming languages. The term rewriting system extracted from a *purely* functional program is indeed orthogonal since each rule describes the application of an independent function on independent reduced values. This is not the case however for non-purely functional programs introducing side effects or "non-deterministic functions" such as a random number generator.

2.2.3 Usage in Functional Program Verification

We now explore in more details the definition and properties of TRSs encoding first-order and higher-order functional programs, and how it can be used to formalize program verification problems.

First-Order

A first-order functional program can be encoded using a first-order functional TRS defined as follows.

Definition 2.2.11 (First-Order Functional TRS)**.** *A TRS $\mathcal{R}$ is a first-order functional TRS iff it is a left-linear TRS defined over a ranked alphabet $\Sigma = \mathcal{C} \cup \mathcal{F}$ and each rule is of the form*

$$f(p_1, \ldots, p_n) \to r$$

where $f \in \mathcal{F}_n$, $p_i \in \mathcal{T}(\mathcal{C}, \mathcal{X})$ for each $i \in [1, n]$ and $r \in \mathcal{T}(\Sigma, \mathcal{X})$.

In this definition, $\mathcal{F}$ is a ranked alphabet of *function symbols* where the arity of the symbol is the arity of the corresponding function, while $\mathcal{C}$ is a ranked alphabet of *constructor symbols* building values. Each rule of a first-order functional TRS defines the application of a function represented by a function symbol on its value parameters.

Example 2.2.1 (From OCaml to first-order functional TRS)**.** *Consider the following OCaml program defining the insertion-sort algorithm on lists:*

```
let rec sort = function
        | [] -> []
        | x::l -> insert x (sort l)

let rec insert x = function
        | [] -> x::[]
        | y::l -> if x <= y then x::y::l else y::(insert x l)
```

This program can be translated into a first-order functional TRS as follows:

$$\begin{aligned} sort(nil) &\to nil \\ sort(cons(\underline{x}, \underline{l})) &\to insert(\underline{x}, sort(\underline{l})) \\ \\ insert(\underline{x}, nil) &\to cons(\underline{x}, nil) \\ insert(\underline{x}, cons(\underline{y}, \underline{l})) &\to ite(leq(\underline{x}, \underline{y}), cons(\underline{x}, cons(\underline{y}, \underline{l})), cons(\underline{y}, insert(\underline{x}, \underline{l}))) \end{aligned}$$

$$\begin{aligned} ite(\text{true}, \underline{x}, \underline{y}) &\to \underline{x} & leq(0, \underline{y}) &\to \text{true} \\ ite(\text{false}, \underline{x}, \underline{y}) &\to \underline{y} & leq(s(\underline{x}), 0) &\to \text{false} \\ & & leq(s(\underline{x}), s(\underline{y})) &\to leq(\underline{x}, \underline{y}) \end{aligned}$$

The symbols ite and leq are introduced to encode the **`if-then-else`** *control structure and built-in comparison operator <= (here on natural numbers only). The associated rules on those symbols are not extracted from the program but from the OCaml language specification.*

Definition 2.2.12 (Values)**.** *Let $\mathcal{R}$ be a first-order functional TRS defined over $\Sigma = \mathcal{C} \cup \mathcal{F}$. The set $\mathcal{T}(\mathcal{C})$ engendered by the* constructor symbols *alphabet $\mathcal{C}$ is called the set (or language) of* values*. It represents all the possible data values manipulated by the program. Note that no rule can apply on a value alone: $\mathcal{T}(\mathcal{C}) \subseteq IRR(\mathcal{R})$.*

Definition 2.2.13 (Determinism)**.** *Because of the constraints applied on the rules, a first-order functional TRS is* deterministic *iff it is* orthogonal*, in which case it is also* confluent *as we have previously seen.*

Definition 2.2.14 (Completeness)**.** *A first-order functional TRS $\mathcal{R}$ is complete iff $IRR(\mathcal{R}) = \mathcal{T}(\mathcal{C})$: every non-value can be rewritten. In other words, every function of the program is exhaustive.*

Completeness is a strong property that almost never applies to functional programs. To ensure that the execution never stops before reaching a value, most functional languages provide a weaker completeness property on a subset of terms: well-typed terms, which notion usually depends on the type system employed by the language. In our case, all we are interested in is completeness, we thus derive well-typedness from completeness.

Definition 2.2.15 (Well-Typed Terms)**.** *A term is well typed w.r.t. a TRS $\mathcal{R}$ if its normal forms (if any) are values. We write $\mathcal{W}(\Sigma)_{\mathcal{R}}$ the set of terms that are well-typed w.r.t. $\mathcal{R}$, or just $\mathcal{W}(\Sigma)$ when there are no ambiguities on $\mathcal{R}$. If $\mathcal{R}$ is defined over $\Sigma = \mathcal{C} \cup \mathcal{F}$ we have:*

$$\mathcal{W}(\Sigma)_{\mathcal{R}} = \{\ s \mid s \to^{!}_{\mathcal{R}} t \ \Rightarrow\ t \in \mathcal{T}(\mathcal{C})\ \}$$

Note that diverging terms that have no normal forms are well-typed.

Higher-Order

To encode higher-order functional programs we must be able to partially apply functions. However using our previous encoding for first-order programs, this leads to the apparition of malformed terms.

Example 2.2.2. *Let us consider the following OCaml program defining the higher-order map function:*

```
let rec map f = function
        | [] -> []
        | x::l -> (f x)::(map f l)
```

One may be tempted to translate this program into the following TRS:

$$map(\underline{f}, nil) \to nil$$
$$map(\underline{f}, cons(\underline{x}, \underline{l})) \to cons(\underline{f}(\underline{x}), map(\underline{f}, \underline{l}))$$

However this is not a valid TRS. Since $\underline{f}$ is a variable we should have $ar(\underline{f}) = 0$, it is hence impossible to write $\underline{f}(\underline{x})$.

This problem is solved by Reynolds [Rey69] by introducing a special application symbol @, where $@(f, x)$ encodes the (partial) application of a function f on its first parameter x. Of course, applications can be stacked to fully apply a function: $@(@(@(f, x), y), z)$.

Example 2.2.3. *In the previous example, the map program can be translated into the following (valid) TRS using @:*

$$@(@(map, \underline{f}), nil) \to nil$$
$$@(@(map, \underline{f}), cons(\underline{x}, \underline{l})) \to cons(@(\underline{f}, \underline{x}), @(@(map, \underline{f}), \underline{l}))$$

This time every use of $\underline{x}$ is valid by encoding $\underline{f}(\underline{x})$ into $@(\underline{f}, \underline{x})$.

This leads to a more sophisticated definition for higher-order functional TRSs.

Definition 2.2.16 (Higher-Order Functional TRS)**.** *A TRS $\mathcal{R}$ is a higher-order functional TRS over a ranked alphabet $\Sigma = \mathcal{C} \cup \mathcal{F}$ if it is a left-linear TRS defined over $\Sigma \cup \{ @ \}$ and such that for all rule $l \to r$ there exists some $k > 0$ such that $l \in LHS_k$. The set LHS_k is inductively defined for all k as the smallest set such that:*

$$f(t_1, \ldots, t_n) \in LHS_0 \Leftarrow f \in \mathcal{F}_n,\ t_1, \ldots, t_n \in \mathcal{T}(\mathcal{C}, \mathcal{X})$$
$$@(t_1, t_2) \in LHS_{k+1} \Leftarrow t_1 \in LHS_k,\ t_2 \in \mathcal{T}(\mathcal{C}, \mathcal{X})$$

The arity of a function symbolized by f, written $Ar(f)$ is defined as follows:

$$Ar(f) = n \iff \forall\, @(@(\ldots @(f, p_1), \ldots), p_i) \to r \in \mathcal{R}.\ i = n$$

Note that in general the arity of a symbol differs from the arity of the function it represents, $Ar(f) \neq ar(f)$. For all symbols f appearing in a higher-order functional TRS, $Ar(f)$ must always be defined.

We still make the distinction between functional and constructor symbols, however the set of values is extended to include partially applied functions. In fact, any irreducible term can be considered as a valid value, which complicates the definition of completeness for higher-order functional TRSs.

Definition 2.2.17 (Completeness)**.** *A higher-order functional TRS $\mathcal{R}$ defined over $\Sigma = \mathcal{C} \cup \mathcal{F}$ is* complete *iff for all functional symbol $f \in \mathcal{F}$, for all terms $t_1, \ldots, t_n$ where $n = Ar(f)$ the term $@(@(\ldots @(f, t_1), \ldots), t_i)$ is reducible.*

Higher-order functional TRS are often filled with @ symbols, which can make them hard to read. For the sake of readability in this document we use a lighter notation for higher-order functional TRSs where $@(t_1, t_2)$ is simply written $t_1\ t_2$.

Example 2.2.4 (Higher-order notation)**.** *The higher-order functional TRS defined in the previous example can be simplified into the following:*

$$map\ \underline{f}\ nil \to nil$$
$$map\ \underline{f}\ cons(\underline{x}, \underline{l}) \to cons(\underline{f}\ \underline{x}, map\ \underline{f}\ \underline{l})$$

Remember that this is still a first-order term rewriting system as the special application symbol @ is used underneath.

2.3 Tree Languages, Grammars and Automata

A set of term is called a *tree language*. Tree languages can be possibly infinite, but even then can have a finite *presentation*: we can describe the whole set in a finite manner. The preferred representation for string language is through *grammars*. We hereby describe its extension to tree languages: *tree grammars*. Most of the definitions of this section can be found in *Tree Automata Techniques and Applications* [Tata].

Definition 2.3.1 (Tree Grammar)**.** *A* tree grammar *is a quadruple* $\langle \Sigma, N, S, \Delta \rangle$ *where* Σ *is a ranked alphabet of symbols called* terminals*,* N *a ranked alphabet of symbols called* non-terminals *(disjoint from* Σ*) with* $S \in N$ *a special starting non-terminal such that* $ar(S) = 0$*. Finally,* Δ *is a set of production rules, a term rewriting system defined over* $\Sigma \cup N$*.*

A term t is *produced* by a grammar $\mathcal{G}$ if there exists a rewriting path of the form $S \to_\Delta^* t$, from the starting non-terminal S to the term. The language produced by $\mathcal{G}$ is written $\mathcal{L}(\mathcal{G})$ and defined as $\{\ t \in \mathcal{T}(\Sigma) \mid S \to_\Delta^* t\ \}$. Tree grammars are basically term rewriting systems producing the terms of the described language from a single starting symbol. The family of languages that can be produced using a tree grammar is called "recursively enumerable" tree languages. Such tree language can be recognized using a Turing machine. Recursively enumerable languages are closed under union, intersection and complement.

Example 2.3.1. *The output of a functional program is always a recursively enumerable language (which also explains why a Turing machine is necessary to recognize such language). For instance consider again the TRS* $\mathcal{R}$ *defined in the previous Example 2.2.1 and extracted from the OCaml insert-sort program. We can describe the output of this program as a tree grammar* $\mathcal{G} = \langle \Sigma, N, S, \Delta \rangle$ *where* Δ *is the union of* $\mathcal{R}$ *with the following TRS:*

$$\begin{array}{ll} S \to sort(L) & \\ L \to nil & L \to cons(N, L) \\ N \to 0 & N \to s(N) \end{array}$$

The additional rules produce all the initial states of the program (in our case, it is any call to the sort function with any list). We can the show that $\mathcal{L}(\mathcal{G}) = \mathcal{R}^*(\{\ sort(L) \mid L\ a\ list\ \})$*. The outputs can be isolated by intersection with* $IRR(\mathcal{R})$*.*

Just as with string languages, we distinguish multiple families of tree languages by the complexity of the term rewriting system needed to define the grammar producing them.

2.3.1 Regular Tree Languages

The first and simplest family of tree languages are called "regular" tree languages. It is defined as the class of languages that can be produced by *regular* tree grammars.

Definition 2.3.2 (Regular Tree Grammar)**.** *A* regular *tree grammar $\mathcal{G}$ is a tree grammar $\langle \Sigma, N, S, \Delta \rangle$ where for all non-terminal $A \in N$, $ar(A) = 0$ and each rule of Δ is of the form $A \to \beta$ where A is a non-terminal and β a tree of $\mathcal{T}(\Sigma \cup N)$.*

Example 2.3.2. *Regular tree grammars are often found in modern functional programming languages as a mean to describe algebraic data types. For instance, consider the following OCaml type definition for lists of colors:*

```
type color = Red | Green | Blue

type list =
        | Nil
        | Cons of color * list
```

This in fact directly encodes the following tree grammar $\mathcal{G} = \langle \Sigma, list, \{color, list\}, \Delta \rangle$ where $\Sigma = \{\ Red : 0, Green : 0, Blue : 0, Nil : 0, Cons : 2\}$ and Δ:

$$\begin{array}{ll} color \to Red & list \to Nil \\ color \to Green & list \to Cons(color, list) \\ color \to Blue & \end{array}$$

For instance, the term $Cons(Red, Cons(Green, Cons(Blue, Nil)))$ is generated by $\mathcal{G}$ because of the following rewriting path in Δ:

$$\begin{aligned} & \underline{list} \\ \to\ & Cons(\underline{color}, list) \\ \to\ & Cons(Red, \underline{list}) \\ \to\ & Cons(Red, Cons(\underline{color}, list)) \\ \to\ & Cons(Red, Cons(Green, \underline{list})) \\ \to\ & Cons(Red, Cons(Green, Cons(\underline{color}, list))) \\ \to\ & Cons(Red, Cons(Green, Cons(Blue, \underline{list}))) \\ \to\ & Cons(Red, Cons(Green, Cons(Blue, Nil))) \end{aligned}$$

Note that contrarily to the OCaml definition, we generally prefer to use lowercase for symbols ($red, green, blue, nil, cons$) and capitals for non-terminals ($Color, List$).

Regular tree grammar are more closely related to context-free string grammars than regular string grammars. In particular, if we note $Yield(\mathcal{L})$ the string language defined as $\{\ Yield(t) \mid t \in \mathcal{L}\ \}$, then $Yield(\mathcal{L})$ can be generated using a context-free string grammar, but may not by a regular string grammar (it is a context-free string language). One can see regular tree languages as a sub-family of well-parenthesized context-free string languages. Regular languages are closed under union ($\mathcal{L}_1 \cup \mathcal{L}_2$), intersection ($\mathcal{L}_1 \cap \mathcal{L}_2$) and complement ($\overline{\mathcal{L}}$).

2.3.2 Bottom-Up Tree Automata

Bottom-up tree automata are another structure for representing regular tree languages. Just as we can make a direct correspondence between regular string grammar defining

regular string languages and nondeterministic finite string automata recognizing them, we can make a direct correspondence between regular tree grammars and nondeterministic finite *tree automata*. While tree grammars can define regular languages, tree automata are more suited to answer membership queries.

Definition 2.3.3 (Bottom-Up Tree Automaton)**.** *A (bottom-up) tree automaton $\mathcal{A}$ is a quadruplet $\langle \Sigma, Q, Q_f, \Delta \rangle$ where Σ is a ranked alphabet of symbols, Q a set of* states*, $Q_f \subseteq Q$ a set of* final states *and Δ a set of* transitions *(a rewriting system) of the form $p \to q$ where $q \in \mathcal{Q}$ is a state and p a pattern, called a* configuration*, of $\mathcal{T}(\Sigma, \mathcal{Q})$.*

A term t is *recognized* by a tree automaton $\mathcal{A}$ in state q if there exists a rewriting path of the form $t \to_\Delta^* q$. We write $\mathcal{L}(\mathcal{A}, q)$ the language recognized by $\mathcal{A}$ in q: $\{\ t \mid t \to_\Delta^* q\ \}$. We say that $\mathcal{A}$ is *reduced* if it has no state q such that $\mathcal{L}(\mathcal{A}, q) = \emptyset$. A term t is *recognized* by a tree automaton $\mathcal{A}$ if it is recognized by a final state. We write $\mathcal{L}(\mathcal{A})$ the language recognized by $\mathcal{A}$. It is a *regular tree language*. To simplify, we write $\to_\mathcal{A}$ instead of $\to_\Delta$ when the components of $\mathcal{A}$ are not explicitly defined.

A regular tree grammar can easily be translated into a tree automaton by using non-terminal as states, the starting non-terminal as final state and by inverting the direction of each production rule and use them as transitions.

Example 2.3.3. *The regular tree grammar $\mathcal{G}$ defined in the previous example can be translated into the following tree automaton $\mathcal{A} = \langle \Sigma, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$ where Δ is defined as*

$$\begin{aligned} red &\to q_{color} & \qquad nil &\to q_{list} \\ green &\to q_{color} & cons(q_{list}, q_{color}) &\to q_{list} \\ blue &\to q_{color} \end{aligned}$$

with q_{list} a final state in Q_f. We have $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{G})$. For instance, the term $cons(red, cons(green, cons(blue, nil)))$ is recognized *by $\mathcal{A}$ (into the final state q_{list}) because of the following rewriting path in Δ:*

$$\begin{aligned} cons(red, cons(green, cons(blue, \underline{nil}))) &\to \\ cons(red, cons(green, cons(\underline{blue}, q_{list}))) &\to \\ cons(red, cons(green, \underline{cons(q_{color}, q_{list})})) &\to \\ cons(red, cons(\underline{green}, q_{list})) &\to \\ cons(red, \underline{cons(q_{color}, q_{list})}) &\to \\ cons(\underline{red}, q_{list}) &\to \\ \underline{cons(q_{color}, q_{list})} &\to \\ q_{list} \end{aligned}$$

Using tree automata instead of tree grammars offers several technical advantages. For instance, a tree automaton can have a deterministic transition system while still representing an infinite language, which is not the case with regular tree grammars.

Definition 2.3.4 (Determinism)**.** *We say that a tree automaton $\mathcal{A} = \langle \Sigma, \mathcal{Q}, \mathcal{Q}_f, \Sigma \rangle$ is deterministic if its transition system Δ is deterministic: for all terms $t \in \mathcal{T}(\Sigma, \mathcal{Q})$, there is at most one state $q \in \mathcal{Q}$ such that $t \to_\Delta^* q$.*

Definition 2.3.5 (ϵ-Transitions)**.** *Let $\mathcal{A} = \langle \Sigma, Q, Q_f, \Delta \rangle$ a tree automaton. An ϵ-transition of $\mathcal{A}$ is a transition of the form $q \to q'$ in Δ where q and q' are two*

states of Q. We write $t \to_\Delta^{\epsilon,*} q$ when there is a Δ-rewriting path from t to q using only ϵ-transitions. Conversely, we write $t \to_\Delta^{\not\epsilon,*} q$ when there is a Δ-rewriting path from t to q using no ϵ-transitions.

We say that $\mathcal{A}$ is ϵ-free if it has no ϵ-transitions. We say that $\mathcal{A}$ is $\not\epsilon$-deterministic if for all terms $t \in \mathcal{T}(\Sigma, \mathcal{Q})$ there is at most one state $q \in \mathcal{Q}$ such that $t \to_\Delta^{\not\epsilon,*} q$. We use the abbreviation "REFD" for reduced, ϵ-free deterministic tree automata.

Definition 2.3.6 (Normalized Transitions)**.** *Let $\mathcal{A}$ be a tree automaton. A transition $f(p_1, \ldots, p_n) \to q$ of $\mathcal{A}$ is* normalized *if $p_1, \ldots, p_n$ are states. We say that $\mathcal{A}$ is* normalized *if all its transitions are normalized.*

It is always possible to normalize a tree automaton by adding new states recognizing sub-terms of non-normalized configuration. For this reason we often suppose that the automata considered in the proofs are normalized since it is easier to reason about normalized transitions.

2.3.3 Beyond Regularity

This thesis focus on the use regular languages to verify regular properties over higher-order functional programs. In order to understand the contour of regular problems (cf. Definition 2.4.6) this section discusses the limits of regular languages and what lies beyond them.

Non-Regular Languages

We have seen how regular languages are closely related to the algebraic data type definitions that can be found in modern functional programming languages. In addition, terms found in regular languages are sufficient to represent any state of a functional program execution. Combined with the apparent simplicity of the structure representing regular languages (grammars and automata), this motivates our interest in regular languages applied to program verification. However we can already discuss what regular languages are not good for: relations. Most infinite tree languages expressing relations between sub-terms of the same tree are not regular. This is particularly true for inductive relations. The simplest example of such inductive relation is the equality relation between trees.

Example 2.3.4. *Consider the following TRS $\mathcal{R}$ defining the equality operator eq over natural numbers:*

$$eq(0,0) \to true \qquad eq(0, s(\underline{y})) \to false$$
$$eq(s(\underline{x}), s(\underline{y})) \to eq(\underline{x}, \underline{y}) \qquad eq(s(\underline{x}), 0) \to false$$

The tree language recognizing all and only terms of the form $eq(s,t)$ rewriting to true is not regular. It is not possible to build a tree automaton recognizing such language. This is because rewriting steps at the core of tree automata and regular tree grammars are essentially local operation, whereas the equality decision is essentially a global operation. For the same reason, the comparison operators $\leq$ and $\geq$ cannot be captured by regular languages.

The direct consequence of this example is that any property whose proof relies on such relation is out of the scope of the techniques presented in Chapters 4 and 5. However an extension dealing with such relation will be discussed in Chapter 6. Note

that comparison predicates defined over a finite domain are not inductive relations and can be represented using a regular language.

Example 2.3.5. *Replacing natural numbers with a finite domain, such as* $\{ a, b \}$ *is sometimes sufficient to transform a non-regular problem into a regular one. With this domain, the equality predicate is no longer inductive and can be defined with a simple cases enumeration:*

$$\begin{aligned} eq(a,a) &\to true & eq(a,b) &\to false \\ eq(b,b) &\to true & eq(b,a) &\to false \end{aligned}$$

Also note that in many cases, even when such relations are involved, they are not required in the proof. A regular approximation of the relation may be sufficient to verify the property.

Example 2.3.6. *Consider the following TRS* $\mathcal{R}$*:*

$$\begin{aligned} f(\underline{x}) &\to not(or(eq(0,\underline{x}), eq(s(0),\underline{x}))) \\ or(false,false) &\to false & or(true,false) &\to true \\ or(false,true) &\to true & or(true,true) &\to true \\ not(true) &\to false & not(false) &\to true \end{aligned}$$

Assume we want to verify that for all $n > 1$*,* $f(n)$ *rewrites to true. This property involves eq, however it does not require to compute exactly the problematic irregular language* $\{ eq(s,t) \mid eq(s,t) \to^*_{\mathcal{R}} true \}$*. This is because eq is only used in two special cases where one of the compared value is known. Instead, it is sufficient to compute* $\{ eq(0,t) \mid eq(0,t) \to^*_{\mathcal{R}} true \}$ *and* $\{ eq(s(0),t) \mid eq(s(0),t) \to^*_{\mathcal{R}} true \}$ *that are both regular.*

Hierarchy of languages

In *Three models for the description of language* [Cho56], Chomsky define a hierarchy of string languages. It is possible to transpose this hierarchy to tree languages as pictured on Figure 2.1.

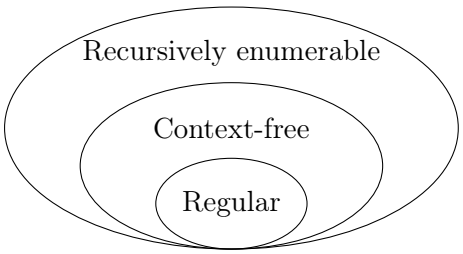


Figure 2.1: Hierarchy of tree language families

We have already seen two member of this hierarchy: Recursively enumerable tree languages can be produced by non constrained tree grammars and recognized by Turing machines, while regular tree languages can be produced by regular tree grammars and recognized by tree automata. Between them we can distinguish the family of context-free tree languages that are recognized by context-free tree grammars.

Definition 2.3.7 (Context-Free Tree Grammar)**.** *A context-free tree grammar $\mathcal{G}$ is a grammar $\langle \Sigma, N, S, \Delta \rangle$ where each production rule in Δ is of the form $A(x_1, \ldots, x_n) \to t$ where A is a non-terminal of arity n, $x_1, \ldots, x_n$ are variables of $\mathcal{X}$ and t a pattern of $\mathcal{T}(\Sigma \cup N, \mathcal{X})$.*

The difference with regular grammars lies in the possibility for non-terminals to be defined with variables. Context-free tree grammars are strictly more powerful than regular grammars and can recognize non regular languages. However it is not yet powerful enough for instance to recognize the irregular languages $\{ eq(s,t) \mid eq(s,t) \to^*_{\mathcal{R}} true \}$ seen in the previous examples.

2.4 Automated Verification and Abstraction

In order to prove or disprove a given property on a program, automated verification techniques generally rely on the automatic abstraction of a representation of the program execution. The representation in itself and the abstraction method depend on the considered technique. For higher-order functional program, we have seen that representing the program execution using term rewriting systems and tree languages offer several advantages, but many other representations have been successfully used over the years in various techniques. To offer a point of comparison, in this section we chose to present the common principles underlying automated verification techniques from the point of view of model checking, which is general enough to be easily related to all the verification techniques presented in Chapter 3.

2.4.1 Formalization as Model Checking

Model Checking is a high-level program verification framework introduced in 1982 by Clarke and Emerson [CE82a], where a program execution, modeled as a directed graph[1], is searched for elaborate patterns in order to check its compliance to a given specification. All the techniques presented in Chapter 3 can be seen as some sort of Model Checking. This point of view is useful to compare the expressive power of each technique, but first requires a proper introduction to the theoretical instruments used in Model Checking.

Model

Formally, if P is a set of *propositions*, a model M is defined as triple $\langle S, \to, V \rangle$ (called a *Kripke structure*) where S is a set of states, $\to$ the transition relation between the states (the program's logic) and V a function mapping each proposition to the set of states for which the proposition is true.

Example 2.4.1. *In our case we consider programs represented as term rewriting systems. If $\mathcal{R}$ is a TRS defined over a ranked alphabet Σ and $I \subseteq \mathcal{T}(\Sigma)$ a set of initial terms, this can be viewed as a model $\langle \mathcal{R}^*(I), \to_{\mathcal{R}}, V \rangle$ where an execution state is a reachable term (the expression currently evaluated), and the transition relation is the rewriting relation w.r.t. $\mathcal{R}$. For instance, consider the following TRS $\mathcal{R}$ representing the even and odd predicates:*

$$\begin{array}{cc} even\ 0 \to true & odd\ 0 \to false \\ even\ s(\underline{n}) \to odd\ \underline{n} & odd\ s(\underline{n}) \to even\ \underline{n} \end{array}$$

[1] We consider here a simplified version of Model Checking for non labeled transition systems.

Using the initial language $I = \{even\ n \mid n \in \mathbb{N}\}$ *it encodes the following (infinite) model* $\langle \mathcal{R}^*(I), \rightarrow_{\mathcal{R}}, V \rangle$*:*

$$\ldots \longrightarrow odd\ s(0) \longrightarrow even\ 0 \longrightarrow true$$

$$\ldots \longrightarrow even\ s(0) \longrightarrow odd\ 0 \longrightarrow false$$

In order to perform any kind of verification on a model, one must first express the property to be checked against the execution graph. In Model Checking, this is traditionally done using modal logics, such as CTL* [EH83] an extension of Computational Tree Logic [CE82b], or Modal μ-Calculus [SdB69].

Modal Logics

Modal logics extend the propositional logic that can only be used to produce assertions about a single execution state by introducing *modalities* that can be used to produce assertions about whole execution *paths*. As an example, we explore in more details here the Modal μ-Calculus [SdB69] (L_μ) that will be useful later in Section 3.2.

If P is the set of *propositions*, and $\mathcal{X}$ a set of variables, the set $\mathcal{F}$ of valid modal μ-calculus formula is given by the smallest set including

$$x \mid p \mid \neg\phi \mid \phi \wedge \psi \mid \phi \vee \psi \mid \Box\phi \mid \Diamond\phi \mid \nu x.\ \phi \mid \mu x.\ \phi$$

where x is a variable of $\mathcal{X}$, p a proposition of P and ϕ and ψ are formula. For a given model M and variable environment σ mapping a variable to a set of states, the semantics $[\![\phi]\!]_\sigma$ of a formula ϕ gives the set of states in which the formula is verified:

$$[\![x]\!]_\sigma = \sigma(x) \qquad [\![\phi \wedge \psi]\!]_\sigma = [\![\phi]\!]_\sigma \cap [\![\psi]\!]_\sigma \qquad [\![\neg\phi]\!]_\sigma = S / [\![\phi]\!]_\sigma$$
$$[\![p]\!]_\sigma = V(p) \qquad [\![\phi \vee \psi]\!]_\sigma = [\![\phi]\!]_\sigma \cup [\![\psi]\!]_\sigma$$

The formula $\Box\phi$ (and its dual $\Diamond\phi$) holds in any state such that ϕ holds in every (resp. some) of its successors:

$$[\![\Box\phi]\!] = \{\ q \in S \mid \forall q'.\ q \rightarrow q' \Rightarrow q' \in [\![\phi]\!]\ \}$$
$$[\![\Diamond\phi]\!] = \{\ q \in S \mid \exists q'.\ q \rightarrow q' \Rightarrow q' \in [\![\phi]\!]\ \}$$

The formula $\nu x.\ \phi$ (and its dual $\mu x.\ \phi$) gives the greatest (resp. least) fixed point of $[\![\phi]\!]_{\sigma[x \mapsto T]}$. It holds in any set of states T such that when x is bound to T, ϕ holds for T:

$$[\![\nu x.\ \phi]\!] = \bigcup \{\ T \subseteq S \mid T \subseteq [\![\phi]\!]_{\sigma[x \mapsto T]}\ \} \qquad \text{greatest fixed point}$$
$$[\![\mu x.\ \phi]\!] = \bigcap \{\ T \subseteq S \mid [\![\phi]\!]_{\sigma[x \mapsto T]} \subseteq T\ \} \qquad \text{least fixed point}$$

Example 2.4.2. *Consider the model* $\langle \mathcal{R}^*(I), \rightarrow_{\mathcal{R}}, V \rangle$ *given in the previous example above representing the execution of the even/odd program. We want to show that for every even number* $2k$*,* (even $2k$) *never returns* false*. Note that in the previous we haven't explicitly defined* V *along with the set of propositions* P*. To verify this property, we now consider* P *with the only proposition* $False$ *that only holds for the state* false*:* $V(False) = \{$ false $\}$*. Our property can now be verified by checking that*

every term of the form (even $2k$) *is* not *contained in the semantics of the following formula:*

$$\phi = \mu x.\ False \vee \Diamond x \qquad\qquad \text{“}false \text{ can be reached”}$$

Following the semantics of L_μ*, we are looking for the smallest set of states for which we have, for each state, either* $False$ *holds, or* ϕ *holds in some of its successors. First, we note that* $[\![False]\!] = V(False) = \{$ false $\}$ *(by definition of* V*). Hence we know that* false $\in [\![\phi]\!]$*. Since* false *is a successor of* (odd 0) *we have* (odd 0) $\in [\![\Diamond x]\!]$*, hence* (odd 0) $\in [\![\phi]\!]$*. Similarly, we have that* (even $s(0)$) $\in [\![\phi]\!]$ *and by induction for all* k*,* (even $2k$+1) $\in [\![\phi]\!]$ *and* (odd $2k$) $\in [\![\phi]\!]$*. Because we used the least fixed point operator* [2]*, no other state is included from which we can conclude that this model verifies our property:* (even $2k$) *never returns* false*.*

Verification Problems

Using the concepts defined above, we can finally formally define verification problems.

Definition 2.4.1 (Verification Problem)**.** *A verification problem* P *is a triple* $\langle M, \phi, I\rangle$ *where* $M = \langle S, \to, V\rangle$ *is a model,* ϕ *a formula and* $I \subseteq S$ *an set of initial states. The conjunction of* ϕ *and* I *describes the* property *to verify on* M*. Solving* P *consists in deciding if* $I \subseteq [\![\phi]\!]_\emptyset$*. If it is, the property is* verified*.*

From the different existing families of problems, the verification techniques developed in this thesis all focus on the family of *safety problems.*

Definition 2.4.2 (Safety Property and Problem)**.** *A formula* ϕ *describe a* safety property *if it can be stated as*

$$\neg(\mu x.\ \psi \vee \Diamond x)$$

A problem $P = \langle M, \phi, I\rangle$ *is a safety problem if* ϕ *describes a safety property. One can see* ψ *as a formula capturing “bad” states. A safety problem then consists in verifying that a bad state can never be reached from an initial state.*

In our settings it is useful to restate this definition using term rewriting systems and tree languages.

Definition 2.4.3 (TRS Safety Problem)**.** *Let* $\mathcal{R}$ *be a term rewriting system and* I*,* O *two tree languages. We note* $\langle \mathcal{R}, I, O\rangle$ *the safety problem over* $\mathcal{R}$ *that consists in verifying the following property:* $\mathcal{R}^*(I) \subseteq O$ *(every reachable term is safely contained in* O*).*

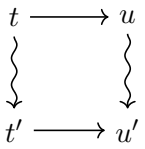
A TRS safety problem $\langle \mathcal{R}, I, O\rangle$ can be expressed as a standard safety problem $\langle M, \phi, I\rangle$ where $M = \langle \mathcal{R}^*(I), \to_{\mathcal{R}}, V\rangle$ and $\phi = \neg(\mu x.\ unsafe \vee \Diamond x)$ with *unsafe* the proposition such that $V(unsafe) = \overline{O}$.

2.4.2 Verification via Abstraction

In general, for a given model $\langle S, \to, V\rangle$ the set of states S is infinite which makes the program's model checking difficult. This is the case in the previous example. Instead the common approach is to first *abstract* the model into a simplified model $\langle S', \to', V'\rangle$ depending on the property of interest that is easier to analyze.

[2] Using the greatest fixed point operator ν, every term of $\mathcal{R}^*(I)$ would have been included: if x is replaced by $\mathcal{T}(\Sigma)$ in $(False \vee \Diamond x)$ then it is verified by every state/term.

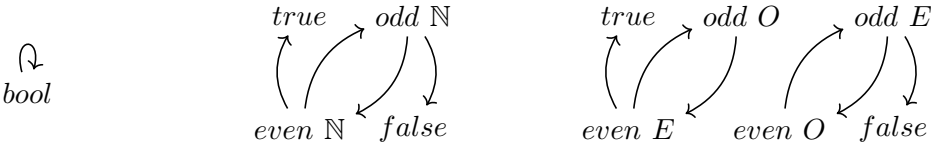
Definition 2.4.4 (Program Abstraction)**.** *Let P a program modeled by $\langle S, \to, V\rangle$. A program abstraction is defined by an abstraction relation $\rightsquigarrow \subseteq S \times S'$ and a (abstract) model $\langle S', \to', V'\rangle$ such that for all $t \in S$, $t' \in S'$ such that $t \rightsquigarrow t'$ then $V(t) \subseteq V'(t')$ and if there exists $u \in S$ such that $t \to u$ then there exists $u' \in S'$ such that $t' \to' u'$ with $u \rightsquigarrow u'$:*



We also say that $\rightsquigarrow$ defines a simulation *of $\langle S, \to, V\rangle$ in $\langle S', \to', V'\rangle$.*

This new model holds an abstraction of the program execution, where each node of the graph represent one or more possible execution state, and each edge a possible transition between those abstracted states.

Example 2.4.3. *The execution of the previous* even/odd *can be finitely abstracted with, for instance, one of the following three models each exposing different properties of the program (the abstraction relation yielding each abstract model is given below):*



$even\ n \rightsquigarrow bool$	$even\ n \rightsquigarrow even\ \mathbb{N}$	$even\ 2k \rightsquigarrow even\ E$
$odd\ n \rightsquigarrow bool$	$odd\ n \rightsquigarrow odd\ \mathbb{N}$	$even\ 2k{+}1 \rightsquigarrow even\ O$
	$true \rightsquigarrow true$	$odd\ 2k \rightsquigarrow odd\ E$
	$false \rightsquigarrow false$	$odd\ 2k{+}1 \rightsquigarrow odd\ O$
		$true \rightsquigarrow true$
		$false \rightsquigarrow false$

The first abstraction essentially abstracts every reachable state into its type, bool. Since the type of a term is in principle preserved by execution, this effectively collapses execution sequences into one single state. Even if the control flow is lost during the transformation, we will see later that the result can still be used to verify safety properties. On the other side of the spectrum, the third abstraction makes the distinction between even and odd numbers. It can be used to verify the same property as in Example 2.4.2 using the same modal μ-calculus formula but on a finite model instead.

This example shows how program abstraction is a critical step in program verification. The range of properties verifiable by the technique depends on which ways a given verification technique can abstract a program.

State Abstraction: Regular and Relational Properties

As we have seen in Example 2.4.1, when a higher-order functional program is represented as term rewriting system, a state of the execution is a term representing the currently evaluated expression. This means that for a given model $\langle S, \to, V\rangle$ abstracted into $\langle S', \to', V'\rangle$, each abstract state $q \in S'$ in the abstraction defines a tree language $\mathcal{L}(q) = \{t \mid t \rightsquigarrow q\}$ that can be represented by a tree grammar. Hence,

verification techniques can be compared against the family of grammars the states can be abstracted into. This can be done using the Chomsky Hierarchy [Cho56] comparing the expressiveness of different classes of grammars. Can the verification technique generate regular grammar abstractions? Context free grammars? etc. In this document we mainly consider two classes of grammars determining the class of properties verifiable by a verification technique.

Regular Problems

Definition 2.4.5 (Regular Abstraction)**.** *An abstraction $\rightsquigarrow$ into $\langle S', \rightarrow', V' \rangle$ is regular when for every abstract state $q \in S'$, $\mathcal{L}(q)$ is regular.*

In a regular abstraction, each state can be stored as a regular tree grammar, or tree automaton. This allows the verification of a specific family of problems: regular problems.

Definition 2.4.6 (Regular Problem)**.** *A regular problem is a problem that can be solved using a regular abstraction of the program. Conversely, non-regular problems cannot be solved without the need of at least context-free grammars abstractions.*

In this thesis we are particularly interested in the verification of the combination of both safety and regular problems using term rewriting systems.

Definition 2.4.7 (TRS Regular Safety Problem)**.** *Following Definitions 2.4.3 and 2.4.6, a TRS regular safety problem $\langle \mathcal{R}, I, O \rangle$ (or just* regular safety problem *for short) is a safety problem that can be solved using a regular abstraction of the program. In our setting this means that I, O are regular languages such that $\mathcal{R}^*(I) \subseteq O$. In other words, a regular safety problem can be solved by either finding a regular over-approximation $\mathcal{L}$ of $\mathcal{R}^*(I)$ such that*

$$\mathcal{R}^*(I) \subseteq \mathcal{L} \subseteq O$$

or by finding a counter-example term of $\mathcal{R}^*(I)$ outside of O. Since I and O are regular languages, we sometimes replace them by tree automata.

Relational Properties Incidentally, we will see in the next chapter that all the automatic verification techniques that can solve non-regular problems are generally not good at verifying regular problems (they are unable to generate arbitrary regular abstractions). These techniques can capture relations between the values, but not the regular structure of those values. In this case, we use the term "*relational properties*" to qualify the range of properties verifiable by these techniques.

Another useful abstraction family to consider is *functional abstractions.*

Definition 2.4.8 (Functional Abstractions)**.** *An abstraction $\rightsquigarrow$ into $\langle S', \rightarrow', V' \rangle$ is functional when $\rightsquigarrow$ is a function. This means that each state is abstracted into (at most) a unique abstract state. All states of S' are disjoint.*

Control Flow Abstraction

The way the program is abstracted can also affect what kind of logic can be *discerned* by the verification technique. For instance we have seen that when states are abstracted by their types, execution paths collapse and the control flow information disappears. Only safety properties can be verified against the resulting model. From

a modal μ-calculus point of view, it means that every formula ϕ is equivalent to a simpler formula ϕ' expressed in propositional logic (without $\Box$ nor $\Diamond$): $[\![\phi]\!]_\sigma = [\![\phi']\!]_\sigma$. Hence, the modal logic cannot be discerned from the non-modal propositional logic, which narrows the range of problems the technique can solve. We name this family of abstractions "collapsing abstractions".

Definition 2.4.9 (Collapsing Abstraction)**.** *Let $\langle S, \to, V\rangle$ be an execution model. An abstraction $\rightsquigarrow$ of $\langle S, \to, V\rangle$ into $\langle S', \to', V'\rangle$ is said to be collapsing when $\to'$ is empty. This implies that for all $t, u \in S$ and t', if $t \rightsquigarrow t'$ and $t \to u$ then $u \rightsquigarrow t'$. The language $\mathcal{L}(t')$ is closed w.r.t. $\to$. As a consequence, the resulting abstraction can only be used to verify safety properties.*

When $\rightsquigarrow$ is a collapsing abstraction into $\langle S', \to', V'\rangle$ with P (the set of propositions) empty, we often write that $\rightsquigarrow$ is an abstraction into S' as both $\to'$ and V' are known to be empty.

Completeness of Verification Techniques

A verification technique is complete w.r.t. to a family of abstractions if it can automatically build any abstraction of this family. For instance, a "regularly complete" verification technique can automatically solve any regular verification problem by generating any necessary regular abstraction of the program.

In the next chapter we explore different automated verification techniques that can be compared against the family of abstractions they can build (regular, relational, etc.) and against the family of logic they can discern (propositional, modal, etc.).

Chapter 3

State of the Art

This chapter summarizes the state of the art of the automatic verification of higher-order functional programs. The techniques presented in this chapter sacrifice expressiveness in favor of automation: the range of verifiable properties is restricted, but the user only needs to state the desired property and is not required to annotate the program or to participate in the proof in any ways. The common approach between these technique consists in finding a finite abstraction (or at least an abstraction with a finite representation) that over-approximates the execution of the program, and then solve the verification problem on the resulting finite abstract model. Techniques are then distinguished by the family of abstractions they are able to automatically generate. One is generally exclusively well suited to solve either relational problems on numerical values using non-regular abstractions, or regular problems on algebraic data types using regular abstractions. At the same time one may allow the user to verify temporal properties over the program, while others are restricted to safety properties.

3.1 Static Type Systems

The use of type annotations statically checked by a type checker can be seen as the most basic kind and most widely used program verification method that does not require program execution. The initial motivation behind the introduction of type systems and type annotations is to ensure the "well-behaveness" of the program execution, i.e. ensure that no execution path leads to a state not covered by the program's semantics: "well-typed programs cannot go wrong" [Mil78]. In practice, the goal is to ensure that at each execution step, the current state of the memory matches the expected memory layout. In this context, a type represents the memory layout of a data value, shared by all the values of the same type. Rapidly the interest grew over the possibility to express more and more complex types describing more than just a memory layout in order to verify more and more complex properties, not only to ensure well-behaveness but also correctness.

3.1.1 Intersection and Set Theoretic Types

One way to extend simple type systems is to allow one term to have multiple types at once. This idea was first proposed by Coppo and Dezani-Ciancaglini [CDC78,CDCS79] in 1978 as a way to assign a type to λ-terms that would normally be impossible to type in simply typed λ-calculus (for instance, $W_* = \lambda x.\ (x\ x)$). The result of this study is a static type system in which terms such that W_* are not assigned a single

type, but are necessarily given multiple types (potentially infinitely many). The idea was then carried on [Rey91, CD80, CDV81, Hin82], giving rise to Intersection Types [BCD83, DCM84, RV84, Roc88], where the type of a term is given by the conjunction (denoted with $\cap$ or $\wedge$) of multiple other types. Intersection Types have been found useful not only with λ-calculi to capture the terminating properties of terms, but also as an interesting programming language feature [Rey96] introducing ad-hoc polymorphism. For instance, where classical ML type systems require two different addition operators for integers and floats (`+` of signature `int -> int -> int` and `+.` of signature `float -> float -> float` in OCaml), Intersection Types can be used to define a single `+` operator whose signature would be

```
(int -> int -> int) ∩ (float -> float -> float)
```

From there, intersection types have been extended with all the usual set operations (union, complement, difference, etc.) giving rise to Set Theoretic Types [HVP00, HP01, HP03, BCF03, FCB02, CPN16] with their own inference algorithm [CNX+14]. However, even if they can be used to describe more complex types, Set Theoretic Types cannot be used to express any regular language or to express relations between the values.

3.1.2 Dependent and Refinement Types

Focusing on theses issues, Refinement Types have been developed on top of the extensively studied foundations of Dependent Types [Nec97, ADLO10, FP91, McK06, OTMW04] to express relational and structural properties over a program. They have already been implemented in a number of programming languages such as Agda [Nor09], Cayenne [Aug98], Coq [Inr16], F* [MI13], Idris [Bra13], etc. In their original form, Refinement Types allow the introduction of predicates in the types, and dependencies between the variables. It has been primarily used to remove runtime array bound checking by encoding the range of integer variables in their type [XP98]. For instance, if we consider the following `nth` function that returns the n^{th} element of a given list. It is possible to statically enforce that n is in bounds by specifying the following signature for `nth`:

```
a:int list -> { n: n >= 0 ∧ n < length a }:int -> int
```

Refinements types are specified as the conjunction of *qualifiers* applied to the subject variable. In the previous example, the two qualifiers `n >= 0` and `n < length` $\star$ are used (where $\star$ can be replaced by any variable). Note that the occurrence of `>=` and `length` in the qualifier does not refer to the actual `>=` and `length` functions, but have their own semantics defined by the refinement type checker: the language of type qualifier is disjoint form the programing language. In 2008, the development of Liquid Types [RKJ08, VSJ14] allow for the relatively automatic inference of refinement types. The inference procedure is able to compose refinement types using a combination of qualifiers picked from a set of predefined qualifiers given by the user, or scrapped from (selected by looking at) the input program. It is further extended in [VRJ13, VBJ15] to allow higher-order refinements, with the presence of predicate variables. In our previous `nth` example, it can be used to transfer a property verified by every element of the list to the returned element:

```
∀(p : int -> bool). ({ e: p(e) } int) list -> int -> { e: p(e) } int
```

By internally manipulating predicate variables as uninterpreted function symbols, this

extension does not require the use of higher-order logics hence preserving decidability of the checking procedure. In parallel, efforts are made to capture structural properties over algebraic data types using Recursive Refinements [KRJ09] which allows the application of qualifiers on the subterms of an inductive type. In addition to the usual notation `{ x: p(x) }` used to refine a type τ, authors introduce a new notation[1] `{ x1, ..., xn: p1(x1), ..., pn(xn) }` to define a *products refinement* on the value `x1, ..., xn` whose type is the product $\tau_1 \times \cdots \times \tau_n$. This is further extended to encompass sum types with the notation `{ x1: p1(x1) | ... | xn: pn(xn) }` which refines the sum type $\tau_1 \vee \cdots \vee \tau_n$. For instance let's consider the following type definition of lists of integers:

```
type ilist = nil | cons of int * ilist
```

Using this pseudo-Caml syntax, it becomes clear that this type is defined as a sum (written `|`) of products (written `*`) between **`int`** and (recursively) **`ilist`**. Then the following *recursive refinement* can be used to denote a list whose *first element* is positive: `{ _ | v, _: v >= 0 } int list`. The first _ is the product refinement for the `nil` constructor, the first term of the sum (product refinements of each term are given in the order of the type definition). It is an empty product so we write _ to ignore it. the variable `v` refers to the first component of the `cons` constructor. It must be positive. Finally the last _ refers to the tail of the list which we ignore. This principle can be used to design more complex recursive types, for instance the type of sorted integer lists:

```
type sorted_ilist =
  nil
| cons of v:int * { _ | w, _: w >= v } sorted_ilist)
```

Note however that Recursive Refinements are only able to capture "local" recursive relations. They are not capable to capture the global shape of a value, that is to describe any regular language. For instance, it is impossible to describe a list whose first part is exclusively composed of even numbers, and second part of odd numbers.

To summarize, we have seen that the different flavors of refinement types allow the expression of various properties on program by encoding them in the type signatures of the functions. However refinement types suffers from several limitations. First, some properties that cannot be expressed with dependent types are out of the scope of the previously detailed Refinement Types techniques. For instance let's consider the following `find` function taking a predicate p and integer i as parameters, and returning the smallest integer after i satisfying p. We are not interested in p, but want to prove that the returned integer is necessarily greater or equal to i. This can be expressed with the following signature:

```
let find :: p:( int -> bool ) -> i:int -> { r: r >= i }:int =
        if p i then i else find p (i + 1)
```

Now instead of returning the value directly, we reformulate `find` so as to take a continuation k as a parameter, and call the continuation when the value is found:

[1] This is not the actual notation used by the authors [KRJ09], but one may find it easier to process than the one used in the original paper.

```
let find :: p:( int -> bool ) -> k:(int -> 'a) -> i:int -> 'a =
        if p i then k i else find p (i + 1)
```

This time however, because k occurs before i in the signature, there is no way to specify that the integer passed to the continuation is greater that i. This problem is tackled in [VBJ15] by introducing Bounded Refinement Types, adding a new layer of complexity to the type system while not covering every cases. Moreover, while well-suited to the verification of relational properties over integers, refinement types can hardly be used to verify structural properties over algebraic data types which compose the most part of regular problems. Finally, contrarily to simple type systems and even set theoretic types for which there exists well known inference algorithms [Hin69, Mil78, CNX+14], the inference of refinement types is undecidable in general. This requires the user to annotate the program to help the type checker, which may jeopardize the readability of the source code and require expertize. The type inference mechanism introduced by Liquid Types [RKJ08] alleviating the number of needed manual annotations is only able to compose refinement types using a combination qualifiers that must be given by the user, or scrapped from the program. Knowing what qualifiers are necessary to infer the refinement type requires expertize while not completely removing the need for annotations.

3.1.3 Deep Specification

The lack of expressiveness of Refinement Types comes from the constrained shape of the predicates (qualifiers) allowed in the refinements. In particular, the language of qualifiers is disjoint from the programming language. This ensure the decidability of the checking procedure at the cost of reducing the range of verifiable specifications. Some language implementation using dependent types such as Dafny [Lei10], F* [MI13] or Halo [VJCR13] have been described as "deep" specification and verification languages in [Vaz16], as opposed to the "shallow" specification of Refinement Types. In these languages, user defined functions can be embedded in the types, extending the range of specification. For instance, the "sortedness" property of a sorting function can be specified by first defining a `sorted` predicate in the language itself, and using it in the signature of the sorting function:

```
let sorted = function
        | nil -> true
        | x::nil -> true
        | x::y::l -> x <= y && sorted y::l

let sort :: l:int list -> { r: sorted(r) }:int list = ...
```

However powerful, this comes at the cost of losing most of the type inference capabilities, and in particular the inference of auxiliary function invariants. For instance in F*, the verification of the insertion sort algorithm is not possible using only the preceding specification. For the type checker to successfully verify the given signature of `sort`, one must also give the signature of the intermediate function `insert`, exposing the necessary invariant (insertion preserves sorting):

```
let insert :: e:int -> { l: sorted(l) }:int list ->
              { r: sorted(r) }:int list = ...

(* type checking this will fail if not given the signature of insert *)
let sort :: l:int list -> { r: sorted(r) }:int list = ...
```

Moreover in addition to losing automation, the type checking procedure becomes undecidable in general. The later limitation is solved by Vazou et al. [Vaz16] using Refinement Reflection [VJ16a, VJ16b] to import deep specifications into Haskell. The idea is to use uninterpreted function symbols to encode user defined functions, preserving a decidable verification procedure. In addition, the authors provide a set of function combinators that can be used to compose proof in the programing language itself, effectively turning Haskell into a theorem prover. The drawback is the lost of automation compared to Liquid Haskell, and a more convoluted proof development mechanism compared to domain specific languages such as F* or Coq.

3.2 Higher-Order Trees Model Checking

Higher-Order Tree Model Checking is a family of automatic verification techniques concerned with the Model Checking of (possibly infinite) trees representing a program execution. These trees are generated by Higher-Order Recursion Schemes (HORS), a kind of higher-order typed tree generating grammar [Ong06]. Contrarily to static type systems seen in the previous section, this technique can preserve the control flow of the program and discern modal logics.

3.2.1 Higher Order Recursion Schemes

According to Ong06, a (deterministic) HORS $\mathcal{G}$ is a tuple $\langle \Sigma, \mathcal{N}, S, \mathcal{R} \rangle$ where Σ is a ranked alphabet of *terminal* symbols, $\mathcal{N}$ is a finite set of typed *non-terminals*, $S \in \mathcal{N}$ is a distinguished start symbol of type o and $\mathcal{R}$ is a finite set of rewriting rules for each non-terminal of the form

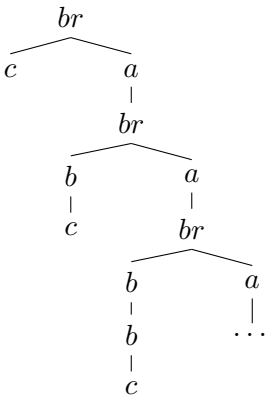
$$F\ \underline{x_1}\ \ldots\ \underline{x_n} \to e$$

where F is a non-terminal of type $\tau_1 \to \ldots \to \tau_n \to o$, each $\underline{x_i}$ is a variable of type τ_i, and t is a *pattern* built from terminals, non-terminal and variables. The type system distinguish trees (of type o) from functions (whose types include $\to$). The order of a HORS is given by the highest order of its non-terminal, where the order of a non-terminal symbol is given by the order of its type:

$$\begin{aligned} order(o) &= 0 \\ order(\tau_1 \to \tau_2) &= max(order(\tau_1) + 1, order(\tau_2)) \end{aligned}$$

Trees are generated by successively rewriting non-terminal symbols following the rules of $\mathcal{R}$, starting with S. For instance, let's consider the following order-1 HORS taken from [Kob09b]

$$\begin{aligned} S &\to F\ c \\ F\ \underline{x} &\to br\ \underline{x}\ (a\ (F\ (b\ \underline{x}))) \end{aligned}$$



Here F is a non-terminal of type $o \to o$. The start symbol S is rewritten as follows: $\underline{S} \to \underline{F\ c} \to br\ c\ (a\ (\underline{F\ (b\ c)})) \to br\ c\ (a\ (br\ (b\ c)\ (a\ (\underline{F\ (b\ (b\ c))}))))) \to \ldots$. If we ignore the br terminal symbol, the generated tree pictured on the side includes all the paths of the form $a^n b^n c$. Overall, HORSs can be viewed as a kind of simply typed lambda calculus with recursion and tree constructors (but no destructor), where any lambda term $(\lambda x.\ t)$ can be modeled by adding the (lambda-lifted) rule $F\ x \to t$ to the HORS. They are hence well suited to model higher-order programs. Note however that, contrarily to term rewriting systems, the lack of tree destructor means that HORSs do not provide any pattern matching mechanism as in most modern higher-order functional programing language, complicating the translation to HORSs. For instance, this implies that the simplest data values such as booleans must be encoded using functions, following the Church encoding.The HORS encoding the ***if-then-else*** control structure becomes:

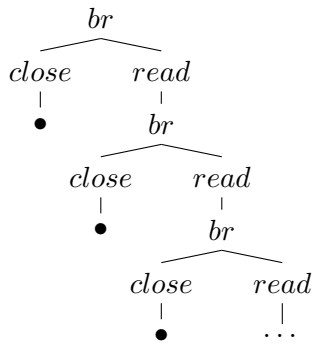
$$IF\ c\ x\ y \to c\ x\ y$$
$$T\ x\ y \to x$$
$$F\ x\ y \to y$$

In his paper [Ong06], Ong proves that, considering finite base types, the modal μ-calculus model-checking of trees generated by HORSs is decidable, turning them into an appealing candidate for program verification. Despite the k-EXPTIME theoretical complexity for order-k HORSs, multiple practical HORS model checker implementations [BK13, Kob09a, RNO14, TK14] have been developed that do not always suffer from the k-EXPTIME bottleneck.

The use of HORS applied to higher-order functional program verification started in 2009, when Kobayashi [Kob09b] introduced a novel method for verifying resource usage problems in higher-order functional programs. His technique consists in a program transformation to HORS that preserves relevant control flow features relative to the resource usage verification. For instance, let's consider the following program taken from the original paper [Kob09b]:

```
let rec g x = if _ then close(x) else (read(x); g(x)) in
        let d = open_in "foo" in g(d)
```

This program opens a file "foo" and manipulates it in the function g, either by reading it recursively, or closing it. We want to verify that the file is always closed, and that no further reading occurs after closure.



To do that, Kobayashi's technique transforms the program into the following HORS:

$$G\ \underline{x}\ \underline{k} \to br\ (close\ \underline{k})\ (read\ (G\ \underline{x}\ \underline{k}))$$
$$S \to G\ d\ \bullet$$

This HORS is very similar to the input programs where the rule g is represented by the non-terminal G, and br a non-deterministic branching (the exact branching condition is lost during the translation). The program is now in Continuation-Passing Style (CPS): the non-terminal G takes an extra k parameter, a continuation, necessary to encode sequences

of instructions in a purely functional manner. The • terminal is the final continuation, denoting the end of the program. The tree generated by this HORS, visible on the side, summarizes all the possible sequences of *read* and *close* that can happen during the program execution. Using existing HORS model checkers, the output tree can be automatically checked against the following two modal μ-calculus formula expressing our verification problem:

$$\mu x.\ close \vee \Box x \qquad \text{the file is eventually closed}$$
$$\neg(close \wedge \Diamond(\mu x.\ read \vee \Diamond x)) \qquad \text{no read occurs after a close}$$

Although useful to verify such resource usage problems, this first endeavor into HORS-based program verification shows the limitation of directly translating programs into HORSs. Since the model-checking of HORS-generated trees is only decidable with finite base types, this technique alone is not capable of dealing with infinite data types such as integers, lists or trees. As always, the program must first be abstracted into a finite base types program before being verified.

3.2.2 Predicate Abstraction

Building on his work on HORSs [Kob09b, Kob09a, KTU10], Kobayashi [KSU11b] and his coauthors define in 2011 an effective way of abstracting functional programs into finite base types HORSs, using *Predicate Abstraction*. In this setting, the input program is first abstracted into a higher-order boolean program where each (numerical) value v is abstracted into its predicates satisfaction. For a set $\langle P_1, \ldots, P_n \rangle$ of predicates, v is replaced by $\langle b_1, \ldots, b_n \rangle$ (where b_i is the result of $P_i(v)$). We directly note b instead of $\langle b_1 \rangle$ when considering a single predicate. For instance, let's consider the following program taken from [KSU11b]:

```
let f x g = g (x+1) in
        let h y = assert (y > 0) in
                let k n = if n > 0 then f n h else () in k(randi())
```

The assert instruction fails if its parameter is $false$ and the function *randi* returns a pseudo-random integer value. We want to verify that the program never fails (*assert false* is never evaluated). Using the predicate $P(x) = x > 0$, the following boolean program is generated:

```
let f x g = g true in
        let h y = assert y in
                let k y = if y then f true h else () in k(randb())
```

All the integer variables have been replaced by their boolean satisfaction of P. Note how *randi* has been replaced by *randb*, generating pseudo-random booleans values. As with the technique presented in the previous section 3.2.1, the obtained boolean program is then lambda-lifted and put in CPS form to translate it into an HORS. The later is finally model checked (using the TRecS [Kob13] model checker in the original paper) to verify that *assert* never fails. Here we have succeeded with $P(x) = x > 0$, however in general it is not obvious what predicates should be used to verify a given property. Hence the main difficulty here is to find the correct abstracting predicates. To do that, Kobayashi et al. defines a Counter-Example Guided Abstraction Refinement (CEGAR) procedure. The idea is as follows: starting with no predicates, at each iteration of the procedure the program is abstracted and verified. If a spurious counter example is found, a new predicate is added to the

abstraction extracted from the spurious rewriting path. The procedure continues adding new predicates until a real counter example is found, or the property is verified. The entire procedure, later improved for scalability [SK17], has been implemented in MoCHi [KSU11a], and been used to successfully and automatically verify a number of relational properties over higher-order functional program manipulating numerical values. However it is not well suited to solve regular verification problems over tree processing programs. This is the purpose of the techniques presented in the next section.

3.3 Regular Verification

As we have seen using our model-checking reference point in Section 2.4, the range of properties verifiable by a technique depends on the range of abstraction it can generate. The techniques presented in the previous sections all focus on the automatic verification of relational properties. This means they can generate various abstractions of the program states, even though it is no clear what exact family of abstraction it can generate. The regular abstraction family on the other hand is holding much attention when it comes to modern higher-order functional programs. Because regular tree languages can be easily represented using grammars or automata there is hope to find an efficient procedure that can generate arbitrary regular abstractions automatically. And since algebraic data types define regular tree languages of values, regular abstractions are particularly useful to verify tree processing programs. In this section we explore techniques focusing on the generation of arbitrary *regular* abstractions of a program, aiming at providing an automatic and complete verification of regular problems.

3.3.1 HORS Extensions

Since the beginning of HORS-based program verification, HORS have been criticized [OR11, KTU10] for their intrinsic inability to represent tree processing program, blaming the lack of tree destructor in this formalism. Algebraic data types can be constructed but never destructed through mechanisms similar to pattern matching, present in most modern functional programing languages. Early after Ong's work on HORSs [Ong06], extensions are proposed to introduce tree destruction mechanisms. On one side, Kobayashi introduces Higher-Order Multi-Parameter Tree Transducers [KTU10], an extension of HORS where the right hand side of each rule can contain a pattern matching term of the form:

$$match\ \underline{x}\ \begin{cases} case_1 & \Rightarrow t_1 \\ & \dots \\ case_n & \Rightarrow t_n \end{cases}$$

where each $case_i$ is a pattern of the form $a\ \underline{y_1} \dots \underline{y_m}$ where a is a terminal symbol of arity m and each $\underline{y_i}$ a variable. This term reduces to one of t_i when $\underline{x}$ matches the pattern $case_i$. The main limitation of HMTTs is the distinction made between *input trees* and *output trees*: Where HORSs define only one base type o representing trees (composable with $\rightarrow$ to denote functions), HMTTs define two base types i for input trees and o for output trees. In this definition, each rule of a HMTT takes input trees, and reduces to an output tree. Only input tree may be destructed with a *match*, and only output trees may be constructed. For instance, let's consider

the following HMTT composed of two rules concatenating two lists of as and bs together (to simplify we allow the use of terminals instead of variables in the matching patterns):

$$App\ \underline{x}\ \underline{y} \to match\ \underline{x} \begin{cases} nil & \Rightarrow Copy\ \underline{y} \\ cons\ a\ \underline{x'} & \Rightarrow cons\ a\ (App\ \underline{x}\ \underline{y'}) \\ cons\ b\ \underline{x'} & \Rightarrow cons\ b\ (App\ \underline{x}\ \underline{y'}) \end{cases}$$

$$Copy\ \underline{x} \to match\ \underline{x} \begin{cases} nil & \Rightarrow nil \\ cons\ a\ \underline{x'} & \Rightarrow cons\ a\ (Copy\ \underline{x'}) \\ cons\ b\ \underline{x'} & \Rightarrow cons\ b\ (Copy\ \underline{x'}) \end{cases}$$

The non-terminal symbol *App* defines the *append* function. The non-terminal *Copy* performs the copy of an input tree (of type i) to an output tree (of type o). This function is necessary because inputs and outputs cannot be mixed up in HMTTs. This also explains why we need to explicitly distinguish the two cases for a and b in *App* instead of just having the case $cons\ \underline{v}\ \underline{x'}$. The (almost) automatic translation from functional programs to HMTT is explored in [UTK10]. Under some linearity conditions, and given correct abstractions for the input trees $\underline{x}$ and $\underline{y}$ in the non-terminal *App* it is then possible to reduce the problem to the verification of a finite base types HORS. Here the abstractions of input variables are defined as tree automata recognizing all the possible input terms. However no procedure is given to automatically find the correct abstractions needed to verify a given property on the program.

On the other side, Ong and Ramsay introduce Pattern-Matching Recursion Schemes (PMRS) [OR11] another extension of HORSs providing a pattern matching mechanism. This time, patterns are allowed to appear on the left-hand side of a rewriting rule in place of the last rule variable. A PMRS rule is of the form:

$$F\ \underline{x_1}\ \ldots\ \underline{x_{n-1}}\ p \to e$$

where p is a pattern again of the form $a\ \underline{y_1}\ \ldots\ \underline{y_m}$ where a is a terminal symbol of arity m. For a set of rules of the form $F\ \underline{x_1}\ \ldots\ \underline{x_{n-1}}\ p_k \to e$, the term $F\ t_1\ \ldots\ t_n$ reduces into e when t_n matches p_k. For instance, let's define again the *append* function, this time using a PMRS:

$$\begin{aligned} App\ \underline{x}\ \underline{y} &\to Pre\ \underline{y}\ \underline{x} \\ Pre\ \underline{x}\ nil &\to \underline{x} \\ Pre\ \underline{x}\ (cons\ \underline{y}\ l) &\to cons\ \underline{y}\ (Pre\ \underline{x}\ l) \end{aligned}$$

Here we have two non-terminal symbols. *App* encodes the interface to the *append* function. However since a pattern can only occur at the last position of a left-hand side, *append* cannot be represented directly. Instead, we use the non-terminal Pre to encode the *prepend* function. *App* defines the *append* function using Pre by swapping $\underline{x}$ and $\underline{y}$. This is necessary because of the restrictions imposed by this formalism. As such, the translation to PMRS is not as direct as the translation to HMTTs. However there is no need for a copy function here, as there is no distinction between inputs and outputs, and the overall size of the model is reduced. In addition, contrarily to HMTTs, in their paper [OR11] Ong and Ramsay propose a fully automatic mechanism to abstract higher-order tree-processing programs so as to verify a given property using PMRSs. However this abstraction mechanism is not complete. In this abstraction

procedure, pattern matched variables are abstracted into regular grammars generated by new non-terminals symbols. For instance, let's consider the following *Filter* function, filtering an input list of natural numbers according to a given predicate p.

$$\begin{aligned} \mathit{Filter}\ \underline{p}\ \mathit{nil} &\to \mathit{nil} \\ \mathit{Filter}\ \underline{p}\ (\mathit{cons}\ \underline{x}\ \underline{l}) &\to \mathit{Ite}\ (\underline{p}\ \underline{x})\ (\mathit{cons}\ \underline{x}\ (\mathit{Filter}\ \underline{p}\ \underline{l}))\ (\mathit{Filter}\ \underline{p}\ \underline{l}) \end{aligned}$$

We omit the definition of *Ite* representing the usual **`if-then-else`** control structure. We want to verify that the output list of this PMRS never contains any odd number when considering the following start symbol definition:

$$\begin{array}{lll} S \to \mathit{Filter}\ \mathit{Even}\ L & N \to 0 & L \to \mathit{nil} \\ & N \to s\ N & L \to \mathit{cons}\ N\ L \end{array}$$

The (non-deterministic) non-terminal L, combined with N defines all the possible input values of *Filter* (here we consider all the possible lists of natural numbers). We omit the definition of the non-terminal *Even* representing the predicate filtering even numbers. The PMRS abstraction procedure starts by abstracting all pattern-matched variables in the right-hand sides of each rules $(\underline{p}, \underline{x}, \underline{l})$ by its associated non-terminal according to the starting rule. Here *Filter* is called with *Even* and L, so the variable $\underline{p}$ is abstracted by *Even*, $\underline{l}$ by L and $\underline{x}$ by N (by unfolding L). We get:

$$\mathit{Filter}\ p\ \mathit{nil} \to \mathit{nil} \tag{3.1}$$

$$\mathit{Filter}\ p\ (\mathit{cons}\ x\ l) \to \mathit{Ite}\ (\mathit{Even}\ N)\ (\mathit{cons}\ N\ (\mathit{Filter}\ \mathit{Even}\ L))\ (\mathit{Filter}\ \mathit{Even}\ L) \tag{3.2}$$

A counter-example run is then searched in the abstraction. Here the following run reduces to a list containing an odd number:

$$\begin{aligned} \underline{S} &\to \\ \mathit{Filter}\ \mathit{Even}\ \underline{L} &\to \\ \underline{\mathit{Filter}\ \mathit{Even}\ (\mathit{cons}\ N\ L)} &\to \\ \mathit{Ite}\ (\mathit{Even}\ \underline{N})\ (\mathit{cons}\ N\ (\mathit{Filter}\ \mathit{Even}\ L))\ (\mathit{Filter}\ \mathit{Even}\ L) &\to \\ \underline{\mathit{Ite}\ (\mathit{Even}\ 0)\ (\mathit{cons}\ N\ (\mathit{Filter}\ \mathit{Even}\ L))\ (\mathit{Filter}\ \mathit{Even}\ L)} &\to^* \\ (\mathit{cons}\ N\ \underline{(\mathit{Filter}\ \mathit{Even}\ L)}) &\to^* \\ (\mathit{cons}\ \underline{N}\ \mathit{nil}) &\to^* \\ (\mathit{cons}\ (s\ 0)\ \mathit{nil}) & \end{aligned}$$

This is *spurious* counter-example, since this run is not valid in the original PMRS: where we have reduced N into *0* and later into $(s\ 0)$ in the abstraction hides in reality a single variable x in the original PMRS, which cannot hold two different values at once. Considering this spurious run, the abstraction is refined by unfolding the non-terminal N so as to not mix up *0* and $(s\ 0)$ anymore. As a result, rule (3.2) is unfolded into:

$$\begin{aligned} \mathit{Filter}\ \underline{p}\ (\mathit{cons}\ 0\ \underline{l}) &\to \mathit{Ite}\ (\mathit{Even}\ 0)\ (\mathit{cons}\ 0\ (\mathit{Filter}\ \mathit{Even}\ L))\ (\mathit{Filter}\ \mathit{Even}\ L) \\ \mathit{Filter}\ \underline{p}\ (\mathit{cons}\ (s\ \underline{x})\ \underline{l}) &\to \mathit{Ite}\ (\mathit{Even}\ (s\ N))\ (\mathit{cons}\ (s\ N)\ (\mathit{Filter}\ \mathit{Even}\ L))\ (\mathit{Filter}\ \mathit{Even}\ L) \end{aligned}$$

The procedure continues searching for counter-examples and unfolding non-terminals until a real counter-example if found, or the property is verified. This particular example however shows that this procedure is unable to generate arbitrary regular abstractions, and may diverge even for regular properties. In our example, it will continue unfolding N into $0, (s\ 0), (s\ (s\ 0)), \mathit{etc.}$, missing the required separation between even and odd numbers.

3.3.2 Regular Tree Languages based Techniques

The interest over the fully automated verification of tree processing programs can be traced back to Jones [JM79, JA07], who first proposed to abstract functional programs using regular tree languages. In this setting, a higher-order functional program is modeled using a term rewriting system $\mathcal{R}$, and the set of input states as a regular tree grammar G. For instance let us consider the following $\mathcal{R}$ (each rule is indexed for later use):

$$\begin{array}{llll} 1: & even(0) \to true & 3: & odd(0) \to false \\ 2: & even(s(\underline{n})) \to odd(\underline{n}) & 4: & odd(s(\underline{n})) \to even(\underline{n}) \end{array}$$

with the following input grammar G (where we call *even* with even numbers):

$$\begin{array}{ll} S \to even(E) & \\ E \to 0 \mid s(O) & O \to s(E) \end{array}$$

Jones and Andersen [JA07] then define an algorithm that computes a new grammar $G^*_{\mathcal{R}}$ over-approximating the set of states (or terms) reachable from G using $\mathcal{R}$:

$$R^*(\mathcal{L}(G)) \subseteq \mathcal{L}(G^*_{\mathcal{R}})$$

The grammar $G^*_{\mathcal{R}}$ is computed by adding new derivations to $\mathcal{G}$. For each rule a new non-terminal will be added that represents (an over-approximation of) the set of terms reachable from the rewriting rule. For each variable of the rule, a non-terminal is added that represents (an over-approximation of) the set of possible instantiations.

Example 3.3.1. *We want to verify that for all even number n, $even(n)$ never rewrite to $false$. To do that, let us build $G^*_{\mathcal{R}}$. From the rule 1 the derivation $R1 \to true$ is added, where $R1$ is the non-terminal for this rule. From the rule 2 the derivation $R2 \to odd(R2_n)$ is added where the non-terminal $R2_n$ represents the possibles instantiations for the variable n in this rule. In the same way, from the rules 3 and 4 the derivations $R3 \to$ false and $R4 \to even(R4_n)$ are added. To find the derivations rules of $R2_n$ and $R4_n$ we follow the initial grammar G. In the rule $S \to even(E)$, since we have $E \to 0$ and $E \to s(O)$, the subterm $even(E)$ can be rewritten using the rule 1 and 2. We hence add the derivations rules $S \to R1$ and $S \to R2$ to the grammar. For rule 2 this means that $R2_n$ can be instantiated with the non-terminal O. We hence add the rule $R2_n \to O$ to the grammar. Now that we know something new about $R2_n$, we can consider the previously added rule $R2 \to odd(R2_n)$. Again, since $R2_n \to O \to s(E)$, the subterm $odd(R2_n)$ rewrites into $even(E)$ with rule 4. We hence add the rule $R2 \to R4$ and $R4_n \to E$. By continuing this way, the final grammar $G^*_{\mathcal{R}}$ is defined by:*

$$\begin{array}{ll} S \to even(E) \mid R1 \mid R2 & \\ E \to 0 \mid s(O) & O \to s(E) \\ R1 \to true & \\ R2 \to odd(R2n) \mid R4 & R2n \to O \\ R3 \to false & \\ R4 \to even(R4n) \mid R1 \mid R2 & R4n \to E \end{array}$$

We can observe that in $G^*_{\mathcal{R}}$, there are no possible paths from S to $false$. Since we have $R^*(\mathcal{L}(G)) \subseteq \mathcal{L}(G^*_{\mathcal{R}})$ we can deduce that our property is verified.

The procedure that we have just illustrated continues until no new rules are added to the grammar. Since a finite amount of non-terminals is added to $G^*_{\mathcal{R}}$, it is guaranteed to terminate. However the way the abstraction $G^*_{\mathcal{R}}$ is built is independent of the verification goal. For a given program, only one abstraction is possible. It is not at all guaranteed that this over-approximation will be enough to verify the input property.

Example 3.3.2. *Let us consider the new (non-deterministic) TRS $\mathcal{R}'$ derived from $\mathcal{R}$ with the additional rules*

$$5: \quad ite(true, x, y) \to x \qquad 7: \quad rand(n) \to ite(even(n), n, rand(n))$$
$$6: \quad ite(false, x, y) \to y \qquad 8: \quad rand(n) \to rand(s(n))$$

Here rand is a non-deterministic function that generates random (even) numbers. Let us consider G' defined as

$$S \to rand(0)$$

We want to verify that rand always generate numbers that are even. Note however that this time, the initial grammar G' does not naturally separate even number from odd numbers. Using Jones's algorithm we compute the grammar $G'^*_{\mathcal{R}}$ over-approximating $\mathcal{R}^*(\mathcal{L}(G'))$. From rule 7, $G'^*_{\mathcal{R}}$ contains the derivation $R7 \to ite(even(R7_n), R7_n, rand(R7_n))$. Because the variable n in this rule (represented by the non-terminal $R7_n$) can be instantiated with an even number, we have $R7_n \to E$. This means that $even(R7_n)$ can be derived into true, which in turn means that $ite(even(R7_n), R7_n, rand(R7_n))$ can be derived into $R7_n$. We have $R7 \to^* R7_n$ in $G'^*_{\mathcal{R}}$. However n can also be instantiated with an odd number: $R7_n \to O$. Because of that, in $G'^*_{\mathcal{R}}$ we have $S \to R7 \to^* R7_n \to^* s(0)$. The computed abstraction contains an odd number, which contradicts the property we want to verify. The over-approximation $G'^*_{\mathcal{R}}$ is too rough.

This issue can be addressed by using the more recent Tree Automata Completion Algorithm [Gen98, GR10]. Just like Jones's algorithm, this algorithm's purpose is to compute the terms reachable from an input regular language, this time using tree automata instead of regular tree grammars. The previously considered grammar G can be translated into the following tree automaton $\mathcal{A} = \langle \{s, even, odd\}, \{q, q_{even}, q_{odd}\}, \{q\}, \Delta \rangle$ with Δ defined by:

$$even(q_{even}) \to q \qquad 0 \to q_{even}$$
$$s(q_{even}) \to q_{odd} \qquad s(q_{odd}) \to q_{even}$$

In its simplest form, given a TRS $\mathcal{R}$ and tree automaton $\mathcal{A}$, the completion algorithm is able to *complete* $\mathcal{A}$ using $\mathcal{R}$ into a new automaton $\mathcal{A}^*$ recognizing exactly $\mathcal{R}^*(\mathcal{L}(\mathcal{A}))$.

Example 3.3.3. *Similarly to Jones's algorithm, this algorithm works iteratively by finding missing transitions in the automaton. In $\mathcal{A}$ we have $even(0) \to_\Delta even(q_{even}) \to_\Delta q$. Since $even(0) \to_{\mathcal{R}} true$, then true should be recognized by $\mathcal{A}^*$. The tree automata completion algorithm proceeds by adding new transitions in $\mathcal{A}^*$ to have $true \to^*_{\Delta^*} q$. One way is to add the two transitions $true \to q_{true}$ and $q_{true} \to q$ (where q_{true} is a new state). In $\mathcal{A}$ we also have $even(s(q_{odd})) \to^*_\Delta q$. In the same way, since $even(s(q_{odd})) \to_{\mathcal{R}} odd(q_{odd})$ we add the transitions $odd(q_{odd}) \to q'$ and $q' \to q$ to $\mathcal{A}^*$. Finally, because of the newly added transitions we have $odd(s(q_{even})) \to^*_\Delta q$*

with $odd(s(q_{even})) \to_{\mathcal{R}} even(q_{even})$. *We already have* $even(q_{even}) \to q$ *so no new transitions are added. We have found every missing transitions and the completion algorithm stops. The output* $\mathcal{A}^*$ *is defined by:*

$$\begin{aligned} even(q_{even}) &\to q & 0 &\to q_{even} \\ q_{true} &\to q & s(q_{odd}) &\to q_{even} \\ q' &\to q & s(q_{even}) &\to q_{odd} \\ odd(q_{odd}) &\to q' & true &\to q_{true} \end{aligned}$$

One of the particularities of the Tree Automata Completion algorithm compared to Jones's algorithm is that this algorithm converges to a tree automaton $\mathcal{A}^*$ recognizing *exactly* $\mathcal{R}^*(\mathcal{L}(\mathcal{A}))$ when $\mathcal{L}(\mathcal{A})$ is finite [GR10]. However contrarily to Jones's algorithm, the Tree Automata Completion algorithm is not guaranteed to converge when the considered TRS is not terminating.

Example 3.3.4. *Let us consider again* $\mathcal{R}'$ *defined in Example 3.3.2, with the initial automaton* $\mathcal{A}$ *defined by:*

$$rand(q^0_{nat}) \to q^0_{rand} \qquad 0 \to q^0_{nat}$$

where q^0_{rand} *is the final state. It recognizes the same language as* G' *from Example 3.3.2. Again we want to verify that rand generates even numbers, this time using the Tree Automata Completion algorithm. First, the algorithm detects that we have* $rand(q^0_{nat}) \to_{\mathcal{R}} rand(s(q^0_{nat}))$ *and* $rand(q^0_{nat}) \to_{\mathcal{R}} ite(even(q^0_{nat}), q^0_{nat}, rand(q^0_{nat}))$. *New states and transitions are added to* $\mathcal{A}^*_{\mathcal{R}}$ *to have* $rand(s(q^0_{nat})) \to^*_{\mathcal{A}^*_{\mathcal{R}}} q^0_{rand}$ *and* $ite(even(q^0_{nat}), q^0_{nat}, rand(q^0_{nat})) \to^*_{\mathcal{A}^*_{\mathcal{R}}} q^0_{rand}$.

$$\begin{aligned} s(q^0_{nat}) &\to q^1_{nat} & even(q^0_{nat}) &\to q^0_{even} \\ rand(q^1_{nat}) &\to q^1_{rand} & ite(q^0_{even}, q^0_{nat}, q^0_{rand}) &\to q^0_{ite} \\ q^1_{rand} &\to q^0_{rand} & q^0_{ite} &\to q^0_{rand} \end{aligned}$$

Note how q^1_{nat} *now recognizes the term* $s(0)$ *and* q^1_{rand} *the term* $rand(s(0))$. *In the next iteration, just like with* q^0_{nat} *and* q^0_{rand}, *because* $rand(q^1_{nat}) \to_{\mathcal{R}} rand(s(q^1_{nat}))$ *and* $rand(q^1_{nat}) \to_{\mathcal{R}} ite(even(q^1_{nat}), q^1_{nat}, rand(q^1_{nat}))$ *the new state* q^2_{nat} *and* q^2_{rand} *are added, then* q_3 *and* q^3_{rand}, *and so on. In fact, for each integer* i, *the following transitions are added:*

$$\begin{aligned} s(q^{i-1}_{nat}) &\to q^i_{nat} & even(q^i_{nat}) &\to q^i_{even} \\ rand(q^i_{nat}) &\to q^i_{rand} & ite(q^i_{even}, q^i_{nat}, q^i_{rand}) &\to q^i_{ite} \\ q^i_{rand} &\to q^{i-1}_{rand} & q^i_{ite} &\to q^i_{rand} \end{aligned}$$

The algorithm diverges.

As illustrated by this example, since $\mathcal{R}^*(\mathcal{A})$ is uncomputable in general, the termination of the completion algorithm is not guaranteed. Multiple families of TRSs have been identified for which the termination is ensured [Gen16]. However in general, one must again rely upon an abstraction. In 2010 Genet et al. [GR10] proposes to use equations between patterns to merge terms in equivalence classes, effectively abstracting the program execution even when $\mathcal{L}(\mathcal{A})$ is finite.

Example 3.3.5. *In the previous example, we can force the completion algorithm to terminate using the equation $x = s(s(x))$. During the completion process, it will detect that state q^0_{nat} recognizes 0 while state q^2_{nat} recognizes $s(s(0))$. Because $0 = s(s(0))$ (according to our equation), states q^0_{nat} and q^2_{nat} are merged in $\mathcal{A}^*_{\mathcal{R}}$. As a consequence, the states recognizing even numbers are merged together, while all the states recognizing odd numbers are merged together. The infinite number of states collapses into a finite number of states, and the Tree Automata Completion algorithm terminates.*

The equation mechanism allows the completion algorithm to terminate, but it also allows to precisely control the precision of the generated abstraction. For a given set of equations E, the language recognized by $\mathcal{A}^*_{\mathcal{R}}$ follows the given inequality [Gen16]:

$$\mathcal{R}^*(\mathcal{L}(\mathcal{A})) \quad \subseteq \quad \mathcal{L}(\mathcal{A}^*_{\mathcal{R}}) \quad \subseteq \quad \mathcal{R}/E^*(\mathcal{L}(\mathcal{A}))$$

where $\mathcal{R}/E$ is the rewriting relation defined by $\{s \to_{\mathcal{R}/E} t \mid s =_E u \to_{\mathcal{R}} v =_E t\}$. The completion algorithm terminates as long as the number of equivalence classes defined by E is finite [Gen16]. By carefully choosing the equivalence classes (E) one can control the shape of the generated abstraction to solve any regular safety verification problem. Some leads have be proposed to automatically find the correct equivalence classes needed to solve a given problem. So far, existing equations generation procedures are not complete and can only guarantee the termination of the completion algorithm for first-order programs. On the other hand Matsumoto et al. [MKU15] define a procedure to infer equivalence classes directly represented with tree grammars, using a counter-example guided abstraction procedure. In this setting, the program is typed with tree automata, and the values abstracted into tree automata states. The tree automata types are then iteratively refined by looking for spurious counter-examples in the abstraction using a higher-order recursion scheme model checker. The refinement is performed by splitting the automata states following the information gathered from the counter-examples and using an SMT solver.

Example 3.3.6. *Consider the following TRS:*

$$\begin{array}{lll} even(0) \to true & even(s(0)) \to false & even(s(s(\underline{x}))) \to even(\underline{x}) \\ double(0) \to 0 & double(s(\underline{x})) \to s(s(double(\underline{x}))) & \end{array}$$

Note that in the original paper, authors are not working with rewriting systems but with their own representation of programs. This can be easily adapted to TRSs. Suppose we want to verify that the output of double is always even: the term $even(double(n))$ never rewrites to false for all natural number n. Using this technique, the program is first typed with tree automata and abstracted into a program manipulating states of these automata. Here the functions are given the following signatures:

$$\begin{aligned} even &: \mathcal{A}_{nat} \to \mathcal{A}_{bool} \\ double &: \mathcal{A}_{nat} \to \mathcal{A}_{nat} \end{aligned}$$

We start with the most general abstraction where $\mathcal{A}_{nat}$ (on the left) and $\mathcal{A}_{bool}$ (on the right) are defined as follows:

$$\begin{array}{ll} 0 \to q & true \to q' \\ s(q) \to q & false \to q' \end{array}$$

The initial TRS is then abstracted into a new TRS manipulating the states of the automata as values:

$$even(q) \to q' \qquad\qquad double(q) \to q$$

There exists a rewriting sequence in the abstraction violating our property: $even(q) \to q'$ *where* q *recognizes* 0 *and* q' *recognizes* $false$*. However this counter example is spurious since* $even(0)$ *does not rewrite to* false*. In practice spurious counter example are found using a higher-order model checker. Then the two abstracting tree automata are refined so that this spurious counter example cannot occur in the next abstraction. This is done by splitting each tree automaton state* q *into* $\{q_1, \ldots, q_n\}$ *(where* n *is called the "split number") and building the transitions of these new states following constraints generated from the counter-example. Starting from* 1*, the split number is increased until a collection of tree automata is found that satisfies the generated constraints using an SMT solver. Here a solution exists for* $n = 2$ *where* q *is split into* $\{q_{even}, q_{odd}\}$ *and* q' *into* $\{q_{true}, q_{false}\}$ *with the following transitions:*

$$\begin{aligned} 0 &\to q_{even} & \quad true &\to q_{true} \\ s(q_{even}) &\to q_{odd} & \quad false &\to q_{false} \\ s(q_{odd}) &\to q_{even} \end{aligned}$$

Minimizing the split number allows this technique to reach any regular abstraction. However because the split number affects every state, the generated abstraction may not be the smallest possible (in the number of states). Moreover a unique type is given to each function independently of the context which can again increase the number of states needed to abstract the function in a way that suits every context. This can become a problem considering that the number of states in the abstraction directly impacts the price of the SMT-constraints solving, especially since the described technique is not modular. In this thesis, we extend this series of work by going further using regular tree languages abstractions and rewriting systems. We explore the limits of the Tree Automata Completion algorithm (Chapter 4), design new ways of generating regular abstractions improving on Matsumoto et al. (Chapters 4 and 5), and define a modular verification technique for regular safety problems (Chapter 5). Finally we extend the reach of regular languages to the verification of relational properties (Chapter 6).

Chapter 4

Higher-Order Equational Abstractions

Since our goal is to verify tree-processing programs, we are mostly interested in the regular verification techniques presented in Section 3.3. In particular the Tree Automata Completion Algorithm (TAC) provides a flexible abstraction mechanism using equations, that can theoretically allow us to reach any regular abstraction of a program execution. However as of now, no automatic equations inference procedure has been formalized, and the only leads given by Genet [Gen16] only apply to first-order programs. In this chapter we pursue the work of Genet at al. [GR10, Gen16, Gen18] and explore how the TAC algorithm can be used to automatically solve regular safety problems over higher-order functional programs.

4.1 Introduction

Using the TAC algorithm a program is represented as a term rewriting system $\mathcal{R}$ and the set of (possibly infinite) inputs to this program as a tree automaton $\mathcal{A}$. The TAC algorithm computes (an over-approximation of) $\mathcal{R}^*(\mathcal{L}(\mathcal{A}))$, the set of reachable terms, as a new automaton $\mathcal{A}^*_{\mathcal{R}}$ by *completing* $\mathcal{A}$ with the missing transitions. The automaton $\mathcal{A}^*_{\mathcal{R}}$ effectively model the execution of the program, allowing us to solve regular safety problems (cf. Definitions 2.4.3 and 2.4.6) of the form $\langle \mathcal{R}, \mathcal{L}(\mathcal{A}), O \rangle$ for some language O.

Example 4.1.1. *The following term rewriting system $\mathcal{R}$ defines the filter function along with the two predicates even and odd on Peano's natural numbers.*

$$\begin{aligned} \mathit{filter}\ \underline{p}\ \mathit{nil} &\to \mathit{nil} \\ \mathit{filter}\ \underline{p}\ \mathit{cons}(\underline{x},\ \underline{l}) &\to \mathit{ite}(\underline{p}\ \underline{x},\ \mathit{cons}(x,\ \mathit{filter}\ \underline{p}\ \underline{l}),\ \mathit{filter}\ \underline{p}\ \underline{l}) \end{aligned}$$

$$\begin{aligned} \mathit{even}\ 0 &\to \mathit{true} \qquad \mathit{odd}\ 0 \to \mathit{false} \\ \mathit{even}\ s(\underline{n}) &\to \mathit{odd}\ \underline{n} \quad \mathit{odd}\ s(\underline{n}) \to \mathit{even}\ \underline{n} \end{aligned}$$

The rewriting rules of ite representing the if-then-else control structure are omitted. This function filters out from the input list any element that does not satisfy the input predicate p. We want to check that for all lists l of natural numbers, (filter even l) filters out every odd numbers. One way to do this is to write a higher-order predicate, for_all, and check that for all list l, (for_all even (filter even l)) never rewrites to false. Let $\mathcal{A}$ be the tree automaton recognizing terms of form

(for_all even (filter even l)) where l is any list of natural numbers. We can verify this property using the TAC algorithm to compute $\mathcal{A}^*_{\mathcal{R}}$*. If false is not recognized by* $\mathcal{A}^*_{\mathcal{R}}$*, since* $\mathcal{R}^*(\mathcal{L}(\mathcal{A})) \subseteq \mathcal{L}(\mathcal{A}^*_{\mathcal{R}})$ *we know in particular that for all list l, the input term does not rewrite to false.*

Since $\mathcal{R}^*(\mathcal{L}(\mathcal{A}))$ is uncomputable in general, the TAC algorithm may diverge. Genet at al. [GR10] proposes to group every term into a finite set of equivalence classes using equations. These equations can force $\mathcal{A}^*_{\mathcal{R}}$ to converge into an over-approximation of $\mathcal{R}^*(\mathcal{L}(\mathcal{A}))$. Depending on the set of equations used, any regular over-approximation can be reached, which means that any *regular safety problem* can theoretically be solved using this technique.

To transform the TAC algorithm into a fully automatic verification technique for regular safety problems, one must first develop a procedure that generates the correct equations set that gives the correct over-approximation of $\mathcal{R}^*(I)$. In his paper [Gen16], Genet draws the outline of a procedure able to quickly find such equations using "contracting equations". However using the generated equations, the termination of the TAC algorithm is only guaranteed in the case of first-order programs. In this chapter, we pursue the development of this equation generation procedure. We propose a solution to shortcomings mentioned above with the following contributions:

- We state and prove a general termination theorem for the Tree Automata Completion algorithm with contracting equations;
- From the conditions of the theorem we characterize a class of higher-order functional programs for which the completion algorithm terminates using contracting equations. This class covers common usage of higher-order features in functional programming languages.
- We discuss the completeness of the abstraction procedure based on contracting equations and propose a novel counter-example guided abstraction refinement (CEGAR) procedure which guaranties regular completeness and completeness in refutation.

The chapter is organized as follows: We start by giving a formal definition of the Tree Automata Completion Algorithm and its properties in Section 4.2. We state and prove a more general termination criterion for the TAC algorithm used with contracting equations in Section 4.3. We use this termination criterion to define a class of higher-order functional TRS on which there always exists a set of contracting equations that makes the TAC algorithm terminate in Section 4.4. We then define our novel abstraction procedure in Section 4.5. In Section 4.6, we present a series of experiments validating our verification technique. Section 4.7 concludes the chapter.

4.2 Tree Automata Completion Algorithm

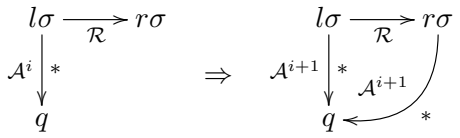
In this section we give more details about the Tree Automata Completion (TAC) algorithm that computes, for a given TRS $\mathcal{R}$ and initial REFD tree automaton $\mathcal{A}^0$, an *over-approximation* of $\mathcal{R}^*(\mathcal{L}(\mathcal{A}^0))$ as a new tree automaton $\mathcal{A}^*$. Remember that, as stated in Section 2.2, for any language $\mathcal{L}$ the set of reachable terms $\mathcal{R}^*(\mathcal{L})$ is defined as $\mathcal{R}^*(\mathcal{L}) = \{\ t \mid \exists s \in \mathcal{L}, s \to^*_{\mathcal{R}} t\ \}$. If $\mathcal{R}$ represents a functional program and $\mathcal{L}(\mathcal{A}^0)$ the set of initial states, then $\mathcal{R}^*(\mathcal{L}(\mathcal{A}^0))$ includes all intermediate computations and, in particular, the *outputs* of the program. This can be used to verify regular safety properties on the program.

4.2.1 Core Algorithm

The TAC algorithm proceeds by iteratively computing new automata $\mathcal{A}^1, \mathcal{A}^2, \ldots$ such that $\mathcal{A}^{i+1} = \mathcal{C}_\mathcal{R}(\mathcal{A}^i)$ until it reaches a fixed point, $\mathcal{A}^*$. Here, $\mathcal{C}_\mathcal{R}(\mathcal{A}^i)$ represents one step of *completion* and is performed by searching the missing transitions in $\mathcal{A}^i$ necessary to recognize $\mathcal{R}^*(\mathcal{L}(\mathcal{A}^i))$. To do that, the algorithm *completes* the *critical pairs* of $\mathcal{A}^i$, corresponding to $\mathcal{R}$-rewriting steps that are not taken into account in the automaton.

Definition 4.2.1 (Critical Pair)**.** *Let $\mathcal{R}$ be a TRS and $\mathcal{A}$ a tree automaton whose states live in $\mathcal{Q}$. Given a rule $l \to r \in \mathcal{R}$ a substitution $\sigma \in \mathcal{X} \mapsto \mathcal{Q}$ and a state $q \in \mathcal{Q}$. The pair $\langle r\sigma, q\rangle$ is* critical *when $l\sigma \to^*_{\mathcal{A}^i} q$ and $r\sigma \not\to^*_{\mathcal{A}^i} q$. To avoid redundancy, in the rest of the document we note such critical pair $\langle l \to r, q, \sigma\rangle$ which gives all the needed information. We name $\mathcal{CP}(\mathcal{R}, \mathcal{A})$ the set of critical pairs in $\mathcal{A}$ w.r.t. $\mathcal{R}$. Since $\mathcal{Q}$ and $\mathcal{X}$ are finite, so is $\mathcal{CP}(\mathcal{R}, \mathcal{A})$.*

A critical pair is *completed* by adding new transitions in the automaton to have $r\sigma \to^*_{\mathcal{A}^i} q$ as illustrated below:



Example 4.2.1. *Let $\Sigma = \{0 : 0, s : 1, add : 2\}$. Let us consider the following TRS $\mathcal{R}$:*

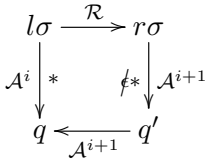
$$add(0, \underline{y}) \to \underline{x} \qquad\qquad add(s(\underline{x}), \underline{y}) \to add(\underline{x}, s(\underline{y}))$$

Let $\mathcal{A} = \langle \Sigma, \mathcal{Q}, \mathcal{Q}_f, \Delta\rangle$ the REFD tree automaton recognizing $add(s(0), s(0))$ using the following transitions:

$$0 \to q^0_{nat} \qquad\qquad s(q^0_{nat}) \to q^1_{nat} \qquad\qquad add(q^1_{nat}, q^1_{nat}) \to q^{1,1}_{add}$$

Because of the rule $add(s(\underline{x}), \underline{y}) \to add(\underline{x}, s(\underline{y}))$ and the substitution $\sigma = \{\underline{x} \mapsto q^0_{nat}, \underline{y} \mapsto q^1_{nat}\}$ the pair $\langle add(s(\underline{x}), \underline{y})\sigma, q^{1,1}_{add}\rangle$ is critical because $add(s(q^0_{nat}), q^1_{nat})$ is recognized by $q^{1,1}_{add}$ in $\mathcal{A}$, but $add(q^0_{nat}, s(q^1_{nat}))$ is not. The pair can be completed by adding the necessary transitions to recognize $add(q^0_{nat}, s(q^1_{nat}))$ in $q^{1,1}_{add}$.

In our case, the completion algorithm completes critical pairs by first adding or reusing transitions in the automaton so to have $r\sigma \to'^*_{\mathcal{A}^{i+1}} q'$ for some state q'. It then adds the ϵ-transition $q' \to_{\mathcal{A}^{i+1}} q$ symbolizing the rewriting step.



Note that the algorithm keeps the automaton *normalized*. As such, it may not directly add the transition $r\sigma \to q'$ in $\mathcal{A}^{i+1}$ since $r\sigma$ may not be a normal configuration (of the form $f(q_1, \ldots, q_n)$). Instead the completion algorithms proceeds by using a *normalization* procedure to ensure that $r\sigma$ is recognized in q' by adding or reusing *normal* transitions.

Definition 4.2.2 (Normalization)**.** *Let $\mathcal{A} = \langle \Sigma, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$ be a tree automaton. For a given pattern $t \in \mathcal{T}(\Sigma, \mathcal{Q})$, the normalization function $Norm(\mathcal{Q}, \Delta, t)$ returns a triple $\langle \mathcal{Q}', \Delta', q' \rangle$ with $\mathcal{Q}' \supseteq \mathcal{Q}$, $\Delta' \supseteq \Delta$ and $q' \in \mathcal{Q}'$ such that in the automaton $\mathcal{A}' = \langle \Sigma, \mathcal{Q}', \mathcal{Q}_f, \Delta' \rangle$ we have $t \to^{\not\epsilon *}_{\Delta'} q'$ using normalized transitions and preserving $\not\epsilon$-determinism. The normalization may add new states taken from an arbitrary infinite reserve of new states $\mathcal{Q}^*$. For any $\mathcal{Q}$, Δ and t, the normalization function is defined as follows:*

$$Norm(\mathcal{Q}, \Delta, q) = \langle \mathcal{Q}, \Delta, q \rangle$$
$$Norm(\mathcal{Q}, \Delta, f(t_1, \ldots, t_n)) = \langle \mathcal{Q}_n \cup \{\ q'\ \}, \Delta_n \cup \{\ f(q_1, \ldots, q_n) \to q'\ \}, q' \rangle$$

where for each $i \in [1..n]$, $\langle \mathcal{Q}_i, \Delta_i, q_i \rangle = Norm(\mathcal{Q}_{i-1}, \Delta_{i-1}, t_i)$ with $\mathcal{Q}_0 = \mathcal{Q}$, $\Delta_0 = \Delta$, and where $q' = q$ if $f(q_1, \ldots q_n) \to q \in \Delta_n$, or $q' \in \mathcal{Q}^* \setminus \mathcal{Q}_n$ (a new state) otherwise.

Example 4.2.2. *In the previous example, we need to add the term $add(q^0_{nat}, s(q^1_{nat}))$ in $\mathcal{A}$ in order to complete the critical pair $\langle add(s(\underline{x}), \underline{y}) \to add(\underline{x}, s(\underline{y})), \sigma, q^{1,1}_{add} \rangle$ where $\sigma = \{\underline{x} \mapsto q^0_{nat}, \underline{y} \mapsto q^1_{nat}\}$. To do that we use the normalization function: $Norm(\mathcal{Q}, \Delta, add(q^0_{nat}, s(q^1_{nat})))$. Transitions are added from bottom to top. By definition we start by computing $Norm(\mathcal{Q}, \Delta, q^0_{nat}) = \langle \mathcal{Q}, \Delta, q^0_{nat} \rangle$. We continue with $Norm(\mathcal{Q}, \Delta, q^1_{nat}) = \langle \mathcal{Q}, \Delta, q^1_{nat} \rangle$. By definition $Norm(\mathcal{Q}, \Delta, s(q^1_{nat}))$ is $\langle \mathcal{Q} \cup \{q^2_{nat}\}, \Delta \cup \{s(q^1_{nat}) \to q^2_{nat}\}, q^2_{nat} \rangle$ where q^2_{nat} is a new state recognizing $s(s(0))$ which was not recognized in $\mathcal{A}$. Finally by definition, $Norm(\mathcal{Q}, \Delta, add(q^0_{nat}, s(q^1_{nat})))$ is*

$$\langle \mathcal{Q} \cup \{q^2_{nat}\} \cup \{q^{0,2}_{add}\}, \Delta \cup \{s(q^1_{nat}) \to q^2_{nat}\} \cup \{add(q^0_{nat}, q^2_{nat}) \to q^{0,2}_{add}\}, q^{0,2}_{add} \rangle$$

where $q^{0,2}_{add}$ is a new state to recognize $add(q^0_{nat}, s(q^1_{nat}))$.

Definition 4.2.3 (One Step Completion)**.** *Let $\mathcal{A} = \langle \Sigma, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$ be a tree automaton, $\mathcal{R}$ be a left-linear TRS. The one step completed automaton is*

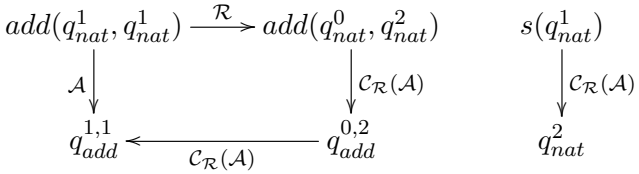
$$\mathcal{C}_{\mathcal{R}}(\mathcal{A}) = \langle \Sigma, \mathcal{Q}', \mathcal{Q}_f, \Delta' \rangle \text{ with } \langle \mathcal{Q}', \Delta' \rangle = Join(\mathcal{Q}, \Delta, \mathcal{CP}(\mathcal{R}, \mathcal{A}))$$

where for any $\mathcal{Q}, \Delta$ and set of critical pairs S, $Join(\mathcal{Q}, \Delta, S)$ is inductively defined by:

$$Join(\mathcal{Q}, \Delta, \emptyset) = \langle \mathcal{Q}, \Delta \rangle$$
$$Join(\mathcal{Q}, \Delta, \{\langle l \to r, q, \sigma \rangle\} \cup S) = Join(\mathcal{Q}', \Delta' \cup \{q' \to q\}, S)$$

with $\langle \mathcal{Q}', \Delta', q' \rangle = Norm(\mathcal{Q}, \Delta, r\sigma)$.

Example 4.2.3. *Let us compute one step of completion for our add example. By definition $\mathcal{C}_{\mathcal{R}}(\mathcal{A}) = \langle \Sigma, \mathcal{Q}', \mathcal{Q}_f, \Delta' \rangle$ with $\langle \mathcal{Q}', \Delta' \rangle = Join(\mathcal{Q}, \Delta, \mathcal{CP}(\mathcal{R}, \mathcal{A}))$. In our case $\mathcal{CP}(\mathcal{R}, \mathcal{A})$ is the singleton $\{\ \langle add(s(\underline{x}), \underline{y}) \to add(\underline{x}, s(\underline{y})), \sigma, q^{1,1}_{add} \rangle\ \}$. Then by definition we have $Join(\mathcal{Q}, \Delta, \mathcal{CP}(\mathcal{R}, \mathcal{A})) = \langle \mathcal{Q}', \Delta' \cup \{q \to q^{1,1}_{add}\} \rangle$ with $\langle \mathcal{Q}', \Delta', q^{0,2}_{add} \rangle = Norm(\mathcal{Q}, \Delta, add(q^0_{nat}, s(q^1_{nat})))$. We already have computed this in the previous example to get $\mathcal{Q}' = \mathcal{Q} \cup \{q^2_{nat}\} \cup \{q^{0,2}_{add}\}$ and $\Delta' = \Delta \cup \{s(q^1_{nat}) \to q^2_{nat}\} \cup \{add(q^0_{nat}, q^2_{nat}) \to q^{0,2}_{add}\}$. The overall difference between $\mathcal{A}$ and $\mathcal{C}_{\mathcal{R}}(\mathcal{A})$ can be summarized as follows:*



4.2.2 Properties of the TAC Algorithm

Lemma 1 (Soundness)**.** *All the definitions stated above ensure by construction that each completion step i, the following properties are verified:*

$$L(\mathcal{A}^i) \subseteq L(\mathcal{A}^{i+1}) \quad \textit{and}$$
$$s \in L(\mathcal{A}^i) \;\Rightarrow\; s \to_{\mathcal{R}} t \;\Rightarrow\; t \in L(\mathcal{A}^{i+1})$$

This implies that, if a fixed point $\mathcal{A}^*$ is found, then it recognizes an over-approximation of $\mathcal{R}^*(L(\mathcal{A}))$.

In the final automaton $\mathcal{A}^*$ produced by the TAC algorithm, ϵ-transitions are exclusively used to connect states that are connected by a rewriting relation. Two states q and q' are connected with an ϵ-transition $q' \to q$ iff rewriting rule $l \to r$ of the considered TRS $\mathcal{R}$ and a substitution σ such that $l\sigma \in \mathcal{L}(\mathcal{A}^*, q)$, $r\sigma \in \mathcal{L}(\mathcal{A}^*, q')$. More precisely, ϵ-transitions represents rewriting step that occurs at the root of terms (position λ). A direct consequence of this particular structure of $\mathcal{A}^*$ is that terms connected by rewriting steps that do not occur at the root are recognized by the same state in $\mathcal{A}^*$.

Lemma 2 (Subterm Collapse)**.** *Let u, v be two terms recognized by $\mathcal{A}^*$. If there exists a rule $l \to r$ a substitution σ and a position $p \in Pos(u)$ such that $p \neq \lambda$, $u|_p = l\sigma$ and $v = u[r\sigma]_p$, then u and v are recognized by the same state in $\mathcal{A}^*$.*

Proof. Let q be the state recognizing u in $\mathcal{A}^*$. Since $\mathcal{A}^*$ is normalized, there exists a context C and a state q_l such that $u = C[l\sigma]$, $l\sigma \to^*_{\mathcal{A}^*} q_l$ and $C[q_l] \to^*_{\mathcal{A}^*} q$. Because $l\sigma \to_{\mathcal{R}} r\sigma$ and $\mathcal{A}^*$ is a fixed point to the TAC algorithm, there exists a state q_r such that $r\sigma \to^*_{\mathcal{A}^*} q_r$ with $q_r \to q_l$. So $v = C[r\sigma] \to^*_{\mathcal{A}^*} C[q_r] \to_{\mathcal{A}^*} C[q_l] \to^*_{\mathcal{A}^*} q$, v is also recognized by q. □

If we transpose this lemma in the context of program verification, considering that the TAC algorithm computes an over-approximation of the reachable states of the program, this means that the TAC algorithm computes in fact an *abstraction* of the program execution where multiple execution states are recognized by the same tree automaton state. We call the particular abstraction realized by the TAC algorithm a *subterm-collapsing abstraction.*

Definition 4.2.4 (Subterm-Collapsing Abstraction)**.** *Let Σ be a ranked alphabet. An abstraction $\rightsquigarrow$ of $\langle \mathcal{T}(\Sigma), \to, V\rangle$ into $\langle \mathcal{Q}, \to', V'\rangle$ is subterm-collapsing iff for all terms $t = f(t_1, \ldots, t_n)$ of $\mathcal{T}(\Sigma)$ and u such that $t_i \to^* u$ for some $i \in [1, n]$, then for all $q \in \mathcal{Q}$ such that $t \rightsquigarrow q$ we also have $t[u]_i \rightsquigarrow q$.*

In addition, because $\mathcal{A}^*$ is $\not\epsilon$-deterministic, the TAC algorithm can only generate functional abstractions (cf. Definition 2.4.8).

4.2.3 Equations

It is in general impossible to compute a tree automaton recognizing exactly $\mathcal{R}^*(L(\mathcal{A}))$. In most cases the TAC algorithm will compute an over-approximation of $\mathcal{R}^*(L(\mathcal{A}))$ (depending on the shape of the initial automaton), or diverge. In their paper [GR10], Genet at al. propose to use equations to merge states during the completion algorithm, which can enforce the termination of the algorithm and control the precision of the approximation. From a program verification point of view, equations can precisely control how execution states are abstracted.

Example 4.2.4. *This example shows how using equations can lead to approximations in tree automata. Let $\mathcal{A}$ be the tree automaton defined by the set of transitions $\Delta = \{0 \to q_0, s(q_0) \to q_1\}$. This automaton recognizes the two terms 0 in q_0 and $s(0)$ (also known as 1) in q_1. Let $E = \{s(\underline{x}) = \underline{x}\}$ containing the equation that equates a number and its successor. For $\sigma = \{\underline{x} \mapsto 0\}$ we have $s(\underline{x})\sigma \to_{\mathcal{A}} q_1$, $\underline{x}\sigma \to_{\mathcal{A}} q_0$ and $s(\underline{x})\sigma =_E \underline{x}\sigma$. Then in the completed automaton, q_0 and q_1 are merged. The resulting automaton has transitions $\{\ 0 \to q_0,\ s(q_0) \to q_0\ \}$, which recognizes $\mathbb{N}$ in q_0.*

Definition 4.2.5 (Equation Set)**.** *An equation set E is composed of equations of the form $l = r$ where $l, r \in \mathcal{T}(\Sigma, \mathcal{X})$. From E we derive the relation $=_E$ as the* smallest congruence *such that for all terms l, r and substitution σ we have:*

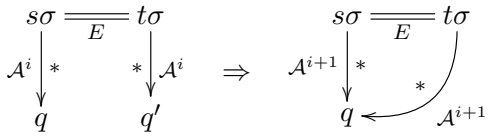
$$l = r \in E \quad \Rightarrow \quad l\sigma =_E r\sigma$$

The set of equivalence classes defined by $=_E$ on $\mathcal{T}(\Sigma)$ is noted $\mathcal{T}(\Sigma)/_{=E}$.

In this chapter we also write $\vec{E}$ for the TRS $\{\ l \to r \mid l = r \in E\ \}$. The tree automata completion algorithm is altered so that at each completion step, the algorithm *simplifies* the automaton by merging states together according to E.

Definition 4.2.6 (Simplification Relation)**.** *Let $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$ be a tree automaton and E be a set of equations. If $s = t \in E, \sigma : \mathcal{X} \mapsto \mathcal{Q}, q, q' \in \mathcal{Q}$ such that $s\sigma \to_{\mathcal{A}}^{\not\varepsilon *} q, t\sigma \to_{\mathcal{A}}^{\not\varepsilon *} q'$ and $q \neq q'$ then $\mathcal{A}$ can be simplified into $\mathcal{A}' = \mathcal{A}\{q' \mapsto q\}$ (where q' has been substituted by q), denoted by $\mathcal{A} \rightsquigarrow_E \mathcal{A}'$.*

We write $\mathcal{S}_E(\mathcal{A})$ for the unique [GR10] automaton (up to renaming) $\mathcal{A}'$ such that $\mathcal{A} \rightsquigarrow_E^* \mathcal{A}'$ and $\mathcal{A}'$ is irreducible by $\rightsquigarrow_E$. One completion step is now defined by $\mathcal{A}^{i+1} = \mathcal{S}_E(\mathcal{C}_{\mathcal{R}}(\mathcal{A}^i))$.



With the simplification phase, equations can be used to enforce termination of the algorithm by over-approximating $\mathcal{R}^*(L(\mathcal{A}))$ even in cases where it would normally diverge [Gen16]. Even better, equations can be used to control the precision of the over-approximation as long as the initial automaton does not recognize multiples equivalence classes of $\mathcal{T}(\Sigma)/_{=E}$ without epsilon transitions with the same state: it is $\mathcal{R}/E$-coherent [Gen16].

Definition 4.2.7 (Coherent Automaton)**.** *Let $\mathcal{R}$ be a term rewriting system, $\mathcal{A}$ a tree automaton and E a set of equations. The automaton $\mathcal{A}$ is said to be $\mathcal{R}/E$-coherent iff for all state $q \in \mathcal{Q}$, for all terms $s, t \in \mathcal{L}(\mathcal{A}, q)$ we have:*

$$s \to_{\mathcal{A}}^{\not\varepsilon *} q \wedge t \to_{\mathcal{A}}^{\not\varepsilon *} q \Rightarrow s =_E t$$
$$s \to_{\mathcal{A}}^{\not\varepsilon *} q \wedge t \to_{\mathcal{A}}^{*} q \Rightarrow s \to_{\mathcal{R}/E}^{*} t$$

where $\mathcal{R}/E$ is the rewriting relation defined as $\{\ s \to_{\mathcal{R}/E} t \mid s =_E u \to v =_E t\}$.

It is always possible to transform any tree automaton into a $\mathcal{R}/E$-coherent automaton. Moreover, $\mathcal{R}/E$-coherence is preserved by completion and simplification. For this reason in the rest of the chapter, we always assume the automata used in the TAC algorithm to be $\mathcal{R}/E$-coherent.

Theorem 17 in Gen16 (Precision). *Let $\mathcal{R}$ be a term rewriting system, E a set of equations and $\mathcal{A}$ an initial automaton. If the the Tree Automaton Completion algorithm on $\mathcal{R}, \mathcal{A}$ and E converges into $\mathcal{A}^*_{\mathcal{R}}$, then the language recognized by $\mathcal{A}^*$ verifies the given inequality:*

$$\mathcal{R}^*(\mathcal{L}(\mathcal{A})) \subseteq \mathcal{L}(\mathcal{A}^*) \subseteq \mathcal{R}/E^*(\mathcal{L}(\mathcal{A}))$$

Remember that because the output $\mathcal{A}^*_{\mathcal{R}}$ is $\not\epsilon$-deterministic, the TAC algorithm can only generate functional abstractions (cf. Definition 2.4.8). Thankfully with this theorem, Genet [Gen18] showed that any regular over-approximation (abstraction) of $\mathcal{R}^*(\mathcal{L}(\mathcal{A}))$ can be reached, given the right equations.

Theorem 32 in Gen18 (Completeness). *Let $\mathcal{R}$ be a left-linear term rewriting system and $\mathcal{A}$ an initial reduced ϵ-free automaton. Let $\mathcal{L}$ be a regular language such that $\mathcal{R}^*(\mathcal{L}(\mathcal{A})) \subseteq \mathcal{L}$. Then there exists a set of equations E such that $\mathcal{L}(\mathcal{A}^*_{\mathcal{R}/E}) \subseteq \mathcal{L}$.*

A consequence of this theorem is that the Tree Automata Completion algorithm combined with equations can theoretically be used to solve any regular problem. The main difficulty of this technique becomes finding E that ensure both termination and verifiability of the considered property.

4.2.4 Contracting Equations

One naive way of finding equations E that ensure both termination and verifiability of the property would be to simply enumerate every possible set of equations. The enumeration must be fair so that if an equations set verifying the property exists, it will eventually be found. This is not practicable of course because the number of possibilities is too high. In his paper [Gen16], Genet proposes a method to reduce the number of possibilities by considering only *contracting equations* over the *constructor* terms (or values). The set of equations is then completed with *reflexive* and $\mathcal{R}$-*transitive* equations.

Definition 4.2.8 (Contracting Equations). *Let $\mathcal{L} \subseteq \mathcal{T}(\Sigma)$. A set of equations is contracting for $\mathcal{L}$, denoted by $E^c_{\mathcal{L}}$, if all equations of $E^c_{\mathcal{L}}$ are of the form $u = u|_p$ with u a linear term of $\mathcal{T}(\Sigma, \mathcal{X})$, $p \neq \lambda$ and if the set of normal forms of $\mathcal{L}$ w.r.t the TRS $\vec{E^c_{\mathcal{L}}} = \{\ u \to u|_p \mid u = u|_p \in E^c_{\mathcal{L}}\ \}$ is finite.*

Example 4.2.5 (Contracting Equations). *Assume that $\Sigma = \{\ 0 : 0, s : 1\ \}$. The set $E^c_{\mathcal{L}} = \{\ s(x) = x\ \}$ is contracting for $\mathcal{L} = \mathcal{T}(\Sigma)$ because the set of normal forms of $\mathcal{T}(\Sigma)$ with respect to $\vec{E^c_{\mathcal{L}}} = \{\ s(x) \to x\ \}$ is the (finite) set $\{\ 0\ \}$. The set $E^c_{\mathcal{L}} = \{\ s(s(x)) = x\ \}$ is contracting because the normal forms of $\{\ s(s(x)) \to x\ \}$ are $\{\ 0, s(0)\ \}$.*

Combined with *reflexive equations*, contracting equations can enforce termination of the TAC algorithm.

Definition 4.2.9 (Reflexive Equations). *Let Σ be a ranked alphabet. We name E_r the set of* reflexive equations *defined as:*

$$E_r = \{\ f(x_1, \ldots, x_n) = f(x_1, \ldots, x_n) \mid f \in \Sigma^n\ \}$$

Theorem 62 in Gen16 (Termination with Contracting Equations). *Let $\mathcal{R}$ be a left-linear term rewriting system defined over $\mathcal{T}(\Sigma)$, $\mathcal{A}$ an initial automaton and $E^c_{\mathcal{T}(\Sigma)}$ a set of contracting equations for $\mathcal{T}(\Sigma)$. If $E \supseteq E^c_{\mathcal{T}(\Sigma)} \cup E_r$ then the Tree Automaton Completion algorithm terminates on $\mathcal{R}$, $\mathcal{A}$ and E.*

Contracting equations alone are not enough to significantly reduce the equations sets search space when it comes to solving a verification problem on a functional TRS. When the considered TRS $\mathcal{R}$ is a first-order functional TRS defined over $\Sigma = \mathcal{F} \cup \mathcal{C}$, Genet proposes to focus on contracting the values manipulated by the program by defining contracting equations over $\mathcal{T}(\mathcal{C})$ instead of $\mathcal{T}(\Sigma)$. In this setting, the set of equations is then completed with $\mathcal{R}$-transitive and reflexive equations to ensure the termination of the TAC algorithm.

Definition 4.2.10 ($\mathcal{R}$-Transitive Equations)**.** *Let $\mathcal{R}$ be a TRS. We name $E_{\mathcal{R}}$ the set of* $\mathcal{R}$-transitive equations *defined as:*

$$E_{\mathcal{R}} = \{\ l = r \mid l \to r \in \mathcal{R}\ \}$$

Lemma 64 in Gen16 (Termination with Contracting Equations on Constructors)**.** *Let $\mathcal{R}$ be a first-order function TRS defined over $\mathcal{T}(\Sigma)$ with $\Sigma = \mathcal{F} \cup \mathcal{C}$, $\mathcal{A}$ an initial automaton and $E^c_{\mathcal{T}(\mathcal{C})}$ a set of contracting equations for $\mathcal{T}(\mathcal{C})$. If $E \supseteq E^c_{\mathcal{T}(\mathcal{C})} \cup E_{\mathcal{R}} \cup E_r$ then the Tree Automaton Completion algorithm terminates on $\mathcal{R}$, $\mathcal{A}$ and E.*

This effectively reduce the search space for E by focusing the search on $E^c_{\mathcal{T}(\mathcal{C})}$, ignoring function symbols.

Example 4.2.6 (Contracting equations in action)**.** *This example is derived from Example 3.3.4 developed in Section 3.3.2. Consider the following first-order functional TRS $\mathcal{R}$ defining a non-deterministic rand function generating an even random number greater or equal to its input:*

$$rand(\underline{n}) \to ite(even(\underline{n}), \underline{n}, rand(\underline{n}))$$
$$rand(\underline{n}) \to rand(s(\underline{n}))$$

The ite rules defining the standard **`if-then-else`** *control structure are omitted but can be found in the original example. We want to show that $even(rand(0))$ cannot rewrite to false: $false \notin \mathcal{R}^*(I)$ with $I = \{\ even(rand(0))\ \}$. Even if the input language is finite, this cannot be decided simply by evaluating the initial term because of the non-determinism of $\mathcal{R}$. We hence use the TAC algorithm to compute an over-approximation of $\mathcal{R}^*(I)$. The smallest initial automaton $\mathcal{A}$ recognizing I is defined by the following transitions:*

$$0 \to q_0 \qquad rand(q_0) \to q^0_{rand} \qquad even(q^0_{rand}) \to q^0_{even}$$

where q^0_{even} is the unique final state. As shown in the original example, without equations, the TAC algorithm will diverge while trying to generate all the possible transitions of the form

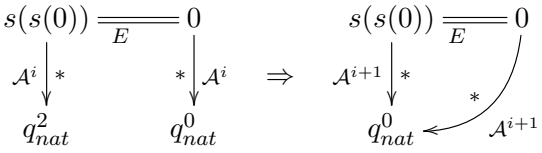
$$s(q^{n-1}_{nat}) \to q^n_{nat} \qquad even(q^n_{nat}) \to q^n_{even}$$
$$rand(q^n_{nat}) \to q^n_{rand} \qquad ite(q^n_{even}, q^n_{nat}, q^n_{rand}) \to q^n_{ite}$$
$$q^n_{rand} \to q^{n-1}_{rand} \qquad q^n_{ite} \to q^n_{rand}$$

for each number n of $\mathbb{N}$. To solve this verification problem, another set of equations must be used. This is where contracting equations become handy. By using the contracting equations set $E^c_{\mathcal{T}(\mathcal{C})} = \{\ s(s(0)) = 0\ \}$, even and odd number will be merged into two separate equivalence classes, allowing the TAC algorithm to terminate

with a more precise over-approximation of $\mathcal{R}^*(I)$*. By definition the full set of equations* E *used during the completion is* $E = E^c_{\mathcal{T}(\mathcal{C})} \cup E_{\mathcal{R}} \cup E_r$ *with*

$$E_{\mathcal{R}} \supseteq \left\{ \begin{array}{l} rand(\underline{n}) = ite(even(\underline{n}), \underline{n}, rand(\underline{n})) \\ rand(\underline{n}) = rand(s(\underline{n})) \end{array} \right\}$$

Here the initial automaton $\mathcal{A}$ *is already* $\mathcal{R}/E$*-coherent. Using this initial automaton and equations, the TAC algorithm will merge every state of the form* q_f^{2k} *in one equivalence class and all the states of the form* q_f^{2k+1} *in another class during the simplification phase:*



The TAC algorithm will generate the following transitions when converging to $\mathcal{A}^*$*:*

$$\begin{array}{rr} & true \to q^0_{even} \\ s(q^1_{nat}) \to q^0_{nat} & even(q^0_{nat}) \to q^0_{even} \\ rand(q^0_{nat}) \to q^0_{rand} & ite(q^0_{even}, q^0_{nat}, q^0_{rand}) \to q^0_{ite} \\ q^0_{rand} \to q^1_{rand} & q^0_{ite} \to q^0_{rand} \end{array}$$

$$\begin{array}{rr} & false \to q^1_{even} \\ s(q^0_{nat}) \to q^1_{nat} & even(q^1_{nat}) \to q^1_{even} \\ rand(q^1_{nat}) \to q^1_{rand} & ite(q^1_{even}, q^1_{nat}, q^1_{rand}) \to q^1_{ite} \\ q^1_{rand} \to q^0_{rand} & q^1_{ite} \to q^1_{rand} \end{array}$$

Note however that because of the set of $\mathcal{R}$*-transitive equations* $E_{\mathcal{R}} \subseteq E$*, states connected by epsilon transitions are also be merged during the completion. The actual transitions added into* $\mathcal{A}^*$ *are:*

$$\begin{array}{rr} true \to q^0_{even} & false \to q^1_{even} \\ s(q^1_{nat}) \to q^0_{nat} & even(q^0_{nat}) \to q^0_{even} \\ s(q^0_{nat}) \to q^1_{nat} & even(q^1_{nat}) \to q^1_{even} \\ rand(q^0_{nat}) \to q_{rand} & rand(q^1_{nat}) \to q_{rand} \\ ite(q^0_{even}, q^0_{nat}, q_{rand}) \to q_{ite} & ite(q^1_{even}, q^1_{nat}, q_{rand}) \to q_{ite} \\ q_{rand} \to q_{rand} & q_{ite} \to q_{rand} \end{array}$$

Note how false *is recognized by the automaton, but not by the final state* q^0_{even}*. The over-approximation is more precise and proves that* false *is not in* $\mathcal{R}^*(I)$*, solving our verification problem.*

Because of the equation set $E_{\mathcal{R}}$, the resulting abstraction is not only subterm-collapsing, but collapsing (cf. Definition 2.4.9). However this allows us to focus on $E^c_{\mathcal{T}(\mathcal{C})}$ which considerably reduce the search space. Termination is ensured when $\mathcal{R}$ is complete because every term eventually rewrites into a term of $\mathcal{T}(\mathcal{C})$ which is captured by $E^c_{\mathcal{T}(\mathcal{C})}$. However this is not the case for higher-order functional TRSs

where the special symbol @ is used to encode higher-order functions into first-order TRSs. This means that termination guaranties are lost on higher-order programs as we show in the following example.

Example 4.2.7 (Unbounded partial applications stack)**.** *Let $\Sigma = \mathcal{F} \cup \mathcal{C}$ be a ranked alphabet such that $\mathcal{F} = \{f, g\}$ and $\mathcal{C} = \{a\}$. Consider the following higher-order functional TRS $\mathcal{R}$ defined by the rules:*

$$f\ \underline{x} \to g\ (f\ \underline{x}) \qquad\qquad g\ \underline{h}\ \underline{y} \to \underline{h}\ \underline{y}$$

We want to an over-approximation of the $\mathcal{R}^*(I)$ where I is the singleton $\{\ (f\ a)\ \}$. This TRS is not terminating but it is still possible to find a set of equations E such that the TAC algorithm terminates. The problem is that E cannot be of the form $E^c_{\mathcal{T}(\mathcal{C})} \cup E_{\mathcal{R}} \cup E_r$ where $E^c_{\mathcal{T}(\mathcal{C})}$ is a set of contracting equations over $\mathcal{C}$. Since $\mathcal{C}$ contains a single constant a, the only contracting equations set over $\mathcal{C}$ is $E^c_{\mathcal{T}(\mathcal{C})} = \emptyset$. This is not enough for the TAC algorithm to terminate on this instance. To see why, we can use $\mathcal{R}$ to rewrite the initial term $(f\ a)$ and get:

$$f\ a \to g\ (f\ a) \to g\ (g\ (f\ a)) \to g\ (g\ (g\ (f\ a))) \to \dots$$

Note how g is never fully applied here. Contracting equations are only defined over constructor symbols (a), but g and @ the hidden symbol encoding higher-order (cf. Definition 2.2.16) are not constructor symbols. This means that each term of the form $g\ (\dots\ (g\ (f\ a)))$ is in its own unique equivalence class w.r.t. E. As a consequence, the TAC algorithm will diverge trying to add infinitely many states in the automaton to recognize these terms.

Of course one solution to solve this problem would be to extend the contracting equation set to contract partial applications. However this greatly expands the equations sets search space which makes it impracticable with current equations inference techniques. Moreover in practice, because partial applications are often part of total applications, contracting them often leads to many equations sets generating poorly precise abstractions of the control flow. In the next section we see how to preserve termination while keeping the equations search space small by defining a more general termination theorem of the TAC algorithm with contracting equations. We then define a general class of higher-order TRS for which the theorem applies with contracting equation over the values ($\mathcal{T}(\mathcal{C})$).

4.3 Termination Criterion Using Contracting Equations

In the previous section we have seen how equations can be used to control the precision of the TAC algorithm. In particular how *contracting equations on values* can be used to reduce the search space when looking for an equations set able to verify (prove or disprove) a given (regular) property. Contracting equations have been proposed by Genet et al. [GR10], and successfully used to rather efficiently verify first-order programs. However as we have seen in the previous section, the termination of the completion algorithm with contracting equations on values has only be proved for first-order TRSs. In this section we prove a more general termination theorem (Theorem 1) for the TAC algorithm with contracting equations, that does not rely on the order of the considered TRSs. We show that termination of the completion algorithm with a set of equations E is ensured under the following conditions: if (i) for every step k of completion, $\mathcal{A}^k$ is reduced and $\not\epsilon$-deterministic (REFD); (ii) every

term of $\mathcal{A}^k$ can be rewritten into a term of a given language $\mathcal{L} \subseteq \mathcal{T}(\Sigma)$ using $\mathcal{R}$; (iii) $\mathcal{L}$ has a finite number of equivalence classes w.r.t. E. Condition (i) is ensured by showing that, in our verification setting, completion preserves REFD. Condition (ii) is ensured for instance when $\mathcal{R}$ is terminating. To simplify, in the rest of the chapter we assume that $\mathcal{R}$ is terminating. The last condition is ensured by having $E \supseteq E^c_{\mathcal{L}}$ where $E^c_{\mathcal{L}}$ is a set of *contracting equations* over $\mathcal{L}$ (cf. Definition 4.2.8). The exact theorem is as follows:

Theorem 1. *Let $\mathcal{A}$ be an REFD tree automaton, $\mathcal{R}$ a left-linear TRS, E a set of equations and $\mathcal{L}$ a language closed by subterms such that for all $k \in \mathbb{N}$ and for all $s \in L_{\trianglerighteq}(\mathcal{A}^k)$, there exists $t \in \mathcal{L}$ s.t. $s \to^*_{\mathcal{R}} t$. If $E \supseteq E^r \cup E^c_{\mathcal{L}} \cup E_{\mathcal{R}}$ then the completion of $\mathcal{A}$ by R and E terminates with a REFD $\mathcal{A}^*$.*

Remember that for any language $\mathcal{L}$, following Definition 2.1.7 we write $\mathcal{L}_{\trianglerighteq}$ for the smallest superset of $\mathcal{L}$ closed under the subterm relation. The idea of the proof developed in this sections and represented on Figure 4.1 is the following: We identify the set G of normal forms of $\mathcal{L}$ w.r.t. $\vec{E^c_{\mathcal{L}}}$. G is finite by definition of contracting equations. We start by showing that, thanks to the contracting equations $E^c_{\mathcal{L}}$, the number of states in $\mathcal{A}^*$ needed to recognize $\mathcal{L}$ is bounded by $|G|$. We then conclude that, thanks to the $\mathcal{R}$-transitive equations $E_{\mathcal{R}}$, the number of states in $\mathcal{A}^*$ is bounded by $|G|$.

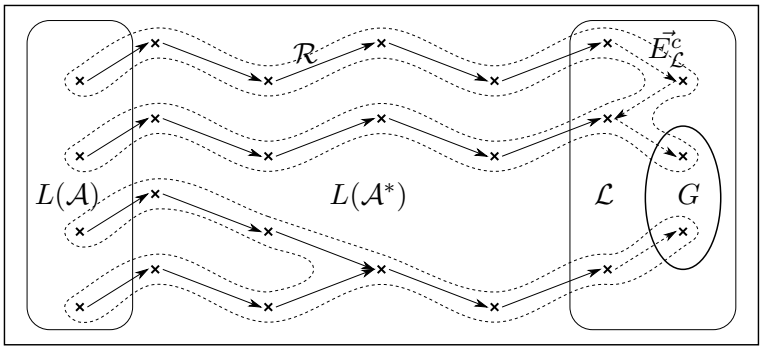


Figure 4.1: Completion termination proof idea. Each cross is a term. Plain arrows are rewriting steps and all terms along rewriting steps are equal w.r.t. $E_{\mathcal{R}}$. Dotted arrows represent *contractions* over $\vec{E^c_{\mathcal{L}}}$. We show that by using $E^c_{\mathcal{L}}$ and $E_{\mathcal{R}}$ the number of equivalence classes (circled with dashed lines) is bounded by $|G|$.

4.3.1 The Role of Contracting Equations

To prove termination of the TAC algorithm, we first prove that it is possible to bound the number of states needed in $\mathcal{A}^*$ to recognize a language $\mathcal{L}$ by the number of normal forms of $\mathcal{L}$ with respect to $\vec{E^c_{\mathcal{L}}}$ (Lemma 4). In our case $\mathcal{L}$ will be the set of output terms of the program. Since $\mathcal{A}^*$ does not only recognize the output terms, we need additional states to recognize intermediate computation terms. In the proof of Theorem 1 we show that with $E_{\mathcal{R}}$, the simplification steps will merge the states recognizing the intermediate computation with the states recognizing the outputs. If the latter set of states is finite then using Lemma 4 we can show that $\mathcal{A}^*$ is finite.

This theorem only holds for REFD tree automata. Thus we first need to prove that completion preserves REFD. It is already known that completion preserves

$\not\epsilon$-determinism and $\not\epsilon$-reduction [Gen16]. Considering that an $\not\epsilon$-deterministic and $\not\epsilon$-reduced automaton that is ϵ-free is REFD by definition, Lemma 3 proves preservation of ϵ-freeness.

Lemma 3. *Let $\mathcal{A}$ be an ϵ-free tree automaton and E a set of equations. If $E \supseteq E_{\mathcal{R}}$ then $\mathcal{S}_E(\mathcal{C}_{\mathcal{R}}(\mathcal{A}))$ is ϵ-free.*

Proof. $\mathcal{A}$ is ϵ-free so for each critical pair $\langle l \to r, \sigma, q\rangle$ we have $l\sigma \to^{\not\epsilon}_{\mathcal{A}} q$. The resolution of the critical pair only adds ϵ-transitions of form $q \to q'$ with q' a new state [Gen16]. So, in $\mathcal{C}_{\mathcal{R}}(\mathcal{A})$, for every transition $q \to q'$ such that $q \neq q'$ there exist s, t such that $s \to^{\not\epsilon} q$, $t \to^{\not\epsilon} q'$ and $s \to_{\mathcal{R}} t$. Now, since $E \supseteq E_{\mathcal{R}}$, we have $s =_E t$, and q and q' are merged in the simplified automaton $\mathcal{S}_E(\mathcal{C}_{\mathcal{R}}(\mathcal{A}))$. The only ϵ-transitions that remain after the simplification steps are of the form $q \to q$ and can be removed without altering the recognized language. □

The following Lemma 4 states that with an REFD tree-automaton $\mathcal{A}$, the number of states needed in $\mathcal{S}_E(\mathcal{A})$ to recognize a language $\mathcal{L}$ is bounded by the number of normal forms of $\mathcal{L}$ w.r.t. $\vec{E^c_{\mathcal{L}}}$.

Lemma 4. *Let $\mathcal{A}$ be a REFD tree automaton, $\mathcal{L}$ a language and E a set of equations such that $E \supseteq E_r \cup E^c_{\mathcal{L}}$. Let G be the set of normal forms of $\mathcal{L}$ w.r.t. $\vec{E^c_{\mathcal{L}}}$. If G is finite then $\mathcal{S}_E(\mathcal{A})$ is a deterministic automaton such that terms of $\mathcal{L}$ are recognized by no more states than terms in G.*

Proof. First we prove that for all terms $s \in \mathcal{L}$, if we additionally have $s \to^*_{\mathcal{S}_E(\mathcal{A})} q$ then there exists $t \in G$ such that $t \to^*_{\mathcal{S}_E(\mathcal{A})} q$. We show this by induction on the structure of s. If $s \in G$ then it is trivially true. If s is not in normal form w.r.t. $\vec{E^c_{\mathcal{L}}}$ then there exists a subterm t of s such that $s \to_{\vec{E^c_{\mathcal{L}}}} t$. Since t is a subterm of s, there exists a state q' such that $t \to^*_{\mathcal{S}_E(\mathcal{A})} q'$, and since $s =_{E^c_{\mathcal{L}}} t$ it guarantees that $q = q'$ in the simplified automaton $\mathcal{S}_E(\mathcal{A})$. By hypothesis of induction, there exists term $t' \in G$ of such that $t' \to^*_{\mathcal{S}_E(\mathcal{A})} q$. $\mathcal{S}_E(\mathcal{A})$ being deterministic (Lemma 3), for all terms $t \in G$ there is at most one state q such that $t \to^*_{\mathcal{S}_E(\mathcal{A})} q$. Moreover, we have just seen that every state recognizing a term of $\mathcal{L}$ recognizes a term of G. Hence there are no more states in $\mathcal{S}_E(\mathcal{A})$ than terms in G. □

4.3.2 The Role of Transitive Equations

We now show that by adding transitive equations $E_{\mathcal{R}}$ (and reflexive equations E_r), the termination of the TAC algorithm is ensured. In the previous section we showed that thanks to $E^c_{\mathcal{L}}$ each term of $\mathcal{L}$ is recognized by the same state as a term of G. Here we show that thanks to $E_{\mathcal{R}}$ each term of $\mathcal{A}^*$ is recognized by the same state as a term of $\mathcal{L}$. This effectively show that the number of states in $\mathcal{A}^*$ is bounded by $|G|$. This is a proof of the general termination theorem, Theorem 1.

Proof. For every index k and state q of $\mathcal{A}^k$, q satisfies at least one of the following properties:

- $P_0(q)$: There exists a transition $f \to q$ in the automaton $\mathcal{A}^k$, with $f \in \Sigma^0$
- $P_{i+1}(q)$: There exists a transition $f(q_1, \ldots, q_n) \to q$ in the automaton $\mathcal{A}^k$ such that $P_i(q_1) \wedge \cdots \wedge P_i(q_n)$.

If not, then $L(\mathcal{A}, q) = \emptyset$ which contradicts the fact that $\mathcal{A}$ is reduced. We know there are a finite number of states recognizing terms of $\mathcal{L}$ (Lemma 4). Let $\tilde{Q}$ be the set of those states. First we prove that for every symbol f and index k such that $\mathcal{A}^k$ contains a transition $f(q_1, \ldots, q_n) \to q$, with $q_1, \ldots, q_n \in \tilde{Q}$, there exists an index k_f such that for all $k' > k_f$, if $\mathcal{A}^{k'}$ contains $f(q_1, \ldots, q_n) \to q'$, then $q' \in \tilde{Q}$. By hypothesis, for any term s such that $s \to^*_{\mathcal{A}^k} f(q_1, \ldots, q_n) \to_{\mathcal{A}^k} q$, there exists a term $t \in \mathcal{L}$ such that $s \to^l_{\mathcal{R}} t$. After at most l successive completion steps, and because $E \supseteq E_{\mathcal{R}}$, we will have q' merged with a state q_f of $\tilde{Q}$. Moreover, having $E \supseteq E_r$ ensures that for every transition $f(q_1, \ldots, q_n) \to q \in \mathcal{A}^{k'}$ such that $k' \geq k + l$ we have $q = q_f$.

Let $\tilde{k} = max\{\ k_f \mid f \in \Sigma\ \}$. We show, by induction on the property P, that for all $k \geq \tilde{k}$, all states of $\mathcal{A}^k$ are in $\tilde{Q}$. For every q such that $P_0(q)$, there exists a transition $f \to q$ with $f \in \Sigma^0$. Since $k > k_f$, we have shown earlier that $q \in \tilde{Q}$. Assume now that it holds for any state q and index i such that $P_i(q)$. For every q such that $P_{i+1}(q)$, there exists a transition $f(q_1, \ldots, q_n) \to q$ with $P_i(q_1) \wedge \cdots \wedge P_i(q_n)$. By induction hypothesis, $q_1, \ldots, q_n \in \tilde{Q}$. As already shown, this implies $q \in \tilde{Q}$. So, after at most $\tilde{k}$ steps, all states of $\mathcal{A}^k$ are merged into $\tilde{Q}$. Since $\tilde{Q}$ is finite, and $\mathcal{A}^k$ REFD for all k, the algorithm terminates. □

4.4 A Class of Analyzable Programs

The next step is to identify a class of TRS and a language $\mathcal{L}$ for which Theorem 1 applies. One trivial possibility is to directly chose $\mathcal{L} = \mathcal{T}(\Sigma)$ while providing a set of contracting equations $E^c_{\mathcal{T}(\Sigma)}$. In this case, the termination theorem above proves that the completion algorithm terminates on any functional program $\mathcal{R}$. If this works in theory, in practice we want to avoid introducing equations over the application symbol (such as $@(x, y) = y$). Contracting equations on applications makes sense in certain cases, *e.g.*, with idempotent functions ($@(sort, @(sort, x)) = @(sort, x)$), but in most cases, such equations dramatically lower the precision of the completion algorithm. Hence, we want to identify a language $\mathcal{L}$ with no contracting equations over @ in $E^c_{\mathcal{L}}$. Such a language $\mathcal{L}$ still has to have a finite number of normal forms w.r.t. $\vec{E^c_{\mathcal{L}}}$ (required by Theorem 1). As a consequence it cannot include terms containing an unbounded *stack* of applications. For instance, $\mathcal{L}$ cannot contain all the terms of the form $@(f, x), @(f, @(f, x)), @(f, @(f, @(f, x))$, etc. The @ stack must be bounded, even if the applications symbols are interleaved with other symbols (e.g. $@(f, s(@(f, s(@(f, s(x))))))$). To do that:

1. we define a set $\mathcal{B}^d$ of all terms where the size of such stacks of @s is bounded by $d \in \mathbb{N}$;

2. we define a set $\mathcal{K}^n$ (where n depends on the considered TRS) and a class of TRS called $\mathcal{K}$-TRS such that for any TRS $\mathcal{R}$ in this class, $\mathcal{K}^n$ is closed by $\mathcal{R}$ and $\mathcal{K}^n \cap IRR(\mathcal{R}) \subseteq \mathcal{B}^k$ for some k. This is done by first introducing a type system over the terms;

3. finally we define $\mathcal{L} = \mathcal{B}^k \cap IRR(\mathcal{R})$ that can be used to instantiate Theorem 1.

4.4.1 Bounded Applications Stacks

First, if we want to forbid equations over the @ symbol encoding applications, we need to control the maximum size of application stacks in $IRR(\mathcal{R})$. That way

when equations are applied we still have a finite number of equivalence classes (the abstraction is finite). In this perspective, we start by defining the set $\mathcal{B}^d$ of terms where the size of each application stack is bounded by d.

Definition 4.4.1 (Bounded Applications Stacks)**.** *For a given alphabet $\Sigma \cup \{@\}$, for any* depth $d \in \mathbb{N}$*, $\mathcal{B}^d$ is the smallest set defined by:*

$$f(t_1, \ldots, t_n) \in \mathcal{B}^i \Leftarrow f \in \Sigma^n \wedge t_1 \ldots t_n \in \mathcal{B}^i$$
$$@(t_1, t_2) \in \mathcal{B}^{i+1} \Leftarrow t_1, t_2 \in \mathcal{B}^i$$

Note that for any term $t \in \mathcal{B}^i$ implies $t \in \mathcal{B}^{i+1}$. This is because the first rule applies for any constant symbol $f \in \Sigma^0$, for any $i \in \mathbb{N}$.

In Section 4.5.1, we show how to produce a set of contracting equations over $\mathcal{B}^d \cap IRR(\mathcal{R})$, with a finite number of equivalence classes and no equations on @. To be able to instantiate the termination theorem with $\mathcal{L} = \mathcal{B}^d \cap IRR(\mathcal{R})$, we still need to ensure that every term encountered during completion reduces into a term of $\mathcal{B}^d$. More formally, for all completion step k, for all term $t \in L_{\trianglerighteq}(\mathcal{A}^k)$ we need to have a term $s \in \mathcal{B}^d \cap IRR(\mathcal{R})$ s.t. $t \to_{\mathcal{R}}^* s$. However such bound d does not exist in general: we can write programs that will generate unbounded stacks of partial applications (cf. Example 4.2.7). As a consequence, Theorem 1 cannot be directly instantiated with $\mathcal{L} = \mathcal{B}^d \cap IRR(\mathcal{R})$, not for any TRS $\mathcal{R}$. Instead we define (i) a set $\mathcal{K}^n \subseteq \mathcal{T}(\Sigma)$ and function ϕ such that $\mathcal{K}^n \cap IRR(\mathcal{R}) \subseteq \mathcal{B}^{\phi(n)}$ and (ii) a class of TRS, called $\mathcal{K}$-TRS for which $L_{\trianglerighteq}(\mathcal{A}^k) \subseteq \mathcal{K}^n_{\trianglerighteq}$. In $\mathcal{K}$-TRS, the right hand sides of TRS rules are constrained to forbid the construction of unbounded partial applications during rewriting. If the initial automaton $\mathcal{A}^0$ satisfies $L_{\trianglerighteq}(\mathcal{A}^0) \subseteq \mathcal{K}^n_{\trianglerighteq}$ then we can instantiate Theorem 1 with $\mathcal{L} = \mathcal{B}^{\phi(n)} \cap IRR(\mathcal{R})$ and prove termination.

4.4.2 Type System

Our goal is to make sure that when a term of $\mathcal{K}^n$ is reduced with a $\mathcal{K}$-TRS, then the size of partial application stacks is bounded by some $\phi(n)$. To do that we require terms in $\mathcal{K}^n$ and $\mathcal{K}$-TRS to be well-typed so applications are easier to keep track of and control. Our type system definition closely follows the one of [BN98], where partial application are annotated with arrow types, which will allow us to control the construction and reduction of stack of applications.

Definition 4.4.2 (Types)**.** *Let $\mathscr{A}$ be a non-empty set of* base types*. The set of* types *$\mathscr{T}$ is inductively defined as the least set containing $\mathscr{A}$ and all function types:*

$$A \in \mathscr{T} \Leftarrow A \in \mathscr{A}$$
$$\tau_1 \to \tau_2 \in \mathscr{T} \Leftarrow \tau_1, \tau_2 \in \mathscr{T}$$

The function type constructor $\to$ is assumed to be right-associative. The arity *of a type τ is inductively defined on the structure of τ by:*

$$ar(A) = 0 \qquad \Leftarrow A \in \mathscr{A}$$
$$ar(\tau_1 \to \tau_2) = 1 + ar(\tau_2) \qquad \Leftarrow \tau_1 \to \tau_2 \in \mathscr{T}$$

Definition 4.4.3 (Typed Alphabet and Symbol Signature)**.** *Remember that a ranked alphabet defines a set of symbols along with their arity. A* typed alphabet Σ *defines a set of symbols along with their type. We write $f : (\tau_1, \ldots, \tau_n) \to \tau \in \Sigma$ for the symbol f of arity n whose type signature is $(\tau_1, \ldots, \tau_n) \to \tau$. A single symbol f can have multiple signatures in Σ but a single arity.*

Intuitively for a term t of the form $f(t_1, \ldots, t_n)$, τ_i give the type of each subterm t_i while τ is the type of t. For higher-order encoding TRSs, the special symbol @ has multiple signatures of the form $(\tau_1 \to \tau_2, \tau_1) \to \tau_2$ to encode every possible typed application.

Definition 4.4.4 (Type Environment)**.** *For a set of variable $\mathcal{X}$, a type environment $\Gamma : \mathcal{X} \mapsto \mathscr{T}$ associates each variable with its type.*

Definition 4.4.5 (Well typed terms)**.** *Let Σ be a typed alphabet with $\mathcal{X}$ a set of variables and Γ a type environment. We write $\Sigma, \Gamma \vdash t : \tau$ when the term $t \in \mathcal{T}(\Sigma, \mathcal{X})$ can by typed with τ using Σ and Γ. The type judgment $\vdash$ is inductively defined by:*

$$cons\frac{f : (\tau_1, \ldots, \tau_n) \to \tau \in \Sigma \quad \Sigma, \Gamma \vdash t_i : \tau_i}{\Sigma, \Gamma \vdash f(t_1, \ldots, t_n) : \tau} \qquad var\frac{\Gamma(\underline{x}) = \tau}{\Sigma, \Gamma \vdash \underline{x} : \tau}$$

We name $\mathcal{W}_\Gamma(\Sigma, \mathcal{X})$ the set of all well-typed terms defined by

$$\{\ t \mid t \in \mathcal{T}(\Sigma, \mathcal{X}) \ \wedge\ \exists \tau \in \mathscr{T}.\ \Sigma, \Gamma \vdash t : \tau\ \}$$

Note that for higher-order encoding TRSs, the function application is just a special case of the *cons* rule with the symbol @:

$$\text{cons}\frac{@ : (\tau_1 \to \tau_2, \tau_1) \to \tau_2 \in \Sigma \quad \Sigma, \Gamma \vdash t_1 : \tau_1 \to \tau_2 \quad \Sigma, \Gamma \vdash t_2 : \tau_1}{\Sigma, \Gamma \vdash @(t_1, t_2) : \tau_2}$$

To simplify, when there are no ambiguities about which Σ and Γ are considered, we directly write $t : \tau$ instead of $\Sigma, \Gamma \vdash t : \tau$, and $\mathcal{W}(\Sigma, \mathcal{X})$ instead of $\mathcal{W}_\Gamma(\Sigma, \mathcal{X})$. We also write $\mathcal{W}(\Sigma)$ for well-typed closed terms.

Definition 4.4.6 (Typed Rewriting Systems)**.** *A term rewriting system $\mathcal{R}$ is (well) typed w.r.t. a type environment Γ if for all rule $l \to r \in \mathcal{R}$, there exists a single type τ such that $l : \tau$ and $r : \tau$. We then often write $l : \tau \to r : \tau$. A typed* functional *TRS is* complete *iff every well typed term reduces to a value (a term of $\mathcal{T}(\mathcal{C})$).*

In the same way, an equation $s = t$ is well typed if both s and t have the same type. In the rest of this chapter we only consider well typed equations and TRSs.

Types provide information about how a term can be rewritten. For instance we expect every term of the form $@(f : \tau_1 \to \tau_2, t : \tau_1) : \tau_2$ to be rewritten into a value by every *complete* (no partial function) TRS $\mathcal{R}$ when $ar(\tau_1 \to \tau_2) = 1$ and $\tau_2 \in \mathscr{A}$ Furthermore, for certain types, we can guarantee the absence of partial applications in the result of a computation using the type's *order*.

Definition 4.4.7 (Order)**.** *For a given typed alphabet $\Sigma = \mathcal{C} \cup \mathcal{F} \cup \{@\}$, the function $ord : \mathscr{T} \mapsto \mathbb{N}$ gives the* order *of each type. It is inductively defined as the unique function that gives the minimum order for each type such that:*

$$ord(\tau \in \mathscr{A}) = max\{\ max\{\ ord(\tau_1), \ldots, ord(\tau_n)\ \} \mid f : \tau_1 \to \cdots \to \tau_n \to \tau \in \mathcal{C}^n\ \}$$
$$ord(\tau_1 \to \tau_2) = max\{\ ord(\tau_1) + 1, ord(\tau_2)\ \}$$

At first glance, it may seem strange that for $\tau \in \mathscr{A}$, $ord(\tau)$ depends on the constructor symbols. Notice that constructor symbols are used to define the algebraic structure of base types. In particular, it is not unusual in modern functional programming languages to see algebraic data types embedding functions in their constructors. The order of those functions must contribute to the order of the type.

Example 4.4.1. *Consider the two base types list and list′. The type list defines lists of int (corresponding to the* `int list` *type in OCaml) with the constructor:*

$$cons : int \to list \to list \in \mathcal{C}$$

The second type list′ defines lists of functions (corresponding to the `(int -> int) list` *type in OCaml) with the constructor:*

$$cons : (int \to int) \to list' \to list' \in \mathcal{C}$$

The importance of looking at the signatures of cons in the definition of $ord(list)$ and $ord(list')$ is manifest here: In the first case a fully reduced term of type list cannot contain any partial application (no @ symbol in the term) whereas in the second case it can. It is reflected by the order of the types: $ord(list) = max\{ord(int), ord(list)\} = 0$ and $ord(list') = max\{ord(int \to int), ord(list')\} = 1$.

Lemma 5. *If $\mathcal{R}$ is a complete functional TRS and τ a type such that $ord(\tau) = 0$, then all* closed *terms t of type τ are rewritten into an irreducible term with no partial applications:*

$$\forall s \in IRR(\mathcal{R}), \quad t \to_{\mathcal{R}}^{*} s \Rightarrow s \in \mathcal{B}^0.$$

Proof. By induction on the length of the rewriting path. If $t = s$ then we proceed by induction on the structure of s. Since $ord(\tau) = 0$, we can't have $s = @(f, u)$. Otherwise because $\mathcal{R}$ is a functional TRS we would have $s \notin IRR(\mathcal{R})$ which contradict the hypothesis. Thus $s = f$ with $f : \tau' \in \mathcal{F}^0$ and by definition $s \in \mathcal{B}^0$. Now if $t \to_{\mathcal{R}}^{k+1} s$. If $t = @(f, u)$ then we know that there is $v : \tau$ such that $t \to_{\mathcal{R}} v$ since $\mathcal{R}$ is a functional TRS. Then $u \to_{\mathcal{R}}^{k} s$ and by hypothesis of induction, $s \in \mathcal{B}^0$. If $t = f$, as previously we have by definition $s \in \mathcal{B}^0$. □

4.4.3 The $\mathcal{K}$-TRS Class

Recall that we want to define (i) a set $\mathcal{K}^n \subseteq \mathcal{T}(\Sigma)$ and ϕ such that $\mathcal{K}^n_{\trianglerighteq} \cap IRR(\mathcal{R}) \subseteq \mathcal{B}^{\phi(n)}$ and (ii) a class of TRSs, $\mathcal{K}$-TRS, for which $L_{\trianglerighteq}(\mathcal{A}^k) \subseteq \mathcal{K}^n_{\trianglerighteq}$. Assuming that $L_{\trianglerighteq}(\mathcal{A}) \subseteq \mathcal{K}^n_{\trianglerighteq}$ we instantiate Theorem 1 with $\mathcal{L} = \mathcal{K}^n_{\trianglerighteq} \cap IRR(\mathcal{R})$ and prove termination.

Definition 4.4.8 ($\mathcal{K}$-TRS)**.** *Let $\Sigma \cup \{@\}$ be a typed alphabet. A TRS $\mathcal{R}$ is part of $\mathcal{K}$-TRS if it is functional TRS and if for all rules $l \to r \in \mathcal{R}$, $r \in \mathcal{K}$ where $\mathcal{K}$ is inductively defined by:*

$$\underline{x} \in \mathcal{K} \Leftarrow \underline{x} \in \mathcal{X}$$

$$f(t_1, \ldots, t_n) : \tau \in \mathcal{K} \Leftarrow f \in \Sigma^n \wedge t_i \in \mathcal{K} \wedge arity(\tau) = 0$$

$$f(t_1 : \tau_1, \ldots, t_n : \tau_n) : \tau \in \mathcal{K} \Leftarrow f \in \Sigma^n \wedge t_i \in \mathcal{K} \wedge ord(\tau_i) = 0 \tag{4.1}$$

$$@(t_1 : \tau_1 \to \tau_2, t_2 : \tau_1) : \tau_2 \in \mathcal{K} \Leftarrow t_1 \in \mathcal{Z}, t_2 \in \mathcal{K} \wedge arity(\tau_2) = 0 \tag{4.2}$$

$$@(t_1 : \tau_1 \to \tau_2, t_2 : \tau_1) : \tau_2 \in \mathcal{K} \Leftarrow t_1, t_2 \in \mathcal{K} \wedge ord(\tau_1) = 0 \tag{4.3}$$

with $\mathcal{Z}$ defined by:

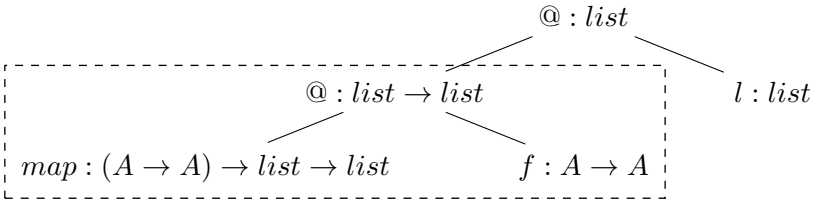
$$t \in \mathcal{Z} \Leftarrow t \in \mathcal{K}$$

$$@(t_1, t_2) \in \mathcal{Z} \Leftarrow t_1 \in \mathcal{Z}, t_2 \in \mathcal{K}$$

By constraining the form of the right hand side of each rule of $\mathcal{R}$, $\mathcal{K}$ defines a set of TRS that cannot construct unbounded partial applications during rewriting.

The definition of $\mathcal{K}$ takes advantage of the type system and Lemma 5. The rules (4.2) and (4.3) ensure that an application $@(t_1, t_2)$ is either: (4.2) a total application, and the whole term can be rewritten; or (4.3) a partial application where t_2 can be rewritten into a term of $\mathcal{B}^0$ (Lemma 5). Rule (4.1) can be seen as the uncurryfied version of (4.3). In (4.2), $\mathcal{Z}$ allows partial applications inside the total application of a multi-parameter function.

Example 4.4.2. *Consider the classical map function over lists of elements of type A, where A is a base type of order 0. A typical call to this function is $@(@(map, f), l)$ of type list, where f is a mapping function, and l a list.*



The whole term belongs to $\mathcal{K}$ because of rule (4.2): list is a base type ($arity(list) = 0$) and its subterm $@(map, f) : list \to list$ framed above, belongs to $\mathcal{Z}$. This subterm is a partial application, but there is no risk of stacking partial applications as it is part of a complete call (to the map function).

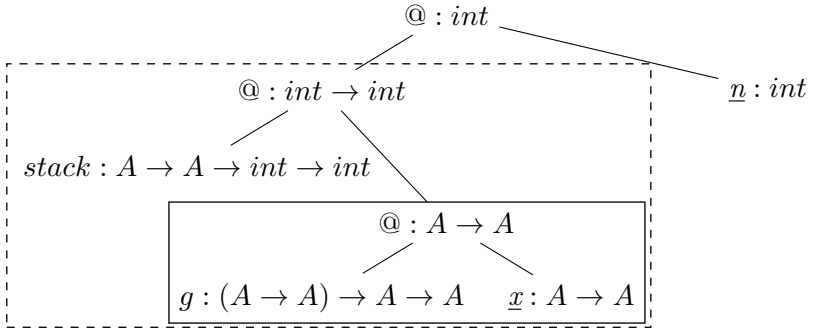
Example 4.4.3. *Consider the higher-order functional TRS $\mathcal{R}$ defining a function stack as follows:*

$$@(@(stack, \underline{x}), 0) \to \underline{x}$$
$$@(@(stack, \underline{x}), s(\underline{n})) \to @(@(stack, @(g, \underline{x})), \underline{n})$$

Here g is a function of type $(A \to A) \to A \to A$. The stack function returns a stack of partial applications whose height is equal to the input parameter:

$$@(@(stack, f), \underbrace{s(s(s \dots s}_{k}(0) \dots))) \to^*_{\mathcal{R}} \underbrace{@(g, @(g, @(g, \dots @(g}_{k}, f) \dots)))$$

The depth of partial applications stacks in the output language is not bounded. With no equations on the @ symbol, the completion algorithm may not terminate. Notice that $\underline{x}$ is a function and $@(g, \underline{x})$ a partial application (represented by a solid frame below). Hence the term $@(stack, @(g, \underline{x}))$ (represented in a dashed frame) is not in $\mathcal{Z}$, which means that $@(@(stack, @(g, \underline{x})), \underline{n})$ is not in $\mathcal{K}$. The TRS $\mathcal{R}$ does not belong to the $\mathcal{K}$-TRS class.



This peculiar way of stacking partial applications may seem useless at first. However this programming pattern frequently appears on higher-order program written in Continuation Passing Style (CPS), where each function is passed a continuation function to be called with the output of the function. In this setting the partial application stack contains the entire program's execution stack. CPS is mostly used as an intermediate representation useful for program analysis.

In the rest of this section we show how we can bound the size of partial application stacks generated by $\mathcal{K}$-TRSs. More precisely, we define a set of terms $\mathcal{K}^n \subseteq \mathcal{T}(\Sigma)$ as $\{\ t\sigma \mid t \in \mathcal{K}, \quad \sigma : \mathcal{X} \mapsto \mathcal{B}^n \cap IRR(\mathcal{R})\ \}$ and claim that if $\mathcal{R}$ is a $\mathcal{K}$-TRS and if $\mathcal{A}$ is a REFD tree automaton with $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{K}^n$ then we can use Theorem 1 to prove that the completion of $\mathcal{A}$ with $\mathcal{R}$ terminates.

Theorem 2. *Let $\mathcal{A}$ be a $\mathcal{K}^n$-coherent REFD tree automaton, $\mathcal{R}$ a $\mathcal{K}$-TRS and E a set of equations. Let $\mathcal{L} = \mathcal{B}^{n+2B} \cap IRR(\mathcal{R})$. If $E = E^r \cup E^c_{\mathcal{L}} \cup E_{\mathcal{R}}$ then the completion of $\mathcal{A}$ by $\mathcal{R}$ and E terminates.*

In this theorem, B is a fixed upper bound on the arity of all the types of the program. The notion of $\mathcal{K}^n$-coherence (cf. Definition 4.4.9) of a tree automaton allows us to ensure that $L_{\trianglerighteq}(\mathcal{A}) \subseteq \mathcal{K}^n$. The idea of the proof developed in this section is the following:

- First we prove that $\mathcal{K}^n_{\trianglerighteq}$ is closed by $\to_{\mathcal{R}}$ (Lemma 10).
- Prove that if $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{K}^n_{\trianglerighteq}$, then it is preserved by completion using the notion of $\mathcal{K}^n$-coherence of $\mathcal{A}$ (Lemma 17).
- Prove that $\mathcal{K}^n_{\trianglerighteq} \cap IRR(\mathcal{R}) \subseteq \mathcal{B}^{n+2B} \cap IRR(\mathcal{R})$ where $B \in \mathbb{N}$ is a fixed upper bound of the arity of all the types of the program (Lemma 19).
- Finally, we use those three properties combined to instantiate Theorem 1 with $\mathcal{L} = \mathcal{B}^{n+2B} \cap IRR(\mathcal{R})$ to prove Theorem 2.

The rest of this section is dedicated to the proof of each of these items, leading to the proof of 2 at the end of the section. First, to prove that $\mathcal{K}^n$ is closed by $\to_{\mathcal{R}}$, we need some intermediate properties over $\mathcal{K}^n$. In particular make sure that, when we apply a rule, the considered substitution gives for each variable a term of $\mathcal{K}^n$: all rules are applied with a substitution of $\mathcal{K}^n$.

Lemma 6. *For all well typed constructor terms $t \in \mathcal{W}(\mathcal{C}, \mathcal{X})$, For all terms $s \in \mathcal{K}^n$, if there exists σ such that $t\sigma = s$, then for all $x \in Var(t)$, $\sigma(x) \in \mathcal{K}^n$.*

Proof. By induction on t.

- $t = x$. If $t\sigma = s$, $s \in \mathcal{K}^n$, then $\sigma(x) = s$ and $\sigma(x) \in \mathcal{K}^n$.
- $t = f(t_1, \ldots, t_n), f \in \mathcal{C}^n$. If $t\sigma = s$, then $s = f(s_1, \ldots, s_n) = f(t_1\sigma, \ldots, t_n\sigma)$. By hypothesis of induction, for all t_i, for all $x \in Var(t_i), \sigma(x) \in \mathcal{K}^n$. Since $Var(t) = \bigcup_{i=1}^n Var(t_i)$, then for all $x \in Var(t), \sigma(x) \in \mathcal{K}^n$.

□

Lemma 7. *For all terms $t \in LHS$ (the left-hand side of a functional TRS, cf. Definition 2.2.16), for all terms $s \in \mathcal{K}^n$, if there exists σ such that $t\sigma = s$, then for all $x \in Var(t)$, $\sigma(x) \in \mathcal{K}^n$.*

Proof. By induction on t.

- $t = f(t_1, \ldots, t_m), f \in \Sigma^m \wedge t_1, \ldots, t_m \in \mathcal{W}(\mathcal{C}, \mathcal{X})$. For all t_i, using Lemma 6 we get for all variables $x \in Var(t_i)$, $\sigma(x) \in \mathcal{K}^n$. Since $Var(t) = \bigcup_{i=1}^{m} Var(t_i)$, then for all $x \in Var(t), \sigma(x) \in \mathcal{K}^n$.
- $t = @(t_1, t_2)$ with $t_1 \in LHS$ and $t_2 \in \mathcal{W}(\mathcal{C}, \mathcal{W})$. Using Lemma 6 we get for all variables $x \in Var(t_2)$, $\sigma(x) \in \mathcal{K}^n$. By induction hypothesis, for all variables $x \in Var(t_1)$, $\sigma(x) \in \mathcal{K}^n$. Since $Var(t) = Var(t_1) \cup Var(t_2)$, then for all variables $x \in Var(t), \sigma(x) \in \mathcal{K}^n$.

□

Lemma 8. *Let $\mathcal{R}$ be a functional TRS. For all rules $l \to r \in \mathcal{R}$, For all terms $t \in \mathcal{K}^n$, if there exists σ such that $l\sigma = t$, then for all $x \in Var(l)$, $\sigma(x) \in \mathcal{K}^n$.*

In other words, each time a rule is used, we know that all variables are substituted with a term of $\mathcal{K}^n$.

Proof. Since $\mathcal{R}$ is a functional TRS, we have $l \in LHS$ (cf. Definition 2.2.16). Using Lemma 7 we have for all $x \in Var(l)$, $\sigma(x) \in \mathcal{K}^n$. □

Lemma 9. *For all $t \in \mathcal{K}^n$, for all rules $l \to r \in \mathcal{R}$ if there exists a position p and a substitution σ with $l\sigma = t|_p$, then $t|_p \in \mathcal{K}^n$, and for all terms $s \in \mathcal{K}^n$, $t[s]_p \in \mathcal{K}^n$.*

Proof. By induction on t.

- $t \in \mathcal{B}^n \cap IRR(\mathcal{R})$. Since $t \in IRR(\mathcal{R})$, there is no p and σ s.t. $l\sigma = t|_p$.
- $t = f(t_1 : \tau_1, \ldots, t_n : \tau_n) : \tau, arity(\tau) = 0$. Two cases:
  - if $p = \lambda$, then $t|_p = t$, and $t \in \mathcal{K}^n$. $t[s]_p = s$, $s \in \mathcal{K}^n$.
  - if $p = i.q$, Since $t_i \in \mathcal{K}^n$, by induction hypothesis we have $t_i|_q \in \mathcal{K}^n$ and $t_i[s]_q \in \mathcal{K}^n$.
    $t|_p = t_i|_q$, so $t|_p \in \mathcal{K}^n$. $t[s]_p = f(t_1, \ldots, t_i[s]_q, \ldots, t_n)$, and since $t_i[s]_q \in \mathcal{K}^n$, $t[s]_p \in \mathcal{K}^n$.
- $t = f(t_1 : \tau_1, \ldots, t_n : \tau_n) : \tau$ with $ord(\tau_1) = 0, \ldots, ord(\tau_n) = 0$. We can use exactly the same reasoning (note that the order of the arity of each type never change).
- $t = @(t_1 : \tau_1, t_2 : \tau_2) : \tau$ with $ord(\tau_2) = 0, t_1, t_2 \in \mathcal{K}$. We can use exactly the same reasoning.
- $t = @(t_1 : \tau_1, t_2 : \tau_2) : \tau$ with $arity(\tau) = 0, t_1 \in Z, t_2 \in \mathcal{K}$. We can use the same reasoning if $p = \lambda$ or $p = 2.q$. If $p = 1.q$ we have $t_1 \in \mathcal{Z}^n$ and $t|_p = t_1|_q$. Let us prove by induction on t_1 that we have $t_1|_q \in \mathcal{K}^n$, and $t_1[s]_q \in \mathcal{Z}^n$.
  - $t_1 \in \mathcal{K}^n$. By induction hypothesis (on t), $t_1|_q \in \mathcal{K}^n$, $t_1[s]_q \in \mathcal{K}^n$, and since $\mathcal{K}^n \subseteq \mathcal{Z}^n$, $t_1[s]_q \in \mathcal{Z}^n$.
  - $t_1 = @(t_1', t_2') : \tau'$.
    - If $q = \lambda$, then since $\mathcal{R}$ is a functional TRS, $arity(\tau') = 0$, which means $t_1 \in \mathcal{K}^n$, and by induction hypothesis (on t), $t_1|_q \in \mathcal{K}^n$ and $t_1[s]_q \in \mathcal{K}^n$. So $t_1[s]_q \subseteq \mathcal{Z}^n$.

* If $q = 1.q'$, then by induction hypothesis on t_1, $t_1|_q \in \mathcal{K}^n$ and $t_1[s]_q \in \mathcal{Z}^n$.
* If $q = 2.q'$, then by induction hypothesis on t, $t_1|_q \in \mathcal{K}^n$ and $t_1[s]_q \in \mathcal{K}^n$. So $t_1[s]_q \subseteq \mathcal{Z}^n$.

$t[s]_p = @(t_1[s]_q, t_2)$, and since $t_1[s]_q \in \mathcal{Z}^n, t_2 \in \mathcal{K}^n$ and $arity(\tau) = 0$, then by definition $t[s]_p : \tau \in \mathcal{K}^n$.

□

Lemma 10 ($\mathcal{K}^n$ is closed by $\rightarrow_{\mathcal{R}}$)**.** *Let assume that for all rules $l \rightarrow r \in \mathcal{R}$ we have $r \in \mathcal{K}$. For all $t \in \mathcal{K}^n$, for all u such that $t \rightarrow_{\mathcal{R}} u$ we have $u \in \mathcal{K}^n$.*

Proof. By induction on t.

- If $t \in \mathcal{B}^n \cap IRR(\mathcal{R})$ then t is irreducible.
- If $t = f(t_1, \ldots, t_n), f \in \Sigma^n$. For all rules $l \rightarrow r \in \mathcal{R}$ and position p such that $l\sigma = t|_p$ and $u = t[r\sigma]_p$. By definition of functional TRS, there is no rule for such term at the root, p cannot be λ. So $p = p'.q$. Let us define $t' = t|p'$, and $u' = t'[r\sigma]_q$. We have $u = t[u']_{p'}$. By induction hypothesis, $u' \in \mathcal{K}^n$, so using Lemma 9 we have $t[u']_{p'} \in \mathcal{K}^n$.
- $t = @(t_1 : \tau \rightarrow \tau', t_2 : \tau) : \tau'$ with $ord(\tau) = 0, t_1, t_2 \in \mathcal{K}^n$. We can use exactly the same reasoning. For all rules $l \rightarrow r \in \mathcal{R}$ and position p such that $l\sigma = t|_p$ and $u = t[r\sigma]_p$. Two cases:
  - if $p = \lambda$, for all $x \in Var(l)$, $t \rhd x\sigma$, which implies $x\sigma \in \mathcal{K}^n$ (Using Lemma 8). Finally, $r\sigma \in \mathcal{K}^n$.
  - if $p = p'.q$, we can use the same reasoning as above.
- $t = @(t_1, t_2) : \tau'$ with $arity(\tau') = 0, t_1 \in \mathcal{Z}^n, t_2 \in \mathcal{K}^n$. For all rules $l \rightarrow r \in \mathcal{R}$ and position p such that $l\sigma = t|_p$ and $u = t[r\sigma]_p$. Two cases:
  - if $p = \lambda$, we can use the same reasoning as above.
  - if $p = 1.q$, using Lemma 9 again we have $t_1|_q \in \mathcal{K}^n$, and by induction hypothesis, $t'_1 = t_1[r\sigma]_q \in \mathcal{K}^n$, thus $@(t'_1, t_2) \in \mathcal{K}^n$.
  - if $p = 2.q$, let us define $t'_2 = t_2[r\sigma]_q$. We have $u = @(t_1, t'_2)$. By induction hypothesis, $t'_2 \in \mathcal{K}^n$, so $@(t_1, t'_2) \in \mathcal{K}^n$.

□

In the same way we can prove that $\mathcal{Z}^n$ is closed by $\rightarrow_{\mathcal{R}}$. To prove that after each step of completion the recognized language stays in $\mathcal{K}^n$, we require the considered automaton to be $\mathcal{K}^n$*-coherent.*

Definition 4.4.9 ($\mathcal{K}^n$-Coherence)**.** *Let $\mathcal{L} \subseteq \mathcal{W}(\Sigma)$ and $n \in \mathbb{N}$. $\mathcal{L}$ is $\mathcal{K}^n$-coherent if*

$$\mathcal{L} \subseteq \mathcal{K}^n \ \vee \ \mathcal{L} \subseteq \mathcal{Z}^n \setminus \mathcal{K}^n$$

Note that $\mathcal{K}^n$-coherence is preserved by inclusion (any subset of $\mathcal{L}$ is also $\mathcal{K}^n$-coherent). By extension we say that a tree-automaton $\mathcal{A} = \langle \Sigma, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$ is $\mathcal{K}^n$-coherent if the language recognized by all states $q \in \mathcal{Q}$ is $\mathcal{K}^n$-coherent.

If $\mathcal{K}^n$-coherence is not preserved during completion, then some states in the completed automaton may recognize terms outside of $\mathcal{K}^n_{\trianglerighteq}$. Our goal is to show that it is preserved by $\mathcal{C}_{\mathcal{R}}(\cdot)$ (Lemma 15) then by $\mathcal{S}_E(\cdot)$ (Lemma 16).

Lemma 11 (Normalization preserves $\mathcal{K}^n$-coherence)**.** *For all $k \in \mathbb{N}, k > 0$, for all REFD $\mathcal{A} = \langle \Sigma, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$, $\mathcal{A}' = \langle \Sigma, \mathcal{Q} \cup \mathcal{Q}', \mathcal{Q}_f, \Delta \cup \Delta' \rangle$ such that $\langle \mathcal{Q} \cup \mathcal{Q}', \Delta \cup \Delta', q \rangle = Norm(\mathcal{Q}, \Delta, t)$ with $t \in \mathcal{T}(\Sigma, \mathcal{Q})$. If $\{t\sigma \mid \sigma \in \mathcal{Q} \mapsto \mathcal{T}(\Sigma)\}$ is $\mathcal{K}^n$-coherent and $\mathcal{A}$ is $\mathcal{K}^n$-coherent, then $\mathcal{A}'$ is $\mathcal{K}^n$-coherent.*

Proof. We need to check that for each new state $q \in \mathcal{Q}'$, $\mathcal{L}(\mathcal{A}', q)$ is $\mathcal{K}^n$-coherent and that for each old state $q \in \mathcal{Q}$, $\mathcal{L}(\mathcal{A}', q)$ is *still* $\mathcal{K}^n$-coherent.

The later one is easy to prove since by definition of $Norm$, no transitions of the form $u \to q$ are created in Δ' with $q \in \mathcal{Q}$. For each old state $q \in \mathcal{Q}$, $\mathcal{L}(\mathcal{A}', q)$ is unchanged and remains $\mathcal{K}^n$-coherent. As for $q \in \mathcal{Q}'$, by definition of $Norm$ each such new state recognizes a subset of $\{t\sigma \mid \sigma \in \mathcal{Q} \mapsto \mathcal{T}(\Sigma)\}$. By hypothesis, this set is $\mathcal{K}^n$-coherent. Since $\mathcal{K}^n$-coherence is closed by inclusion, $\mathcal{L}(\mathcal{A}', q)$ is $\mathcal{K}^n$-coherent. □

Lemma 12. *For all context C and terms $s : \tau, t : \tau$ such that $s, t \in \mathcal{Z}^n \setminus \mathcal{K}^n$ or $s, t \in \mathcal{K}^n$, $C[s] \in \mathcal{K}^n \iff C[t] \in \mathcal{K}^n$.*

Proof. This can be done by induction on the context C by seeing in the definition of K that we can swap a subterm of a $\mathcal{K}^n$ term as long as the type of the subterm is the same, and that a $\mathcal{K}^n$ term is not replaced by a $\mathcal{Z}^n$ term. □

Lemma 13 (Linking two states together preserves $\mathcal{K}^n$-coherence)**.** *Let consider the tree automata $\mathcal{A} = \langle \Sigma, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$ and $\mathcal{A}' = \langle \Sigma, \mathcal{Q}, \mathcal{Q}_f, \Delta' \rangle$ such that $\Delta' = \Delta \cup \{q_a \to q_b\}$ and $\mathcal{L}(\mathcal{A}, q_b) \subseteq \mathcal{K}^n \iff \mathcal{L}(\mathcal{A}, q_a) \subseteq \mathcal{K}^n$, with both languages sharing the same type τ. If $\mathcal{A}$ is $\mathcal{K}^n$-coherent, then $\mathcal{A}'$ is $\mathcal{K}^n$-coherent.*

Proof. For all state $q \in \mathcal{Q}$, for all term $s, t \in \mathcal{K}^n_{\trianglerighteq}$ such that we have $C[t] \to^*_\Delta C[q_a] \to_{\Delta'} C[q_b] \to^*_\Delta q$ with $C[s] \to^*_\Delta C[q_b] \to^*_\Delta q$. $C[t]$ is recognized in q with only one transition $q_a \to q_b$. Let us prove that $\mathcal{L}(\mathcal{A}, q) \in \mathcal{K}^n \iff C[t] \in \mathcal{K}^n$. If $s \in \mathcal{K}^n$ (or $t \in \mathcal{K}^n$), then since $\mathcal{A}$ is $\mathcal{K}^n$-coherent, $\mathcal{L}(\mathcal{A}, q_b) \subseteq \mathcal{K}^n$ (or $\mathcal{L}(\mathcal{A}, q_a) \subseteq \mathcal{K}^n$). Then since by hypothesis $\mathcal{L}(\mathcal{A}, q_b) \subseteq \mathcal{K}^n \iff \mathcal{L}(\mathcal{A}, q_a) \subseteq \mathcal{K}^n$, $s \in \mathcal{K}^n$ *and* $t \in \mathcal{K}^n$. Using Lemma 12 we have that $C[s] \in \mathcal{K}^n \iff C[t] \in \mathcal{K}^n$. If $\mathcal{L}(\mathcal{A}, q) \in \mathcal{K}^n$ then $C[s] \in \mathcal{K}^n$ and then $C[t] \in \mathcal{K}^n$. We can use the exact same reasoning if $s, t \in \mathcal{Z}^n \setminus \mathcal{K}^n$. We can generalize this reasoning to any number of $q_a \to q_b$ transitions in the derivation paths and see that $\mathcal{L}(\mathcal{A}', q)$ remains $\mathcal{K}^n$-coherent, for any state q. $\mathcal{A}'$ is $\mathcal{K}^n$ coherent. □

Lemma 14 (Solving a critical-pair preserves $\mathcal{K}^n$-coherence)**.** *Let $\mathcal{A} = \langle \Sigma, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$ a REFD automaton that contains the critical pair $\langle l \to r, q, \sigma \rangle$ and $\mathcal{A}' = \langle \Sigma, \mathcal{Q} \cup \mathcal{Q}', \mathcal{Q}_f, \Delta \cup \Delta' \cup \{q' \to q\} \rangle$ where $\langle \mathcal{Q} \cup \mathcal{Q}', \Delta \cup \Delta', q' \rangle = Norm(\mathcal{Q}, \Delta, r\sigma)$. If $\mathcal{A}$ is $\mathcal{K}^n$-coherent, then $\mathcal{A}'$ is $\mathcal{K}^n$-coherent.*

Proof. Let $\mathcal{A}$ be $\mathcal{K}^n$-coherent. Let $\mathcal{A}''$ the tree automaton $\langle \Sigma, \mathcal{Q} \cup \mathcal{Q}', \mathcal{Q}_f, \Delta \cup \Delta'' \rangle$ result of the normalization. By Lemma 11 we know that $\mathcal{A}''$ is $\mathcal{K}^n$-coherent. By definition of a critical pair we have $l\sigma \to^*_\Delta q$ and $l\sigma \to_{\mathcal{R}} r\sigma$. Since $\mathcal{K}^n$ is closed by $\to_{\mathcal{R}}$ we have $l\sigma \in \mathcal{K}^n \Rightarrow r\sigma \in \mathcal{K}^n$. Because $\mathcal{A}$ is $\mathcal{K}^n$-coherent, $l\sigma \in \mathcal{K}^n \iff \mathcal{L}(\mathcal{A}, q) \subseteq \mathcal{K}^n$, and $\mathcal{L}(\mathcal{A}, q) \subseteq \mathcal{K}^n \Rightarrow r\sigma \in \mathcal{K}^n \Rightarrow \mathcal{L}(\mathcal{A}, q') \subseteq \mathcal{K}^n$. We can do the opposite demonstration do deduce that $\mathcal{L}(\mathcal{A}, q') \in \mathcal{K}^n \Rightarrow \mathcal{L}(\mathcal{A}, q) \in \mathcal{K}^n$ by first showing that $\mathcal{Z}^n \setminus \mathcal{K}^n$ is closed by rewriting. So we have $\mathcal{L}(\mathcal{A}, q) \iff \mathcal{L}(\mathcal{A}, q')$. Since typed TRSs preserves type by rewriting we know that $\mathcal{L}(\mathcal{A}, q)$ and $\mathcal{L}(\mathcal{A}, q')$ share the same type. Using Lemma 13 we deduce that adding the transition $q \to q'$ preserve $\mathcal{K}^n$-coherence, so $\mathcal{A}'$ is $\mathcal{K}^n$-coherent. □

Lemma 15 ($\mathcal{C}_\mathcal{R}(\mathcal{A})$ preserves $\mathcal{K}^n$-coherence)**.** *Let $\mathcal{A}$ be a REFD tree automaton. If $\mathcal{A}$ is $\mathcal{K}^n$-coherent, then $\mathcal{C}_\mathcal{R}(\mathcal{A})$ is $\mathcal{K}^n$-coherent.*

Proof. Let $\mathcal{A} = \langle \Sigma, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$. By definition, $\mathcal{C}_\mathcal{R}(\mathcal{A}) = \langle \Sigma, \mathcal{Q}', \mathcal{Q}_f, \Delta' \rangle$ where $\langle \mathcal{Q}', \Delta' \rangle = Join(\mathcal{Q}, \Delta, \mathcal{CP}(\mathcal{R}, \mathcal{A}))$. Let us proceed by induction on the structure of $\mathcal{CP}(\mathcal{R}, \mathcal{A})$.

- $\mathcal{CP}(\mathcal{R}, \mathcal{A}) = \emptyset$. Then $\mathcal{C}_\mathcal{R}(\mathcal{A}) = \mathcal{A}$, which is $\mathcal{K}^n$-coherent.
- $\mathcal{CP}(\mathcal{R}, \mathcal{A}) = \{ \langle l \to r, q, \sigma \rangle \} \cup S$. $Join(\mathcal{Q}, \Delta, \mathcal{CP}(\mathcal{R}, \mathcal{A})) = Join^S(\mathcal{Q}'', \Delta'' \cup \{q \to q'\}, S)$ with $\langle \mathcal{Q}'', \Delta'', q' \rangle = Norm(\mathcal{Q}, \Delta, r\sigma)$. Let $\mathcal{A}' = \langle \Sigma, \mathcal{Q}'', \mathcal{Q}_f, \Delta'' \cup \{q \to q'\} \rangle$. By Lemma 14, $\mathcal{A}'$ is $\mathcal{K}^n$-coherent. By definition $\mathcal{CP}(\mathcal{R}, \mathcal{A}') = S$. By hypothesis of induction $\mathcal{C}_\mathcal{R}(\mathcal{A}')$ is $\mathcal{K}^n$-coherent. Also note that $Join(\mathcal{Q}'', \Delta'' \cup \{q \to q'\}, \mathcal{CP}(\mathcal{R}, \mathcal{A}')) = Join(\mathcal{Q}'', \Delta'' \cup \{q \to q'\}, S)$ so $\mathcal{C}_\mathcal{R}(\mathcal{A}) = \mathcal{C}_\mathcal{R}(\mathcal{A}')$. Thus $\mathcal{C}_\mathcal{R}(\mathcal{A})$ is $\mathcal{K}^n$-coherent.

□

Lemma 16 ($\mathcal{S}_E(\mathcal{A})$ preserves $\mathcal{K}^n$-coherence)**.** *Let $\mathcal{A}, \mathcal{A}'$ two REFD tree automata, $\mathcal{R}$ a $\mathcal{K}$-TRS and E a set of equations such that $E = E^r \cup E_n^c \cup E_\mathcal{R}$. If $\mathcal{A} \rightsquigarrow_E \mathcal{A}'$ and $\mathcal{A}$ is $\mathcal{K}^n$-coherent then $\mathcal{A}'$ is $\mathcal{K}^n$-coherent and $\mathcal{S}_E(\mathcal{A})$ is $\mathcal{K}^n$-coherent.*

Proof. Let us name q_a and q_b the two states merged from $\mathcal{A}$ to $\mathcal{A}'$. The proof is based on the idea that merging those two states is equivalent to adding the transitions $q_a \to q_b$ and $q_b \to q_a$. Let $\mathcal{A}''$ be the tree automaton defined from $\mathcal{A}$ by $\langle \Sigma, \mathcal{Q}, \mathcal{Q}_f, \Delta \cup \{q_a \to q_b, q_b \to q_a\} \rangle$. Note that for all states $q \in \mathcal{Q} \setminus \{q_a\}$, $L(\mathcal{A}', q) = L(\mathcal{A}'', q)$. Let us show that $\mathcal{A}''$ is $\mathcal{K}^n$-coherent instead of $\mathcal{A}'$. First we show that $L(\mathcal{A}, q_a) \subseteq \mathcal{K}^n \iff L(\mathcal{A}, q_b) \subseteq \mathcal{K}^n$. Since q_a and q_b are being merged, it means that there exists two terms s and t such that $s \to_\Delta^* q_a$, $t \to_\Delta^* q_b$ and $s =_E t$. Here we have three cases.

1. $s =_{E^r} t$, then $s = t$ and $s \in \mathcal{K}^n \iff t \in \mathcal{K}^n$.
2. $s =_{E_n^c}$, then $t = s|_p$ for some p meaning that there is an equation $u = u|_p$ and a substitution σ such that $u\sigma = s$ and $u|_p\sigma = t$. Recall that we restrain ourselves to well-typed equations where $u \in \mathcal{W}(\mathcal{C}, \mathcal{X})$. Then using the definition of $\mathcal{K}^n$ we have $s \in \mathcal{K}^n \iff s|_p \in \mathcal{K}^n$.
3. $s =_{E_\mathcal{R}} t$, then $s \to^\mathcal{R} t$ and using Lemma 10 (and its equivalent with $\mathcal{Z}^n$) we have $s \in \mathcal{K}^n \iff t \in \mathcal{K}^n$.

So we have $s \in \mathcal{K}^n \iff t \in \mathcal{K}^n$. Now since $\mathcal{A}$ is $\mathcal{K}^n$-coherent, $L(\mathcal{A}, q_a) \subseteq \mathcal{K}^n \iff L(\mathcal{A}, q_b) \subseteq \mathcal{K}^n$. By Lemma 13 we have $\mathcal{A}''$ $\mathcal{K}^n$-coherent. By extension since for all states $q \in \mathcal{Q} \setminus \{q_a\}$, $L(\mathcal{A}', q) = L(\mathcal{A}'', q)$, $\mathcal{A}'$ is $\mathcal{K}^n$-coherent and $\mathcal{S}_E(\mathcal{A})$ is $\mathcal{K}^n$-coherent. □

By using Lemma 15 and Lemma 16, we can prove that the completion algorithm, which is a composition of $C_\mathcal{R}(\mathcal{A})$ and $\mathcal{S}_E(\mathcal{A})$, preserves $\mathcal{K}^n$-coherence.

Lemma 17 (Completion preserves $\mathcal{K}^n$-coherence)**.** *Let $\mathcal{A}^0 = \langle \Sigma, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$ an initial REFD tree automaton for the completion algorithm, $\mathcal{R}$ a f$\mathcal{K}$-TRS and E a set of equations. If $E = E^r \cup E_\mathcal{L}^c \cup E_\mathcal{R}$ with $\mathcal{L} = \mathcal{B}^{n+2B} \cap IRR(\mathcal{R})$ and $\mathcal{A}^0$ is $\mathcal{K}^n$-coherent then for all $k \in \mathbb{N}$, $\mathcal{A}^k$ is $\mathcal{K}^n$-coherent. In particular, $\mathcal{A}^*$ is $\mathcal{K}^n$-coherent.*

Proof. By induction on k. If $k = 0$, by hypothesis $\mathcal{A}_0$ is $\mathcal{K}^n$-coherent. If $k = i + 1$, by definition $\mathcal{A}_{i+1} = \mathcal{S}_E(C_\mathcal{R}(\mathcal{A}_i))$. By hypothesis of induction $\mathcal{A}_i$ is $\mathcal{K}^n$-coherent. We then apply Lemma 15 to show that $C_\mathcal{R}(\mathcal{A}_i)$ is $\mathcal{K}^n$-coherent, and Lemma 16 to show that $\mathcal{S}_E(C_\mathcal{R}(\mathcal{A}_i))$ is $\mathcal{K}^n$-coherent. We conclude that if $\mathcal{A}^*$ exists, it is $\mathcal{K}^n$-coherent. □

By construction we can prove that the depth of irreducible $\mathcal{K}^n_\trianglerighteq$ terms is bounded, which correspond to the following lemmas.

Lemma 18. *Let $B \in \mathbb{N}$ be the maximum function arity in the program. For all $t : \tau \in K^n$, $t : \tau \in IRR(\mathcal{R}) \Rightarrow t : \tau \in \mathcal{B}^{n+B-arity(\tau)}$.*

Proof. By induction on $t : \tau$.

- $t : \tau \in \mathcal{B}^n$. Since B is the maximum program arity, $arity(\tau) \leq B$, so $n \leq n + B - arity(\tau)$. Then $t : \tau \in \mathcal{B}^{n+B-arity(\tau)}$.
- $t = f(t_1 : \tau_1, \ldots, t_n : \tau_n) : \tau, arity(\tau) = 0$. Since $\mathcal{R}$ is a functional TRS, if $t \in IRR(\mathcal{R})$ and $arity(\tau) = 0$ then $f \in \mathcal{C}^n$. By induction hypothesis, for all $t_i, t_i \in \mathcal{B}^{n+B-arity(T_i)}$, with $\mathcal{B}^{n+B-arity(T_i)} \subseteq \mathcal{B}^{n+B}$. This implies $t_1, \ldots, t_n \in \mathcal{B}^{n+B}$, so by definition, $t \in \mathcal{B}^{n+B}$. Since $arity(\tau) = 0$, $t \in \mathcal{B}^{n+B-arity(\tau)}$.
- $t = f(t_1 : \tau_1, \ldots, t_n : \tau_n) : \tau$ with $order(\tau_1) = 0, \ldots, order(\tau_n) = 0$. If $arity(\tau) = 0$ then we can use the same reasoning. Otherwise, for all t_i since $order(T_i) = 0$ we have $t_i \in \mathcal{B}^0$, so $t \in \mathcal{B}^1$. Since $n > 0$, then $B > 0, B > arity(\tau)$ and $1 \leq n + B - arity(\tau)$. So by definition, $t \in \mathcal{B}^{n+B-arity(\tau)}$.
- $t = @(t_1 : \tau_1, t_2 : \tau_2)$. Since $\mathcal{R}$ is a functional TRS, if $t \in IRR(\mathcal{R})$ then we can't have $arity(\tau) = 0$, so by definition of K^n, $order(\tau_2) = 0$ and $t_1, t_2 \in K^n$. Moreover, since $t_2 \in IRR(\mathcal{R})$, $t_2 \in \mathcal{B}^0$, $t_2 \in \mathcal{W}(\mathcal{C})$. By induction hypothesis, since $t_1 \in IRR(\mathcal{R})$ we have $t_1 \in \mathcal{B}^{n+B-arity(\tau_2)}$. By definition, $t \in \mathcal{B}^{n+B-arity(\tau_2)+1}$, and since $arity(\tau_2) = arity(\tau) - 1$, $t \in \mathcal{B}^{n+B-arity(\tau)}$.

□

Lemma 19. *For all $t : \tau \in \mathcal{K}^n_\trianglerighteq$, $t : \tau \in IRR(\mathcal{R}) \Rightarrow t : \tau \in \mathcal{B}^{n+2B-arity(\tau)}$.*

Proof. Note that if $t \in \mathcal{K}^n_\trianglerighteq$ then $t \in \mathcal{Z}^n$. We reason by induction on $t : \tau$.

- $t : \tau \in \mathcal{K}^n$, then by Lemma 18 we have $t : \tau \in \mathcal{B}^{n+B-arity(\tau)}$, thus $t : \tau \in \mathcal{B}^{n+2B-arity(\tau)}$.
- $t = @(t_1 : \tau_1, t_2 : \tau_2) : \tau$ with $t_1 \in \mathcal{Z}^n, t_2 \in \mathcal{K}^n$. By Lemma 18 we have $t_2 : \tau_2 \in \mathcal{B}^{n+B}$. By induction hypothesis we have $t_1 : \tau_1 \in \mathcal{B}^{n+2B-arity(\tau_2)}$. Since $arity(\tau_2) = arity(\tau) + 1$, $t_1 : \tau_1 \in \mathcal{B}^{n+2B-arity(\tau)-1}$.

  No that know the depth of t_1 and t_2, we can deduce the depth of t by taking the maximum depth of t_1 and t_2, which gives us $t : \tau \in \mathcal{B}^{max(n+2B-arity(\tau),n+B+1)}$.

  However $arity(\tau_2) \leq B$ implies $arity(\tau) < B$ and then $B - arity(\tau) - 1 \geq 0$. Thus $n + B + 1 \leq n + 2B - arity(\tau)$. Finally $t : \tau \in \mathcal{B}^{n+2B-arity(\tau)}$.

□

Reminder of Theorem 2. *Let $\mathcal{A}$ be a $\mathcal{K}^n$-coherent REFD tree automaton, $\mathcal{R}$ a $\mathcal{K}$-TRS and E a set of equations. Let $\mathcal{L} = \mathcal{B}^{n+2B} \cap IRR(\mathcal{R})$. If $E = E^r \cup E^c_\mathcal{L} \cup E_\mathcal{R}$ then the completion of $\mathcal{A}$ by $\mathcal{R}$ and E terminates.*

Proof. According to Lemma 17, for all $k \in \mathbb{N}$, the completed automaton $\mathcal{A}^k$ is $\mathcal{K}^n$-coherent. By definition this implies that $\mathcal{L}_{\trianglerighteq}(\mathcal{A}^k) \subseteq \mathcal{K}^n_{\trianglerighteq}$. Moreover, we know that $IRR(\mathcal{R}) \cap \mathcal{K}^n_{\trianglerighteq} \subseteq \mathcal{B}^{n+2B}$ (Lemma 19). Let $\mathcal{L} = \mathcal{B}^{n+2B} \cap IRR(\mathcal{R})$. $\mathcal{R}$ is terminating, so for every term $s \in \mathcal{L}_{\trianglerighteq}(\mathcal{A}^k)$ there exists $t \in \mathcal{L}$ such that $s \to^*_{\mathcal{R}} t$. Since the number of normal form of $\mathcal{L}$ is finite w.r.t. $\vec{E}$, Theorem 1 implies that the completion of $\mathcal{A}$ by $\mathcal{R}$ and E terminates. □

4.5 Verification Procedure

In the previous section we have identified a class of TRS, $\mathcal{K}$-TRS, for which contracting equations over the values can be used while preserving termination of the TAC algorithm. We now want to automatically generate those contracting equations to abstract the program execution using the TAC algorithm and solve a given regular safety problem (cf. Definition 2.4.7).

4.5.1 Contracting Equations Generation

Theorem 2 states a number of hypotheses that must be satisfied in order to guarantee termination of the TAC algorithm:

- The initial automaton $\mathcal{A}$ must be REFD and $\mathcal{K}^n$-coherent for some n.
- $\mathcal{R}$ is a $\mathcal{K}$-TRS. This is a straightforward syntactic check. If it is not verified, we can reject the TRS before starting the completion.
- $\mathcal{R}$ must be terminating.
- The set of equations E must be of the form $E^c_{\mathcal{L}} \cup E^r \cup E_{\mathcal{R}}$ with $\mathcal{L} = \mathcal{B}^{n+2B}$. That is what we want to automatically generate here. The equation sets E^r and $E_{\mathcal{R}}$ are determined directly from the syntactic structure of $\mathcal{R}$ (cf. Definitions 4.2.10 and 4.2.9). However, there is no unique suitable set of contracting equations $E^c_{\mathcal{L}}$. This set must be generated carefully in order to prove or disprove a given property.

In this section, we describe a method for generating all possible sets of contracting equations $E^c_{\mathcal{L}}$. Following the previous section, $\mathcal{L}$ should be replaced by $\mathcal{B}^{n+2B}$ (cf. Theorem 2). However to simplify we only present the case where $\mathcal{L} = \mathcal{W}(\mathcal{C})$ (with $IRR(\mathcal{R}) \subseteq \mathcal{W}(\mathcal{C})$, *i.e.*, all results are first-order terms). We generate the contracting equations iteratively, as a series of equation sets $\mathbb{E}^c_k$ where the equations only equate terms of depth at most k.

Recall that according to Definition 4.2.8, a contracting equation is of the form $u = u|_p$ with $p \neq \lambda$, *i.e.*, it equates a term with one of its strict subterm (of the same type). A set of contracting equations over the set $\mathcal{W}(\mathcal{C})$ is then generated as follows: (i) generate the set of left-hand side of equations as a *covering set of patterns* [Kou92], so that for each term $t \in \mathcal{W}(\mathcal{C})$ there exists a left-hand side u of an equation and a substitution σ such that $t = u\sigma$. (ii) for each left-hand side, generate all possible equations of the form $u = u|_p$, satisfying that both sides have the same type. (iii) from all those equations, we build all possible $E^c_{\mathcal{L}}$ (with $\mathcal{L} = \mathcal{W}(\mathcal{C})$) such that the set of normal forms of $\mathcal{W}(\mathcal{C})$ w.r.t. $\vec{E^c_{\mathcal{L}}}$ is finite. Since $\vec{E^c_{\mathcal{L}}}$ is left-linear and $\mathcal{L} = \mathcal{W}(\mathcal{C})$, this can be decided efficiently [Com00].

Example 4.5.1. *Assume that* $\mathcal{C} = \{\ 0 : 0, s : 1\ \}$. *For* $k = 0$, $\mathbb{E}_c^0 = \emptyset$ *because there is no covering set of terms with depth* 0. *Thus, there is no set* $E_{\mathcal{L}}^c$ *satisfying Definition 4.2.8 for* $k = 0$. *For* $k = 1$, *the covering set is* $\{s(x), 0\}$ *and* $\mathbb{E}_c^1 = \{\{s(x) = x\}\}$. *For depth* 2, *the covering set is* $\{s(s(x)), s(0), 0\}$ *and* $\mathbb{E}_c^2 = \mathbb{E}_c^1 \cup \{\{s(s(x)) = x\}, \{s(s(x)) = s(x)\}, \{s(0) = 0\}, \{s(0) = 0, s(s(x)) = x\}, \{s(0) = 0, s(s(x)) = s(x)\}\}$. *All equation sets of* $\mathbb{E}_c^1$ *and* $\mathbb{E}_c^2$ *satisfy Definition 4.2.8 and lead to different approximations.*

To find every left-hand side we use the notion of *covering set of patterns* inspired by Kounalis [Kou92].

Definition 4.5.1 (Covering Patterns)**.** *A set of patterns* $P \subseteq \mathcal{W}(\Sigma, \mathcal{X})$ *is covering for* $t \in \mathcal{W}(\mathcal{C}, \mathcal{X})$ *if for all substitution* σ, *there exists a term* $s \in P$ *and a substitution* σ' *such that* $t\sigma = s\sigma'$. *Similarly,* P *is covering for a type* τ *if it is covering for the set* $\{t \mid t : \tau\}$. *We write* $P_k(\tau)$ *the set of patterns of depth (at most)* k *covering the type* τ. *It is inductively defined by:*

$$P_0(\tau) = \{\ \{\ \underline{x}\ \}\ \}$$
$$P_{k+1}(\tau) = \{\ f(t_1, \ldots, t_n) \mid f : (\tau_1, \ldots, \tau_n) \to \tau \wedge t_i \in P_k(\tau_i)\ \}$$

Note that in $P_k(\tau)$, *each term of* $\{t \mid t : \tau\}$ *is covered by exactly one pattern.*

From a pattern u of type τ we can extract a set of possible contracting equations, $C(u : \tau)$ defined as:

$$C(u : \tau) = \begin{cases} \emptyset & \text{if } u \text{ is closed} \\ \{\ u = u|_p \mid u : \tau \wedge u|_p : \tau\ \} & \text{otherwise} \end{cases}$$

For any $k > 0$, the set of depth-k contracting equation set $\mathbb{E}_k^c$ is the smallest set such that:

$$E^c \in \mathbb{E}_k^c \quad \Leftarrow \quad \forall \tau.\ \forall u \in P_k(\tau).\ \exists!\ u = u|_p \in C(u : \tau) \text{ s.t. } u = u|_p \in E^c$$

By construction, for each contracting set E^c in $\mathbb{E}_k^c$, $\vec{E^c}$ is deterministic, complete over $\mathcal{W}(\mathcal{C})$ (it can rewrite any term of $\mathcal{W}(\mathcal{C})$) with a finite number of normal forms. In each E^c, the set $P_k(\tau)$ control the number of normal forms for each type τ while the normal form attributed to each term u depends on what equation have been picked from $C(u : \tau)$. In the following, we demonstrate that by choosing the right k and by picking the right equations in $C(u : \tau)$, any regular abstraction of $\mathcal{W}(\mathcal{C})$ can be built. In other words, for any regular abstraction of $\mathcal{W}(\mathcal{C})$, there exists k such that some equation set in $\mathbb{E}_k^c$ produces this abstraction (Lemma 21).

Note that a regular abstraction of $\mathcal{W}(\mathcal{C})$ can be represented by a tree automaton where each state recognizes one element (equivalence class) of the abstraction. We first show that for any of such automaton, it is possible to find k such that each state of type τ is covered by different patterns in $P_k(\tau)$.

Lemma 20. *Let* $\mathcal{A}$ *be a deterministic and reduced tree automaton recognizing terms of* $\mathcal{W}(\mathcal{C})$. *Let us assume that* $\mathcal{A}$ *is type-coherent, meaning that for all state* q *there exists a unique base type* $\tau \in \mathscr{A}$ *such that,* $\mathcal{L}(\mathcal{A}, q) \subseteq \{t \mid t : \tau\}$ *which we write* $q : \tau$. *Let us write* $Q_\tau = \{q \mid q : \tau\}$. *Then there exists* $k \in \mathbb{N}$ *for which for each base type* $\tau \in \mathscr{A}$, *there exists a partition* P *of* $P_k(\tau)$ *such that each state* $q \in \mathcal{Q}_\tau$ *is covered by a different element of* P. *In other words, each pattern of* $P_k(\tau)$ *cover (part of) a unique state of* $\mathcal{A}$.

Proof. We can show this by contradiction. If no such k exists, it means that for all $k \in \mathbb{N}$, for all partition P of $P_k(\tau)$ there is always two states q and q' covered by the same element of P. This means that there is a pattern $p \in P$ and two substitutions σ, σ' such that $p\sigma \rightarrow^*_{\mathcal{A}} q$ and $p\sigma' \rightarrow^*_{\mathcal{A}} q'$. Since this is true for all k, it must be because the same term t is recognized by both states. However this is not possible because $\mathcal{A}$ is deterministic. □

Lemma 21 (Any regular abstraction is in $\mathbb{E}^c_k$)**.** *For any regular abstraction of $\mathcal{W}(\mathcal{C})$, there exists k such that some equation set in $\mathbb{E}^c_k$ produces this abstraction.*

Proof. A regular abstraction can be represented with a tree automaton $\mathcal{A}$ where each equivalence class (abstraction) is represented by a state. We already showed with Lemma 20 that there exists k for which $P_k(\cdot)$ generates a covering set that covers each state of the $\mathcal{A}$ with different patterns. Let us name $P_k(q)$ the set of patterns covering state q. We now need to prove that there exists some $k' \geq k$ for which there exists $E^c \in \mathbb{E}^{k'}_c$ where each state q are represented by different normal forms in $\vec{E^c}$. We show that there exists $k' \geq k$ with the following properties:

- For each recursive state q, with k' we have in addition that each covering pattern u in $P_{k'}(q)$ uses recursion: there exists some position p such that for all σ, $u|_p\sigma$ is also recognized by q. We write $Rec_q(u)$ (one of) such position for a given covering pattern u.

- For each non-recursive state q, with k' we have in addition that for each covering pattern u in $P_{k'}(q)$ if there exists a substitution σ, a context C and recursive state q' such that $u\sigma \rightarrow^*_{\mathcal{A}} C[q'] \rightarrow^*_{\mathcal{A}} q$ then there exists some position p such that for all σ, $u|_p\sigma$ is also recognized by q'. We write $Rec_q(u) = p$ (one of) such position for a given covering pattern u.

Using the contracting equations $E^c = \{\ u = u|_{Rec_q(u)} \mid q \in \mathcal{Q} \wedge u \in P_{k'}(q)\ \}$, we ensure that every term recognized by q are represented by the same term, unique to q. Note that for each state q, for each $u \in P_{k'}(q)$ we have $u \in P_{k'}(\tau)$ with $q : \tau$ and $u = u|_{Rec_q(u)} \in C(u : \tau)$. By definition this means that we have $E^c \in \mathbb{E}^{k'}_c$. Note that there are potentially multiple solutions for E^c since there are potentially multiple candidates for each $Rec_q(u)$. The generated E^c gives in general a more precise abstraction compared to the one represented by $\mathcal{A}$. Two terms represented by the same state may rewrite to two different normal forms w.r.t. $\vec{E^c}$. □

4.5.2 Equations Exploration

In the following, we define a simple verification procedure that uses the Tree Automata Completion algorithm and the equation generation method described in the previous section to verify regular safety problems of the form $\langle \mathcal{R}, \mathcal{A}, O \rangle$ where $\mathcal{R}$ is a $\mathcal{K}$-TRS, $\mathcal{A}$ a $\mathcal{K}^n$-coherent tree automaton (where n follows Theorem 2) and O some regular

language.

Input : A regular safety problem $\langle \mathcal{R}, \mathcal{A}, O \rangle$
Output : *Success* if the problem's property is verified or
Fail if a counter-example is found.

```
1  forall k from 1 to ∞ do
2  |  A*_k ← Completion_R(A_k);
3  |  if L(A*_k) ⊈ O then
4  |  |  return Fail;
5  |  else
6  |  |  forall E^c ∈ E^k_c do
7  |  |  |  E ← E^r ∪ E_R ∪ E^c;
8  |  |  |  A* ← Completion_{R,E}(A);
9  |  |  |  if L(A*) ∩ O = ∅ then
10 |  |  |  |  return Success;
```

Algorithm 1: Verification Procedure

The procedure defined by Algorithm 1 searches for a set of contracting equations E^c such that verification succeeds, *i.e.* $\mathcal{L}(\mathcal{A}^*) \subseteq O$. Starting from $k = 1$, we apply the following algorithm:

1. We first complete the tree automaton $\mathcal{A}_k$ recognizing the *finite* subset of $\mathcal{L}(\mathcal{A})$ of terms of maximum depth k. Since $\mathcal{L}(\mathcal{A}_k)$ is finite and $\mathcal{R}$ is terminating, the set of reachable terms is finite, completion terminates without equations and computes an automaton $\mathcal{A}^*_k$ recognizing exactly the set $\mathcal{R}^*(L(\mathcal{A}_k))$ [GR10].

2. If $\mathcal{L}(\mathcal{A}^*_k)$ does violates the property, *i.e.* $\mathcal{L}(\mathcal{A}^*_k) \not\subseteq O$, then verification fails: a counterexample is found.

3. Otherwise for all E^c of $\mathbb{E}^k_c$, we try to complete $\mathcal{A}$ with $\mathcal{R}$ and $E = E_r \cup E_\mathcal{R} \cup E^c$ and check the property on the completed automaton. If $\mathcal{A}^* \subseteq O$ then verification succeeds. Otherwise, we try the next E^c.

4. If no E^c remains, we start again with $k = k + 1$.

Because of $E_\mathcal{R}$ the generated abstraction is necessarily closed w.r.t. $\mathcal{R}$ which makes the abstraction collapsing (cf. Definition 2.4.9). Furthermore, because the tree automaton generated by the TAC algorithm is $\not\epsilon$-deterministic, the generated abstraction is necessarily functional (cf. Definition 2.4.8).

Theorem 3 (Functional & Collapsing Regular Completeness)**.** *Let $P = \langle \mathcal{R}, \mathcal{A}, O \rangle$ a regular safety problem over $\mathcal{K}$-TRS. Let $\mathcal{L}$ be the contracted language as defined in Theorem 2. If there exists a set of contracting equations $E^c_\mathcal{L}$ such that $E_r \cup E_\mathcal{R} \cup E^c_\mathcal{L}$ is able to verify P, this algorithm will eventually find $E^{c'}_\mathcal{L}$ such that $E_r \cup E_\mathcal{R} \cup E^{c'}_\mathcal{L}$ is also able to verify P.*

Proof. First note that each use of the completion algorithm terminates. For each k, The completion of $\mathcal{A}_k$ without equations terminates because $\mathcal{L}(\mathcal{A}_k)$ if finite. The completion of $\mathcal{A}$ with E terminates because $\mathcal{R}$ is a $\mathcal{K}$-TRS (Theorem 2). $E^c_\mathcal{L}$ represents a regular abstraction of $\mathcal{W}(\mathcal{C})$. Thanks to Lemma 21 we know there exists some k such that $\mathbb{E}^k_c$ contains some $E^{c'}_\mathcal{L}$ representing the same abstraction. Then if $E^{c'}_\mathcal{L}$ is enough to verify P, so is $E^{c'}_\mathcal{L}$. Since this algorithm does an exhaustive exploration of k and $\mathbb{E}^k_c$, and since because $\mathcal{R}$ is a $\mathcal{K}$-TRS each use of the completion algorithm terminates, it will eventually find it. □

Theorem 4 (Completeness in Refutation)**.** *Let $P = \langle \mathcal{R}, \mathcal{A}, O \rangle$ a regular safety problem over $\mathcal{K}$-TRS. Let $\mathcal{L}$ be the contracted language as defined in Theorem 2. If there is a counter-example to P, this procedure will eventually find it.*

Proof. At each step k, we complete $\mathcal{A}_k$ to find counter-examples. If there exists a term in $\mathcal{L}(\mathcal{A})$ that rewrites outside of O, then there exists some k such that this term is recognized by $\mathcal{A}_k$. □

4.6 Experiments

The verification technique described in this chapter has been integrated in the Timbuk library [Tbk3] (version 3.2). We implemented the naive equation generation procedure described in Section 4.5.2 where all possible equation sets E_c are enumerated. Despite the evident scalability issues of this simple version of the verification procedure, we have been able to verify a series of properties on several classical higher-order functions such as *map*, *filter*, *exists*, *forall*, *foldLeft*, *foldRight* as well as higher-order sorting functions parameterized by an ordering function.

4.6.1 Test Suite

Our test suite, publicly available on Timbuk's website [Exp3], is composed of positive and negative regular safety problems of the form $\langle \mathcal{R}, \mathcal{A}, O \rangle$ where $\mathcal{R}$ is a $\mathcal{K}$-TRS representing the functional program, $\mathcal{A}$ a $\mathcal{K}^n$-coherent tree automaton representing the possible initial states, and O a regular language. The goal for Timbuk is to prove the safety of the program by proving that $\mathcal{R}^*(\mathcal{L}(\mathcal{A})) \subseteq O$, or disprove it by finding a counter example. Most problems are taken from or inspired by [OR11, KSU11a], and Tons of Inductive Problems [CJRS15]. We expect some problem instances to be similar to the ones used in [MKU15]. However this cannot be verified since the test suite used in this paper is not publicly available.

Timbuk will prove the input property by exhibiting some tree automaton $\mathcal{A}^*$ over-approximating $\mathcal{R}^*(\mathcal{L}(\mathcal{A}))$ using the verification technique described in this chapter. For each problem $\langle \mathcal{R}, \mathcal{A}, O \rangle$ in the test suite, we have been able to verify the *correctness of the verification*, i.e. the fact that $\mathcal{R}^*(\mathcal{L}(\mathcal{A})) \subseteq \mathcal{L}(\mathcal{A}^*)$, using a proof assistant embedding a formalization of rewriting and tree automata [BGJ08]. To do that, it is enough to prove that (a) $\mathcal{L}(\mathcal{A}^*) \supseteq \mathcal{L}(\mathcal{A})$ and that (b) for all critical pairs $\langle l \to r, \sigma, q \rangle$ of $\mathcal{A}^*$ we have $r\sigma \to^*_{\mathcal{A}^*} q$. Property (a) can be checked using standard algorithms on tree automata. Property (b) can be checked by enumerating all critical pairs of $\mathcal{A}^*$ (there are finitely many) and by proving that all of them satisfy $r\sigma \to^*_{\mathcal{A}^*} q$. Since there exists algorithms for checking properties (a) and (b), the complete proof of correctness can automatically be built in the proof assistant. For instance, the automaton $\mathcal{A}^*$ can be used as a certificate to build the correctness proof in Coq [Inr16] and in Isabelle/HOL [NPW02]. Besides, since verifying (a) and (b) is automatic, the correctness proof may be run outside of the proof assistant (in a more efficient way) using a formally verified external checker extracted from the formalization. All our (successful) verification attempts has been certified automatically using an external certified checker defined in Coq [BGJ08].

4.6.2 Experimental Results

Table 4.1 and 4.2 give the results of our experiments with Timbuk on an Intel® i7-7600U CPU, 4 2.80GHz cores. The first column gives the name of the problem.

Name	Time (s)			Memory (MiB)
	Completion	Equations	Total	
allA	0.0	0.0	0.001 ±0.0	4.93
iteEvenOdd	0.0	0.0	0.001 ±0.0	5.58
mult	0.0	0.0	0.001 ±0.0	5.83
delete	0.0	0.01	0.01 ±0.0	6.24
plusEven	0.01	0.0	0.01 ±0.0	5.71
headReverse	0.02	0.0	0.03 ±0.0	8.92
insertionSort	0.02	0.05	0.07 ±0.03	8.96
reverse	0.01	0.07	0.08 ±0.01	7.24
appendTheorem	0.03	0.06	0.09 ±0.0	10.18
makelist	0.4	0.01	0.42 ±0.08	99.56
reverseImplies	0.09	0.45	0.54 ±0.12	9.61
insertionSort2	0.44	0.21	0.65 ±0.07	109.94
incTree	0.07	0.92	0.99 ±0.08	46.12
treeDepth	0.45	1.0	1.45 ±0.13	118.89
orderedTreeTraversal	0.08	1.44	1.52 ±0.12	11.83
orderedTree	0.1	5.14	5.24 ±0.48	13.8
memberAppend	18.38	2.38	20.76 ±2.37	2971.76
square	47.02	0.12	47.15 ±2.77	3997.26
deleteImplies	-	-	Timeout	-
equalLength	-	-	Timeout	-
evenOdd	-	-	Timeout	-
heightTree	-	-	Timeout	-
insertTree	-	-	Timeout	-
memberTree	-	-	Timeout	-
mergeSort	-	-	Timeout	-
replaceTree	-	-	Timeout	-
split	-	-	Timeout	-
zipUnzip	-	-	Timeout	-
simple	0.0	0.0	0.001 ±0.0	3.42
appendTheoremBug	0.01	0.0	0.01 ±0.0	6.36
memberAppendError	0.01	0.0	0.01 ±0.0	6.69
plusEvenError	0.01	0.0	0.01 ±0.0	6.06
reverseBUGimplies	0.01	0.0	0.01 ±0.0	6.09
deleteBUG	0.06	0.0	0.06 ±0.0	7.41
insertionSort2BUG	0.09	0.0	0.09 ±0.01	9.18
orderedTreeTraversalBug	0.18	0.0	0.18 ±0.05	8.96
orderedTreePredicate2	0.72	0.01	0.73 ±0.04	12.78

Table 4.1: First-order problems

Name	Time (s)			Memory (MiB)
	Completion	Equations	Total	
foldRightMult	0.01	0.0	0.01 ±0.0	6.53
foldDiv	0.02	0.0	0.03 ±0.04	8.59
forallLeq	0.01	0.01	0.03 ±0.0	8.47
forallNotEqNotExists	0.02	0.0	0.03 ±0.01	8.9
mapPlus	0.01	0.06	0.07 ±0.0	8.89
filterNz	0.02	0.1	0.12 ±0.01	8.95
insertionSortHO	0.04	0.1	0.14 ±0.01	9.82
forallImpliesExists	0.04	0.13	0.17 ±0.01	9.91
filterEquivExists	0.04	0.13	0.18 ±0.01	9.15
foldLeftPlus	0.01	0.19	0.2 ±0.01	8.95
filterEven	0.03	0.28	0.31 ±0.0	8.98
mapFilter	0.03	0.36	0.39 ±0.88	8.98
map2AddImplies	-	-	Timeout	-
mapSquare	-	-	Timeout	-
mapTree	-	-	Timeout	-
mergeSortHO	-	-	Timeout	-
forallImpliesExists2	0.0	0.0	0.001 ±0.0	3.74
forallNotEqNotForall	0.0	0.0	0.001 ±0.0	3.93
filterEvenBug	0.1	0.0	0.1 ±0.01	8.96
insertionSortHObug	0.35	0.0	0.35 ±0.16	9.54
mergeSortHObug	0.42	0.0	0.42 ±0.04	10.91

Table 4.2: Higher-order problems

The next three columns give the execution time before reaching a solution or finding a counter example, averaging on 10 executions: the time spent by the Tree Automata Completion algorithm, the time spent generating equations and the total time (± standard deviation). The timeout threshold is set to 120s. The last column gives the memory usage of Timbuk. Each table is split in two, with on the top positive problems where the property is proved, and on the bottom negative problems where the property is disproved.

Execution Time

The tables show mostly low execution times (far below the second) for most problems, in particular when dealing with lists or incorrect properties. However we see that simply enumerating all possible equations sets becomes a problem when dealing with binary trees (treeDepth, orderedTreeTraversal, orderedTree) where the execution time start rising above the second. This cannot be improved without developing a more efficient equation generation procedure that doesn't need to go through all possible equations.

Memory Usage

We also see that the memory usage is extremely volatile, ranging from 4MiB in the best case to 4GiB in the worst case. This happens because the whole program is analyzed at once, on the whole set of inputs. It makes the Tree Automata Completion algorithm generate huge automata, especially during the exact completion phase (without equations) in the equation generation procedure, Algorithm 1 (line 2). This could be improved be adding modularity, analyzing each function separately when possible (which is one of the purposes of the next chapter).

Completeness

Finally, many problems has not been solved before the timeout threshold. For the most part, this is because this technique is not *regularly complete* (does not cover every regular safety problem). We proved in the previous section the "functional and collapsing" completeness of this technique (Theorem 3), which means that it can prove any regular problem which can be proved using a collapsing functional abstraction, i.e. each term is abstracted into a unique equivalence class and each class is $\mathcal{R}$-closed. It is enough to handle non-trivial problems such as proving that the output of the insertion-sort algorithm is sorted (insertionSort). However it is not enough to prove the merge-sort algorithm. In fact, it is not even enough to verify some simpler functions.

Example 4.6.1. *Consider the following TRS $\mathcal{R}$ defining a pred function computing the predecessor of a natural number. The term $pred(0)$ rewrites into the special error state $fail$.*

$$pred(0) \to fail \qquad\qquad pred(s(\underline{x})) \to \underline{x}$$

We want to show that $pred(n)$ when $n > 0$ never fail. Using the technique described in this chapter, the natural approach would be to (automatically) search for the contracting equation set $E^c = \{s(s(0)) = s(0)\}$ (or similar) that separates the natural numbers into two equivalence classes $\mathbb{O}$ and $\mathbb{N}^+$. This doesn't work because the generated set of equation E contains E^c but also $E_{\mathcal{R}}$. This means that for all $n \in \mathbb{N}^+$, since $pred(n)$

rewrites into a natural number it must be abstracted into either $\mathbb{O}$ *or* $\mathbb{N}^+$*. If we take* $n = s(0)$ *then* $pred(s(0))$ *rewrites into* 0 *so* $pred(s(0)) \in \mathbb{O}$*. Now if we take* $n = s(s(0))$ *then* $pred(s(s(0)))$ *rewrites into* $s(0)$ *so* $pred(s(s(0))) \in \mathbb{N}^+$*. However both* $s(0)$ *and* $s(s(0))$ *are in the same class* $\mathbb{N}^+$*, which means that* $pred(s(0))$ *and* $pred(s(s(0)))$ *should be in the same class, although we have just seen that they are not (*$pred(s(0)) \in \mathbb{O}$ *and* $pred(s(s(0))) \in \mathbb{N}^+$*). This shows that there are no sets of equations of the form* $E^c \cup E_{\mathcal{R}} \cup E_r$ *that can verify this function. As a result, this verification technique is bound to diverge even on this simple problem. There are two ways to solve this problem:*

- *A first solution is to remove* $E_{\mathcal{R}}$ *so that the final abstraction may not be* collapsing*. This would allow* $pred(s(0))$ *to not be automatically abstracted by* $\mathbb{O}$ *but in a third abstract value* $\mathbb{N}$ *for instance, along with* $pred(s(s(0)))$*.*
- *Another solution would be to generate non-functional abstractions, so that* $pred(s(0))$ *can be abstracted by both* $\mathbb{O}$ *(because of* $E_{\mathcal{R}}$*)* and $\mathbb{N}^+$*.*

This example shows that the only way to reach full regular completeness is to generate either non-functional abstraction or non-collapsing abstractions [GR10]. Remember that by definition the Tree Automata Completion algorithm can only generate functional abstractions. Generating non-collapsing abstractions would mean generating equation sets without $E_{\mathcal{R}}$, which was introduced to reduce the number of equations to consider [Gen16]. In the light of our experimental results this is to be reconsidered.

4.6.3 Related Work

To our knowledge, the technique developed by Matsumoto et al. [MKU15] is the only other fully automatic verification technique targeting regular safety problems. As mentioned in Section 3.3.2, this paper defines a counter-example guided refinement procedure to directly find equivalence classes represented as tree automata instead of equations (each automaton recognizing one equivalence class). The abstraction procedure is based on SMT solving and is regularly complete as it generates regular abstractions that are not necessarily functional or $\mathcal{R}$-closed. However because of that, the abstraction procedure is computationally expensive. Even if in our case enumerating equation sets can also be expensive, we seem to have overall better performances. However this cannot be properly verified as neither the implementation nor the test suite of used in the paper [MKU15] are publicly available.

4.7 Conclusion

In this chapter we have used the Tree Automata Completion algorithm to automatically verify regular safety properties on higher-order functional programs. We have defined the class of $\mathcal{K}$-TRS defining higher-order functional programs on which contracting equations on values could be used to generate regular abstractions without compromising the termination of the completion algorithm. This has been used to automatically verify regular properties on higher-order programs with good time performances, in particular on list processing programs.

However we have seen with Example 4.4.3 that the $\mathcal{K}$-TRS does not include higher-order functional programs written in CPS form, which is an important family of functional programs. Moreover we also showed that the current abstraction procedure

based on contracting equations can only generate functional, collapsing abstractions. We have seen that this prevents us to reach regular completeness: some regular problems as simple as the *pred* problem (cf. Example 4.6.1) cannot be handled. We need to find another way to generate abstractions. We have seen that in their paper [MKU15] Matsumoto et al. define a regularly complete abstraction procedure by using tree automata instead of equations. However by generating regular abstractions that are not necessarily functional or collapsing, this procedure seems expensive. In the next chapter we show that we can still generate collapsing only abstractions while preserving regular completeness. By using regular languages as types, we define a new complete and modular verification technique designed to scale.

Chapter 5

Regular Language Type Inference

5.1 Introduction

In this chapter we design a novel technique for the verification of regular safety properties on higher-order functional programs that is complete and modular. This continues the effort developed in the last chapter where we ended off with three problems to solve:

- (Abstraction) Using contracting equations on values with the Tree Automata Completion algorithm can only generate abstractions that are both functional and collapsing. We have seen with Example 4.6.1 that this prevents the use of contracting equations in a regularly complete verification technique for safety properties. We need a new way of abstracting the program in a non-functional manner.
- (Inference) For now we used a very naive way of generating abstractions that just enumerates all the possibles abstractions. This is relatively fast in practice for small programs as shown in Section 4.6.2. However this is favored by the use of contracting equations on values, which we cannot use in a complete verification technique. We need to define a proper abstraction inference procedure.
- (Modularity) Finally, analyzing whole programs at once is surely not practicable with larger programs. We need to design a modular verification procedure able to break down the verification problem into smaller parts.

Our new technique is defined as a type inference system which aims at typing each term of the input term rewriting system using regular languages as types.

5.1.1 Abstraction solution: regular abstract interpretation

Our first problem (Abstraction) is solved by explicitly defining the abstraction relation $\rightsquigarrow$ (cf. Definition 2.4.4) using term rewriting systems. This defines a framework of *regular abstract interpretation*, a special case of abstract interpretation where: (a) the concrete domain is $\mathcal{T}(\Sigma)$ the set of terms/states, (b) the abstract domain is $\Sigma^\#$ a set of abstract states, (c) the abstraction relation is defined as a rewriting system $\Delta^\#$ that rewrites concrete terms into abstract states. We write $\Lambda = \langle \Sigma^\#, \Delta^\# \rangle$ the tree automaton defining the abstraction of $\mathcal{T}(\Sigma)$. In this automaton, each $\Sigma^\#$ recognizes a regular language. (d) the abstract semantics of the program is defined by a TRS $\mathcal{R}^\#$, extracted from $\mathcal{R}$, that rewrites abstract terms. The relation $\rightsquigarrow$ is defined as $\rightarrow^!_{\Delta^\# \cup \mathcal{R}^\#}$ (remember that we write $s \rightarrow^!_{\mathcal{R}} t$ when $s \rightarrow^* t$ and $t \in IRR(\mathcal{R})$).

Example 5.1.1. *Let us consider the following property over the TRS $\mathcal{R}$ defining a filter function over lists of natural numbers (List). We want to verify that for each list l, when filter is called with the even predicate, all odd numbers are filtered out of the l in the output. This can be stated as follows using the for_all predicate function:*

$$\forall l \in List.\ for_all\ \text{even}\ (\text{filter even}\ l) \not\rightarrow^*_{\mathcal{R}}\ \text{false}$$

All the functions (for_all, even, filter, etc.) are defined in $\mathcal{R}$. To verify this particular property, in the context of regular abstract interpretation we can restate our property as

$$for_all\ \text{even}\ (\text{filter even}\ List^\#)\ \not\rightsquigarrow \text{false}^\#$$

where $List^\#$ and $\text{false}^\#$ are some elements of the abstract domain $\Sigma^\#$ and $\rightsquigarrow$ the relation $\rightarrow^!_{\Delta^\# \cup \mathcal{R}^\#}$. Each abstract value of $\Sigma^\#$ recognizes a concrete language as defined by $\Delta^\#$:

$$\begin{array}{rrr} 0 \rightarrow Nat^\# & nil \rightarrow List^\# & true \rightarrow true^\# \\ s(Nat^\#) \rightarrow Nat^\# & cons(Nat^\#, List^\#) \rightarrow List^\# & false \rightarrow false^\# \end{array}$$

Note how the $\forall$ disappears. A solution to this problem is a complete definition of $\mathcal{R}^\#$ (which was not yet given) and $\Lambda = \langle \Sigma^\#, \Delta^\# \rangle$ such that the new (abstract) property is verified when $\rightsquigarrow$ is an abstraction of $\mathcal{R}$ in $\Sigma^\#$. In this case, a possible solution contains $\Delta^\#$ defined by (in addition to the transitions above):

$$\begin{array}{rrr} 0 \rightarrow Even^\# & s(Even^\#) \rightarrow Odd^\# & nil \rightarrow EvenList^\# \\ s(Odd^\#) \rightarrow Even^\# & even \rightarrow even^\# & cons(Even^\#, EvenList^\#) \rightarrow EvenList^\# \end{array}$$

and $\mathcal{R}^\#$ defined as [1]*:*

$$\begin{array}{r} \text{filter even}^\#\ List^\# \rightarrow EvenList^\# \\ for_all\ \text{even}^\#\ EvenList^\# \rightarrow \text{true}^\# \end{array}$$

In this abstraction, (filter even l) rewrites into $EvenList^\#$ using $\Delta^\#$ and $\mathcal{R}^\#$. In the same way, (for_all even $EvenList^\#$) rewrites into $\text{true}^\#$ and only $\text{true}^\#$, which means that overall we have (for_all even (filter even $List^\#$)) $\not\rightsquigarrow \text{false}^\#$. (remember that $\rightsquigarrow = \rightarrow^!_{\Delta^\# \cup \mathcal{R}^\#}$).

The verification problem boils down to the problem of automatically finding $\Lambda = \langle \Sigma^\#, \Delta^\# \rangle$ and $\mathcal{R}^\#$ that proves or disproves the user's property.

5.1.2 Modularity solution: regular language types

Note how each rule of $\Delta^\#$ and $\mathcal{R}^\#$ acts as a type signature for its associated constructor symbol or function symbol respectively. To find $\Sigma^\#$, $\Delta^\#$ and $\mathcal{R}^\#$ we embrace the type perspective by defining a modular type inference algorithm that can find the type signatures of each symbol necessary to solve a given regular safety verification problem. The algorithm is a backward static analysis that uses the input expression target type to find the input type of each function call recursively.

[1] For $\rightsquigarrow$ to really be an abstraction of $\mathcal{R}$ according to Definition 2.4.4, some other rules are needed in $\mathcal{R}^\#$ and $\Delta^\#$ to abstract intermediate states. This will be detailed later on.

Example 5.1.2. *Our previous (abstract) rewriting problem becomes a regular typing problem stated as:*

$$\Lambda, \mathcal{R}^{\#}, \pi \not\vdash for_all\ even\ (filter\ even\ x) : false^{\#}$$

where $\vdash$ *is our type judgment relation with* $\Lambda, \mathcal{R}^{\#}$ *the typing environment and* $\pi : \mathcal{X} \mapsto \Sigma^{\#}$ *the substitution binding each variable to its type. We want to make sure that it is* impossible *to type this input term with* $false^{\#}$*. It can be resolved by inferring every possible type for* $(for_all\ even\ (filter\ even\ x))$ *and its subterms, and check that it is never typed with* $false^{\#}$*.*

To solve this problem, we have developed a regular type inference procedure. Given an input term (such as $(for_all\ even\ (filter\ even\ x))$) and a target type (such as $false^{\#}$), this procedure is able to infer $\Lambda, \mathcal{R}^{\#}$ and all the possible type substitutions π for which the given term is typed with the given type. This typing algorithm analyzes each function separately, solving our modularity problem.

Example 5.1.3. *To find* $\Lambda, \mathcal{R}^{\#}, \pi$ *such that* $(for_all\ even\ (filter\ even\ x))$ *can be typed with the target type* $false^{\#}$*, our type inference algorithm starts by solving the simpler problem of finding* $\Lambda, \mathcal{R}^{\#}, \pi$ *such that* $(for_all\ \underline{y}\ \underline{z})$ *can be typed with the same target* $false^{\#}$*, which does not involve* filter *nor* even*. It will find/generate input types for* y *and* z *that can then be recursively used to analyze the subterms* even *and* $(filter\ even\ x)$*, and so on. If it cannot find a suitable type for* x *(or any other term), then we know* $(for_all\ even\ (filter\ even\ x))$ *cannot be typed with* $false^{\#}$ *and the property is verified.*

In this algorithm, each function is analyzed separately. [2] For each function f, it needs to solve the following problem: Given a target output type for f, infer all the possible inputs types to the function that leads to this type. We see in Section 5.3.3 that for non-recursive functions, this can be done using well known operations on tree automata by simply looking at the output type and the rewriting rules of the function. For recursive functions however, a more complex invariant learning procedure is needed.

5.1.3 Inference solution: Regular language learning

Analyzing a recursive function is much harder than non-recursive functions, as it requires finding a regular fixpoint for the function. We do that by solving our last problem (Inference) with the definition of a new regular invariant learning procedure. The core of the invariant inference procedure is an example guided regular language learning procedure, based on the Tree Automata Completion Algorithm [Gen16] and SMT constraints solving. For a given symbol f and target regular language partition P (a set of disjoint regular languages), our procedure is able to give, for all $\mathcal{L} \in P$, all the input regular languages $\mathcal{L}_1, \ldots, \mathcal{L}_n$ such that for all $t_i \in \mathcal{L}_i$, $f(t_1, \ldots, t_n)$ rewrites into a term belonging to $\mathcal{L}$.

Example 5.1.4. *Let us focus on the following fragment of the TRS* $\mathcal{R}$ *used in the previous examples:*

$$even\ 0 \to true \qquad even\ s(\underline{x}) \to odd\ \underline{x} \qquad odd\ 0 \to false \qquad odd\ s(\underline{x}) \to even\ \underline{x}$$

[2] More accurately, the typing algorithm analyzes mutually recursive functions at the same time.

By giving $\mathcal{R}$, even, and the target partition $\{\{\text{true}\},\{\text{false}\}\}$ to our procedure, it is able to automatically learn that the interesting input languages for even are the languages $Even = \{n \mid n \,\%\, 2 = 0\}$ and $Odd = \mathbb{N} \setminus Even$. It will also infer that $\forall n \in Even.\ even\ n \to^* true$ and that $\forall n \in Odd.\ even\ n \to^* false$.

As we will see in Section 5.3.4, learning is done by providing a series of positive and negative examples to the procedure. For instance by providing (*even* 0) and (*even* $s(0)$), because both rewrite to two different elements of the target partition, we learn that 0 and $s(0)$ must be in different input languages. In practice, the target language partition is given in terms of *abstract* values in an input abstraction domain. In our example, the input partition would be $\{true^\#, false^\#\}$. Similarly, the output of the invariant inference procedure is a refinement of the abstract domain (rules to add to $\Delta^\#$) and an abstract TRS $\mathcal{R}^\#$. In our example, the output of the procedure would be:

$0 \to Even^\#$	$s(Even^\#) \to Odd^\#$	$s(Odd^\#) \to Even^\#$	added to $\Delta^\#$
	$even\ Even^\# \to true^\#$	$even\ Odd^\# \to false^\#$	added to $\mathcal{R}^\#$
	$oddOdd^\# \to false^\#$	$odd\ Even^\# \to true^\#$	

Note that it also gives information about the *odd* function, since the definition of even is mutually recursive with *odd*. In Section 5.3.4 we show that this procedure is *regularly complete* and *complete in refutation*. This means that, if there exists a regular abstraction ($\Delta^\#$ and $\mathcal{R}^\#$) that satisfies the input partition, then the procedure will find it. In addition, if there exists an input term rewriting into two different elements of the input partition P, the procedure terminates and provides such a counter-example term.

The rest of the chapter is structured as follows. Section 5.2 introduces our regular abstract interpretation framework. Section 5.3 presents the regular language type system and the type inference algorithm. Section 5.4 exposes the results of our experiments carried on a custom implementation of the technique in a new version of Timbuk [Tbk4], and compare them with the technique defined in Chapter 4. Finally, Section 5.5 concludes the chapter.

5.2 Regular Abstract Interpretation

In this section we define a regular abstract interpretation framework that adapts classical abstract interpretation to term rewriting systems and regular languages. In this framework, terms representing states of the execution are abstracted into a regular language, denoted by a state of a tree automaton. For readability, we denote those states by identifiers of the form $name^\#$, where *name* is an arbitrary symbol whose role is to provide an intuition of the language recognized by $name^\#$. In the following, we also call those states *abstract values*. The set of abstract value is the *abstract domain*, denoted by $\Sigma^\#$. The semantics of the program $\mathcal{R}$ is represented by an abstract TRS $\mathcal{R}^\#$ that operates over the abstract domain of regular languages.

We start with an example. Define I as the set $\{nil, cons(a, nil), cons(b, nil), ...\}$ of (not nested) lists of as and bs and assume that the safety property we wish to verify is

$$(1) \quad \forall l \in I.\ sorted\ (sort\ l) \not\to^*_{\mathcal{R}} false$$

This can be translated into an abstract interpretation problem where the goal is to find an *abstract* domain $\Sigma^\#$ provided with an abstraction function $\alpha : \mathcal{T}(\Sigma) \to \mathcal{P}(\Sigma^\#)$

and a concretization function $\gamma : \Sigma^\# \to \mathcal{P}(\mathcal{T}(\Sigma))$, and an abstract semantics $\to^\#$ extracted from $\mathcal{R}$ which faithfully models $\mathcal{R}$ such that

$$(2) \quad \mathit{sorted}\ (\mathit{sort}\ \mathit{ab_list}^\#) \not\to^{\#*} \mathit{false}^\#$$

where $\mathit{ab_list}^\#$ and $\mathit{false}^\#$ are two abstract values of $\Sigma^\#$ such that $\alpha(I) = \{\mathit{ab_list}^\#\}$ and $\gamma(\mathit{false}^\#) = \{\mathit{false}\}$. Note that having an abstraction α returning a set of abstract elements is non-standard in abstract interpretation but we have ensured that it is used consistently throughout the chapter. In this particular first abstraction, all lists of a and b are abstracted by the same abstract element $\mathit{ab_list}^\#$ thus $\alpha(I)$ is the singleton $\{\mathit{ab_list}^\#\}$. Since $\Sigma^\#$ is finite, it is easier to verify property (2) than property (1). Then, if the abstraction is correct and property (2) is true, we can then deduce that property (1) is also true. In the rest of this section, we show how to define $\to^\#$, α and γ in terms of term rewriting systems and how to infer them with an abstract domain $\Sigma^\#$ which together will allow us to verify a given property. In the rest of this chapter, we use σ to denote *concrete* substitutions from $\mathcal{X}$ to $\mathcal{T}(\Sigma)$, and π to denote *abstract* substitutions from $\mathcal{X}$ to $\Sigma^\#$. In addition we name $\gamma(\pi)$ the set of all concrete substitutions extracted from π, i.e., $\gamma(\pi) = \{\ \sigma \mid \forall x \in \mathit{Dom}(\pi).\ \sigma(x) \in \gamma(\pi(x))\ \}$.

5.2.1 Regular Abstract Domain

The *concrete domain* of a regular abstract interpretation is $\mathcal{T}(\Sigma)$. The elements of the *abstract domain* of a regular abstract interpretation are arbitrary symbols, each of them representing a regular set of terms of $\mathcal{T}(\Sigma)$. One originality of our approach resides in representing an abstract domain by a tree automaton, where each state corresponds to a regular set of terms. More precisely, an abstraction is represented by a tree automaton $\Lambda = \langle \Sigma, \Sigma^\#, \Sigma^\#, \Delta^\# \rangle$ where $\Sigma^\#$ is the set of abstract values and $\Delta^\#$ a term rewriting system defining the associated abstraction function α that describes how concrete terms are abstracted into abstract values. For readability, we write $\Lambda = \langle \Sigma^\#, \Delta^\# \rangle$. In the previous example, Λ can be defined using the following $\Delta^\#$:

$$a \to \mathit{ab}^\# \qquad b \to \mathit{ab}^\# \qquad \mathit{nil} \to \mathit{ab_list}^\# \qquad \mathit{cons}(\mathit{ab}^\#, \mathit{ab_list}^\#) \to \mathit{ab_list}^\#$$

Note that a configuration of Λ is a term of $T(\Sigma, \Sigma^\#)$, a mix between abstract and concrete terms which allows the transition from the concrete domain to the abstract one. Each term of $T(\Sigma, \Sigma^\#)$ is called an *abstract pattern*.

Definition 5.2.1 (Regular Abstract Domain)**.** *Let Σ be an alphabet. A $T(\Sigma)$-abstraction $\Lambda = \langle \Sigma^\#, \Delta^\# \rangle$ is a* regular abstract domain *iff the corresponding tree automaton $\langle \Sigma, \Sigma^\#, \Sigma^\#, \Delta^\# \rangle$ is normalized and complete. The abstraction and concretization functions are then defined by*

$$\alpha(t) = \{a^\# \mid t \to^*_{\Delta^\#} a^\#\} \qquad\qquad \gamma(a^\#) = \{t \mid t \to^*_{\Delta^\#} a^\#\}$$

Note that in this definition α returns the set of all possible abstract values for any term t. Since Λ can be non-deterministic, this set may contain more than one abstract element.

5.2.2 Abstract Semantics

By rewriting a term t using $\Delta^\#$, we can abstract some subterms of t by elements of $\Sigma^\#$. However, this abstracted term can no longer be rewritten using $\mathcal{R}$. We need

to introduce a new rewriting system $\mathcal{R}^\#$ as an abstraction of $\mathcal{R}$, rewriting *abstract patterns* while *preserving the behavior* of $\mathcal{R}$. To deduce a property of $\mathcal{R}$ using $\mathcal{R}^\#$, the abstract $\mathcal{R}^\#$ must itself respect specific constraints w.r.t. $\mathcal{R}$.

Definition 5.2.2 (Abstraction of a TRS)**.** *Let $\mathcal{R}$ be a TRS, $\Lambda = \langle \Sigma^\#, \Delta^\# \rangle$ a regular abstract domain over $IRR(\mathcal{R})$. Let $\mathcal{R}^\#$ be a TRS over $T(\Sigma^\#)$. $\mathcal{R}^\#$ is an* abstraction *of $\mathcal{R}$ over Λ iff*

- *every rule of $\mathcal{R}^\#$ has the form $f(a_1^\#, \ldots, a_n^\#) \to a^\#$ with $f \in \Sigma^n$, $a_1^\#, \ldots, a_n^\#, a^\# \in \Sigma^\#$;*
- *(soundness) for all rules $f(a_1^\#, \ldots, a_n^\#) \to a^\#$ of $\mathcal{R}^\#$, for all terms t of $\{f(t_1, \ldots, t_n) \mid t_i \in \gamma(a_i^\#)\}$, for all terms $u \in IRR(\mathcal{R})$ such that $t \to_\mathcal{R}^* u$ then $u \in \gamma(a^\#)$;*
- *(completeness) for all symbols f used in $\mathcal{R}^\#$, for all terms $t = f(t_1, \ldots, t_n)$ where $t_i \in IRR(\mathcal{R})$ for all i, there must exist some rule $f(a_1^\#, \ldots, a_n^\#) \to a^\#$ such that for each i, $t_i \in \gamma(a^\#)$. In other words, for a given used symbol, each possible abstract input is mapped to at least one abstract output.*

Theorem 5 (Correctness)**.** *Let $\mathcal{R}$ be a TRS, $\Lambda = \langle \Sigma^\#, \Delta^\# \rangle$ an abstract domain over $IRR(\mathcal{R})$. If $\mathcal{R}^\#$ is an abstraction of $\mathcal{R}$ over Λ then we have for all pattern p, abstract substitution π and abstract value $v^\# \in \Sigma^\#$:*

$$p\pi \to_{\mathcal{R}^\# \cup \Delta^\#}^* u^\# \quad \Rightarrow \quad \forall \sigma \in \gamma(\pi).\ \forall u \in IRR(\mathcal{R}).\ p\sigma \to_\mathcal{R}^* u \Rightarrow u \in \gamma(u^\#)$$

with $\gamma(\pi) = \{\ \sigma \mid \forall x \in Dom(\pi).\ \sigma(x) \in \gamma(\pi(x))\ \}$ the set of all possible concretized substitutions.

Proof. This can be proved by a simple induction on the size of the (abstract) rewriting path using the *soundness* property of $\mathcal{R}^\#$. If $p\pi = u^\#$ then p is some variable x. For all $\sigma \in \gamma(\pi)$, $p\sigma$ is a ground term. So for all $u \in IRR(\mathcal{R})$ such that $p\sigma \to_\mathcal{R}^* u$ we have $u = p\sigma$, with $u \in \gamma(u^\#)$. Next, if $p\pi \to_{\mathcal{R}^\# \cup \Delta^\#}^{k+1} u^\#$. By definition (of $\mathcal{R}^\#$ and $\Sigma^\#$), there exists a context C, a pattern $l = f(l_1, \ldots, l_n)$ and an abstract value $r^\#$ such that $p = C[l]$ and $p\pi \to_{\mathcal{R}^\# \cup \Delta^\#} C[r^\#] \to_{\mathcal{R}^\# \cup \Delta^\#}^k u^\#$. Now let us consider $\sigma \in \gamma(\pi)$ and $u \in IRR(\mathcal{R})$. Since $C[l]\pi \to_{\mathcal{R}^\# \cup \Delta^\#} C[r^\#]$ then by definition of $\mathcal{R}^\#$ (soundness of abstraction $\mathcal{R}^\#$) there exists some term r such that $l\sigma \to_\mathcal{R}^* r$ so that $C[l]\sigma \to_\mathcal{R}^* C[r]$ which we can write as (1) $p\sigma \to_\mathcal{R}^* C[r]\sigma$. Finally let x be a fresh variable and π' be the abstract substitution such that $\pi' = \pi \cup \{x \mapsto r^\#\}$. By construction we have $C[r^\#]\pi = C[x]\pi'$. Using the induction hypothesis on $C[x]\pi' \to_{\mathcal{R}^\# \cup \Delta^\#}^* u^\#$, we get that (2) $\forall \sigma' \in \gamma(\pi').\ \forall u \in IRR(\mathcal{R}).\ C[x]\sigma' \to_\mathcal{R}^* u \Rightarrow u \in \gamma(u^\#)$. Since $\pi' = \pi \cup \{x \mapsto r^\#\}$, for all $\sigma' \in \gamma(\pi')$ there exists $r \in \gamma(r^\#)$ and $\sigma \in \gamma(\pi)$ such that $\sigma' = \sigma \cup \{x \mapsto r\}$. In this case, $C[x]\sigma' = C[r]\sigma$ and we can connect (1) and (2) to obtain that $p\sigma \to_\mathcal{R}^* u \Rightarrow u \in \gamma(u^\#)$. □

Note that it is not sufficient to have *some* abstraction of $\mathcal{R}$ to be able to verify the desired property. On our previous example, the following TRS is a correct abstraction of $\mathcal{R}$:

$$sort\ ab_list^\# \to ab_list^\# \qquad\qquad sorted\ ab_list^\# \to bool^\#$$

however since all lists of as and bs are always abstracted by the same element $ab_list^\#$, this abstraction is too coarse to prove that for all list l, $sorted\ (sort\ l) \not\to_\mathcal{R}^* false$. To succeed, we need to make sure that our abstraction provides the additional property:

$$\forall v^\#.\ sorted\ (sort\ v^\#) \not\to_{\mathcal{R}^\# \cup \Lambda}^* false^\# \Rightarrow \forall l.\ sorted\ (sort\ l) \not\to_\mathcal{R}^* false$$

Another way to phrase it is for $\mathcal{R}^\#$ to be "complete" w.r.t. *sorted* (*sort* l) and $false^\#$, according to the following definition:

Definition 5.2.3 (Complete abstraction)**.** *Let $\mathcal{R}^\#$ be a TRS abstraction of $\mathcal{R}$ over $\Lambda = \langle \Sigma^\#, \Delta^\# \rangle$. We say that $\mathcal{R}^\#$ is complete with regard to a pattern p and abstract value $v^\#$ when for all term $t \in \gamma(v^\#)$, for all substitution $\sigma : \mathcal{X} \mapsto \mathcal{T}(\Sigma)$, if $p\sigma \to^*_{\mathcal{R}} t$ then there exists an abstract substitution $\pi : \mathcal{X} \mapsto \Sigma^\#$ such that $\sigma \in \gamma(\pi)$ and $p\pi \to^*_{\Delta^\# \cup \mathcal{R}^\#} v^\#$. This can be seen as the the contraposition of Theorem 5 for a particular case of p and $v^\#$.*

In our example, for $\mathcal{R}^\#$ to be complete w.r.t. *sorted* (*sort* l) and $\mathit{false}^\#$ we need at least three elements to abstract lists of as and bs: $\mathit{sorted}^\#$, $\mathit{unsorted}^\#$, and $b_list^\#$ recognizing respectively sorted a and b lists, unsorted lists, and lists of bs. The abstract domain Λ becomes:

$$\begin{array}{lll} a \to a^\# & cons(b^\#, b_list^\#) \to b_list^\# & cons(b^\#, \mathit{sorted}^\#) \to \mathit{unsorted}^\# \\ b \to b^\# & cons(a^\#, b_list^\#) \to \mathit{sorted}^\# & cons(b^\#, \mathit{unsorted}^\#) \to \mathit{unsorted}^\# \\ nil \to b_list^\# & cons(a^\#, \mathit{sorted}^\#) \to \mathit{sorted}^\# & cons(a^\#, \mathit{unsorted}^\#) \to \mathit{unsorted}^\# \end{array}$$

and the abstract TRS $\mathcal{R}^\#$ becomes:

$$\begin{array}{lll} sort\ b_list^\# \to b_list^\# & sorted\ b_list^\# \to true^\# & sort\ \mathit{sorted}^\# \to \mathit{sorted}^\# \\ sorted\ \mathit{sorted}^\# \to true^\# & sort\ \mathit{unsorted}^\# \to \mathit{sorted}^\# & sorted\ \mathit{unsorted}^\# \to \mathit{false}^\# \end{array}$$

Using those abstract elements, domain and TRS, we can show:

$$\left.\begin{array}{l} sorted\ (sort\ \mathit{sorted}^\#) \not\to^*_{\mathcal{R}^\# \cup \Lambda} \mathit{false}^\# \\ sorted\ (sort\ \mathit{unsorted}^\#) \not\to^*_{\mathcal{R}^\# \cup \Lambda} \mathit{false}^\# \\ sorted\ (sort\ b_list^\#) \not\to^*_{\mathcal{R}^\# \cup \Lambda} \mathit{false}^\# \end{array}\right\} \quad \Rightarrow \quad \forall l \in I.\ sorted\ (sort\ l) \not\to^*_{\mathcal{R}} \mathit{false}$$

To use this for verification, we need to *infer* Λ with an $\mathcal{R}^\#$ that is *complete* with regard to our desired property: this is the *inference problem* we solve in this chapter.

Definition 5.2.4 (Inference problem)**.** *Assume that we are given a TRS $\mathcal{R}$, a pattern $p \in T(\Sigma, \mathcal{X})$, an initial abstract domain $\Lambda_* = \langle \Sigma^\#_*, \Delta^\#_* \rangle$ and a target abstract value $v^\# \in \Sigma^\#_*$. A solution to the inference problem is*

1. *an abstract domain $\Lambda = \langle \Sigma^\#, \Delta^\# \rangle$ such that $\Sigma^\# \supseteq \Sigma^\#_*$ and $\Delta^\# \supseteq \Delta^\#_*$;*
2. *an abstraction $\mathcal{R}^\#$ of $\mathcal{R}$ in Λ, complete w.r.t. p and $v^\#$;*
3. *the set Π of all the abstract substitutions π such that $p\pi \to^*_{\Delta^\# \cup \mathcal{R}^\#} v^\#$.*

Intuitively, a solution provides a set of substitutions from variables to abstract values such that if the pattern rewrites to a result that belongs to $v^\#$ then there is an abstract substitution in Π such that the pattern rewrites to $v^\#$. Note that if the resulting set Π is empty, we can deduce that for all $t \in \gamma(v^\#)$, for all substitution σ we have $p\sigma \not\to^*_{\mathcal{R}} t$.

5.2.3 Abstraction Inference Challenges

To automatically solve inference problems, we designed an abstraction refinement procedure where an initial coarse abstraction is iteratively refined from the informations gathered at previous iterations, until it converges to a satisfying abstraction able to prove or disprove the desired property. A classical approach to abstraction refinement is the Counter-Example Guided Abstraction Refinement (CEGAR) procedure [CGJ+00], where the refinement information comes from spurious counter-examples introduced by the considered abstractions. In our setting, using CEGAR would mean to start the procedure with some initial abstractions $\mathcal{R}^\#$ and Λ, and search for spurious counter-examples introduced by the abstraction to refine it. For instance, consider the inference problem defined by $p = (\mathit{sorted}\ \underline{l})$ and $v^\# = \mathit{false}^\#$. By Definition 5.2.4 we want to find an abstract domain $\Lambda = \langle \Sigma^\#, \Delta^\# \rangle$ (with $\mathit{false}^\# \in \Sigma^\#$ and $\mathit{false} \to \mathit{false}^\# \in \Delta^\#$) an abstraction $\mathcal{R}^\#$ of $\mathcal{R}$ in Λ that is complete w.r.t. p and $\mathit{false}^\#$. Assume we start from the basic abstraction Λ where $\Delta^\#$ is defined by

$$\begin{array}{rr} \mathit{true} \to \mathit{true}^\# & \mathit{false} \to \mathit{false}^\# \\ a \to \mathit{ab}^\# & b \to \mathit{ab}^\# \\ \mathit{nil} \to \mathit{ab_list}^\# & \mathit{cons}(\mathit{ab}^\#, \mathit{ab_list}^\#) \to \mathit{ab_list}^\# \end{array}$$

and the initial abstraction $\mathcal{R}^\#$:

$$\mathit{sorted}\ \mathit{ab_list}^\# \to \mathit{true}^\# \qquad \mathit{sorted}\ \mathit{ab_list}^\# \to \mathit{false}^\#$$

Obviously this abstraction is too coarse because it does not distinguish between sorted and unsorted lists. As a result we get:

$$\mathit{sorted}\ \mathit{ab_list}^\# \to^*_{\mathcal{R}^\# \cup \Lambda} \mathit{true}^\#$$

even though there exists an (unsorted) list $l \in \gamma(\mathit{ab_list}^\#)$ such that

$$\mathit{sorted}\ l \not\to^*_{\mathcal{R}} \mathit{true}$$

$\mathcal{R}^\#$ is not a sound abstraction of $\mathcal{R}$, which is required to be a valid solution of our inference problem. We need to refine it. To do that using CEGAR, we need to find such a $(\mathit{sorted}\ l)$ that does not rewrite to *true*. It is a counter-example to the soundness of the abstraction, and we must make sure it does not occur again in the refined abstraction. For instance here, a possible counter-example is $l = \mathit{cons}(b, \mathit{cons}(a, \mathit{nil}))$, where $l \in \gamma(\mathit{ab_list}^\#)$ and $\mathit{sorted}\ l \not\to^*_{\mathcal{R}} \mathit{true}$. In the next abstraction we need to abstract l into an abstract value $l^\#$ such that

$$\mathit{sorted}\ l^\# \not\to^*_{\mathcal{R}^\# \cup \Lambda} \mathit{true}^\#$$

To concretely design such a procedure we have to solve three problems. First, building a new abstract domain by separating only one concrete value at each refinement step may lead to non-termination. For instance in our example, one could refine the abstraction by separating $\mathit{cons}(b, \mathit{cons}(a, \mathit{nil}))$ from the other lists, then separating $\mathit{cons}(a, \mathit{cons}(b, \mathit{cons}(a, \mathit{nil})))$ at the next iteration and so on. If we do this, we end up creating infinitely many abstract values, one for each unsorted list instead of creating one single abstract value for all unsorted lists. Remember that this is the cause of the incompleteness of the PMRS abstraction procedure designed by Ong et al. [OR11] as showed in Section 3.3.1. Thankfully, there are ways to explore the set

of possible abstractions so that the refinement procedure is guaranteed to terminate if there exists a regular abstraction $\mathcal{R}^\#$ and Λ proving or disproving the property. These exploration techniques generally uses SMT solvers to explore the set of possible abstractions $\Sigma^\#$ w.r.t. their cardinal [MKU15]. The second problem is that, given an abstract domain, finding spurious counter-examples is not easy. Indeed, the TRS $\mathcal{R}^\#$ does not explicitly encode the rewriting relation between terms. As a consequence, even if we know that a counter-example may exist, extracting a concrete rewriting sequence leading to the counter-example from $\mathcal{R}^\#$ and Λ is computationally expensive. Moreover, in the above example, the rewritings only depend on a single function: *sorted*. In practice, to find counter-examples on complex programs it is necessary to explore a much larger abstraction of the whole program. Thus, the computational cost for finding a counter-example grows with the size of the program to verify. Finally, the third problem is that the "abstract and refine" procedure presented above is not modular. From a set of constraints that we extract from a spurious counter-example, the next abstraction is recomputed for the *whole* program. As a result the size of the program directly and greatly impacts the efficiency of two main steps of the procedure: the search for counter-examples and the abstraction refinement. This is why we want to define a *modular* procedure able to analyze functions independently and whose *termination* is guaranteed if there exists a regular abstraction satisfying the property. As we will see in Section 5.3.4, contrarily to classical CEGAR, our procedure extracts counter-examples directly from the concrete TRS itself instead of the abstraction to reduce the computation cost. To ensure completeness, we also use an SMT-based technique exploring possible abstractions w.r.t. the size of the abstract domain $\Sigma^\#$. Finally for modularity, we design a *type system* attaching abstraction information to each function (to each symbol of the TRS). In the next section, we show how to translate the above inference problem into a *type inference problem* over regular language types, that will allow us to design a modular inference procedure for those types.

5.3 Regular Language Types

A convenient way of modularizing the abstract interpretation is by introducing a type system to attach abstraction information to each term and symbol. In this approach, the *set of types* is the abstract set of values $\Sigma^\#$ and a term has type τ if it rewrites to τ using $\Delta^\#$ and $\mathcal{R}^\#$. In practice, each type represents a regular language. A *type substitution* $\pi \in \mathcal{X} \to \Sigma^\#$ maps variables to types. From this perspective, we say that the abstract semantics $\mathcal{R}^\#$ is the *type environment* of symbols. In the following, we define a typing judgment $\vdash$ which can be used to give types to patterns, relative to a given $\Lambda, \mathcal{R}^\#$ and substitution π which assigns a type to each variable of the pattern. The typing judgment $\Lambda, \mathcal{R}^\#, \pi \vdash p : \tau$ means that the pattern $p \in T(\Sigma, \mathcal{X})$ can be typed with $\tau \in \Sigma^\#$ using the substitution π, Λ and $\mathcal{R}^\#$.

Definition 5.3.1 (Typing rules)**.** *Let $\Lambda = \langle \Sigma^\#, \Delta^\# \rangle$ be an abstraction of $T(\Sigma)$, $\mathcal{R}^\#$ a* type environment *and π a substitution from $\mathcal{X}$ to $\Sigma^\#$. We define the typing judgment $\vdash$ via the following inference rules. In the rules we use the rewriting rules $p \to \tau$ of $\Delta^\#$ to deduce a type for constructor and function applications. Recall that Σ is the disjoint union of constructor symbols $\mathcal{C}$ and function symbols $\mathcal{F}$.*

$$var\frac{\pi(\underline{x}) = \tau}{\Lambda, \mathcal{R}^\#, \pi \vdash \underline{x} : \tau}$$

$$\textit{constructor}\ \frac{\Lambda, \mathcal{R}^\#, \pi \vdash p_1 : \tau_1 \quad \dots \quad \Lambda, \mathcal{R}^\#, \pi \vdash p_n : \tau_n \quad f \in \mathcal{C} \quad f(\tau_1, \dots, \tau_n) \to \tau \in \Delta^\#}{\Lambda, \mathcal{R}^\#, \pi \vdash f(p_1, \dots, p_n) : \tau}$$

$$\textit{sub-typing}\ \frac{\Lambda, \mathcal{R}^\#, \pi \vdash f(p_1, \dots, p_n) : \tau' \quad \tau' \to \tau \in \Delta^\#}{\Lambda, \mathcal{R}^\#, \pi \vdash f(p_1, \dots, p_n) : \tau}$$

$$\textit{application}\ \frac{\Lambda, \mathcal{R}^\#, \pi \vdash p_1 : \tau_1 \quad \dots \quad \Lambda, \mathcal{R}^\#, \pi \vdash p_n : \tau_n \quad f \in \mathcal{F} \quad f(\tau_1, \dots, \tau_n) \to \tau \in \mathcal{R}^\#}{\Lambda, \mathcal{R}^\#, \pi \vdash f(p_1, \dots, p_n) : \tau}$$

The var *rule uses the type substitution π to type a single variable. The* application *rule follows the abstract semantics $\mathcal{R}^\#$ to type a pattern which can be rewritten. The* constructor *rule uses the abstraction Λ, and in particular $\Delta^\#$, to give a type to patterns built from a constructor symbol. The* sub-typing *rule does the same using the ϵ-transitions of $\Delta^\#$. From this rule we extract a sub-typing relation $\preceq$ where $\tau_1 \preceq \tau_2$ means that $\tau_1 \to^*_{\Delta^\#} \tau_2$.*

The typing judgment definition makes a bridge between the rewriting world and the typing world. Each typing rule correspond to a rewriting step with either $\mathcal{R}^\#$ which becomes the *type environment*, or $\Delta^\#$ which includes the *types definitions*. The following lemma states the correctness of the type system using the abstract interpretation defined in the previous section. Its proof uses the rewriting system $\mathcal{R}^\# \cup \Delta^\#$ as an intermediate step between the type system and abstract interpretation.

Theorem 6 (Correctness)**.** *Let $\mathcal{R}$ be a TRS, $\Lambda = \langle \Sigma^\#, \Delta^\# \rangle$ an abstraction of $T(\Sigma)$ and $\mathcal{R}^\#$ an abstraction of $\mathcal{R}$ over Λ. For all patterns p and types τ (which are abstract values of $\Sigma^\#$) we have:*

$$\Lambda, \mathcal{R}^\#, \pi \vdash p : \tau \quad \iff \quad p\pi \to^*_{\mathcal{R}^\# \cup \Delta^\#} \tau$$

Proof. First, let's show that $\Lambda, \mathcal{R}^\#, \pi \vdash p : \tau$ implies $p\pi \to^*_{\mathcal{R}^\# \cup \Delta^\#} \tau$. We show this with a simple induction on the type inference rules.

- (var) $\Lambda, \mathcal{R}^\#, \pi \vdash \underline{x} : \tau$. Then $\pi(\underline{x}) = \tau$, $p = \underline{x}$ and thus $p\pi = \tau$.
- (constructor) $\Lambda, \mathcal{R}^\#, \pi \vdash f(p_1, \dots, p_n) : \tau$ with $f \in \mathcal{C}$. Then $\Lambda, \mathcal{R}^\#, \pi \vdash p_i : \tau_i$. By induction hypothesis $p_i\pi \to^*_{\mathcal{R}^\# \cup \Delta^\#} \tau_i$. By definition $f(\tau_1, \dots, \tau_n) \to \tau \in \Delta^\#$ so $f(p_1, \dots, p_n)\pi \to^*_{\mathcal{R}^\# \cup \Delta^\#} \tau$.
- (sub-typing) $\Lambda, \mathcal{R}^\#, \pi \vdash f(p_1, \dots, p_n) : \tau$ with $\Lambda, \mathcal{R}^\#, \pi \vdash f(p_1, \dots, p_n) : \tau'$ and $\tau' \to \tau \in \Delta^\#$. By induction hypothesis we have that $f(p_1, \dots, p_n)\pi \to^*_{\mathcal{R}^\# \cup \Delta^\#} \tau'$. Using $\tau' \to \tau \in \Delta^\#$ we conclude that $f(p_1, \dots, p_n)\pi \to^*_{\mathcal{R}^\# \cup \Delta^\#} \tau$.
- (application) $\Lambda, \mathcal{R}^\#, \pi \vdash f(p_1, \dots, p_n) : \tau$ with $f \in \mathcal{F}$. Then $\Lambda, \mathcal{R}^\#, \pi \vdash p_i : \tau_i$. By induction hypothesis, $p_i\pi \to^*_{\mathcal{R}^\# \cup \Delta^\#} \tau_i$. By def. $f(\tau_1, \dots, \tau_n) \to \tau \in \mathcal{R}^\#$ so $f(p_1, \dots, p_n)\pi \to^*_{\mathcal{R}^\# \cup \Delta^\#} \tau$.

Then we show that $p\pi \to^*_{\mathcal{R}^\# \cup \Delta^\#} \tau$ implies $\Lambda, \mathcal{R}^\#, \pi \vdash p : \tau$, by strong induction on the length of the rewriting path. If $p\pi = \tau$ then p is a variable $\underline{x}$ and we have $\pi(x) = \tau$. By the inference rule (var) we have $\Lambda, \mathcal{R}^\#, \pi \vdash \underline{x} : \tau$. Now if $p\pi \to^{k+1}_{\mathcal{R}^\# \cup \Delta^\#} \tau$, we have the following cases:

- $p\pi \to^{k}_{\mathcal{R}^\# \cup \Delta^\#} f(\tau_1, \ldots, \tau_n) \to_{\Delta^\#} \tau$ with $f \in \mathcal{C}$ and $p = f(p_1, \ldots, p_n)$. For all $i = 1 \ldots n$, we have $p_i\pi \to^{k_i}_{\mathcal{R}^\# \cup \Delta^\#} \tau_i$ with $k_i \leq k$. Applying the induction hypothesis we get $\Lambda, \mathcal{R}^\#, \pi \vdash p_i : \tau$ for $i = 1 \ldots n$. The *constructor* rule then gives $\Lambda, \mathcal{R}^\#, \pi \vdash p : \tau$.
- $p\pi \to^{k}_{\mathcal{R}^\# \cup \Delta^\#} \tau' \to_{\Delta^\#} \tau$. Using the induction hypothesis on $p\pi \to^{k}_{\mathcal{R}^\# \cup \Delta^\#} \tau'$ we get that $\Lambda, \mathcal{R}^\#, \pi \vdash p : \tau'$. We can then apply the *sub-typing* rule to get $\Lambda, \mathcal{R}^\#, \pi \vdash p : \tau$.
- $p\pi \to^{k}_{\mathcal{R}^\# \cup \Delta^\#} f(\tau_1, \ldots, \tau_n) \to_{\mathcal{R}^\#} \tau$ with $f \in \mathcal{F}$ and $p = f(p_1, \ldots, p_n)$. For all $i = 1 \ldots n$, we have $p_i\pi \to^{k_i}_{\mathcal{R}^\# \cup \Delta^\#} \tau_i$ with $k_i \leq k$. Thus, we can apply the induction hypothesis and get $\Lambda, \mathcal{R}^\#, \pi \vdash p_i : \tau$ for $i = 1 \ldots n$. Applying the *application* rule gives $\Lambda, \mathcal{R}^\#, \pi \vdash p : \tau$.

□

Remark 5.3.1. *Recall that we encode higher-order programs using the dedicated @ symbol (Section 2.2.3). For instance $@(@(f, x), y)$ is the total application of f on two parameters x and y. A reader familiarized with usual ML-like type systems may notice the lack of arrow type for partial applications such as $@(f, x)$. We do not need them in our type system as they can be represented using regular languages, just like any other type. The ML-type $\tau_1 \to \tau_2$ is represented by the regular language of all terms $t \in \mathcal{T}(\Sigma)$ such that the application $@(t, x)$ rewrites to a term of τ_2 when x rewrites to a term of τ_1.*

Example 5.3.1. *Consider $\mathcal{R}$ defining a (buggy) delete function as*

$$(d1) \qquad delete\ \underline{x}\ nil \to nil$$
$$(d2) \qquad delete\ \underline{x}\ cons(\underline{y}, \underline{l}) \to if(eq(\underline{x}, \underline{y}), delete\ \underline{x}\ \underline{l}, delete\ \underline{x}\ \underline{l})$$

The definition of the equality predicate eq and of the if-then-else symbol if are omitted but present in $\mathcal{R}$. The delete function is supposed to remove every occurrence of $\underline{x}$ in the given list. In the last rule however, we forgot to put $\underline{y}$ back in the list when $\underline{x} \neq \underline{y}$. As a result, this delete function always returns nil. This can be spotted by noticing that there exists an abstraction Λ and $\mathcal{R}^\#$ such that $\Lambda, \mathcal{R}^\#, \pi \vdash delete\ \underline{x}\ \underline{l} : nil^\#$ with $\pi(\underline{x}) = ab^\#$ and $\pi(\underline{l}) = ab_list^\#$. Let the abstraction Λ of $T(\Sigma)$ be defined by

$$\begin{array}{llll} a \to ab^\# & true \to bool^\# & nil \to ab_list^\# & nil \to nil^\# \\ b \to ab^\# & false \to bool^\# & cons(ab^\#, ab_list^\#) \to ab_list^\# & nil^\# \to ab_list^\# \end{array}$$

and let the abstraction $\mathcal{R}^\#$ of $\mathcal{R}$ over Λ be defined by

$$eq(ab^\#,\ ab^\#) \to bool^\#$$
$$if(bool^\#,\ nil^\#,\ nil^\#) \to nil^\#$$
$$delete\ ab^\#\ ab_list^\# \to nil^\#$$

It is then easy to show $\Lambda, \mathcal{R}^\#, \pi \vdash delete\ \underline{x}\ \underline{l} : nil^\#$ by the following typing derivation ($\Lambda, \mathcal{R}^\#, \pi$ are omitted to preserve the readability):

$$\dfrac{\dfrac{\pi(\underline{x}) = ab^\#}{\vdash \underline{x} : ab^\#} \qquad \dfrac{\pi(\underline{l}) = ab_list^\#}{\vdash \underline{l} : ab_list^\#} \qquad delete\ ab^\#\ ab_list^\# \to nil^\# \in \mathcal{R}^\#}{\vdash delete\ \underline{x}\ \underline{l} : nil^\#}$$

It is possible to build an abstraction $\mathcal{R}^\#$ from Λ and $\mathcal{R}$ so as to give the most precise regular type information to each function. The outline of the procedure is as follows: For each function f of $\mathcal{R}$, for every combination of input and output types, make the hypothesis that this combination is valid and try to type both sides of each rule defining f in $\mathcal{R}$ with this combination of types. If it is possible, then the combination is valid. From the set of valid combinations, keep the most precise.

Example 5.3.2. *In the previous example, the combination delete $ab^\#$ $ab_list^\# \to bool^\#$ is not valid for the delete function since it is impossible to type the right-hand-side of the rule $(d1)$ of delete with $bool^\#$. The combination delete $ab^\#$ $ab_list^\# \to nil^\#$ that has been selected is valid. The right-hand side of the rule $(d1)$ of delete has the type $nil^\#$ by the typing derivation:*

$$\frac{nil \to nil^\# \in \Delta^\#}{\vdash nil \ : nil^\#}$$

The typing judgment of the right-hand-side of rule $(d2)$ with $nil^\#$ comes from the following derivation (where we omit the rules of if, eq and delete in $\mathcal{R}^\#$ for readability):

$$\frac{\dfrac{\dfrac{\pi(\underline{x}) = ab^\#}{\vdash \underline{x} : ab^\#} \quad \dfrac{\pi(\underline{y}) = ab^\#}{\vdash \underline{y} : ab^\#}}{\vdash eq(\underline{x}, \underline{y}) : bool^\#} \quad \dfrac{\dfrac{\pi(\underline{x}) = ab^\#}{\vdash \underline{x} : ab^\#} \quad \dfrac{\pi(\underline{l}) = ab_list^\#}{\vdash \underline{l} : ab_list^\#}}{\vdash delete\ \underline{x}\ \underline{l} : nil^\#}}{\vdash if(eq(\underline{x}, \underline{y}),\ delete\ \underline{x}\ \underline{l},\ delete\ \underline{x}\ \underline{l}) : nil^\#}$$

Note that the combination (delete $ab^\#$ $ab_list^\# \to ab_list^\#$) is also valid for the delete function, but is less precise than (delete $ab^\#$ $ab_list^\# \to nil^\#$).

This example illustrates that using this type system and a given abstraction Λ of $T(\Sigma)$, there exists a procedure to build an abstraction $\mathcal{R}^\#$ of $\mathcal{R}$ from Λ that gives precise information about the functions of the program. In this example, it is convenient to have $nil^\#$ as part of Λ, allowing us to type the *delete* function with $nil^\#$ as output, which in turns allowed us to spot the mistake in *delete* (cf. Section 5.4). In general however, Λ does not contain enough information to prove or disprove the wanted property. The abstraction Λ must be *refined* along with $\mathcal{R}^\#$. This transforms the *abstraction inference problem* introduced in the previous section into the following *type inference problem.*

Definition 5.3.2 (Regular Language Type Inference Problem)**.** *Let $\mathcal{R}$ be a term rewriting system and p a pattern of $\mathcal{T}(\Sigma, \mathcal{X})$. Let $\Lambda_* = \langle \Sigma_*^\#, \Delta_*^\# \rangle$ be an initial abstract domain, and $\tau \in \Sigma_*^\#$ a (target) type. A solution to the type inference problem is 1) an abstract domain $\Lambda = \langle \Sigma^\#, \Delta^\# \rangle$ such that $\Sigma^\# \supseteq \Sigma_*^\#$ and $\Delta^\# \supseteq \Delta_*^\#$; 2) an abstraction $\mathcal{R}^\#$ of $\mathcal{R}$ in Λ, complete w.r.t. p and $v^\#$; 3) the set Π of all the type environments π such that $\Lambda, \mathcal{R}^\#, \pi \vdash p : v^\#$.*

Note that type inference problems are usually concerned with finding *some* type substitution π such that $\pi \vdash p : \tau$. In our case however, since we want to capture the entire behavior of the input pattern with regard to the target type τ, we are interested in finding an abstraction containing *all* such type substitutions, in Π. For instance if we consider the pattern $p = xor(\underline{x}, \underline{y})$ with the target type $\tau = true^\#$, a solution to the regular language type inference problem must include $\Lambda, \mathcal{R}^\#$ with Π containing π_1, π_2 such that $\pi_1 = \{x \mapsto true^\#, y \mapsto false^\#\}$ and $\pi_2 = \{x \mapsto false^\#, y \mapsto true^\#\}$. These two substitutions are necessary (and sufficient) to capture the entire behavior of the *xor* function on its input with regard to the target type $true^\#$.

5.3.1 Type Partitions

The type inference procedure we define in this chapter is fundamentally an inductive inference procedure working on the structure of the given input pattern. However having multiple possible type environments for a single pattern and target type is not convenient, since we would need to analyze every case and "split" the analysis at each induction step.

Example 5.3.3. *We want to type the term $xor(f(\underline{x}), \underline{y})$ with target type $true^\#$. The only possible abstraction for xor separating* true *and* false *is the following $\mathcal{R}^\#$:*

$$xor(true^\#, true^\#) \to false^\# \qquad xor(false^\#, true^\#) \to true^\#$$
$$xor(true^\#, false^\#) \to true^\# \qquad xor(false^\#, false^\#) \to false^\#$$

This means that to have $xor(f(\underline{x}), \underline{y}) : true^\#$, we may have either $f(\underline{x}) : true^\#$ or $f(\underline{x}) : false^\#$. We need to analyse both cases, splitting the analysis of f into two branches. Then, depending on the definition of f, we may need to split again each of the two branches to analyze $\underline{x}$, etc. This can result into an exponential blow-up.

Instead, we would like to have one single environment to pass along. The fundamental idea to achieve this is to use *type partition* environments instead of using type environments. A *type partition* is a set of types which represent non-overlapping (regular) sets of values and which together cover the whole domain of values.

Definition 5.3.3 (Type Partition)**.** *Let $\Lambda = \langle \Sigma^\#, \Delta^\# \rangle$ be an abstraction of $\mathcal{T}(\Sigma)$. A set $T = \{ \tau_1, \ldots, \tau_n \} \subseteq \Sigma^\#$ is a* type partition *if for all $i, j \in [1, n]$, $\gamma(\tau_i) \cap \gamma(\tau_j) = \emptyset$ when $i \neq j$, and*

$$\bigcup_{\tau \in T} \gamma(\tau) = \mathcal{T}(\Sigma)$$

We write $P^\#(\Lambda)$ the set of type partitions defined over Λ. Type partitions form a semi-lattice w.r.t. $\sqsubseteq$, where $T_1 \sqsubseteq T_2$ iff for all element $a_1^\#$ of T_1 there exists an element $a_2^\#$ of T_2 such that $\gamma(a_1^\#) \subseteq \gamma(a_2^\#)$. The greatest lower bound of two partitions T_1 and T_2 is the partition T with the fewest elements such that $T \sqsubseteq T_1$ and $T \sqsubseteq T_2$. Following tree automata usage, we call this the product *$T_1 \otimes T_2$.*

Definition 5.3.4 (Type Partition Environment)**.** *Let Λ be an abstraction of $\mathcal{T}(\Sigma)$. A type partition environment is a mapping from variables to type partitions $\mathcal{X} \mapsto P^\#(\Lambda)$. If Π is a set of type environments ($\mathcal{X} \mapsto \Sigma^\#$), then we write $\widetilde{\Pi}$ the type partition environment where for all variable $\underline{x} \in \mathcal{X}$, $\widetilde{\Pi}(\underline{x})$ is the product of all the types associated to $\underline{x}$ in Π:*

$$\widetilde{\Pi}(\underline{x}) = \bigotimes_{\pi \in \Pi} \{ \pi(\underline{x}), \overline{\pi(\underline{x})} \}$$

where $\overline{\tau}$ refers to the element of $\Sigma^\#$ complement of τ such that $\gamma(\overline{\tau}) = \overline{\gamma(\tau)}$ (we assume such element exists since we can always add it otherwise). We call $\{ \pi(\underline{x}), \overline{\pi(\underline{x})} \}$ the partitioned domain *of the variable $\underline{x}$ in π.*

Example 5.3.4. *Let $\Sigma^\# = \{ a^\#, b^\#, c^\#, ab^\#, bc^\# \}$. We assume that $\gamma(ab^\#) = \gamma(a^\#) \cup \gamma(b^\#)$ and $\gamma(bc^\#) = \gamma(b^\#) \cup \gamma(c^\#)$. Let x be a variable, $\pi_1 = \{ \underline{x} \mapsto a^\# \}$, $\pi_2 = \{ \underline{x} \mapsto c^\# \}$, and $\Pi = \{ \pi_1, \pi_2 \}$. The* partitioned domain *of $\underline{x}$ in π_1 is $\{ a^\#, bc^\# \}$ since $\overline{a^\#} = bc^\#$. The* partitioned domain *of $\underline{x}$ in π_2 is $\{ ab^\#, c^\# \}$. The* partitioned domain *of $\underline{x}$ in Π is $\widetilde{\Pi}(\underline{x}) = \{ a^\#, bc^\# \} \otimes \{ ab^\#, c^\# \} = \{ a^\#, b^\#, c^\# \}$.*

To solve the regular language type inference problem of a pattern p for the target type τ, it is sufficient to compute $\widetilde{\Pi}$ instead of Π. Indeed from $\widetilde{\Pi}$ we can extract another set of type environments, $\Pi' = \{\ \pi \mid \forall \underline{x} \in Var(p).\ \pi(\underline{x}) \in \widetilde{\Pi}(\underline{x})\ \}$, for which by construction for all $\pi \in \Pi'$ we have either $\pi \vdash p : \tau$ or $\pi \vdash p : \overline{\tau}$. So Π' is a super-set of Π extracted from $\widetilde{\Pi}$. Hence in the rest of this chapter, we shall focus on finding $\widetilde{\Pi}$ instead of Π. This is the *type partition inference problem.*

Definition 5.3.5 (Regular Language Type Partition Inference Problem)**.** *Let $\mathcal{R}$ be a term rewriting system and p a pattern of $\mathcal{T}(\Sigma, \mathcal{X})$. Let $\Lambda_* = \langle \Sigma_*^\#, \Delta_*^\# \rangle$ be an initial abstract domain, and a target type partition $T \in \mathcal{P}(\Sigma_*^\#)$. A solution to the type partition inference problem is 1) an abstract domain $\Lambda = \langle \Sigma^\#, \Delta^\# \rangle$ such that $\Sigma^\# \supseteq \Sigma_*^\#$ and $\Delta^\# \supseteq \Delta_*^\#$; 2) an abstraction $\mathcal{R}^\#$ of $\mathcal{R}$ in Λ, complete w.r.t. p and every $v^\# \in T$; 3) a type partition environment $\widetilde{\Pi}$, from which we can derive $\Pi = \{\ \pi \mid \forall \underline{x} \in Var(p).\ \pi(\underline{x}) \in \widetilde{\Pi}(\underline{x})\ \}$ containing all the substitutions π such that $\Lambda, \mathcal{R}^\#, \pi \vdash p : v^\#$ with $v^\# \in T$.*

Example 5.3.5. *We return to the previous xor example and type the term $xor(f(\underline{x}), \underline{y})$ with the target type partition $\{true^\#, false^\#\}$. Again, the only possible abstraction for xor in $\mathcal{R}^\#$ is to have*

$$xor(true^\#, true^\#) \to false^\# \qquad xor(false^\#, true^\#) \to true^\#$$
$$xor(true^\#, false^\#) \to true^\# \qquad xor(false^\#, false^\#) \to false^\#$$

But now we only have one single inductive case to type f, which is to consider $f(\underline{x})$ with the type partition $\{\ true^\#, false^\#\ \}$. If we assume that f is the identity function, then the result of the analysis gives $\widetilde{\Pi} = \{\ \underline{x} \mapsto \{\ true^\#, false^\#\ \}, \underline{y} \mapsto \{\ true^\#, false^\#\ \}\ \}$. The resulting Π includes the four possible combinations, which are $\pi_1 = \{\underline{x} \mapsto true^\#, \underline{y} \mapsto true^\#\}$, $\pi_2 = \{\ \underline{x} \mapsto true^\#, \underline{y} \mapsto false^\#\ \}$, $\pi_3 = \{\ \underline{x} \mapsto false^\#, \underline{y} \mapsto true^\#\ \}$ and $\pi_4 = \{\ \underline{x} \mapsto false^\#, \underline{y} \mapsto false^\#\ \}$.

The rest of this section defines an inference procedure for solving this type partition inference problem. We first present the general algorithm. We then present the invariant learning procedure which uses SMT solving for minimizing the size of the abstraction automaton.

5.3.2 Inference Algorithm

This section introduces the type inference algorithm for any TRS given on the next page as Algorithm 2. The main function of this algorithm, *partitions-inference* is in charge of solving the type partition inference problem, that is to find correct type partitions for every variable of a given pattern we wish to type with a given type partition. This algorithm uses the auxiliary functions *analyze-function* and *merge*. The role of the *merge* function will be detailed below. The role of *analyze-function* is to compute the *type partitions signature* of a given symbol f for a given output type partition T. This correspond to the input type partitions needed for every term $t_1, \ldots, t_n$ to type $f(t_1, \ldots, t_n)$ with the given output type partition T. For instance, the expected type partition signature of *xor* of Example 5.3.5 for the output partition $\{true^\#, false^\#\}$ is $(\{true^\#, false^\#\}, \{true^\#, false^\#\}) \to \{true^\#, false^\#\}$.

Definition 5.3.6 (Type Partitions Signature)**.** *Let $\mathcal{R}$ be a TRS, $\Lambda = \langle \Sigma^\#, \Delta^\# \rangle$ an abstract domain, and $\mathcal{R}^\#$ an abstraction of $\mathcal{R}$ defined over Λ. Let f be a symbol of Σ. Let $T, T_1, \ldots, T_n$ be type partitions over $\Sigma^\#$. We say that $(T_1, \ldots, T_n) \to T$ is*

```
1 function partitions-inference
    input  : A TRS R, an abstraction Λ*, pattern p, and type partition T.
    output : A solution Λ, R^# and Π̃ to the partition inference problem of
             R, Λ*, p and T.
2   match p with
3     when x then
4       return { x ↦ T }
5     when f(p1, ..., pn) then
6       let Λ', R'^#, (T1, ..., Tn) → T = analyze-function(R, Λ*, f, T);
7       foreach pattern pi do
8         let Λi, Ri^#, Π̃i = partitions-inference(R'^#, Λ', pi, Ti);
9       return merge(Λ1, ..., Λn, R1^#, ..., Rn^#, Π̃1, ..., Π̃n)
```

Algorithm 2: Inference of Type Partitions

a type partition signature *for* f *in* $\Lambda, \mathcal{R}^\#$ *if for all* π*, all patterns* p_i*, and* $\tau_i \in T_i$*, such that* $\pi, \Lambda, \mathcal{R}^\# \vdash p_i : \tau_i$*, for* $i = 1 \dots n$*, then there exists* $\tau \in T$ *such that* $\pi, \Lambda, \mathcal{R}^\# \vdash f(p_1, \dots, p_n) : \tau$*.*

The second auxiliary function *merge* deduces the final solution $\Lambda = \langle \Sigma^\#, \Delta^\# \rangle, \mathcal{R}^\#, \widetilde{\Pi}$ of the type partition inference problem from all the $\Lambda_i = \langle \Sigma_i^\#, \Delta_i^\# \rangle, \mathcal{R}_i^\#$ found for all sub-patterns p_i. It is defined as the smallest sets such that 1) $\Sigma^\# \supseteq \Sigma_1^\# \cup \dots \cup \Sigma_n^\#$; 2) $\Delta^\# \supseteq \Delta_1^\# \cup \dots \cup \Delta_n^\#$; 3) $\mathcal{R}^\# = \mathcal{R}_*^\# \cup \mathcal{R}_1^\# \cup \dots \cup \mathcal{R}_n^\#$; 4) $\widetilde{\Pi} = \widetilde{\Pi}_1 \cup \dots \cup \widetilde{\Pi}_n$ where the union of two type partition environments $\widetilde{\Pi} = \widetilde{\Pi}_1 \cup \widetilde{\Pi}_2$ is defined for each variable of $Dom(\widetilde{\Pi}_1) \cup Dom(\widetilde{\Pi}_2)$ by

$$\widetilde{\Pi}(\underline{x}) = \begin{cases} \widetilde{\Pi}_1(\underline{x}) & \text{if} \quad \underline{x} \notin Dom(\widetilde{\Pi}_2) \\ \widetilde{\Pi}_2(\underline{x}) & \text{if} \quad \underline{x} \notin Dom(\widetilde{\Pi}_1) \\ \widetilde{\Pi}_1(\underline{x}) \otimes \widetilde{\Pi}_2(\underline{x}) & \text{otherwise} \end{cases}$$

Lemma 22 (Merge)**.** *Let* $p = f(p_1, \dots, p_n)$*. Let* $(T_1, \dots, T_n) \to T$ *be a type partitions signature for the symbol* f *in* $\Lambda_* = \langle \Sigma_*^\#, \Delta_*^\# \rangle$*,* $R_*^\#$*. Assume we have a solution* $\Lambda_i, R_i^\#, \widetilde{\Pi}_i$ *to the type partition inference problem with* $\mathcal{R}, \Lambda_*, p_i, T_i$*. If* $\Lambda = \langle \Sigma^\#, \Delta^\# \rangle, \mathcal{R}^\#$ *and* $\widetilde{\Pi}$ *(the output of merge) are the smallest sets such that 1)* $\Sigma^\# \supseteq \Sigma_1^\# \cup \dots \cup \Sigma_n^\#$*; 2)* $\Delta^\# \supseteq \Delta_1^\# \cup \dots \cup \Delta_n^\#$*; 3)* $\mathcal{R}^\# = \mathcal{R}_*^\# \cup \mathcal{R}_1^\# \cup \dots \cup \mathcal{R}_n^\#$*; 4)* $\widetilde{\Pi} = \widetilde{\Pi}_1 \cup \dots \cup \widetilde{\Pi}_n$ *is defined in* Λ*. Then* $\Lambda, \mathcal{R}^\#$ *and* $\widetilde{\Pi}$ *are a solution to the type partition inference problem of* p *with* T *in* $\Lambda_*, \mathcal{R}$*.*

Proof. We prove that for all $\pi \in \Pi$, there exists a type $\tau \in T$ such that $\pi, \Lambda, \mathcal{R}^\# \vdash p : \tau$. Let $\pi \in \Pi$. By definition of the union of type partition environments, for each i, for each $\pi_i \in \Pi_i$, for each variable $x \in Var(p_i)$ we have either $\pi(\underline{x}) = \pi_i(\underline{x})$ or there exists an epsilon transition $\pi(\underline{x}) \to \pi_i(\underline{x}) \in \Delta^\#$. Hence we have $\tau_i \in T_i$ such that $\pi, \Lambda, \mathcal{R}^\# \vdash p_i : \tau_i$. Now by definition, since $(T_1, \dots, T_n) \to T$ is a signature of the symbol f in $\mathcal{R}_*^\#$ with $\mathcal{R}_*^\# \subseteq \mathcal{R}^\#$, we know that there exists $\tau \in T$ such that $\pi, \Lambda, \mathcal{R}^\# \vdash p : \tau$. □

Assuming that *analyze-function* is correct, we can then prove that the whole *partitions-inference* algorithm is correct.

Theorem 7. *Let* $\mathcal{R}$ *be a TRS,* $\Lambda_* = \langle \Sigma_*^\#, \Delta_*^\# \rangle$ *be an initial abstract domain,* p *a pattern with* T *a partition of* $\Sigma_*^\#$*. If* $\Lambda, \mathcal{R}^\#, \widetilde{\Pi} =$ *partitions-inference*$(\mathcal{R}, \Lambda_*, p, T)$ *then it is a solution to the type partition inference problem of* p *with* T*,* Λ_* *and* $\mathcal{R}$*.*

Proof. By induction on the pattern p.

- If $p = \underline{x}$, then the output of *partitions-inference* is $\Lambda = \Lambda_*$, $\mathcal{R}^\# = \emptyset$ and $\widetilde{\Pi} = \{\underline{x} \mapsto T\}$. We have (1) $\Sigma^\# \subseteq \Sigma^\#_*$ by construction (2) $\mathcal{R}^\#$ is an abstraction of $\mathcal{R}$ since it is empty (3) for all $\pi \in \Pi$, if we take $\tau = \pi(\underline{x})$ then $\pi, \Lambda, \mathcal{R}^\# \vdash \underline{x} : \pi$.

- In the other case, if $p = f(p_1, \ldots, p_n)$, then we have $\Lambda', \mathcal{R}'^\#, (T_1, \ldots T_n) \to T =$ analyze-function$(\mathcal{R}, \Lambda_*, f, T)$. Since we assume that *analyze-function* is correct, we known that $(T_1, \ldots, T_n) \to T$ is the type partitions signature of f in $\Lambda', \mathcal{R}'^\#$. We write $\Sigma^\#_i, \Lambda^\#_i, R^\#_i$ and $\widetilde{\Pi}_i$ for the outputs of *partitions-inference*$(R, \Lambda', R'^\#, p_i, T_i)$ for all i. By hypothesis of induction, we know those are solutions to the type partition inference problem with $\mathcal{R}, \Lambda', R'^\#, p_i, T_i$ as inputs. Then by definition of *merge* and Lemma 22, we conclude that the returned $\Sigma^\#, \Lambda, R^\#, \widetilde{\Pi}$ are solutions to the type partition inference problem of p with Λ_*, R and T.

□

This algorithm is independent of the actual implementation of *analyze-function*. We detail in Section 5.3.3 a direct implementation of *analyze-function* that efficiently computes the input partitions signature, but only for non-recursive functions. In Section 5.3.4, we propose an implementation of *analyze-function* based on a regular language invariant learning procedure that can analyze any function, including recursive functions. In our implementation (cf. Section 5.4), we combine the two versions to be more efficient.

5.3.3 Non-Recursive TRS

In the case of non-recursive TRSs, it is possible to directly find a solution to the type partition inference problem, without relying on any learning technique needed in the general case. We give the following implementation of *analyze-function* as Algorithm 3 that works in this specific setting. This algorithm makes the distinction between two cases: constructor symbols for which no rules of $\mathcal{R}$ apply, and functional symbols for which at least one rule apply. The constructor case is handled using a *projection operator* whose idea is as follows: If a regular language $\mathcal{L}$ is shaped as $\{f(t_1, \ldots, t_n) \mid t_1 \in \mathcal{L}_1, \ldots, t_n \in \mathcal{L}_n\}$, then the projection of $\mathcal{L}$ on the symbol f at position i is $\mathcal{L}_i$. For instance, if we consider the language of lists of As and Bs defined by $\mathcal{L} = \{nil, cons(A, nil), cons(B, nil), cons(A, cons(A, nil)), \ldots\}$, then $proj(\mathcal{L}, cons, 1)$ is $\{A, B\}$ and $proj(\mathcal{L}, cons, 2) = \mathcal{L}$. In our case, since a type basically represents a regular language, we directly define the projection on types. We then further extend this definition to partitions of types.

Definition 5.3.7 (Type Projection)**.** *Let $\Lambda = \langle \Sigma^\#, \Delta^\# \rangle$. The projection of the type $\tau \in \Sigma^\#$ on a given symbol f at position i is*

$$proj(\tau, f, i) = unify\ \{\tau_i \mid f(\tau_1, \ldots, \tau_i, \ldots, \tau_n) \to^* \tau \in \Delta^\#\}$$

where unify is the type unification function. We write $unify(\{\tau_1, \ldots, \tau_n\})$ for the most precise type τ w.r.t. $\preceq$ such that for all $i, 1 \leq i \leq n$, $\tau_i \preceq \tau$. [3] *By convention, we write $proj(\tau, f, i) = \bot$ when the projection is not defined, that is when f is not*

[3] Note that we can always build such unified type. In the worst case we can unify two arbitrary types by introducing a new abstract value $any^\#$ such that $\gamma(any^\#) = \mathcal{T}(\mathcal{F})$

function *analyze-function*
  input : A TRS $\mathcal{R}$, an initial abstract domain Λ_*, a symbol $f \in \mathcal{F}^n$ and an output type partition T
  output : An abstraction $\mathcal{R}^\#$ and a partitions signature $(T_1, \ldots, T_n) \to T$ of f in $\mathcal{R}^\#$
  Let $\mathcal{R}_f$ be the set of rules applicable on f;
  if $\mathcal{R}_f$ *is empty* **then**
    `/* f is a constructor symbol, for a value */`
    return $(proj(T, f, 1), \ldots, proj(T, f, n))$
  else
    `/* f is a function symbol that can be rewritten */`
    Let $K \leftarrow |\mathcal{R}_f|$;
    foreach *rule;*
    $rule_k = f(p_1, \ldots, p_n) \to r \in \mathcal{R}$ **do**
      Let $\Lambda_k, \mathcal{R}_k^\#, (T_{k,1}, \ldots, T_{k,n}) \leftarrow$ *rule-signature*$(\mathcal{R}, \Lambda_*, rule_k, T)$;
    Let $\Lambda \leftarrow \Lambda_1 \cup \cdots \cup \Lambda_K$;
    Let $\mathcal{R}^\# \leftarrow \mathcal{R}_1^\# \cup \cdots \cup \mathcal{R}_K^\#$;
    return *merge-rules*$(\Lambda, \mathcal{R}^\#, (T_{1,1}, \ldots, T_{1,n}), \ldots, (T_{K,1}, \ldots, T_{K,n}))$

Algorithm 3: Non-Recursive Function Analysis

recognized by τ in $\Delta^\#$. If we consider the type partition $T = \tau_1, \ldots, \tau_n$, the projection of T on the symbol f at position i is defined as

$$proj(T, f, i) = remove_\bot\{proj(\tau_1, f, i), \ldots, proj(\tau_n, f, i)\}$$

where $remove_\bot$ filters out the $\bot$ element from the set. Note that $proj(T, f, i)$ is also a type partition.

The projection operator (Definition 5.3.7) is used in the algorithm to directly find the type partition signature of a constructor symbol. We can show that picking the projection of every sub-position of the constructor symbol on the target partition gives us the type partition signature of this symbol for this target type partition.

Lemma 23 (Projection on Constructor)**.** *Let $p = f(p_i, \ldots, p_n)$ be a non-functional pattern (without function symbol) and T a type partition. Then $(T_1, \ldots, T_n) \to T$ where $T_i = proj(T_i, f, i)$ is a type partitions signature of f.*

Proof. Let $(\tau_1, \ldots, \tau_n)$ be a combination of $(T_1, \ldots, T_n)$. Let $\tau \in T$ be the type such that for all i, $\tau_i = proj(\tau, f, i)$. Let π be a type environment such that $\Lambda^\#, \mathcal{R}^\#, \pi \vdash p_i : \tau_i$. By definition of the projection, we have $f(\tau_1, \ldots, \tau_n) \to^*_{\Lambda^\#} \tau$. We can then conclude by construction of the type judgment rules that we have $\Lambda^\#, \mathcal{R}^\#, \pi \vdash p : \tau$. □

Finding the type partitions signature of a functional symbol is a bit harder since it involves following the rewriting rules associated to this symbol. To do that that we first introduce the notion of "rule type partitions signature". Informally, for a given rule of a symbol, a rule type partitions signature is a type partitions signature for the symbol that is correct when considering this symbol's rule alone. This allows us to analyze every rule separately. We then show that we can merge all the rule signatures into one symbol signature.

Example 5.3.6. *Consider the following rules encoding the if-then-else:*

$$ite(true, \underline{x}, \underline{y}) \to \underline{x} \qquad ite(false, \underline{x}, \underline{y}) \to \underline{y}$$

Consider the type $nat^\#$ *with two sub-types* $even^\#$ *and* $odd^\#$ *and the target type partition* T *defined as* $T = \{even^\#, odd^\#\}$. *The* rule signature *of the first rule for this target type partition would be* $(\{true^\#, false^\#\}, \{even^\#, odd^\#\}, \{nat^\#\}) \to T$. *For this rule, there is no constraints on* $\underline{y}$, *so the most general partition* $\{nat^\#\}$ *is given. Conversely, the* rule signature *of the second rule for the same target type partition would be* $(\{true^\#, false^\#\}, \{nat^\#\}, \{even^\#, odd^\#\}) \to T$. *We can then merge both rule-signatures into one input partitions signature for the symbol if by using the product of each sub-partitions. The resulting input partitions signature is* $(\{true^\#, false^\#\}, \{even^\#, odd^\#\}, \{even^\#, odd^\#\}) \to T$.

The following gives the formal definition of *rule signatures* and shows how to merge those rules signatures into a symbol signature for a given target type partition. Note that we do not explicitly give an implementation of the *merge-rules* function used in the algorithm and assume it strictly follows the merge instructions given by Lemma 24.

Definition 5.3.8 (Rule Type Partitions Signature)**.** *Let* R *be a TRS,* $\Lambda = \langle \Sigma^\#, \Delta^\# \rangle$ *an abstract domain, and* $R^\#$ *an abstraction of* R *defined over them. Let* T *be a type partition over* $\Sigma^\#$. *Let* $l \to r$ *be a rule of* R *such that* $l = f(p_1, \ldots, p_n)$. *We say that* $(T_1, \ldots, T_n) \to T$ *is the type partitions signature for this rule in* $\Lambda, R^\#$ *if for all* $\tau_1, \ldots, \tau_n$ *where* $\tau_i \in T_i$ *there exists* $\tau \in T$ *such that for all* π *such that for all* i, $\pi, \Sigma^\#, \Lambda^\#, R^\# \vdash p_i : \tau_i$ *then we have* $\pi, \Sigma^\#, \Lambda^\#, R^\# \vdash r : \tau$.

Lemma 24 (From rule signatures to symbol signatures)**.** *Let* $\mathcal{R}$ *be a TRS,* $\Lambda = \langle \Sigma^\#, \Delta^\# \rangle$ *an abstract domain, and* $\mathcal{R}^\#$ *an abstraction of* R *defined over them. Let* T *be a type partition over* $\Sigma^\#$ *and* f *a symbol associated with a non-empty set of rules* $R_f \subseteq R$. *Let us index each rule of* R_f *by* k *from* 1 *to* $|R_f|$ *so that each rule is of the form* $f(p_{k,1}, \ldots, p_{k,n}) \to r_k$. *For all rule* k *let's write* $(T_{k,1}, \ldots, T_{k,n})$ *the input signature of this rule for* T *in* $\Lambda, R^\#$. *Let us define* $(T_1, \ldots, T_n)$ *such that*

$$\forall i, 1 \leq i \leq n. \quad T_i = T_{1,i} \otimes \ldots \otimes T_{|R_f|,i}$$

Let us write $T_{k,i}(\tau)$ *the type* $\tau' \in T_{k,i}$ *such that* $\tau \preceq \tau'$. *For all* $\tau \in T_i$, *by definition this type should exists and be unique. Let us also write* $\Pi_k(\tau_1, \ldots, \tau_n) = \{ \pi \mid \forall i. \quad \pi, \Sigma^\#, \Lambda^\#, R^\# \vdash p_{k,i} : T_{k,i}(\tau_i) \}$. *If we define* $\mathcal{R}'^\#$ *as*

$$\mathcal{R}^\# \cup \{ f(\tau_1, \ldots, \tau_n) \to \tau \mid \forall i. \tau_i \in T_i \quad \wedge \quad \forall k. \forall \pi \in \Pi_k(\tau_1, \ldots, \tau_n).\ \pi, \Lambda, R^\# \vdash r_k : \tau \}$$

then $\mathcal{R}'^\#$ *is an abstraction of* $\mathcal{R}$ *such that* $(T_1, \ldots, T_n) \to T$ *is the type partitions signature of the symbol* f.

Proof. For all $f'(\tau_1, \ldots, \tau_n) \to \tau \in \mathcal{R}'^\#$. If $f \neq f'$, soundness and completeness are ensured by including $\mathcal{R}^\#$ in $\mathcal{R}'^\#$. Otherwise if $f = f'$, we first show that $\mathcal{R}'^\#$ is a sound abstraction of $\mathcal{R}$. Let $t = f(t_1, \ldots, t_n)$ such that $t_i \in \gamma(\tau_i)$. Let us consider $u \in IRR(R)$ such that $t \to^*_{\mathcal{R}} u$. Since we consider functional TRSs, there exists a rule $f(p_{k,1}, \ldots, p_{k,n}) \to r_k \in \mathcal{R}$ and a substitution $\sigma : X \mapsto IRR(\mathcal{R})$ such that $t_i = p_{k,i}\sigma$ and $t \to_{\mathcal{R}} r\sigma \to^*_{\mathcal{R}} u$. By definition we have $p_{k,i}\sigma \in \gamma(\tau)$, but also by construction, $p_{k,i}\sigma \in \gamma(T_{k,i}(\tau_i))$. Hence by definition there exists π such that for all i, $\pi, \Lambda, R^\# \vdash p_{k,i} : T_{k,i}(\tau_i)$. Note that by construction, $\pi \in \Pi_k(\tau_1, \ldots, \tau_n)$. Since

$(T_{k,1}(\tau_1), \ldots, T_{k,n}(\tau_n))$ is a combination of the rule signature $(T_{k,1}, \ldots, T_{k,n}) \to T$ of the rule k with $\Lambda, R^\#$, we know there exist a type $\tau \in T$ such that $\pi, \Lambda, R^\# \vdash r_k : \tau$. Since $\mathcal{R}^\#$ is an abstraction of $\mathcal{R}$ for Λ, by definition this means that $u \in \gamma(\tau)$. Now let's show $\mathcal{R}'^\#$ is a complete abstraction of $\mathcal{R}$. Let us consider any $(\tau_1, \ldots, \tau_n)$ where $\tau_i \in T_i$. First note that $\mathcal{R}_f$ is not empty. Since the TRS is complete, there exists at least one rule f such that $\Pi_k(\tau_1, \ldots, \tau_n)$ is not empty. Since $\mathcal{R}^\#$ is an abstraction of $\mathcal{R}$, there exists a unique τ such that for all rule k and all $\pi \in \Pi_k(\tau_1, \ldots, \tau_n)$, $\pi, \Lambda, R^\# \vdash r_k : \tau$. Then by definition there exists the rule $f(\tau_1, \ldots, \tau_n) \to \tau$ in $R'^\#$. □

The last missing piece left is the *rule-signature* algorithm that will allow us to actually compute the signature of each rule. For the considered rule $l \to r$, it will type the right-hand-side r of the rule in order to find the type partition of each variable (for this rule), and deduce a type for the left-hand-side from the found type partitions.

```
1 function rule-signature
    input  : A TRS R, an initial abstract domain Λ*, a rule
             f(p1, ..., pn) → r ∈ R and an output type partition T
    output : An abstraction R# and the rule signature (T1, ..., Tn) → T of
             the given rule in R#
2   Let Λ, R#, Π̃ ← partitions-inference(R, Λ*, r, T);
3   foreach sub-pattern pi do
4     Let Λi, Ti ← bottom-up-type-pattern(Λ, pi, Π̃);
5   Let Λ' ← Λ1 ∪ ··· ∪ Λn;
6   return Λ', R#, (T1, ..., Tn)
```

The goal of the *bottom-up-type-pattern* function is to give a fitting type partition to the left-hand-side of the rule from a given type partition environment. This type can easily be built by typing the pattern from bottom to top.

Example 5.3.7 (Bottom-Up Typing)**.** *Let us imagine we wish to type the irreducible pattern* $cons(s(\underline{x}), \underline{y})$ *with the type partition environment* $\widetilde{\Pi} = \{\underline{x} \mapsto \{0^\#, N+^\#\}, \underline{y} \mapsto \{[nat\ list]^\#\}\}$ *where* $\Delta^\#$ *is*

$$\begin{aligned} 0 &\to 0^\# & nil &\to [nat\ list]^\# \\ s(0^\#) &\to N+^\# & cons(0^\#, [nat\ list]^\#) &\to [nat\ list]^\# \\ s(N+^\#) &\to N+^\# & cons(N+^\#, [nat\ list]^\#) &\to [nat\ list]^\# \end{aligned}$$

Since $\widetilde{\Pi}(\underline{x}) = \{0^\#, N+^\#\}$ *we can create two new types* $[s(0)]^\#$ *and* $[s(N+)]^\#$ *in* $\Sigma^\#$ *with the associated rules* $s(0^\#) \to [1]^\#$ *and* $s(N+^\#) \to [2+]^\#$ *in* $\Delta^\#$ *such that a valid type partition for* $s(\underline{x})$ *is* $\{[1]^\#, [2+]^\#\}$*. We also need to add the sub-typing rules* $[1]^\# \to N+^\#$ *and* $[2+]^\# \to N+^\#$ *in* $\Delta^\#$*. Now that we have a partition for* $s(\underline{x})$*, we can continue up and type the whole pattern. This gives us the partition* $\{[cons(1, nat\ list)]^\#, [cons(2+, nat\ list)]^\#\}$ *associated to the rules*

$$\begin{aligned} cons([1]^\#, [nat\ list]^\#) &\to [cons(1, nat\ list)]^\# \\ cons([2+]^\#, [nat\ list]^\#) &\to [cons(2+, nat\ list)]^\# \end{aligned}$$

The actual specification of the *bottom-up-type-pattern* function is given by the following definition.

Definition 5.3.9. *Bottom-Up Typing Let $\mathcal{R}$ be a functional TRS. Let Λ be an abstract domain and $\mathcal{R}^\#$ an abstraction of $\mathcal{R}$ defined over it. Let p be an* irreducible *pattern and $\widetilde{\Pi}$ a type partition environment. Let us name $\widetilde{\Pi}|_p = \{x \mapsto \widetilde{\Pi}(x) \mid x \in Var(p)\}$ the projection of $\widetilde{\Pi}$ on the variables of p (we discard other variables). The function bottom-up-type-pattern returns Λ', T such that there exists a bijection $\xi : \Pi|_p \mapsto T$ such that for all $\pi \in \Pi|_p$, $\quad \pi, \Lambda, \mathcal{R}^\# \vdash p : \tau(\pi)$.*

From this definition, it is possible to show that the *rule-signature* algorithm is correct if we suppose the *partitions-inference* algorithm correct. We later see that it is easy to prove this assumption while considering non-recursive TRSs.

Lemma 25 (*rule-signature* is correct)**.** *Let $\mathcal{R}$ be a TRS, Λ an abstract domain and $\mathcal{R}^\#$ an abstraction of $\mathcal{R}$ on this domain. Let $l \to r$ be a rule of $\mathcal{R}$, and T a type partition. Let $\Lambda', \mathcal{R}'^\#, (T_1, \ldots, T_n) \to T =$ rule-signature$(\Lambda, \mathcal{R}^\#, l \to r, T)$. Then $(T_1, \ldots, T_n)$ is an input signature for the rule for T over $\Lambda', \mathcal{R}'^\#$.*

Proof. Let $l = f(p_1, \ldots, p_n)$. Let $\Lambda_r^\#, \mathcal{R}_r^\#, \widetilde{\Pi_r}$ be the result of *partitions-inference* called with $\mathcal{R}, \Lambda, \mathcal{R}^\#, r$ and T. We can show by induction on the cardinal of $\mathcal{R}$ that this recursive call to *partitions-inference* gives a valid solution to the type partition inference problem for the right-hand side. First, if $\mathcal{R}$ is empty, then *partitions-inference* is never called back from *analyze-function*. If $\mathcal{R}$ is not empty, *partitions-inference* is called after using one rule $l \to r$ of $\mathcal{R}$. Then, since $\mathcal{R}$ is not recursive, we can consider that the result of *partitions-inference* will be equivalent if called with $\mathcal{R} - \{l \to r\}$. By hypothesis of induction on the cardinal of $\mathcal{R}$, it is a valid solution. Now for each sub pattern p_i of the left-hand side, let's call $\Lambda_i, T_i =$ *bottom-up-type-pattern*$(\Lambda_r^\#, \mathcal{R}_r^\#, \widetilde{\Pi_r}, p_i)$. Let us name $\Lambda' = \Lambda_1^\# \cup \ldots \cup \Lambda_n^\#$ Let us show that $(T_i, \ldots, T_n) \to T$ is a type partitions signature for the given rule in $\Lambda', \mathcal{R}_r^\#$. Let $(\tau_1, \ldots, \tau_n)$ be a combination of $(T_1, \ldots, T_n)$. Let us prove that there exists $\tau \in T$ such that for all π such that for all i, $\pi, \Lambda', \mathcal{R}_r^\# \vdash p_i : \tau_i$ then we have $\pi, \Lambda', \mathcal{R}_r^\# \vdash p : \tau$. First we prove that there is only one π such that $\pi, \Lambda', \mathcal{R}_r^\# \vdash p_i : \tau_i$. Let π_1, π_2 be two different type environments following this property. Let observe that because ξ_i is a bijection we should have $\pi_1|_{p_i} = \pi_2|_{p_i}$, for all i. This can only happen if $\pi_1 = \pi_2$ which contradicts our initial hypothesis that $\pi_1 \neq \pi_2$. Now we prove that for this π, there exists $\tau \in T$ such that $\pi, \Lambda', \mathcal{R}_r^\# \vdash p : \tau$. By definition of *bottom-up-type-pattern* we have that $\pi \in \Pi_r$. Hence by construction there exists $\tau \in T$ such that $\pi, \Lambda', \mathcal{R}_r^\# \vdash r : \tau$. □

Note that this definition of *analyze-function* only works for non-recursive functions. Despite the amount of proofs needed to prove it correct, this algorithm combined with *partitions-inference* only involves some basic operations on tree automata to directly find a solution to the type partition inference problem. In the next section, we introduce a new definition of *analyze-function* intended for recursive functions.

5.3.4 Invariant Learning

The main difficulty in functional program analysis is recursion, or in our case, the analysis of functional symbols defined with mutually recursive rewriting rules. In this section, we define an implementation of *analyze-function* based on an original invariant learning procedure. For a given (recursive function) symbol and target type partition, this procedure finds correct input regular languages partitions that completes the symbol's type partitions signature. It follows the standard outline of a counter-example guided abstraction refinement (CEGAR) procedure, where

a rough abstraction is iteratively refined using constraints learned from previous iterations. Those new constraints are defined by finding a *spurious* counter-example generated because of a faulty previous abstraction. Constraints are accumulated until no *spurious* counter-example can be found in which case the invariant has been found, or by finding a real counter-example. In this section we show how to use the *Tree Automata Completion Algorithm* [GR10] to adapt this family of techniques to Term Rewriting Systems and Regular Languages, allowing us to learn recursive symbol partitions signatures.

```
1  function analyze-function
       input  : An input TRS R, an initial abstraction Λ*, a function symbol
                f, and a target type partition T.
       output : An abstraction Λ ⊇ Λ*,an abstraction R# of R over Λ and a
                type partitions signature (T1,...,Tn) → T of f in R#
2      let L = {f(t1,...,tn) | t1...tn ∈ IRR(R)};
3      let A0 = finite-subset(L);
4      let i = 0;
5      forever
           /* (1) Tree Automata Completion                    */
6          let Ai* = tree-automata-completion(Ai, R);
           /* (2) Counter-example/constraints generation      */
7          let φ = S(Ai*, T);
8          if φ is unsatisfiable then return counter-example ;
           /* (3) Abstraction                                 */
9          let Ai# = φ(Ai*);
           /* (4) Validity Check / Termination                */
10         if Ai# is R-closed and IRR-complete then
11             let R# = { f(τ1,...,τn) → τ | f ∈ F };
12             let Δ# = { f(τ1,...,τn) → τ | f ∈ C };
13             let Λ = ⟨states_of(A#), Δ#⟩;
14             let Ti = { τi | f(...,τi,...) → τ ∈ R# };
15             return Λ, R#, (T1,...,Tn) → T
16         else
17             Ai+1 = grow(Ai, Ai#);
18             i = i + 1;
```

Algorithm 4: Invariant Learning Procedure

To find a type partition signature for a symbol f with the target type partition T, the procedure showed as Algorithm 4 computes a series of tree automata $\mathcal{A}_0^\#, \mathcal{A}_1^\#, \ldots$ according to the following outline: 1. We start by using the Tree Automata Completion Algorithm on $\mathcal{R}$ and an automaton $\mathcal{A}_0$ recognizing a finite subset $\mathcal{L}_0$ of the language $\mathcal{L} = \{f(t_1, \ldots, t_n) \mid t_1 \ldots t_n \in \mathit{IRR}(\mathcal{R})\}$. The result is an automaton $\mathcal{A}_0^*$ recognizing $\mathcal{R}^*(\mathcal{L}_0)$. This automaton is guaranteed to exist if $\mathcal{R}$ is terminating [GR10]. 2. We then check for any counter-example: any input term that violate the target type partition T by rewriting to two different types of the partition. For now, no abstraction has been done. If a counter-example is found in $\mathcal{A}_0^*$ recognizing $\mathcal{R}^*(\mathcal{L}_0)$, it is a real counter-example, no such signature exists for f and the property is disproved. Otherwise, using $\mathcal{A}_0^*$ and T, we build a set of disequality constraints over its states. 3. We then merge the states of $\mathcal{A}_0^*$ according to those disequality constraints to build

an *abstraction* $\mathcal{A}_0^\#$ as the *smallest automaton* respecting those constraints. 4. If $\mathcal{A}_0^\#$ is complete and $\mathcal{R}$-closed (cf. Definition 5.3.11), then we know it contains a valid abstraction of $\mathcal{R}$ and Σ, and we can extract a signature for the symbol f. If not, we need to start over from (1) with a new automaton $\mathcal{A}_1$, recognizing $\mathcal{L}_1$ another finite subset of $\mathcal{L}$ such that $\mathcal{L}_1 \supset \mathcal{L}_0$. In Algorithm 4, this is done by the *grow* function (cf. Definition 5.3.13). We continue until a counter-example is found, or a type partitions signature is found.

Example 5.3.8. *Consider again the TRS defined by*

$$even(0) \to true \quad odd(0) \to false \quad even(s(\underline{x})) \to odd(\underline{x}) \quad odd(s(\underline{x})) \to even(\underline{x})$$

We want to find a partitions signature for the symbol even for the target type partition $T = \{true^\#, false^\#\}$*. The language* $\mathcal{L}$ *generated by this symbol is* $\mathcal{L} = \{even(n) \mid n \in \mathcal{N}\}$*. We start (1) with* $\mathcal{A}_0$ *recognizing* $\mathcal{L}_0 = \{even(0)\}$*, and use the tree automata completion algorithm on it, which gives us a new tree automaton* $\mathcal{A}_0^*$ *recognizing the reachable terms* $\{even(0), true\}$ *and defined by:*

$$(a) \qquad 0 \to q_0 \qquad even(q_0) \to q_{e0} \qquad true \to q_t \qquad q_t \to q_{e0}$$

There is (2) no violation of T *yet. Let* $S(\mathcal{A}_0^*, T)$ *be the set of constraints to consider to (3) build an abstraction from* $\mathcal{A}_0^*$*, in which terms typed with different types of* T *are recognized by different states in the abstraction. For now, the only constraints to consider are well-typedness constraints:* $S(\mathcal{A}_0^*, T) = \{q_0 \neq q_{e0}, q_t \neq q_0\}$*. Indeed,* q_0 *recognizes a fragment of type* $\mathcal{N}$ *and* q_t, q_{e0} *a fragment of type bool, they must not be merged. We then use an SMT solver to build the smallest renaming* ϕ *from* $\mathcal{Q}$ *to* $\Sigma^\#$ *respecting the given constraints. The result,* $\phi(\mathcal{A}_0^*)$*, is as follows (for the sake of readability we give comprehensible names to the new elements of* $\Sigma^\#$*):*

$$0 \to nat^\# \qquad even(nat^\#) \to true^\# \qquad true \to true^\#$$

This automaton is not complete (4): the well-typed term $even(s(0))$ *is not recognized. We hence start over (1) with* $\mathcal{A}_1$ *recognizing* $\mathcal{L}_1 = \{even(0), even(s(0))\}$*. After using the completion algorithm,* $\mathcal{A}_1^*$ *contains the transition set* (a) *plus the following new transitions:*

$$(b) \quad s(q_0) \to q_1 \quad even(q_1) \to q_{e1} \quad odd(q_0) \to q_{e2} \quad false \to q_f \quad q_f \to q_{e2} \quad q_{e2} \to q_{e1}$$

Building the associated abstraction (3), we use the set of constraints $S(\mathcal{A}_1^*, T) = S(\mathcal{A}_0^*, T) \cup \{q_{e0} \neq q_{e1}, q_{e0} \neq q_{e2}, q_{e0} \neq q_f\} \cup \{q_t \neq q_{e1}, q_t \neq q_{e2}, q_t \neq q_f\}$ *where we separate* q_{e0}, q_t *from* q_{e1}, q_{e2}, q_f *because they match two different elements of the type partition* T*, respectively* true *and* false*. The resulting* $\phi(\mathcal{A}_1^*)$ *is*

$$\begin{array}{rrr} 0 \to nat^\# & true \to true^\# & even(nat^\#) \to true^\# \\ s(nat^\#) \to nat^\# & \mathsf{false} \to \mathsf{false}^\# & odd(nat^\#) \to \mathsf{false}^\# \end{array}$$

This automaton is not $\mathcal{R}$*-closed w.r.t. the rule* $odd(s(\underline{x})) \to even(\underline{x})$*: if we instantiate* $\underline{x}$ *with* $nat^\#$*, since* $even(nat^\#)$ *is recognized in* $true^\#$ *and* $odd(s(nat^\#))$ *in* $\mathsf{false}^\#$*, for it to be* $\mathcal{R}$*-closed it should include the transition* $true^\# \to \mathsf{false}^\#$*, which it does not. So we start again (1) with* $\mathcal{A}_2$ *recognizing* $\mathcal{L}_2 = \{even(0), even(s(0)), odd(s(0))\}$ *where* $odd(s(0))$ *has been generated from* $odd(s(nat^\#))$*. After using the completion algorithm,* $\mathcal{A}_2^*$ *contains transition sets* (a)*,* (b) *and the new transition* $odd(q_1) \to q_{e0}$*. The associated set of constraints is* $S(\mathcal{A}_2^*, T) = S(\mathcal{A}_1^*, T) \cup \{q_0 \neq q_1 \vee q_{e0} = q_{e1}\}$*.*

The new constraint $q_0 \neq q_1$ is added because with the two transitions $odd(q_0) \to q_{e1}$ and $odd(q_1) \to q_{e0}$ if the abstraction chooses $q_0 = q_1$ then the resulting abstraction automaton would no longer be deterministic. The resulting $\phi(\mathcal{A}_2^*)$ is defined by

$$\begin{array}{rrr} 0 \to 0^\# & s(1^\#) \to 0^\# & s(0^\#) \to 1^\# \\ true \to true^\# & false \to false^\# & even(0^\#) \to true^\# \\ even(1^\#) \to false^\# & odd(0^\#) \to false^\# & odd(1^\#) \to true^\# \end{array}$$

This automaton is completed, $\mathcal{R}$-closed and contains an abstraction of $\mathcal{R}$ and of Σ. From it, we extract a symbol signature for even with output partition $T = \{true, false\}$ which is $(\{0^\#, 1^\#\}) \to T$.

The rest of this section details each step of one iteration of the procedure followed by proofs of correctness (Theorem 8), regular completeness (Theorem 9) and completeness in refutation (Theorem 10).

Tree Automata Completion

Let $\mathcal{R}$ be a functional TRS, and f the symbol of Σ we are searching a type partitions signature for. Each iteration i of the procedure starts by using the Tree Automata Completion algorithm to complete the automaton $\mathcal{A}_i$. This automaton is an ϵ-free tree automaton in which each state q recognizes exactly one term. The language recognized by $\mathcal{A}_i$ is a finite subset of $\mathcal{R}^*(\mathcal{L})$ where $\mathcal{L} = \{f(t_1, \ldots, t_n) \mid t_1 \ldots t_n \in IRR(\mathcal{R})\}$. A state q is final in $\mathcal{A}_i$ if it recognizes a term of $\mathcal{L}$. We write $\mathcal{A}_i^*$ for the output of this step of the procedure. It is a new tree automaton recognizing exactly $\mathcal{R}^*(\mathcal{L}(\mathcal{A}_i))$ and having additional properties ensured by the Tree Automata Completion algorithm: $\mathcal{A}_i$ is syntactically included in $\mathcal{A}_i^*$, and for all state q, q' of $\mathcal{A}_i^*$ and term t (resp. t') recognized by q (resp. q'), if $t \to_\mathcal{R} t'$ then there exists a transition $q' \to q$ in $\mathcal{A}_i^*$ (t' is also recognized by q).

Counter-Example Finding and Constraints Generation

Since $\mathcal{A}_i^*$ converges to $\mathcal{R}^*(\mathcal{L})$, it can be used to search for a counter-example. Note that at this point, no abstraction has been made: a counter-example found in $\mathcal{A}_i^*$ is not spurious. Note also that the structure of a completed automaton allows us to easily build the rewriting path from an initial term of $\mathcal{L}$ to its faulty outcomes.

Abstraction Generation

An abstraction $\mathcal{A}_i^\#$ is built from $\mathcal{A}_i^*$ by first computing a set $S(\mathcal{A}_i^*, T)$ of constraints over the states of $\mathcal{A}_i^*$ (which will depends on the target type partition T).

Definition 5.3.10 (SMT Constraints)**.** *For any tree automaton $\mathcal{A} = \langle \Sigma, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$ and type partition T, $S(\mathcal{A}, T)$ is the smallest set of SMT constraints such that:*

$$(q \neq q') \in S(\mathcal{A}, T) \Leftarrow \exists \tau, \tau' \in T. \begin{cases} q \in T_\mathcal{A}(\tau) \wedge \\ q' \in T_\mathcal{A}(\tau') \wedge \\ \tau \neq \tau' \end{cases} \tag{5.1}$$

$$(q_1 \neq q_1' \vee \cdots \vee q_n \neq q_n \vee q = q') \in S(\mathcal{A}, T) \Leftarrow \begin{cases} f(q_1, \ldots, q_n) \to q \in \mathcal{A} \wedge \\ f(q_1', \ldots, q_n') \to q' \in \mathcal{A} \end{cases} \tag{5.2}$$

$$(v \neq v' \vee q = q') \in S(\mathcal{A}, T) \Leftarrow v \to q \in \mathcal{A} \wedge v' \to q' \in \mathcal{A} \tag{5.3}$$

where $T_{\mathcal{A}}(\tau) = \{\ q_f \mid q_f \in Q_f \wedge \exists t \in \gamma(\tau).\ t \to^*_{\mathcal{A}} q_f\ \}$. *In other words,* $T_{\mathcal{A}}(\tau)$ *contains all the final states recognizing terms of type* τ.

In this definition, the first type of constraints (5.1) ensures that the target type partition is respected: for any two final states q, q' (recognizing each a term of $\mathcal{L}$), if they rewrite into members of two different types of T, then we must have the constraint $q \neq q'$. The two other kind of constraints, (5.2) and (5.3), ensure determinism. We then use an SMT solver to find *the smallest* renaming ϕ from Q to $\Sigma^\#$ such that $\mathcal{A}_i^\# = \phi(\mathcal{A}_i^*)$ satisfies $\phi(S(\mathcal{A}_i^*, T))$.

Termination

We stop the procedure when we detect that $\mathcal{A}_i^\#$ contains an abstraction of $\mathcal{R}$. It is the case when $\mathcal{A}_i^\#$ is $\mathcal{R}$-closed and *IRR*-complete w.r.t. $\mathcal{R}$.

Definition 5.3.11 ($\mathcal{R}$-Closed Tree Automaton)**.** *Let* $\mathcal{R}$ *be a term rewriting system and* $\mathcal{A}$ *a tree automaton. For all state* q *of* $\mathcal{A}$*, rule* $l \to r$ *of* $\mathcal{R}$ *and substitution* σ *of* $\mathcal{X} \to \mathcal{Q}$ *such that* $l\sigma \to^*_{\mathcal{A}} q$*, the pair* $\langle l\sigma, q\rangle$ *is called a* critical pair*. It is* resolved *if there exists* q' *such that* $r\sigma \to^*_{\mathcal{A}} q'$ *and* $q' = q$ *or* $q' \to q \in \mathcal{A}$*.* $\mathcal{A}$ *is* $\mathcal{R}$*-closed if every critical pair is resolved.*

Definition 5.3.12 (*IRR*-Completeness)**.** *An automaton* $\mathcal{A}$ *is IRR-complete w.r.t.* $\mathcal{R}$ *if for all symbols* f *in* $\mathcal{A}$*, for every term* $t = f(t_1, \ldots, t_n)$ *with* $t_i \in IRR(\mathcal{R})$ *there exists a state* q *such that* $t \to^*_{\mathcal{A}} q$.

Note that in our case, $IRR(\mathcal{R})$ is easily computable since it is the language of values. If $\mathcal{A}_i^\#$ is not *IRR*-complete and $\mathcal{R}$-closed, we start a new iteration of the procedure using $\mathcal{A}_{i+1} = grow(\mathcal{A}_i, \mathcal{A}_i^\#)$ where the purpose of *grow* is to add new samples to the completed automaton while eliminating the causes of the non-*IRR*-completeness and non-$\mathcal{R}$-closure.

Definition 5.3.13 (The *grow* Function)**.** *The automaton* $\mathcal{A}_{i+1} = grow(\mathcal{A}_i, \mathcal{A}_i^\#)$ *is defined as follows.*

(i) if $\mathcal{A}_i^\#$ *contains an unresolved critical pair* $\langle l\sigma, q\rangle$ *(not* $\mathcal{R}$*-closed), then a rewriting path has not been taken into account yet. Let* t *be a term such that* $t \to^*_{\mathcal{A}_i^\#} l\sigma$*. We add to* $\mathcal{A}_{i+1}$ *the necessary transitions to recognize* t*, to ensure that this critical pair will be solved in the next iteration.*

(ii) if $\mathcal{A}_i^\#$ *has no unresolved critical pair but it is not IRR-complete, then we know that the set* $E = \mathcal{L} \setminus \mathcal{L}(\mathcal{A}_i^\#)$ *is not empty. Let* t *be one term of* E *having the smallest number of symbols. Then we add to* $\mathcal{A}_{i+1}$ *the necessary transitions to recognize* t*. This ensures that in the next iteration,* $\mathcal{A}_{i+1}^\#$ *recognizes it.*

When $\mathcal{A}_i^\# = \langle \Sigma, \mathcal{Q}, \mathcal{Q}_f, \Delta\rangle$ is *IRR*-complete $\mathcal{R}$-closed it contains an abstraction $\Lambda = \langle \Sigma^\#, \Delta^\#\rangle$ and an abstraction $\mathcal{R}^\#$ of $\mathcal{R}$ where $\Sigma^\#$ is $\mathcal{Q}$ the set of states of $\mathcal{A}_i^\#$, and where $\Delta^\#$ and $\mathcal{R}^\#$ are defined as

$$\mathcal{R}^\# = \{f(\tau_1, \ldots, \tau_n) \to \tau \mid f \in \mathcal{F} \wedge f(\tau_1, \ldots, \tau_n) \to \tau \in \Delta\} \qquad \Delta^\# = \Delta \setminus \mathcal{R}^\#$$

Then $\mathcal{A}_i^\#$ is returned by *analyze-function*.

Lemma 26 (The procedure outputs an abstraction of $\mathcal{R}$)**.** *Let $\Sigma = \mathcal{C} \cup \mathcal{F}$ be a ranked alphabet, with $\mathcal{R}$ a rewriting system, functional w.r.t. $\mathcal{C}$ and $\mathcal{F}$. Let $\mathcal{A}$ be a normalized, IRR-complete, and $\mathcal{R}$-closed automaton that is ϵ-deterministic (deterministic once ϵ-transitions are removed). Let ϕ be a renaming respecting the constraints $S(\mathcal{A}, T)$. Let $\mathcal{R}^\# = \{f(\tau_1, \ldots, \tau_n) \to \tau \mid f \in \mathcal{F} \wedge f(\tau_1, \ldots, \tau_n) \to \tau \in \Delta\}$. If the resulting automaton $\phi(\mathcal{A})$ is $\mathcal{R}^\#$-closed and is IRR-complete w.r.t. $\mathcal{R}$, then $\mathcal{R}^\#$ is an abstraction of $\mathcal{R}$.*

Proof. First we prove that for all terms t and type τ such that $t \to^*_{\mathcal{A}} \tau$, for all k and all term u such that $t \to^k_{\mathcal{R}} u$ then $u \to^*_{\mathcal{A}} \tau$. We proceed by induction on k. First if $k = 0$. Then $t = u$ and since we already know that $t \to^k_{\mathcal{A}} \tau$, we have $u \to^*_{\mathcal{A}} \tau$. Second, if $k = k' + 1$. Then by definition there exists a rule $l \to r \in \mathcal{R}$, a substitution σ and a position p such that $t|_p = l\sigma$ and $t \to_{\mathcal{R}} t[r\sigma]_p \to^{k'}_{\mathcal{R}} u$. Since $t \to^*_{\mathcal{A}} \tau$ and $\mathcal{A}$ is normalized, there exists τ_p such that $t|_p \to^*_{\mathcal{A}} \tau_p$. Since $\mathcal{A}$ is $\mathcal{R}$-closed and $t|_p \to_{\mathcal{R}} r\sigma$ this also means that we have $r\sigma \to^*_{\mathcal{A}} \tau_p$. Note that we hence have $t[r\sigma]_p \to^*_{\mathcal{A}} t[\tau_p]_p \to^*_{\mathcal{A}} \tau$. By induction hypothesis on k this means that $u \to^*_{\mathcal{A}} \tau$.

Now we prove that $\mathcal{R}^\#$ is an abstraction of $\mathcal{R}$. For soundness, let $f(\tau_1, \ldots, \tau_n) \to \tau$ be a rule of $\mathcal{R}^\#$. Let $t = f(t_1, \ldots, t_n)$ be such that for all i, $t_i \in \gamma(\tau_i)$. Let $u \in \mathit{IRR}(\mathcal{R})$ a term such that $t \to^*_{\mathcal{R}} u$. Then thanks to what we just proved, we know that $u \to^*_{\mathcal{A}} \tau$. Since u is irreducible, we deduce that $u \to^*_{\Delta^\#} \tau$. By definition this means that $u \in \gamma(\tau)$. To prove completeness, let $t = f(t_1, \ldots, t_n)$ be such that for $i = 1 \ldots n$, $t_i \in \mathit{IRR}(\mathcal{R})$. By assumption, we know that $\mathcal{A}$ is *IRR*-complete, thus there must exist a rule $f(\tau'_1, \ldots, \tau'_n) \to \tau'$ such that $t_i \in \gamma(\tau'_i)$ for $i = 1 \ldots n$. □

Theorem 8 (Correctness of the invariant learning procedure)**.** *Let $\Lambda = \langle \Sigma^\#, \Delta^\# \rangle, \mathcal{R}^\#$ be the output of the invariant learning procedure of $\mathcal{R}$ with $\Lambda_* = \langle \Sigma^\#_*, \Delta^\#_* \rangle$, the pattern p and type partition T. Let Π be the set of all substitutions π such that $p\pi \to^*_{\Delta^\# \cup \mathcal{R}^\#} v^\#$. Then Λ, $\mathcal{R}$, $\widetilde{\Pi}$ is a solution to the type partition inference problem.*

Proof. First, since transitions are successively added to the automata $\mathcal{A}_i$, the facts $\Sigma^\# \supseteq \Sigma^\#_*$ and $\Delta^\# \supseteq \Delta^\#_*$ are ensured by construction of $\mathcal{A}_i$. Second, thanks to Lemma 26 we know that $\mathcal{R}^\#$ is an abstraction of $\mathcal{R}$. We need to show that this abstraction is complete w.r.t. p and every $v^\#$ of T (cf. Definition 5.2.3). Consider $v^\# \in T$, a term $t \in \gamma(v^\#)$ and a (concrete) substitution σ such that $p\sigma \to^*_{\mathcal{R}} t$. Let $\mathcal{A}$ be the last automaton considered during the inference procedure. Since $\phi(\mathcal{A})$ is *IRR*-complete there must exist an (abstract) substitution π such that $\sigma \in \gamma(\pi)$. Since each term in $\mathcal{A}$ is recognized by a unique state, t will be abstracted in $\phi(\mathcal{A})$ by a unique abstract value. By definition of $S(\mathcal{A}, T)$ this abstract value is $v^\#$. [4] Hence, if $p\pi \not\to^*_{\phi(\mathcal{A})} v^\#$ this means that $\phi(\mathcal{A})$ is not complete, which contradicts our hypothesis (otherwise the procedure would still continue). We have $p\pi \to^*_{\phi(\mathcal{A})} v^\#$ and by definition, $p\pi \not\to^*_{\Delta^\# \cup \mathcal{R}^\#} v^\#$. □

By Theorem 8, if we let T_i denote the set $\{\tau_i \mid f(\tau_1, \ldots, \tau_i, \ldots, \tau_n) \to \tau \in \mathcal{R}^\#\}$ then this procedure gives $(T_1, \ldots, T_n) \to T$ as a correct signature of f in $\mathcal{R}^\#$ and $\Lambda^\#$. We can thus replace the *analyze-function* algorithm in *partition-inference* with this procedure when the input symbol is *recursive*, that is, its associated TRS fragment is recursive. It is worth noticing that this procedure provides two guarantees: 1. *regular completeness*, if there exists a regular abstraction $\mathcal{R}^\#$ providing a signature for f,

[4] In practice, t could be abstracted into a more precise abstract value $w^\#$ such that $\gamma(w^\#) \subset \gamma(v^\#)$. In this rare case, an epsilon transition $w^\# \to v^\#$ is added to the automaton to retain the information. The rest of the proof is unchanged.

then we will eventually find it; and 2. *completeness in refutation*, if there exists a counter-example, we will eventually find it.

Theorem 9 (Regular Completeness)**.** *If there exists a regular abstraction* $\Lambda = \langle \Sigma^\#, \Delta^\# \rangle$ *of* $\mathcal{T}(\Sigma)$ *and a regular abstraction* $\mathcal{R}^\#$ *of* $\mathcal{R}$ *providing a type partition signature for* f*, Algorithm 4 will eventually find it.*

Proof. In the following, given automata $\mathcal{A}$ and $\mathcal{A}'$, $\mathcal{A} \subseteq \mathcal{A}'$ denotes that transitions of $\mathcal{A}$ are included in transitions of $\mathcal{A}'$ modulo state renaming. In the same way, $\mathcal{A} = \mathcal{A}'$ means that transitions sets are equal modulo state renaming. If Λ and $\mathcal{R}^\#$ exists, then we can build $\mathcal{A}^\#$ defined with the union of $\Delta^\#$ and $\mathcal{R}^\#$. At each cycle i of the procedure, there exists a renaming ϕ respecting $S(\mathcal{A}_i^*, T)$ such that $\phi(\mathcal{A}_i^*) \subseteq \mathcal{A}^\#$. Since the procedure builds the *smallest* renaming ϕ' respecting $S(\mathcal{A}_i^*, T)$ (meaning that $\phi'(\mathcal{A}_i^*)$ has the smallest number of states), and since there is a finite number of automata for a given number of states, then we will eventually have $\phi = \phi'$. The procedure stops when $\phi'(\mathcal{A}_i^*)$ is *IRR*-complete, which means we cannot have $\phi'(\mathcal{A}_i^*) \subset \mathcal{A}^\#$, but only $\phi'(\mathcal{A}_i^*) = \mathcal{A}^\#$. □

Theorem 10 (Completeness in Refutation)**.** *For a given input function symbol* f *and target type partition* T*, if there exists a term* $f(t_1, \ldots, t_n)$ *that rewrites to two different elements of* T*, Algorithm 4 will eventually find it.*

Proof. By adding new terms in the automaton $\mathcal{A}_i$ at each new cycle in a fair manner (smallest terms first in step (ii) of *grow* function), we guarantee that at some point we will add the counter-example. It will be detected as a counter-example after the tree automata completion phase. □

Recall that our overall goal is to prove safety properties on programs, i.e., properties of the form $t \not\to^* \mathit{false}$. For this, we can identify a subset of *forbidden types* (such as $\mathit{false}^\#$) in the target type partition, and define a counter-example as a term t typable with one of those forbidden types. In this way we can turn our type inference algorithm into a verification tool for safety properties, as illustrated in the next section.

5.4 Experiments

This section details our implementation [Tbk4] of the verification technique developed in this chapter and the associated experimental results [Exp4]. It consists of an OCaml program along with several libraries able to resolve together the regular language type inference problem from an input term rewriting system $\mathcal{R}$, pattern p and target type partition T. It outputs an abstraction Λ and $\mathcal{R}^\#$ and all the possible type substitutions π with the associated $\tau \in T$ such that $\Lambda, \mathcal{R}^\#, \pi \vdash p : \tau$. This allows us to verify complex properties on programs by expressing the property using a predicate defined in the program itself, and verifying that it can never be typed with $\mathit{false}^\#$. For instance to prove that a list sorting function *sort* is correct (in the sense that the output list is sorted), we first define a *sorted* predicate as

$$\mathit{sorted}(\mathit{nil}) \to \mathit{true} \qquad \mathit{sorted}(\mathit{cons}(\underline{x}, \mathit{nil})) \to \mathit{true}$$
$$\mathit{sorted}(\mathit{cons}(\underline{x}, \mathit{cons}(\underline{y}, \underline{z}))) \to \mathit{if}(\mathit{leq}(\underline{x}, \underline{y}), \mathit{sorted}(\mathit{cons}(\underline{y}, \underline{z})), \mathit{false})$$

and use our implementation with the input term $\mathit{sorted}(\mathit{sort}(\underline{x}))$ with the target type partition $\{\mathit{true}^\#, \mathit{false}^\#\}$. If all the output type substitutions π are such that $\pi \vdash \mathit{sorted}(\mathit{sort}(\underline{x})) : \mathit{true}^\#$, then the property is verified.

5.4.1 Implementation Details

The implementation [Tbk4] follows the theoretical algorithms presented in this chapter with some optimizations and limitations. We describe here the various specificities of the implementation.

Preliminary Typing Phases To simplify our argumentation in this chapter, we required every type partition to be a partition of $IRR(\mathcal{R})$, the entire set of possible values. In practice, trying to compute type partitions over $IRR(\mathcal{R})$ would be both inefficient and unnecessary since in most modern programming languages functions can only take given subsets of values (types) as parameters. For this reason, our implementation is equipped with a preliminary Hindley-Milner type inference phase [Hin69, Mil78] with let-polymorphism. The first-order types used by this inference phase are defined by the user as an input tree automaton where each state is a type. From the input pattern, the term rewriting system is then *monomorphized* so that we know the input domain of every function call, and thus the domain of each type partition we are looking for. This greatly improves the performances of the analysis.

Constants sub-typing phase Consider the simple term rewriting system defining the equality predicate on natural numbers

$$eq(0,0) \to true \quad eq(s(\underline{x}),0) \to false \quad eq(0,s(\underline{y})) \to false \quad eq(s(\underline{x}),s(\underline{y})) \to eq(\underline{x},\underline{y})$$

together with the initial abstract domain $\{true^\#, false^\#, nat^\#\}$. The problem of finding the signature of eq for the target partition $\{true^\#, false^\#\}$ is not regular if we consider the input domains of the arguments of eq to be $nat^\#$. In practice however, if eq is applied on a constant value, such as in $eq(\underline{x}, s(0))$, we can reduce the domain of the second parameter by adding an abstract value $1^\#$ abstracting $s(0)$ in the initial abstract domain. The only possible partition of $1^\#$ is $\{1^\#\}$, which transform the problem of finding the (type partition) signature of eq for $\{true^\#, false^\#\}$ into a regular problem. Its solution is $(\{2{+}^\#, 1^\#, 0^\#\}, \{1^\#\})$. In our implementation, in addition to the first-order types given by the user, we build precise types for the constant values used in the program. This allows us to find a solution to otherwise irregular problems. It is however a prototypical optimization that must be used with caution since it may lead to a computational overhead.

Counter-examples When using our implementation to type a given input pattern with a given type partition, it is possible to specify that some types in the partition are *invalid*. For instance, while typing $sorted(sort(L))$ with the partition $\{true^\#, false^\#\}$, if the *sort* function is correct we expect the pattern to not be typable by $false^\#$. In this case, once the analysis is finished, if we find a way to type the pattern with $false^\#$, then we are able to generate a counter example: an instantiation of the input pattern that rewrites to *false*. Note that this implementation choice has two drawbacks: 1. Since the "forbidden type" information is only used once the analysis is done, if there is no regular types that satisfies the target type partition, our implementation may diverge even though there is a counter example to the desired property. 2. In the best case, even if everything is regular, this may still delay the finding of counter examples. It is possible to insert the notion of *forbidden type* into the analysis to avoid these issues, modulo some adaptations of the presented algorithms.

5.4.2 Test Suite

We tested our implementation over a collection of more than 80 problems [Tbk4] coming from Timbuk 3 [Tbk3], some regular problems from the MoCHi test suite [KSU11a] and some original challenges created for the occasion inspired by Tons of Inductive Problems [CJRS15]. We expect some problem instances to be similar to the ones used in [MKU15], although this cannot be verified since the test suite used in this paper is not publicly available. The test suite, publicly available on Timbuk's website [Exp4], is composed of a variety of problems over first-order and higher-order tree-processing functional programs including:

- Positive tests: regular properties intended to be proved by our implementation. This includes properties such as $\forall L.sorted(merge\text{-}sort(L)) = true$ where L is a list of As and Bs, or $\forall X, T.\ member(X, T) \iff member(X, mirror(T))$ where T is a binary tree, etc.

- Negative tests: false properties or buggy programs, where there exists a counter example to the target property.

- Typing challenges with no property to verify but only regular language types to find. *E.g.*, using the input pattern $only\text{-}As(L)$ and the target type partition $\{true^{\#}, false^{\#}\}$, our implementation should type L with either the type "lists of As" or "lists with at least one B".

- Intentionally non-regular problems, for which we expect the invariant learning procedure to diverge. For instance, $\forall N : nat.\ N = N$ is not a regular property.

Even if some of the problems in our test suite are taken from the MoCHi test suite [KSU11a], MoCHi does not handle regular languages and does not target the same family of properties.

5.4.3 Experimental Results

Table 5.1 and 5.2 give the result of our experiments, comparing our implementation over our test suite against Timbuk 3 which is, to our knowledge, the only higher-order tree-processing program verification tool that is publicly available. In particular, we have not been able to compare to the regular-complete verification procedure presented in [MKU15] which does not offer a public implementation, even if it also targets regular properties. Each table presents the time performances and memory usage (averaged over 10 executions) of the two implementations over the compatible regular test instances on a Intel® i7-7600U CPU, 4 2.80GHz cores. Positive instances (first half of the table) and negative instances (second half of the table) are separated by a line. A timeout is set at 120 seconds.

Execution Time & Completeness

The "Time" columns show that when it succeeds, Timbuk 3 is on average faster than our implementation. This is expected since our preliminary typing phase and sub-typing phase have a cost. However this also shows that in many cases, even on first-order programs our implementation terminates where Timbuk 3 diverges. In particular this is the case for the non-trivial *mergeSort* algorithm for which we successfully identify the regular language of sorted lists. This is also the case for binary-tree processing algorithms such as *memberTree* where we automatically

Name	Time (s)		Memory (MiB)	
	Ours	Timbuk 3.2	Ours	Timbuk 3.2
all-a	0.06 ±0.01	0.0 ±0.0	14.24	4.93
mult	0.13 ±0.03	0.0 ±0.0	14.39	5.83
plusEven	0.17 ±0.01	0.01 ±0.0	14.87	5.71
iteEvenOdd	0.18 ±0.01	0.0 ±0.0	14.86	5.58
appendTheorem	0.32 ±0.12	0.09 ±0.0	14.95	10.18
delete	0.37 ±0.04	0.01 ±0.0	14.99	6.24
evenOdd	0.4 ±0.02	Timeout	14.98	-
incTree	0.42 ±0.04	0.99 ±0.08	15.09	46.12
makelist	0.49 ±0.08	0.42 ±0.08	15.3	99.56
memberAppend	0.51 ±0.08	20.76 ±2.37	15.26	2971.76
square	0.55 ±0.09	47.15 ±2.77	15.12	3997.26
insertionSort	0.63 ±0.05	0.07 ±0.03	15.28	8.96
split	0.69 ±0.1	Timeout	15.81	-
deleteImplies	0.7 ±0.04	Timeout	15.44	-
replaceTree	0.75 ±0.1	Timeout	15.12	-
insertionSort2	0.82 ±0.08	0.65 ±0.07	15.8	109.94
treeDepth	0.92 ±0.15	1.45 ±0.13	15.06	118.89
headReverse	1.26 ±0.09	0.03 ±0.0	19.96	8.92
reverseImplies	1.4 ±0.12	0.54 ±0.12	18.42	9.61
mergeSort	1.53 ±0.13	Timeout	20.68	-
equalLength	1.63 ±0.14	Timeout	14.57	-
reverse	2.35 ±0.24	0.08 ±0.01	18.44	7.24
memberTree	4.25 ±0.42	Timeout	17.96	-
orderedTreeTraversal	4.72 ±0.47	1.52 ±0.12	21.54	11.83
orderedTree	7.1 ±0.81	5.24 ±0.48	14.73	13.8
heightTree	Timeout	Timeout	-	-
insertTree	Timeout	Timeout	-	-
zipUnzip	Timeout	Timeout	-	-
simple	0.05 ±0.0	0.0 ±0.0	5.69	3.42
plusEvenError	0.17 ±0.01	0.01 ±0.0	14.84	6.06
deleteBUG	0.32 ±0.03	0.06 ±0.0	14.98	7.41
appendTheoremBug	0.35 ±0.03	0.01 ±0.0	14.87	6.36
reverseBUGimplies	0.42 ±0.04	0.01 ±0.0	15.26	6.09
memberAppendError	0.5 ±0.09	0.01 ±0.0	15.42	6.69
insertionSort2BUG	0.82 ±0.1	0.09 ±0.01	15.61	9.18
orderedTreePredicate2	40.79 ±2.59	0.73 ±0.04	14.82	12.78
orderedTreeTraversalBug	47.85 ±6.55	0.18 ±0.05	39.66	8.96

Table 5.1: First-order problems

Name	Time (s)		Memory (MiB)	
	Ours	Timbuk 3.2	Ours	Timbuk 3.2
foldDiv	0.24 ±0.01	0.03 ±0.04	15.31	8.59
forallLeq	0.4 ±0.11	0.03 ±0.0	14.94	8.47
mapPlus	0.42 ±0.03	0.07 ±0.0	14.97	8.89
filterNz	0.42 ±0.04	0.12 ±0.01	15.01	8.95
filterEven	0.46 ±0.03	0.31 ±0.0	15.09	8.98
forallImpliesExists	0.53 ±0.04	0.17 ±0.01	14.9	9.91
forallNotEqNotExists	0.54 ±0.04	0.03 ±0.01	14.96	8.9
mapTree	0.56 ±0.03	Timeout	15.09	-
filterEquivExists	0.82 ±0.08	0.18 ±0.01	15.0	9.15
mapFilter	0.91 ±0.13	0.39 ±0.88	15.56	8.98
mergeSortHO	1.67 ±0.14	Timeout	20.61	-
map2AddImplies	Timeout	Timeout	-	-
mapSquare	Timeout	Timeout	-	-
foldRightMult	Timeout	0.01 ±0.0	-	6.53
forallImpliesExists2	0.46 ±0.05	0.0 ±0.0	15.0	3.74
forallNotEqNotForall	0.51 ±0.04	0.0 ±0.0	15.04	3.93
filterEvenBug	0.54 ±0.12	0.1 ±0.01	15.03	8.96
mergeSortHObug	Timeout	0.42 ±0.04	-	10.91

Table 5.2: Higher-order problems

show that it is invariant by mirroring. We see the same tendency on higher-order programs where Timbuk 3 also diverges on the merge-sort algorithm (where this time the comparison operator is a parameter of the sorting function), and on binary-tree processing programs such as *mapTree*, where we succeed. Interestingly, our implementation is unable to find a counter-example to a *mergeSortBug* property, where we try to check that $\forall L.\ \mathit{sorted}\ (\leq)\ (\mathit{merge\text{-}sort}\ (\geq)\ L)$, which is wrong. Timbuk 3 is able to find a counter example, where we reach a timeout. As discussed in Section 5.4.1 this is due to the fact that, in our current modular implementation, counter-examples are searched once the whole analysis is finished.

Memory Usage

Another feature of our implementation is its consistent memory usage (which includes the memory usage of the SMT solver), shown by the "Memory" columns in the tables. Thanks to modularization, the memory footprint of our implementation stays low even on time consuming instances. This is an improvement compared to the non-modular Timbuk 3, where on some problems the memory usage can grow up to several GB of data. The difference is especially visible for instance on the *memberAppend* problem where the goal is to verify that for all lists L_1, L_2 and element X, $(\mathit{member}\ X\ (\mathit{append}\ L_1\ L_2)) \iff ((\mathit{member}\ X\ L_1) \vee (\mathit{member}\ X\ L_2))$.

5.5 Conclusion

We have developed a regular language type inference procedure on top of a regular abstract interpretation of term rewriting systems. This allows us to automatically

verify safety properties on higher-order tree-processing functional programs. This improves on existing verification techniques based on type annotations which generally require considerable expertize to determine all the necessary annotations to carry out the proof. For instance in F* it is necessary to give complex type annotations on every intermediate functions in the definition of the *merge-sort* algorithm (corresponding to intermediate lemma) for the type-checker to accept the program, only to show that its output is sorted. On the other hand we can automatically carry out the same proof with only the expected output type of the main function.

The type inference mechanism uses type partitions to reduce the complexity of the underlying algorithms. For a given input term with variables and output type partition, we compute, for each variable of the input term, the input type partitions needed to respect the output partition. Using a type system allows for modularity: a type partition signature is attached to each symbol, which summarizes the behavior of associated rewriting rules. The type partition signature can be inferred independently for independent functions. It can be inferred directly for non-recursive functions. Recursive functions are handled using a novel invariant learning procedure based on the tree automata completion algorithm, following the precepts of a counter-example guided abstraction refinement procedure (CEGAR). This particular variant of CEGAR allows us to refine the abstractions without the need of a complete spurious counter-example rewriting sequence, at the price of requiring the input TRS to be terminating. The resulting procedure is regularly-complete and complete in refutation, meaning that if it is possible to give a regular language type to a term then we will eventually find it, and if there is no possible type (regular or not) then we will eventually find a counter-example. Regular completeness is achieved by the generation of non-functional abstractions, which Timbuk 3 could not do. Our implementation of this technique shows encouraging performances and is able to verify properties that were not covered by previous similar techniques. In the following we list some possible further improvements.

Using a more restricted type language helps us to enjoy more automation for type inference. In particular, annotations for intermediate functions are automatically inferred. For example, from a TRS encoding the insertion sort algorithm, and the final term $sorted(sort(\underline{x}))$, we automatically infer the type annotation for the insert function as $insert(any^{\#}, sorted^{\#}) \to sorted^{\#} \in \mathcal{R}^{\#}$, where the abstract value $sorted^{\#}$ is inferred from the side analysis of the *sorted* symbol: $sorted(sorted^{\#}) \to true^{\#} \in \mathcal{R}^{\#}$. This annotation reveals a necessary intermediate proof step: we first need to prove that when inserting *any* element into a *sorted* list, we obtain a *sorted* list. We believe that starting from the two above generated rules of $\mathcal{R}^{\#}$ it is possible to infer the adequate type annotations for F* or Liquid Types and automate the proofs, i.e., infer the annotation $insert(X : int, Y : \{l : int\ list \mid sorted(l)\}) : \{l : int\ list \mid sorted(l)\}$.

Another strand of further work concerns the extension to properties on lazy functional programs. Jones' abstraction technique [JA07] can prove properties on functional programs using call-by-name evaluation. With our current invariant inference procedure we need for the input program to be terminating. However, this inference part is not limited to call-by-value evaluation, as it is in [JA07]. We could broaden the image computation technique so that *unfinished* non terminating computations are taken into account. We experimented with this and there are some interesting cases, like call-by-value evaluations, for which this is sufficient to build a correct abstraction automaton.

There are known abstraction inference techniques for domains mixing structures and base types [LGJ07], e.g., queues of integers. These techniques rely on an

extension of word automata called lattice automata. There exists a similar extension for tree automata called lattice tree automata making it possible to abstract recursive structures containing, e.g., integer values [GLGLM13].

The main limitation of our technique is the impossibility to express or verify relational properties. There exists ways to use tree automata to represent relations using, for instance, automatic structures [KN95, BG00]. The idea is to use one automaton to represent multiple terms at once, or *convolution of terms*. For instance, the equality relation can be recognized by the two automaton transitions

$$0 \otimes 0 \to q \qquad\qquad s \otimes s(q) \to q$$

where $\otimes$ is a *convolution operator*. Checking if two terms t_1, t_2 are equals is then equivalent to testing if the convoluted term $t_1 \otimes t_2$ is recognized by the automaton above. Convolutions introduces however new challenges that remains to be solved in order to integrate them into this technique. This is the topic of the next chapter.

Chapter 6

Regular relations

The contribution presented in this chapter has mainly been realized during a three month collaboration with Prof. Naoki Kobayashi at the University of Tokyo.

6.1 Introduction

Until now, in the previous chapters we have seen how to use regular languages to tackle regular problems, and in particular regular safety problems. One particularity of regular problems is that they can be solved without exhibiting any relations between the variables of the program. As a consequence, most problems involving arithmetic, or any kind of structural comparison between infinite domain values, is not a regular problem and cannot be handled by the techniques defined in the previous chapters. For instance, consider the following term rewriting system $\mathcal{R}$ defining the equality predicate eq over natural numbers:

$$\begin{array}{rr} eq(0,\ 0) \to \mathit{true} & eq(s(\underline{x}),\ 0) \to \mathit{false} \\ eq(s(\underline{x}),\ s(\underline{y})) \to eq(\underline{x},\ \underline{y}) & eq(0, s(\underline{y})) \to \mathit{false} \end{array}$$

We want to check that, for all n, eq is reflexive:

$$\forall n.\ eq(n, n) \not\to_{\mathcal{R}}^{*} \mathit{false}$$

As we have seen throughout the document, verifying this property requires computing, or over-approximating, $\mathcal{R}^*(I)$ for $I = \{\ eq(n, n) \mid n \in \mathbb{N}\ \}$. Since $\mathcal{R}$ is simple, we know that $\mathcal{R}^*(I)$ is the union of I and $\{\ \mathit{true}\ \}$. However I is not a regular language, and any regular over-approximation of I would add *false* to the over-approximation of $\mathcal{R}^*(I)$. From the point of view of regular language typing, we can say that there exists no regular language signature for eq for the target type $\{\ \mathit{true}\ \}$. This shows why our problem is not regular. In general, any problem involving non-modulo arithmetic is not regular.

This limitation is directly due to the non-regularity of tree languages representing tree relations such as I. A similar problems arises in the world of string languages for string relations. In the case of strings, this problem has been solved using *automatic relations* [KN95, BG00]. This chapter discusses how string automatic relations can be extended into tree automatic relations in order to represent relations between trees using regular languages [Tata, Lin12]. We then propose a procedure to learn tree automatic relations using ICE [GLMN14, GNMR16], a robust counter-example guided learning model applied to constrained Horn clause (CHC) solving. We implemented

a regular CHC solver [Hau19] based on this procedure and tested it on several case studies.

The rest of the chapter is structured as follows. First we see in Section 6.2 what are automatic relations on strings. We then extend this concept to tree automatic relations in Section 6.3. In Section 6.4 we propose a learning procedure for tree automatic relations based on ICE learning. In Section 6.5 we expose the results of our implementation of this procedure in our Regular CHC solver. Finally, Section 6.6 concludes the chapter.

6.2 String Automatic Relations

Automatic Relations on strings have first been introduced in [KN95] as a mean to present algebraic structures (on strings) using finite automata to encode the structure's operations and relations. For instance, consider the following structure $\langle \mathbb{N}, \leq \rangle$ where $\mathbb{N}$ is the string language of natural numbers $(0, s0, ss0, sss0, \dots)$ and $\leq$ the usual comparison operator. One way to represent the relation $\leq$ is by considering the string language $Leq = \{s^n 0 s^m 0 \mid n \leq m\}$ that encodes every pair of the relation as its concatenation. This is very similar to our previous example with the (tree) language Eq: Leq is not regular as it cannot be recognized by a simple finite automaton (it can be recognized by a finite push-down automaton). In [KN95], authors propose a new way of encoding such relations using regular languages: instead of encoding each pair of the relation by concatenation, each pair is encoded by *convolution*.

Definition 6.2.1 (String convolution)**.** *Let Σ be an alphabet. The convolution of two words α, β of Σ^*, written $\alpha \otimes \beta$, is recursively defined as:*

$$\begin{array}{rcll} a\alpha & \otimes & b\beta & = \frac{a}{b}(\alpha \otimes \beta) \\ a\alpha & \otimes & \epsilon & = \frac{a}{\cdot}(\alpha \otimes \epsilon) \\ \epsilon & \otimes & b\beta & = \frac{\cdot}{b}(\epsilon \otimes \beta) \\ \epsilon & \otimes & \epsilon & = \epsilon \end{array}$$

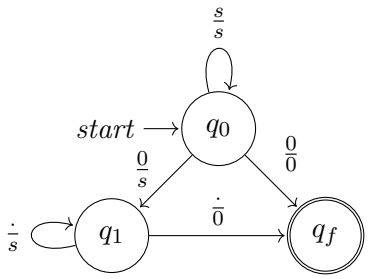
where ϵ is the empty string, $\frac{a}{b}$ denotes the convolution of the two symbols a, b and $\cdot$ is a special padding symbol not in Σ. The resulting string is defined over the alphabet $\{ \frac{a}{b} \mid a, b \in \Sigma \cup \{ \cdot \} \}$.

Example 6.2.1. *Consider $\Sigma = \{ a, b \}$ with the two words $\alpha = ababa$ and $\beta = baa$. The convolution $\alpha \otimes \beta$ is equal to $\frac{a}{b}\frac{b}{a}\frac{a}{a}\frac{b}{\cdot}\frac{a}{\cdot}$. It can easily be seen as the superposition of the two strings on top of each other.*

A string automatic relation, or regular relation, is defined as any relation for which the convolution defines a regular language. Every element of this language represents then a convoluted pair of the relation. A simple deconvolution can reconstruct the two items of the pair.

Definition 6.2.2 (Regular Relation)**.** *A (binary) relation R over strings is* regular *if the language $\mathcal{L}(R) = \{ x \otimes y \mid R(x, y) \}$ is regular. Then it can be recognized by a finite automaton. Such an automaton is called an* automatic relation*. This definition extends to any n-ary relations.*

Example 6.2.2. *Let us consider again the structure $\langle \mathbb{N}, \leq \rangle$ of natural number provided with the comparison operator $\leq$. the relation $\leq$ can be encoded with the language $\mathcal{L}(\leq) = \{ x \otimes y \mid x \leq y \}$. This language contains for instance $\frac{0}{0}$, $\frac{s}{s}\frac{0}{s}\frac{\cdot}{s}\frac{\cdot}{0}$, but not $\frac{s}{0}\frac{0}{\cdot}$. We can show that $\mathcal{L}(\leq)$ is regular since it can be recognized with the following finite automaton:*



The state q_0 recognizes the common prefix between the two strings (the minimum of the two numbers), while q_1 recognizes the suffix (the difference of the two numbers). The final state q_f only accepts elements of the relation.

Contrarily to the concatenation encoding, Regular relations encoded with convolutions are closed under union ($\mathcal{L}(R \cup R') = \mathcal{L}(R) \cup \mathcal{L}(R')$ is regular), intersection ($\mathcal{L}(R \cap R') = \mathcal{L}(R) \cap \mathcal{L}(R')$ is regular), and under complement ($\mathcal{L}(\Sigma^{*2} \setminus R) = \mathcal{L}(\Sigma^{*2}) \setminus \mathcal{L}(R)$ is regular). In the rest of this chapter, we explore how we can extend string automatic relations to tree automatic relations, and how to automatically learn such relations.

6.3 Tree Automatic Relations

As we have seen in the previous section, the nature of automatic relations is closely related to the definition of a convolution operator on the considered domain. In order to generalize string automatic relations to trees, we must first define a convolution operator on trees.

6.3.1 Standard Convolution

The standard definition of tree convolution [Tata] used to define tree automatic relations consists in overlapping the two convoluted trees as pictured in Figure 6.1.

Definition 6.3.1 (Standard Convolution)**.** *Let Σ be a ranked alphabet. The standard convolution of two terms $s, t \in \mathcal{T}(\Sigma)$ written $s \oplus t$ is recursively defined by:*

$$f(s_1,\ \ldots,\ s_n) \oplus g(t_1,\ \ldots,\ t_m) = \frac{f}{g}(s_1 \oplus t_1,\ \ldots,\ s_N \oplus t_N)$$

where $N = max(n, m) = ar(\frac{f}{g})$, for all $i > n$, $s_i = \cdot$ and for all $i > m$, $t_i = \cdot$. Once again, $\cdot$ is a special padding symbol not in Σ. The resulting term is defined over the ranked alphabet $\Sigma_\oplus = \{\ \frac{f}{g} \mid f, g \in \Sigma \cup \{\ \cdot\ \}\ \}$ where $ar(\cdot) = 0$.

Definition 6.3.2 (Regular Tree Relation)**.** *A tree relation R is* regular *(w.r.t. the standard convolution) if the language $\mathcal{L}(R) = \{\ x \oplus y \mid R(x, y)\ \}$ is regular. Then it can be recognized by a finite tree automaton. Such a tree automaton is a* tree automatic relation*. This definition extends to any n-ary relations. We name $R_\oplus$ the set of regular relations w.r.t. the standard convolution.*

This definition is very similar to regular string relations (Definition 6.2.2), and is also closed under union, intersection and complement. It can be used to represent, with a simple tree automaton, relations that are not regular otherwise.

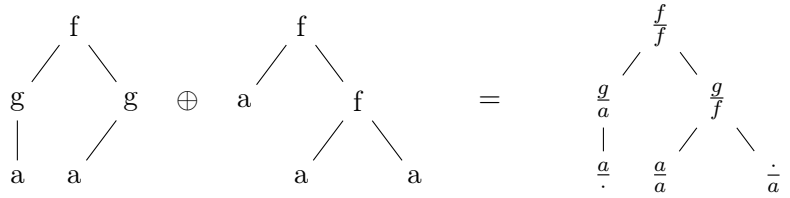


Figure 6.1: Standard convolution between the terms $f(g(a), g(a))$ and $f(a, f(a, a))$. This shows how the convolution can be pictured as the overlapping of the two trees.

Example 6.3.1. *Consider the $=$ relation on trees defined over the alphabet $\Sigma = \{a : 0, b : 1, c : 2\}$. It is a regular relation as $\mathcal{L}(=)$ can be recognized by the following tree automaton:*

$$\frac{a}{a} \to q \qquad \frac{b}{b}(q) \to q \qquad \frac{c}{c}(q, q) \to q$$

However this standard definition of convolution cannot capture relations between trees that never overlap. For instance, consider the ranked alphabet $\Sigma = \{0 : 0, s : 1, nil : 0, cons : 2, a : 0, a : 0\}$ ready to encode numbers from $\mathbb{N}$ and lists of as and as from $List$. Then the relation $R = \{(l, n) \mid n \in \mathbb{N}, l \in List.\ length(l) = n\}$ that relates every list with its length is not regular. This is because when convoluting a number with a list, the tree branch representing the number does not overlap the tree branch representing the list. One simple way to solve this case is to encode the lists

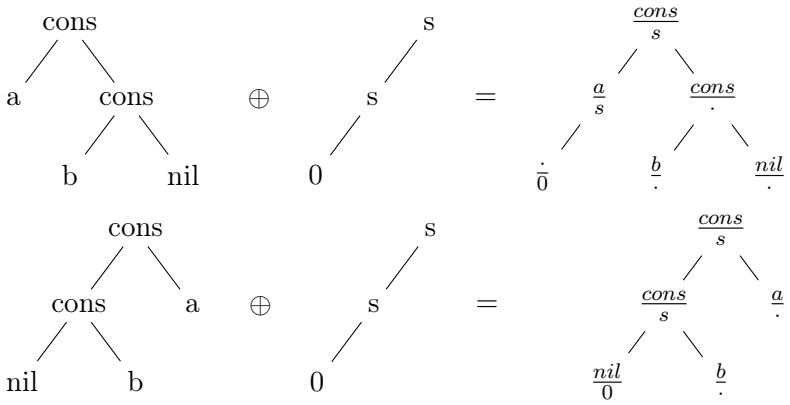


Figure 6.2: Standard convolution between the terms $s(s(0))$ and $cons(a, cons(b, nil))$ (on top) and the terms $s(s(0))$ and $cons(cons(nil, b), a)$ (below). The relation between the length of the list and $s(s(0))$ cannot be captured on top because the two trees do not overlap on relevant positions. This is resolved below by putting the recursion site of lists at the first position of the *cons* symbol.

the other way around as shown on Figure 6.2, writing $cons(cons(nil, b), a)$ instead of $cons(a, cons(b, nil))$. In this case the relation can be represented by the following tree automaton in the final state q_R:

$$\frac{cons}{s}(q_R, q_v) \to q_R \qquad \frac{a}{\cdot} \to q_v$$

$$\frac{nil}{0} \to q_R \qquad \frac{b}{\cdot} \to q_v$$

This can be generalized to any algebraic datatype with a single recursion site such as numbers and lists. However it doesn't work with tree structure with multiple recursion sites such as binary trees, where each node of the tree has two children (two recursion sites). For instance, if we note $BTree$ the set of binary trees, the relation $\{ (t, n) \mid t \in Btree, n \in \mathbb{N}. |t| \leq n \}$ is not regular w.r.t. the standard convolution. This is because the term n cannot overlap all the branches of the tree t whatever the encoding we choose for $BTree$.

6.3.2 Full Convolution

In the previous section we exposed the standard convolution operator definition which can be found in Tata. In this section we extend this definition and define a new convolution operator on trees designed to solve the overlapping problem of the standard convolution operator. The idea of this new convolution is, instead of relating each term of one tree with only one term of the other tree, to relate every term of one tree with every term of the other tree at the same depth.

Definition 6.3.3 (Full Convolution)**.** *Let Σ be a ranked alphabet. The (full) convolution of two terms $s, t \in \mathcal{T}(\Sigma)$ written $s \otimes t$ is recursively defined by:*

$$f(s_1, \ \ldots, \ s_n) \otimes g(t_1, \ldots, t_m) = \begin{cases} \frac{f}{g}(s_1 \otimes \cdot, \ldots, s_n \otimes \cdot) & \text{if } m = 0 \\ \frac{f}{g}(\cdot \otimes t_1, \ldots, \cdot \otimes t_m) & \text{if } n = 0, \text{ otherwise:} \\ \frac{f}{g}(s_1 \otimes t_1, \ldots, s_1 \otimes t_m, \ldots, s_n \otimes t_1, \ldots, s_n \otimes t_m) \end{cases}$$

where each symbol $\frac{f}{g}$ is derived from f and g such that:

$$ar(\frac{f}{g}) = \begin{cases} 0 & \text{if } ar(f) = ar(g) = 0 \\ max(1, ar(f)) * max(1, ar(g)) & \text{otherwise} \end{cases}$$

Once again, $\cdot$ is a special padding symbol not in Σ. The resulting term is defined over the ranked alphabet $\Sigma_\otimes = \{ \frac{f}{g} \mid f, g \in \Sigma \cup \{ \cdot \} \}$ where $ar(\cdot) = 0$. The binary convolution operator can be extended into any n-ary convolution.

In the standard convolution definition, for every two terms $f(s_1, \ \ldots, \ s_n)$ and $g(t_1, \ldots, t_m)$ each sub-term s_i, for $i \in [1, n]$ is convoluted with t_i (if it exists). In this definition, each s_i is convoluted with every t_j for $j \in [1, m]$. This allows us to capture more relations with regular languages. We can hence extend the definition of regular tree relations as follows.

Definition 6.3.4 (Regular Tree Relation)**.** *A tree relation R is* regular *w.r.t. the full convolution if there exists a regular language $\mathcal{L}(R)$ such that for all terms s, t:*

$$s \otimes t \in \mathcal{L}(R) \iff R(s, t)$$

Then it can be recognized by a finite tree automaton. Such an automaton is a tree automatic relation. *Again, this definition extends to any n-ary relations. We name $R_\otimes$ the set of regular relations w.r.t. the full convolution.*

Example 6.3.2 (Binary tree depth)**.** *Let consider the ranked alphabet $\Sigma = \{0 : 0, s : 1, leaf : 0, node : 2\}$ used to encode natural numbers into $\mathbb{N}$ and binary trees into $BTree$. Consider the previously mentioned relation $R = \{ (t, n) \mid t \in BTree, n \in$*

$\mathbb{N}.\ |t| \leq n\ \}$. *We saw earlier that this is not a regular relation with regards to the standard convolution. However it is regular w.r.t. the full convolution. It can be represented using the following tree automaton:*

$$\frac{node}{s}(q,q) \to q \qquad \frac{leaf}{s}(q) \to q \qquad \frac{leaf}{0} \to q$$
$$\frac{\cdot}{s}(q) \to q \qquad \frac{\cdot}{0} \to q$$

For instance, the convolution of $node(node(leaf, leaf), leaf)$ *and* $s(s(0))$ *pictured on Figure 6.3 is recognized by this automaton. On the contrary, the convolution between* $node(leaf, leaf)$ *and* 0*, which is* $\frac{node}{0}(\frac{leaf}{\cdot}, \frac{leaf}{\cdot})$*, is not recognized by this automaton.*

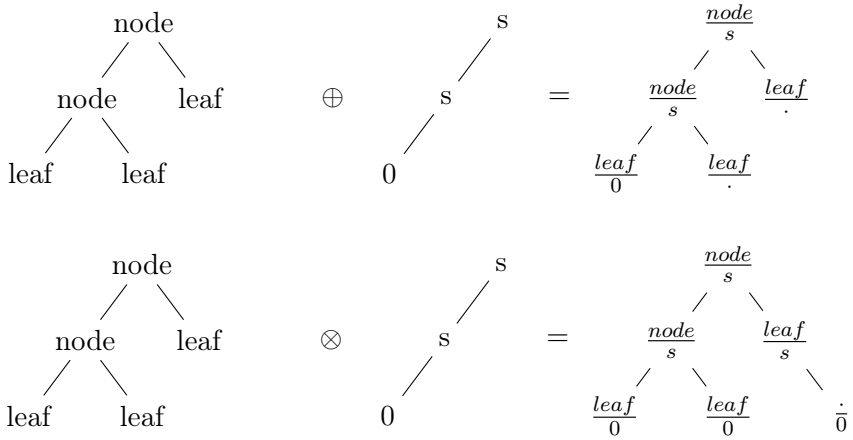


Figure 6.3: Difference between the standard convolution $\oplus$ (on top) and full convolution $\otimes$ (bottom) between the terms $node(node(leaf, leaf), leaf)$ and $s(s(0))$. This shows how the full convolution allows each branch to be in relation with every other branch.

The full convolution is strictly more expressive that the standard convolution as it can be used to encode more relations with regular languages. This extends the concept of regular relations.

Theorem 11 (Standard Regular Relations are Full Regular Relations)**.** *The set of standard regular relations is strictly included in the set of full regular relations:* $R_\oplus \subset R_\otimes$.

Proof. For every relation R of $R_\oplus$, by definition $\mathcal{L}(R)$ is regular and can be represented by a tree automaton $\mathcal{A}$ over $\oplus$ convoluted terms. To prove that R is also in $R_\otimes$ we proceed as follows. First we define a transformation $[\cdot]_\otimes$ from $\mathcal{A}$ to a new tree automaton $[\mathcal{A}]_\otimes$ recognizing the same relation with $\otimes$ convoluted terms. We then show the following property of the transformation:

$$\forall s, t. \quad s \otimes t \in \mathcal{L}([\mathcal{A}]_\otimes) \iff s \oplus t \in \mathcal{L}(\mathcal{A})$$

Let Σ be an initial (before convolution) ranked alphabet. Let $\mathcal{A} = \langle \Sigma_\oplus, \mathcal{Q}_\oplus, \mathcal{Q}_f, \Delta_\oplus \rangle$ where $\Sigma_\oplus$ follows Definition 6.3.1. Then $[\mathcal{A}]_\otimes$ is defined as $\langle \Sigma_\otimes, \mathcal{Q}_\otimes, \mathcal{Q}_f, \Delta_\otimes \rangle$, where $\Sigma_\otimes$ follows Definition 6.3.3, where $\mathcal{Q}_\otimes$ is the smallest set such that

$$\frac{\top}{\top} \in \mathcal{Q}_\otimes \qquad q \in \mathcal{Q}_\oplus \iff \{\ q, \frac{q}{\top}, \frac{\top}{q}\ \} \subseteq \mathcal{Q}_\otimes$$

The state $\frac{\top}{\top}$ recognizes any term convolution and is associated to the following transitions in $\Delta_\otimes$:

$$\frac{f}{g} \in \Sigma_\oplus \iff \frac{f}{g}(\frac{\top}{\top}, \ldots, \frac{\top}{\top}) \to \frac{\top}{\top} \in \Delta_\otimes$$
$$f \in \Sigma \iff \frac{f}{\cdot}(\frac{\top}{\top}, \ldots, \frac{\top}{\top}) \to \frac{\top}{\top} \in \Delta_\otimes$$
$$g \in \Sigma \iff \frac{\cdot}{g}(\frac{\top}{\top}, \ldots, \frac{\top}{\top}) \to \frac{\top}{\top} \in \Delta_\otimes$$

Similarly, states of the form $\frac{q}{\top}$ (resp. $\frac{\top}{q}$) can recognize every convolution $s \otimes t$ if and only if $s \oplus \cdot$ (resp. $\cdot \oplus t$) is recognized by q in $\mathcal{A}$. This is defined by the following transitions in $\Delta_\otimes$:

$$\frac{f}{\cdot}(q_1, \ldots, q_n) \to q \in \Delta_\oplus \iff \forall g \in \Sigma \cup \{\cdot\}.\ \frac{f}{g}(\frac{q_1}{\top}, \ldots, \frac{q_n}{\top}) \to \frac{q}{\top} \in \Delta_\otimes$$
$$\frac{\cdot}{g}(q_1, \ldots, q_m) \to q \in \Delta_\oplus \iff \forall f \in \Sigma \cup \{\cdot\}.\ \frac{f}{g}(\frac{\top}{q_1}, \ldots, \frac{\top}{q_m}) \to \frac{\top}{q} \in \Delta_\otimes$$

where $m = ar(f)$ and $n = ar(g)$. Finally the remaining transitions in $\Delta_\otimes$ define the transformation itself:

$$\frac{f}{g}(q_1, \ldots, q_N) \to q \in \Delta_\oplus \iff \begin{cases} \frac{f}{g}(q_1, \ldots, q_N) \to q \in \Delta_\otimes \text{ if } ar(f) = 0 \vee ar(g) = 0, \text{ or:} \\ \frac{f}{g}(q_{1,1}, \ldots, q_{1,m}, \ldots, q_{n,1}, \ldots, q_{n,m}) \to q \in \Delta_\otimes \end{cases}$$

where $m = ar(f)$, $n = ar(g)$, $N = max(n, m)$ and for all (i, j), $q_{i,j} = q_i$ if $i = j$, $q_{i,j} = \frac{q_i}{\top}$ if $i > m$, $q_{i,j} = \frac{\top}{q_j}$ if $j > n$ and $q_{i,j} = \frac{\top}{\top}$ in other cases. Here we assume that $\mathcal{A}$ is a REFD automaton. The idea of the transformation is to simulate the standard convolution with the full convolution by using the states of $\mathcal{A}$ in $[\mathcal{A}]_\otimes$ where the terms overlap, and using $\frac{\top}{q}$ and $\frac{q}{\top}$ otherwise when they don't. Let us show that for any term s, t, for any state $q \in \mathcal{Q}$, if $s \oplus t \in \mathcal{L}(\mathcal{A}, q)$ then $s \otimes t \in \mathcal{L}([\mathcal{A}]_\otimes, q)$. We do that by induction on the term $s \oplus t$.

- If $s \oplus t = \frac{f}{g}$. Then for all q such that $s \oplus t \in \mathcal{L}(\mathcal{A}, q)$ because $\mathcal{A}$ is ϵ-free we have $\frac{f}{g} \to q \in \Delta_\oplus$. By definition this means that $\frac{f}{g} \to q \in \Delta_\otimes$. Because $s \otimes t = \frac{f}{g}$, we have $s \otimes t \in \mathcal{L}([\mathcal{A}]_\otimes, q)$.
- If $s \oplus t = \frac{f}{g}(T_1, \ldots, T_k)$, then by definition of $\oplus$, we have $s = f(s_1, \ldots, s_n)$, $t = g(t_1, \ldots, t_m)$ with $k = max(n, m)$ and for each $i \in [1, k]$, $T_i = s_i \oplus t_i$ according to Definition 6.3.1. For all q such that $s \oplus t \in \mathcal{L}(\mathcal{A}, q)$, since $\mathcal{A}$ is ϵ-free we have $\frac{f}{g}(q_1, \ldots, q_k) \to q \in \Delta_\oplus$ with for all i, $s_i \oplus t_i \in \mathcal{L}(\mathcal{A}, q_i)$. Then by hypothesis of induction we have $s_i \otimes t_i \in \mathcal{L}([\mathcal{A}]_\otimes, q_i)$. Here we have two cases:
  - If $ar(f) = 0$ or $ar(g) = 0$ then by definition of $[\mathcal{A}]_\otimes$ the transition $\frac{f}{g}(q_1, \ldots, q_k) \to q$ is in $\Delta_\otimes$. Also by definition we know $s \otimes t$ is equal to $\frac{f}{g}(s_1 \otimes t_1, \ldots, s_k \otimes t_k)$ (where every s_i or every t_i is $\cdot$ according to Definition 6.3.1 and 6.3.3). And we already know that $s_i \otimes t_i \in \mathcal{L}([\mathcal{A}]_\otimes, q_i)$, hence we conclude that $s \otimes t \in \mathcal{L}([\mathcal{A}]_\otimes, q)$.
  - If $ar(f) > 0$ and $ar(g) > 0$ then by definition of the transformation we have $\frac{f}{g}(q_{1,1}, \ldots, q_{1,m}, \ldots, q_{n,1}, \ldots, q_{n,m}) \to q \in \Delta_\otimes$. Moreover the term $s \otimes t$ is by Definition 6.3.3 $\frac{f}{g}(s_1 \otimes t_1, \ldots, s_1 \otimes t_m, \ldots, s_n \otimes t_1, \ldots, s_n \otimes t_m)$. For all $i \in [1, n], j \in [1, m]$, if $i = j$ we have $q_{i,j} = q_i$ and we already

know $s_i \otimes t_i \in \mathcal{L}([\mathcal{A}]_\otimes, q_i)$ which means $s_i \otimes t_i \in \mathcal{L}([\mathcal{A}]_\otimes, q_{i,j})$. If $i > m$ then by definition of $\Delta_\otimes$ we have $q_{i,j} = \frac{q_i}{\top}$. By definition of the standard convolution, $s_i \oplus \cdot \in \mathcal{L}(\mathcal{A}, q_i)$ which implies $s_i \otimes t \in \mathcal{L}([\mathcal{A}]_\otimes, q_{i,j})$ for all t. In particular $s_i \otimes t_j \in \mathcal{L}([\mathcal{A}]_\otimes, q_{i,j})$. Similarly if $j > n$ then by definition of $\Delta_\otimes$ we have $q_{i,j} = \frac{\top}{q_j}$. By definition of the standard convolution, $\cdot \oplus t_j \in \mathcal{L}(\mathcal{A}, q_j)$ which implies $s \otimes t_j \in \mathcal{L}([\mathcal{A}]_\otimes, q_{i,j})$ for all s. In particular $s_i \otimes t_j \in \mathcal{L}([\mathcal{A}]_\otimes, q_{i,j})$. Finally in any other cases $q_{i,j} = \frac{\top}{\top}$ which also means $s_i \otimes t_i \in \mathcal{L}([\mathcal{A}]_\otimes, q_{i,j})$ because $\frac{\top}{\top}$ recognizes any term. We conclude that $s \otimes t \in \mathcal{L}([\mathcal{A}]_\otimes, q)$.

We do a similar induction on the term $s \otimes t$ to show that for any term s, t, for any state $q \in \mathcal{Q}$, if $s \otimes t \in \mathcal{L}([\mathcal{A}]_\otimes, q)$ then $s \oplus t \in \mathcal{L}(\mathcal{A}, q)$. Let us write $s = f(s_1, \ldots, s_n)$ and $t = g(t_1, \ldots, t_m)$.

- If $ar(f) = ar(g) = 0$ then $s \otimes t = \frac{f}{g}$. Let q be the state such that $\frac{f}{g} \to q \in \Delta_\otimes$. By definition of the transformation this means that there exists a transition $\frac{f}{g} \to q \in \Delta_\oplus$. By definition of the standard convolution this means that $s \oplus t \in \mathcal{L}(\mathcal{A}, q)$.

- If $ar(f) = n = 0$ or $ar(g) = m = 0$ then $s \otimes t = \frac{f}{g}(s_1 \otimes t_1, \ldots, s_N \otimes t_N)$ with $N = max(n, m)$. (where every s_i or every t_i is $\cdot$ according to Definition 6.3.1 and 6.3.3). Let q be the state such that $\frac{f}{g}(q_1, \ldots, q_N) \to q \in \Delta_\otimes$ with $s_i \otimes t_i \in q_i$. By definition of the transformation this means that there exists a transition $\frac{f}{g}(q_1, \ldots, q_N) \to q \in \Delta_\oplus$. An by hypothesis of induction, $s_i \oplus t_i \in \mathcal{L}(\mathcal{A}, q)$. By definition of the standard convolution this means that $s \oplus t \in \mathcal{L}(\mathcal{A}, q)$.

- If $ar(f) > 0$ and $ar(g) > 0$ then $s \otimes t$ is equal to $\frac{f}{g}(s_1 \otimes t_1, \ldots, s_1 \otimes t_m, \ldots, s_n \otimes t_1, \ldots, s_n \otimes t_m)$. and it is recognized using the following transition in $\Delta_\otimes$: $\frac{f}{g}(q_{1,1}, \ldots, q_{1,m}, \ldots, q_{n,1}, \ldots, q_{n,m}) \to q$ where $s_i \otimes t_j \in \mathcal{L}([\mathcal{A}]_\otimes, q_{i_j})$. By definition of the transformation this means there exists a transition $\frac{f}{g}(q_1, \ldots, q_N) \to q$ in $\Delta_\oplus$. Let us assume that $n \leq m$. Then $N = m$. For all $i \in [1, n]$ we have $q_{i,i} = q_i$. By hypothesis of induction this means that $s_i \oplus t_i \in \mathcal{L}(\mathcal{A}, q_i)$. For all $i \in [n+1, m]$, we have $q_{i,i} = \frac{\top}{q_i}$, which implies $\cdot \oplus s_i \in \mathcal{L}(\mathcal{A}, q_i)$. By definition of the standard convolution this means that $s \oplus t \in \mathcal{L}(\mathcal{A}, q_i)$. Similarly if we assume that $m \leq n$, then $N = n$ and for all $i \in [1, m]$ we have $q_{i,i} = q_i$. By hypothesis of induction this means that $s_i \oplus t_i \in \mathcal{L}(\mathcal{A}, q_i)$. For all $i \in [m+1, n]$, we have $q_{i,i} = \frac{q_i}{\top}$, which implies $t_i \oplus \cdot \in \mathcal{L}(\mathcal{A}, q_i)$. By definition of the standard convolution this means that $s \oplus t \in \mathcal{L}(\mathcal{A}, q_i)$.

This proves that any relation of $R_\oplus$ is a relation of $R_\otimes$. Besides, we have already exposed a regular relation of $R_\otimes$ that is not in $R_\oplus$ in Example 6.3.2. Overall this proves that $R_\oplus \subset R_\otimes$. □

6.4 Relations Inference

In this section, we define a procedure to automatically learn regular tree relations using the information gathered from the program definition. For instance, consider the following (first-order) program defined with the following term rewriting system $\mathcal{R}$:

$$\begin{array}{cc} \mathit{has\text{-}length}(\mathit{nil}, 0) \to \mathit{true} & \mathit{has\text{-}length}(\mathit{cons}(\underline{x}, \underline{l}), 0) \to \mathit{false} \\ \mathit{has\text{-}length}(\mathit{nil}, s(\underline{y})) \to \mathit{false} & \mathit{has\text{-}length}(\mathit{cons}(\underline{x}, \underline{l}), s(\underline{y})) \to \mathit{has\text{-}length}(\underline{l}, \underline{y}) \end{array}$$

The function *has-length* associates a list with its length. We can easily translate the rewriting system $\mathcal{R}$ into a system of constraints over *has-length*, expressed as first-order logical formulas called *Constrained Horn Clauses* (CHC):

$$\begin{aligned} \textit{has-length}(\textit{nil}, 0) \\ \forall \underline{y}.\ \neg \textit{has-length}(\textit{nil}, s(\underline{y})) \\ \forall \underline{x}, \underline{l}.\ \neg \textit{has-length}(\textit{cons}(\underline{x}, \underline{l}), 0) \\ \forall \underline{x}, \underline{y}, \underline{l}.\ \textit{has-length}(\underline{l}, \underline{y}) \iff \textit{has-length}(\textit{cons}(\underline{x}, \underline{l}), s(\underline{y})) \end{aligned}$$

Solving this constraint system gives us a definition of *has-length* that satisfies the formulas. There already exists a plethora of techniques and tools to solve a CHC system, some based on satisfiability modulo theories [dMB08, BCD+11] machine learning approaches [GLMN14, CCKS18, CKS18], etc. However most of these methods focus on numerical domains and are not well suited to infer regular relations on algebraic data types. We define in this section a regular CHC solving technique based on the ICE framework [GLMN14] that has already successfully been used on numerical domains.

6.4.1 Constrained Horn Clauses Solving

Definition 6.4.1 (Constrained Horn Clause)**.** *Let Σ be a ranked alphabet of constructor symbols, and $\mathscr{R}$ be a ranked alphabet of relation symbols. Let $\mathscr{M}$ be a* partial model *mapping* some *elements p of $\mathscr{R}$ to its interpretation, a relation $\mathscr{M}(p)$ of $T(\Sigma)^n$ where $ar(p) = n$. We say that $p \in \mathscr{R}$ is* concrete *if $\mathscr{M}(p)$ is defined, and* abstract *otherwise. A constrained Horn clause (CHC) over $\mathscr{R}$ and $\mathscr{M}$ is a first-order formula φ of the form*

$$\forall \mathcal{X}.\ \alpha_1 \wedge \cdots \wedge \alpha_n \Rightarrow \alpha_{n+1}$$

where $\mathcal{X}$ is a set of variables, and each α_i is either (i) a concrete formula, i.e. composed of concrete relations and variables of $\mathcal{X}$ (ii) an abstract predicate application over variables of $\mathcal{X}$. In our case, a predicate application is a pattern of the form $p(t_1, \ldots, t_n)$ where $p \in \mathscr{R}$ with $t_1, \ldots, t_n \in \mathcal{T}(\Sigma, \mathcal{X})$. We often write $\vec{t}$ instead of $t_1, \ldots, t_n$ and $\vec{t}_\otimes$ for $t_1 \otimes \cdots \otimes t_n$. The left hand side of the clause is called the "body" of the clause, while the right hand side is the "head".

We say that a model $\mathscr{M}$ satisfies a CHC φ, written $\mathscr{M} \models \varphi$, if it is true considering the interpretation of predicates provided by $\mathscr{M}$. Solving a CHC system S (a set of CHCs) defined over $\mathscr{R}$ and $\mathscr{M}$ consist in extending, or completing, the model $\mathscr{M}$ into a new model $\mathscr{M}'$ by finding interpretations for the abstract predicates such that $\mathscr{M}'$ satisfies every clause:

$$\forall \varphi \in S.\quad \mathscr{M}' \models \varphi$$

We then write $\mathscr{M}' \models S$. In our case, each predicate in the model, a regular relation, is represented by a regular language. Our goal is to automatically infer the missing regular languages that satisfy the input constraint system. To do that, we design an ICE-based inference procedure for regular relations.

6.4.2 ICE-Based Verification

First introduced in [GLMN14], "Implication Counter-Example" (ICE) is a robust framework for learning invariant that is particularly well suited to the resolution of

CHC systems, and has already been used to infer invariants on higher-order functional programs [CCKS18]. It works by combining a *learner* that iteratively produces candidate models, and a *teacher* that verifies that the produced model satisfies the CHC system. When a CHC is violated, the teacher produces *learning constraints* guiding the learner during the process. The procedure can be summarized as follows. Starting from the empty model $\mathscr{M}_0$ where each abstract predicate is mapped to empty relations (in our case, empty tree automatic relations), the procedure generates a series of models $\mathscr{M}_1, \mathscr{M}_2, \ldots$ where $\mathscr{M}_{i+1}$ is given as

$$Learner(\bigcup_{k\in[0,i]} Teacher(\mathscr{M}_k))$$

The learner uses every set of constraints provided by the teacher for the current iteration and the previous ones. The procedure stops when $\mathscr{M}_{i+1} = \mathscr{M}_i$. The output of the teacher (and the input of the learner) is a set of constraints extracted from $\mathscr{M}_i$. We detail in this section the role of the teacher and the learner. The following definitions are adaptations of the definitions found in [CCKS18] to the theory of terms.

The Teacher

The teacher's role is to ensure that the models produced by the learner come closer and closer to the solution. Given the current model $\mathscr{M}_i$, the teacher produces a set of learning constraints to guide the learner. A solution has been found when the output of the teacher is empty. Contrarily to a standard CEGAR procedure, in the ICE framework learning constraints are not solely made of (counter-)examples. ICE defines three different sorts of learning constraints depending on the form of the violated formula.

Positive examples If the violated formula is of the form $\forall\mathcal{X}.p(\vec{t})$, then the teacher can extract an *example* of valuation for which p is not satisfied in the current model (given by the learner), where it should.

Definition 6.4.2 (Positive example)**.** *Let $\mathcal{X}$ be a set of variables. Let Σ be a ranked alphabet of constructor symbols, and $\mathscr{R}$ be a ranked alphabet of relation symbols. A learning* example *is a pair $\langle p(\vec{t}), \sigma\rangle$ where $p \in \mathscr{R}$ is an abstract predicate, $\vec{t}$ are patterns of $\mathcal{T}(\Sigma, \mathcal{X})$ and $\sigma : \mathcal{X} \mapsto \mathcal{T}(\Sigma)$ a substitution. An example $\langle p(\vec{t}), \sigma\rangle$ represents the fact that $p(\vec{t\sigma})$ should be satisfied in the final model.*

For instance, consider again the previous *has-length* example, and assume the learner just produced a model in which *has-length* is instantiated as the empty predicate. Then the CHC *has-length*$(nil, 0)$ is violated. The teacher then produces the example $\langle$*has-length*$(nil, 0), \emptyset\rangle$ for the learner so that in the next iteration the model satisfies *has-length*$(nil, 0)$.

Negative constraint If the formula is of the form $\forall\mathcal{X}.\alpha_1 \wedge \cdots \wedge \alpha_n \Rightarrow false$, then the teacher can extract a *negative constraint*: a valuation for which the formula is satisfied when it should not.

Definition 6.4.3 (Negative constraint)**.** *A learning* negative constraint *is a set of pairs $\{\langle p_1(\vec{t_1}), \sigma_1\rangle, \ldots, \langle p_k(\vec{t_k}), \sigma_k\rangle\}$ where for each $i \in [1, n]$, $p_i \in \mathscr{R}$ is an abstract predicate, $\vec{t_i}$ are patterns of $\mathcal{T}(\Sigma, \mathcal{X})$ and $\sigma_i : \mathcal{X} \mapsto \mathcal{T}(\Sigma)$ a substitution. A negative*

constraint represents the fact that there must exists i such that $p_i(t_i\sigma_i)$ is not *satisfied in the final model.*

For instance, assume the learner just produced a model in which *has-length* is instantiated as the *true* predicate (such that for all l, n, *has-length*(l, n) is satisfied). Then the CHC $\forall y.\neg$*has-length*$(nil, s(\underline{y}))$ is violated. The teacher can then produce the following negative constraint $\langle$*has-length*$(nil, s(\underline{y})), \{\underline{y} \mapsto 0\}\rangle$ for the learner so that in the next iteration the model does *not* satisfy *has-length*$(nil, s(0))$.

Implication constraint The distinctive feature of the ICE framework is to make use of formula of the form $\forall\mathcal{X}.\alpha_1 \wedge \cdots \wedge \alpha_n \Rightarrow p(\vec{t})$ where the head is an abstract predicate application. If it is violated in the current model, then the teacher extracts an *implication constraint.*

Definition 6.4.4 (Implication constraint)**.** *A learning* implication constraint *is a pair $(N, \langle p(\vec{t}), \sigma\rangle)$ where N is a negative constraint. An implication constraint represent the fact that in the final model, if the negative constraint N is not satisfied, then $p(\vec{t\sigma})$ must be satisfied.*

For instance, assume the learner just produced a model in which *has-length* is instantiated as the singleton $\{(nil, 0)\}$ (which satisfies the constraints given in the previous examples). Then the CHC $\forall x, y, l.$*has-length*$(\underline{l}, \underline{y}) \Rightarrow$ *has-length*$(cons(\underline{x}, \underline{l}), s(\underline{y}))$ is violated. The teacher can then produce the following implication constraint $(\{\langle$*has-length*$(\underline{l}, \underline{y}), \sigma\rangle\}, \langle$*has-length*$(cons(\underline{x}, \underline{l}), s(\underline{y})), \sigma'\rangle)$ for the learner, where $\sigma = \{\underline{l} \mapsto nil, \underline{y} \mapsto 0\}$ and $\sigma' = \{\underline{x} \mapsto a, \underline{l} \mapsto nil, \underline{y} \mapsto 0\}$. Then in the next iteration we have either *has-length*$(cons(a, nil), s(0))$ satisfied, or not *has-length*$(nil, 0)$. Of course the later is not possible because of the previous positive example, so the only possibility is to have *has-length*$(cons(a, nil), s(0))$.

Definition 6.4.5 (ICE Constraints Satisfaction)**.** *For an ICE constraint s, we write $\mathscr{M} \models s$ when the model $\mathscr{M}$ verifies s according to the following:*

- *If s is a positive example of the form $\langle p(\vec{t}), \sigma\rangle$, then $\mathscr{M} \models s$ iff $\mathscr{M}(p)(\vec{t\sigma})$;*
- *If s is a negative constraint of the form $\{\ \langle p_1(\vec{t_1}), \sigma_1\rangle, \ldots, \langle p_n(\vec{t_n}), \sigma_n\rangle\ \}$, then $\mathscr{M} \models s$ iff there exists $i \in [1, n]$ such that $\neg\mathscr{M}(p_i)(\vec{t_i\sigma_i})$;*
- *If s is an implication constraint of the form $(\{\ \ldots, \langle p_i(\vec{t_i}), \sigma_i\rangle, \ldots\ \}, \langle p(\vec{t}), \sigma\rangle)$, then $\mathscr{M} \models s$ iff either there exists i such that $\neg\mathscr{M}(p_i)(\vec{t_i\sigma_i})$ or $\mathscr{M}(p)(\vec{t\sigma})$.*

For a set of ICE constraints I, we write $\mathscr{M} \models I$ when for each $s \in I$ we have $\mathscr{M} \models s$

The Learner

The learner's goal is to output a candidate model ($\mathscr{M}_{i+1}$) using as input the constraints issued by the teacher during the previous iterations ($\cup_{k\in[0,i]} Teacher(\mathscr{M}_k)$). It is completely driven by the teacher and does not have access to the CHC system itself. This simplifies the role of the learner as it only has to focus on concrete examples. In [CCKS18], the learner is implemented using a predefined set of *qualifiers* [RKJ08] as in the original ICE framework [GLMN14]. Qualifiers are combined throughout the procedure using a decision tree built using the teacher's learning constraints. The qualifiers generally includes usual relations between numerical values which make it particularly well suited to learn numerical relations, but not regular relations on algebraic data types. In Section 6.4.4 we define a new SMT-based learner implementation that can infer tree automatic relations as defined in Section 6.3.

6.4.3 The Teacher

In this section we give some details on our implementation of the teacher in the case of tree automatic relations (cf. Section 6.3). The complete formalization of our implementation of the teacher is not yet ready, but this section tries to give a general idea of the principles underlying our algorithm. Remember that tree automatic relations are defined as regular relations represented using convoluted regular languages. Hence in our setting, the model $\mathscr{M}$ given to the teacher is defined as an application that maps each abstract predicate p to a regular language $\mathscr{M}(p)$. From $\mathscr{M}$ and a CHC system S, our teacher can extract at least one learning constraint for each violated CHC of S.

Synchronized Convoluted Runs

For a given CHC φ, the problem of finding a valuation that violates φ can be reduced to synchronized search of terms in a set of languages. For instance, consider again the CHC clause $\forall \underline{y}.\neg \mathit{has\text{-}length}(\mathit{nil}, s(\underline{y}))$ with the model $\mathscr{M}$ such that $\mathscr{M}(p) = \{l \otimes n \mid l \in \mathit{List}, n \in \mathbb{N}\}$. To find a negative constraint we need to search in the regular language $\mathscr{M}(p)$ a term that matches the pattern $\mathit{nil} \otimes s(\underline{y})$. For example the term $\mathit{nil} \otimes s(0)$ matches this pattern and gives us the negative constraint $\{\langle \mathit{has\text{-}length}(\mathit{nil}, s(\underline{y})), \{y \mapsto 0\}\rangle\}$.

To generalize, for any CHC of the form $\forall \mathcal{X}.\ p_1(\vec{t_1}) \wedge \cdots \wedge p_n(\vec{t_n}) \Rightarrow \mathit{false}$, we need to find a substitution σ such that for all i, $(\vec{t_i}\sigma)_\otimes \in \mathscr{M}(p_i)$. The same reasoning can be applied with a little more work to other forms of CHC formula. If φ has the form $\forall \mathcal{X}.\ p(\vec{t})$, the teacher can find an (counter-)example by looking for a substitution σ such that $(\vec{t}\sigma)_\otimes \notin \mathscr{M}(p)$ which can be done just as before by considering the complement of $\mathscr{M}(p)$. What we want is then $(\vec{t}\sigma)_\otimes \in \overline{\mathscr{M}(p)}$. Finally, if φ has the form $\forall \mathcal{X}.p_1(\vec{t_1}) \wedge \cdots \wedge p_n(\vec{t_n}) \Rightarrow p(\vec{t})$, we need to find a substitution σ such that $(\vec{t}\sigma)_\otimes \in \overline{\mathscr{M}(p)}$ and for all i, $(\vec{t_i}\sigma)_\otimes \in \mathscr{M}(p_i)$. In all of these situations, finding σ is done by finding a *synchronized convoluted run* in a set of tree automata.

Definition 6.4.6 (Synchronized Convoluted Run)**.** *Let Ω be a set of pairs $\langle \mathcal{A}, \vec{t}\rangle$ where $\mathcal{A}$ is a tree automaton and $\vec{t}$ patterns of $\mathcal{T}(\Sigma, \mathcal{X})$. Note that $\vec{t}$ may not be linear. A synchronized convoluted run over Ω is represented by substitution $\sigma : \mathcal{X} \mapsto \mathcal{T}(\Sigma)$ such that for all $\langle \mathcal{A}, \vec{t}\rangle$, $(\vec{t}\sigma)_\otimes \in \mathcal{L}(\mathcal{A})$. We write $SR(\Omega)$ the set of synchronized convoluted runs over Ω.*

Our teacher works by searching for at least one element of $SR(\Omega)$ for each clause of the system. If $SR(\Omega)$ is empty, the clause if verified. If $SR(\Omega)$ is not empty, the clause is violated and we can use the extracted element of $SR(\Omega)$ to build a learning constraint to give to the learner. In the rest of this section we give a constructive definition of $SR(\Omega)$ that can be used to design a search algorithm. Note that the algorithm itself is not given in this document. It closely follows the inductive definition of $SR(\Omega)$ given below, but require the addition of a loop detection system to cope with the infinite nature of $SR(\Omega)$. First, we start by decomposing the problem by defining $SR(\Omega)$ as follows:

$$SR(\Omega) = \bigcap_{\langle \mathcal{A}, \vec{t}\rangle} SR(\mathcal{A}, \vec{t})$$

where $SR(\mathcal{A}, \vec{t})$ is the set of synchronized runs described by σ such that $\vec{t\sigma}_\otimes \in \mathcal{L}(\mathcal{A})$.

In turn, $SR(\mathcal{A}, \vec{t})$ can be further decomposed as:

$$SR(\mathcal{A}, \vec{t}) = \bigcap_{q \in \mathcal{Q}_f} SR(\mathcal{A}, q, \vec{t})$$

where $\mathcal{Q}_f$ is the set of final states of $\mathcal{A}$, and $SR(\mathcal{A}, q, \vec{t})$ the set of synchronized runs described by σ such that $\vec{t\sigma}_{\otimes} \in \mathcal{L}(\mathcal{A}, q)$. Our objective is now to define $SR(\mathcal{A}, q, \vec{t})$ in a constructive manner. We have seen in Section 6.3 multiple definitions of the convolution operator. The definition of $SR(\mathcal{A}, q, \vec{t})$ depends on the actual definition of the convolution we use.

Standard Convolution

We focus here on the definition of $SR(\mathcal{A}, q, \vec{t})$ for the standard convolution operator, written $SR_{\oplus}(\mathcal{A}, q, \vec{t})$. To simplify, we only consider binary relations. However all the definitions and proofs can easily be extended to n-ary relations.

Definition 6.4.7. *Let $\mathcal{A}$ be a tree automatic binary relation. For each q and pair $(s,t) \in \mathcal{T}(\Sigma, \mathcal{X})^2$, $SR_{\oplus}(\mathcal{A}, q, (s,t))$ is defined as the smallest set inductively defined such that for each transition $\frac{f}{g}(q_1, \ldots, q_k) \to q$ in $\mathcal{A}$:*

(1) if $s = \underline{x}$ then if $\sigma' \in SR_{\oplus}(\mathcal{A}, q, \underline{x}\sigma, t\sigma)$ with $\sigma = \{ \underline{x} \mapsto f(\underline{x_1}, \ldots, \underline{x_n}) \}$ then $\sigma' \circ \sigma \in SR_{\oplus}(\mathcal{A}, q, (s,t))$.

(2) if $t = \underline{y}$ then $\sigma' \in SR_{\oplus}(\mathcal{A}, q, s\sigma, \underline{y}\sigma)$ with $\sigma = \{ \underline{y} \mapsto g(\underline{y_1}, \ldots, \underline{y_m}) \}$ then $\sigma' \circ \sigma \in SR_{\oplus}(\mathcal{A}, q, (s,t))$.

(3) if $s = f(s_1, \ldots, s_n)$ and $t = g(t_1, \ldots, t_m)$ (with $n > 0 \vee m > 0$) then if $\sigma \in SR_{\oplus}(\mathcal{A}, q_i, (s_i, t_i))$ for all $i \in [1, max(n,m)]$ then $\sigma \in SR_{\oplus}(\mathcal{A}, q, (s,t))$.

(4) if $s = f$ and $t = g$ then $\emptyset \in SR_{\oplus}(\mathcal{A}, q, (s,t))$ (where $\emptyset$ is the empty substitution).

The idea of this definition is to build synchronized convoluted runs by unrolling the input patterns following the transitions defined in $\mathcal{A}$. Variables are either eliminated and unrolled until none is left, in which case a synchronized run has been found and can be added to $SR_{\oplus}(\mathcal{A}, q, (s,t))$.

Example 6.4.1. *Consider the following tree automatic relation $\mathcal{A}$ defining the equality between natural numbers:*

$$\frac{0}{0} \to q \qquad\qquad \frac{s}{s}(q) \to q$$

We want to find a synchronized run in $\mathcal{A}$ for the patterns $\underline{x}$ and $s(\underline{y})$. To do that, we follow the definition of $SR_{\oplus}(\mathcal{A}, q, (\underline{x}, s(\underline{y})))$. Using (1) we start by "unrolling" the variable $\underline{x}$ along the transition $\frac{s}{s}(q) \to q$ of $\mathcal{A}$: we build a substitution $\sigma = \{ \underline{x} \mapsto s(\underline{x'}) \}$ and search for a synchronized run in $SR_{\oplus}(\mathcal{A}, q, (s(\underline{x'}), s(\underline{y})))$. Since both patterns start with the same symbol (s), we use (3) to consume this symbol and search for a synchronized run in $SR_{\oplus}(\mathcal{A}, q, (\underline{x'}, \underline{y}))$. Now we can use (1) and (2) to unroll both variables along the transition $\frac{0}{0} \to q$ of $\mathcal{A}$: we build a substitution $\sigma' = \{ \underline{x'} \mapsto 0, \underline{y} \mapsto 0 \}$ and search for a synchronized run in $SR_{\oplus}(\mathcal{A}, q, (0,0))$. According to (4), $\emptyset \in SR_{\oplus}(\mathcal{A}, q, (0,0))$. Now we go back and compose all the produced substitutions to find a synchronized run for our initial problem: $\emptyset \circ \sigma' \circ \sigma \in SR_{\oplus}(\mathcal{A}, q, (\underline{x}, s(\underline{y})))$ where $\emptyset \circ \sigma' \circ \sigma = \{ \underline{x} \mapsto s(0), \underline{y} \mapsto 0 \}$.

Theorem 12. *Let Σ be a ranked alphabet, with $\mathcal{A}$ a tree automatic binary relation for the standard convolution $\oplus$. For all terms $s, t \in \mathcal{T}(\Sigma, \mathcal{X})$ and state q of $\mathcal{A}$:*

$$s\sigma \oplus t\sigma \in \mathcal{L}(\mathcal{A}, q) \Longleftrightarrow \sigma \in SR_{\oplus}(\mathcal{A}, q, (s, t))$$

Proof. First, we prove that for all $s, t \in \mathcal{T}(\Sigma, \mathcal{X})$, for all σ, for all state q, $s\sigma \oplus t\sigma \in \mathcal{L}(\mathcal{A}, q) \Rightarrow \sigma \in SR_{\oplus}(\mathcal{A}, q, (s, t))$. We proceed by strong induction on the maximum depth of both $s\sigma$ and $t\sigma$, $max(|s\sigma|, |t\sigma|)$.

- if $max(|s\sigma|, |t\sigma|) = 1$ then there exists some f and g such that $s\sigma = f$ and $t\sigma = g$. By applying (1) and (2) if s or t are variables we know that $SR_{\oplus}(\mathcal{A}, q, (s, t))$ includes every substitution $\sigma\sigma'$ with $\sigma' \in SR_{\oplus}(\mathcal{A}, q, f, g)$. By (4), we know that $\emptyset \in SR_{\oplus}(\mathcal{A}, q, f, g)$, hence we deduce that $\sigma \in SR_{\oplus}(\mathcal{A}, q, (s, t))$.

- if $max(|s|, |t|) > 1$ then If $s = x$ then we can decompose σ into $\sigma_\alpha\sigma_\beta$ with $\sigma_\alpha = \{x \mapsto f(x_1, \ldots, x_n)\}$. Using (1) we know that $R_{\mathcal{A}}(q, (s, t))$ includes every $\sigma_\alpha\sigma'$ such that $\sigma' \in SR_{\oplus}(\mathcal{A}, q, f(x_1, \ldots, x_n), t\sigma_\alpha)$. We need to show that $\sigma_\beta \in SR_{\oplus}(\mathcal{A}, q, (f(x_1, \ldots, x_n), t\sigma_\alpha))$. If $t\sigma_\alpha = y$ then we can decompose σ_β into $\sigma_\gamma\sigma_\delta$ with with $\sigma_\gamma = \{y \mapsto g(y_1, \ldots, y_n)\}$. Using (2) we know that $R_{\mathcal{A}}(q, (f(x_1, \ldots, x_n), t\sigma_\alpha))$ includes every $\sigma_\gamma\sigma'$ such that $\sigma' \in SR_{\oplus}(\mathcal{A}, q, f(x_1, \ldots, x_n)\sigma_\gamma, g(y_1, \ldots, y_n))$. We need to show that the substitution σ_δ is in $SR_{\oplus}(\mathcal{A}, q, (f(x_1, \ldots, x_n)\sigma_\gamma, g(y_1, \ldots, y_n)))$. By definition, since $f(x_1, \ldots, x_n)\sigma_\gamma\sigma_\delta \oplus g(y_1, \ldots, y_n)\sigma_\delta \in \mathcal{L}(\mathcal{A}, q)$, there exists a transition in $\mathcal{A}$ of the form $\frac{f}{g}(q_1, \ldots, q_k) \to q$ such that for each i, $x_i\sigma_\gamma\sigma_\delta \oplus y_i\sigma_\delta \in \mathcal{L}(\mathcal{A}, q_i)$ (where $x_i\sigma_\gamma\sigma_\delta = \cdot$ and $y_i\sigma_\delta = \cdot$ when undefined, according to Definition 6.3.1). Since $max(|x_i\sigma_\gamma\sigma_\delta|, |y_i\sigma_\delta|) < max(|s\sigma|, |t\sigma|)$, by hypothesis of (strong) induction we deduce that $\sigma_\delta \in SR_{\oplus}(\mathcal{A}, q_i, (x_i\sigma_\gamma, y_i))$. By (3) we deduce that $\sigma_\delta \in SR_{\oplus}(\mathcal{A}, q, (f(x_1, \ldots, x_n)\sigma_\gamma, g(y_1, \ldots, y_n)))$. Going back we deduce that $\sigma_\beta \in SR_{\oplus}(\mathcal{A}, q, (f(x_1, \ldots, x_n), t\sigma_\alpha))$ and that $\sigma \in SR_{\oplus}(\mathcal{A}, q, (s, t))$. If s or t are not variables, we can use the same reasoning without using (1) and (2) and still get $s\sigma \oplus t\sigma \in \mathcal{L}(\mathcal{A}, q)$.

Now we prove that for all s, t, for all σ, $\sigma \in SR_{\oplus}(\mathcal{A}, q, (s, t)) \Rightarrow s\sigma \oplus t\sigma \in \mathcal{L}(\mathcal{A}, q)$. We proceed by induction on the maximum depth of both $s\sigma$ and $t\sigma$, $max(|s\sigma|, |t\sigma|)$.

- If $max(|s\sigma|, |t\sigma|) = 1$. If $s = x$, then using (1) σ can be decomposed into $\sigma_\alpha\sigma_\beta$ with $\sigma_\beta \in SR_{\oplus}(\mathcal{A}, q, f, t\sigma_\alpha)$ and $\sigma_\alpha = \{x \mapsto f\}$. If $t\sigma_\alpha = y$, then using (2) σ_β can be decomposed into $\sigma_\gamma\sigma_\delta$ with $\sigma_\delta \in SR_{\oplus}(\mathcal{A}, q, f\sigma_\gamma, g)$ and $\sigma_\gamma = \{y \mapsto g\}$. By definition we deduce there must exist a transition $\frac{f}{g} \to q$ in $\mathcal{A}$, which means that $\frac{f}{g} \in \mathcal{L}(\mathcal{A}, q)$, $f \oplus g \in \mathcal{L}(\mathcal{A}, q)$. Because $f = f\sigma_\gamma$ and $g = y\sigma_\gamma$ we get $f\sigma_\gamma \oplus y\sigma_\gamma \in \mathcal{L}(\mathcal{A}, q)$. Since both $f\sigma_\gamma$ and $y\sigma_\gamma$ are closed, we can apply σ_δ ans still have $f\sigma_\gamma\sigma_\delta \oplus y\sigma_\gamma\sigma_\delta \in \mathcal{L}(\mathcal{A}, q)$. And because $\sigma_\gamma\sigma_\delta = \sigma_\beta$ we get $f\sigma_\beta \oplus y\sigma_\beta \in \mathcal{L}(\mathcal{A}, q)$. Now because $y = t\sigma_\alpha$ and $f = s\sigma_\alpha$ then $s\sigma_\alpha\sigma_\beta \oplus t\sigma_\alpha\sigma_\beta \in \mathcal{L}(\mathcal{A}, q)$. Finally since $\sigma_\alpha\sigma_\beta = \sigma$ we deduce that $s\sigma \oplus t\sigma \in \mathcal{L}(\mathcal{A}, q)$. Again if s or t are not variables, we can use the same reasoning without using (1) and (2) and still get $s\sigma \oplus t\sigma \in \mathcal{L}(\mathcal{A}, q)$.

- If $max(|s\sigma|, |t\sigma|) > 1$ If $s = x$, then using (1) σ can be decomposed into $\sigma_\alpha\sigma_\beta$ with $\sigma_\beta \in SR_{\oplus}(\mathcal{A}, q, f(x_1, \ldots, x_n), t\sigma_\alpha)$ and $\sigma_\alpha = \{x \mapsto f(x_1, \ldots, x_n)\}$. If $t\sigma_\alpha = y$, then using (2) σ_β can be decomposed into $\sigma_\gamma\sigma_\delta$ with $\sigma_\delta \in SR_{\oplus}(\mathcal{A}, q, f(x_1, \ldots, x_n)\sigma_\gamma, g(y_1, \ldots, y_m))$ and $\sigma_\gamma = \{y \mapsto g(x_1, \ldots, y_n)\}$. The by definition, there exists a transition $\frac{f}{g}(q_1, \ldots, q_k)$ with $k = max(n, m)$ such

that for each $i \in [1,k]$ we have $\sigma \in SR_{\oplus}(\mathcal{A}, q_i, (x_i\sigma_\gamma, y_i))$ where, following (3), $x_i\sigma_\gamma = \cdot$ if $i > n$ and $y_i = \cdot$ if $j > m$. Then by hypothesis of induction we know that $x_i\sigma_\gamma \oplus y_i \in \mathcal{L}(\mathcal{A}, q_i)$, from which we deduce that $f(x_1, \ldots, x_n)\sigma_\gamma \oplus g(y_1, \ldots, y_m) \in \mathcal{L}(\mathcal{A}, q)$. Because $\sigma_\gamma = \{y \mapsto g(x_1, \ldots, y_n)\}$ we have $f(x_1, \ldots, x_n)\sigma_\gamma \oplus y\sigma_\alpha\sigma_\gamma \in \mathcal{L}(\mathcal{A}, q)$. Because $f(x_1, \ldots, x_n)\sigma_\gamma$ and $y\sigma_\alpha\sigma_\gamma$ are closed, we can safely apply σ_δ is still have $f(x_1, \ldots, x_n)\sigma_\gamma\sigma_\delta \oplus y\sigma_\alpha\sigma_\gamma\sigma_\delta \in \mathcal{L}(\mathcal{A}, q)$. Because $\sigma_\beta = \sigma_\gamma\sigma_\delta$ then $f(x_1, \ldots, x_n)\sigma_\beta \oplus y\sigma_\alpha\sigma_\beta \in \mathcal{L}(\mathcal{A}, q)$. Because $\sigma_\alpha = \{x \mapsto f(x_1, \ldots, x_n)\}$ we have $x\sigma_\alpha\sigma_\beta \oplus y\sigma_\alpha\sigma_\beta \in \mathcal{L}(\mathcal{A}, q)$. Finally because $\sigma = \sigma_\alpha\sigma_\beta$ we have $x\sigma \oplus y\sigma \in \mathcal{L}(\mathcal{A}, q)$ from which we conclude that $s\sigma \oplus t\sigma \in \mathcal{L}(\mathcal{A}, q)$. Once again if s or t are not variables, we can use the same reasoning without using (1) and (2) and still get $s\sigma \oplus t\sigma \in \mathcal{L}(\mathcal{A}, q)$.

□

The inductive definition of $SR(\Omega)$ is not quite sufficient to be directly translated into an actual algorithm. Such an algorithm would also require the addition of a loop detection system to cope with the infinite nature of $SR(\Omega)$, which is not given in this document but implemented in our Regular CHC solver [Hau19]. If it follows our definition of $SR(\Omega)$, we can still show that the teacher is sound and complete (when it terminates), but have not yet proved its termination.

Conjecture 6.4.1 (Teacher Termination)**.** *For all CHC constraint system S and model $\mathscr{M}$ of S, $Teacher(\mathscr{M})$ is defined.*

Theorem 13 (Teacher is Sound)**.** *Let S be a CHC constraint system and $\mathscr{M}$ a model for S. For all ICE constraint $s \in Teacher(\mathscr{M})$ we have $\mathscr{M} \not\models s$.*

Proof. If s is a positive example of the form $\langle p(\vec{t}), \sigma\rangle$ then $\sigma \in SR(\{\ \langle\overline{\mathscr{M}(p)}, \vec{t}\rangle\ \})$ which has been computed because there exists a clause $\varphi \in S$ of the form $\forall\mathcal{X}.p(\underline{t})$. By Theorem 12 this means that $(\vec{t\sigma})_\oplus \notin \mathcal{L}(\mathscr{M}(p))$ which means that $\mathscr{M} \not\models \varphi$. Otherwise if s is a negative example of the form $\{\ \langle p_1(\vec{t_1}), \sigma\rangle, \ldots, \langle p_n(\vec{t_n}), \sigma\rangle\ \}$ then $\sigma \in SR(\{\ \langle\mathscr{M}(p_1), \vec{t_1}\rangle, \ldots, \langle\mathscr{M}(p_n), \vec{t_n}\rangle\ \})$ which has been computed because there exists a clause $\varphi \in S$ of the form $\forall\mathcal{X}.p_1(\underline{t_1}) \wedge \cdots \wedge p_n(\underline{t_n}) \Rightarrow \mathit{false}$. By Theorem 12 this means that for all i, $(\vec{t_i\sigma})_\oplus \in \mathcal{L}(\mathscr{M}(p_i))$ which means that $\mathscr{M} \not\models \varphi$. Otherwise s is an implication constraint of the form $\langle\{\ \langle p_1(\vec{t_1}), \sigma\rangle, \ldots, \langle p_n(\vec{t_n}), \sigma\rangle\ \}, \langle p(\vec{t}), \sigma\rangle\rangle$ then $SR(\{\ \langle\mathscr{M}(p_1), \vec{t_1}\rangle, \ldots, \langle\mathscr{M}(p_n), \vec{t_n}\rangle, \langle\overline{\mathscr{M}(p)}, \vec{t}\rangle\ \})$ which has been computed because there exists a clause $\varphi \in S$ of the form $\forall\mathcal{X}.p_1(\underline{t_1}) \wedge \cdots \wedge p_n(\underline{t_n}) \Rightarrow p(\vec{t})$. By Theorem 12 this means that for all i, $(\vec{t_i\sigma})_\oplus \in \mathcal{L}(\mathscr{M}(p_i))$ and $(\vec{t\sigma})_\oplus \notin \mathcal{L}(\mathscr{M}(p))$ which means that $\mathscr{M} \not\models \varphi$. □

Theorem 14 (Teacher is Complete)**.** *Let S be a CHC constraint system. Let $\mathscr{M}$ be a model. If there exists a clause $\varphi \in S$ such that $\mathscr{M} \not\models \varphi$ then $Teacher(\mathscr{M}) \neq \emptyset$*

Proof. If φ is of the form $\forall\mathcal{X}.p(\underline{t})$ then by Theorem 12 there exists a substitution σ in $SR(\{\ \langle\overline{\mathscr{M}(p)}, \vec{t}\rangle\ \})$. Then $\langle p(\vec{t}), \sigma\rangle \in Teacher(\mathscr{M})$. Otherwise if φ is of the form $\forall\mathcal{X}.p_1(\underline{t_1}) \wedge \cdots \wedge p_n(\underline{t_n}) \Rightarrow \mathit{false}$ then by Theorem 12 there exists $\sigma \in SR(\{\ \langle\mathscr{M}(p_1), \vec{t_1}\rangle, \ldots, \langle\mathscr{M}(p_n), \vec{t_n}\rangle\ \})$. Then $\{\ \langle p_1(\vec{t_1}), \sigma\rangle, \ldots, \langle p_n(\vec{t_n}), \sigma\rangle\ \} \in Teacher(\mathscr{M})$. Otherwise φ is of the form $\forall\mathcal{X}.p_1(\underline{t_1}) \wedge \cdots \wedge p_n(\underline{t_n}) \Rightarrow p(\vec{t})$ then by Theorem 12 there exists a substitution σ in $SR(\{\ \langle\mathscr{M}(p_1), \vec{t_1}\rangle, \ldots, \langle\mathscr{M}(p_n), \vec{t_n}\rangle, \langle\overline{\mathscr{M}(p)}, \vec{t}\rangle\ \})$. Then $\langle\{\ \langle p_1(\vec{t_1}), \sigma\rangle, \ldots, \langle p_n(\vec{t_n}), \sigma\rangle\ \}, \langle p(\vec{t}), \sigma\rangle\rangle$ is in $Teacher(\mathscr{M})$. □

Full Convolution

We can naturally define $SR_{\otimes}(\mathcal{A}, q, \vec{t})$ be modifying Definition 6.4.7. The only part of the definition that changes is the rule (3) which actually reflects the behavior of the convolution operator.

Definition 6.4.8. *Let $\mathcal{A}$ be a tree automatic binary relation. For each q and pair (s,t), $SR_{\otimes}(\mathcal{A}, q, (s,t))$ is defined as the smallest set inductively defined such that for each transition $\frac{f}{g}(q_1, \ldots, q_k) \to q$ in $\mathcal{A}$, it includes:*

(1) if $s = \underline{x}$ then if $\sigma' \in SR_{\oplus}(\mathcal{A}, q, \underline{x}\sigma, t\sigma)$ with $\sigma = \{\ \underline{x} \mapsto f(\underline{x_1}, \ldots, \underline{x_n})\ \}$ then $\sigma' \circ \sigma \in SR_{\oplus}(\mathcal{A}, q, (s,t))$.

(2) if $t = \underline{y}$ then $\sigma' \in SR_{\oplus}(\mathcal{A}, q, s\sigma, \underline{y}\sigma)$ with $\sigma = \{\ \underline{y} \mapsto g(\underline{y_1}, \ldots, \underline{y_m})\ \}$ then $\sigma' \circ \sigma \in SR_{\oplus}(\mathcal{A}, q, (s,t))$.

(3) if $s = f(s_1, \ldots, s_n)$ and $t = g(t_1, \ldots, t_m)$ (with $n > 0 \vee m > 0$) then if $\sigma \in SR_{\oplus}(\mathcal{A}, q_i, (s_{l(i)}, t_{r(i)}))$ for all $i \in [1, n \times m]$ where $l(i) = \lceil k/m \rceil$ and $r(i) = k \bmod n$ (following Definition 6.3.3), then $\sigma \in SR_{\oplus}(\mathcal{A}, q, (s,t))$.

(4) if $s = f$ and $t = g$ then $\emptyset \in SR_{\oplus}(\mathcal{A}, q, (s,t))$ (where $\emptyset$ is the empty substitution).

Theorem 15. *Let Σ be a ranked alphabet, with $\mathcal{A}$ a tree automatic binary relation for the full convolution $\otimes$. For all terms $s, t \in \mathcal{T}(\Sigma)$ and state q of $\mathcal{A}$:*

$$s\sigma \otimes t\sigma \in \mathcal{L}(\mathcal{A}, q) \iff \sigma \in SR_{\otimes}(\mathcal{A}, q, (s,t))$$

The proof of this theorem closely follows the one of Theorem 12.

6.4.4 The Learner

In this section we give an implementation of the learner, that uses learning constraints from the teacher to find a model that verifies the CHC system. In our case, the model maps each abstract predicate to a tree automatic relation, a tree automaton. To learn those tree automata we use an SMT-based procedure similar to the one developed in Section 5.3.4. However this time, we need to build multiple automata at the same time, one for each abstract predicate. Let $\mathscr{R}$ be the considered set of abstract predicates. We first define the *working context* for the learner at iteration i, $\mathscr{A}_i$, as a function that maps each predicate p to an automaton recognizing every term for which we know something about p. Then, the constraints on terms emitted by the teacher are translated into SMT constraints about the states of the automata in the working context. As in Section 5.3.4, this gives us a candidate model for the CHC system that is submitted back to the teacher.

Definition 6.4.9 (Learner's Working Context)**.** *Let $\mathscr{R}$ be a set of abstract predicates. A* working context *$\mathscr{A}$ is a function that maps each predicate p to a tree automaton (a tree automatic relation). We write $\mathscr{A}_i$ the working context at iteration i of the procedure, with $\mathscr{A}_0$ the empty working context that maps each predicate to the empty automaton.*

The language recognized by $\mathscr{A}(p)$ contains every term for which the learner knows *something*. Note that this does not only include the members of the relation p but also some other constrained terms (for instance terms that *must not* be members of the relation). The automaton $\mathscr{A}(p)$ essentially recognizes terms from which the

teacher has given the learner some information w.r.t. p. The working context is meant to be enriched at each iteration from the data produced by the teacher starting from $\mathscr{A}_0$, however to simplify here we can consider it empty at the beginning of each iteration. The model $\mathscr{M}_{i+1}$ outputted by the learner at iteration $i+1$ is generated from:

1. The input of the learner, written I, which is the set of ICE constraints accumulated from the teacher, $\cup_{k\in[0,i]} Teacher(\mathscr{M}_k)$;
2. $\mathscr{A}_i$ the working context of the previous iteration;
3. $\langle \mathscr{A}_{i+1}, \Gamma\rangle = \Gamma(\mathscr{A}_i, I)$ where Γ is a set of SMT-constraints extracted from I about the states of the automata defined $\mathscr{A}_{i+1}$ derived from $\mathscr{A}_i$; and
4. $\Psi(\mathscr{A}_{i+1})$ an additional set of SMT-constraints required to keep the model automata deterministic.

The model generated by the learner is defined as a *minimal* (with respect to the number of generated constants) solution ϕ to the SMT-constraints $\Gamma \cup \Psi(\mathscr{A}_{i+1})$:

$$\mathscr{M}_{i+1}(p) = \phi(\mathscr{A}_{i+1}(p))$$

SMT constraints are composed of equality constraints between the tree automata states, and of constraints of the form $p(q)$ (or $\neg p(q)$) deciding if a state q belong to the predicate p. The later constraints are used to decide which tree automata states should be final in the generated model. For instance if $(s \otimes t)$ is recognized by the state q in the automaton $\mathscr{A}_{i+1}(p)$, the SMT-constraint "$p(q)$" means that $p(s,t)$ is true, and q should be instantiated by a final state of $\mathscr{M}_{i+1}(p)$ so that $(s \otimes t) \in \mathcal{L}(\mathscr{M}_{i+1}(p))$.

Definition 6.4.10 (ICE to SMT Constraints)**.** *Let $\mathscr{R}$ be a set of abstract relations. Let $\mathscr{A}$ be a working context mapping each abstract relation to a tree automaton. Let s be a learning constraint emitted by the teacher. The following function $\Gamma(\mathscr{A}, s)$ computes $\langle \mathscr{A}', \gamma\rangle$ where γ is the (unique) SMT constraint associated with the sample s in the new working context $\mathscr{A}'$ derived from $\mathscr{A}$. SMT-constraints are written between " . " in the following definition of $\Gamma(\mathscr{A}, s)$:*

1. *(positive example)* $\Gamma(\mathscr{A}, \langle p(\vec{t}), \sigma\rangle) = \langle \mathscr{A}[p \mapsto \mathcal{A}'], \text{“ } p(q) \text{ ”}\rangle$ *where* $\langle \mathcal{A}', q\rangle = Norm(\mathscr{A}(p), t\vec{\sigma}_{\otimes})$;
2. *(negative constraint)* $\Gamma(\mathscr{A}, \{\langle p_1(\vec{t_1}), \sigma_1\rangle, \ldots, \langle p_n(\vec{t_n}), \sigma_n\rangle\}) =$ $\langle \mathscr{A}_n, \text{“ } \neg p_i(q_i) \vee \cdots \vee \neg p_n(q_n) \text{ ”}\rangle$ *where* $\mathscr{A}_0 = \mathscr{A}$ *and for all* $i \in [1,n]$, $\mathscr{A}_i = \mathscr{A}_{i-1}[p_i \mapsto \mathcal{A}_i]$ *with* $\langle \mathcal{A}_i, q_i\rangle = Norm(\mathscr{A}_{i-1}[p_i], t_i\vec{\sigma}_i)$;
3. *(implication constraint)* $\Gamma(\mathscr{A}, (\{\langle p_1(\vec{t_1}), \sigma_1\rangle, \ldots, \langle p_n(\vec{t_n}), \sigma_n\rangle\}, \langle p(\vec{t}), \sigma\rangle)) =$ $\langle \mathscr{A}_n, \text{“ } p_i(q_i) \wedge \ldots \wedge p_n(q_n) \Rightarrow p(q) \text{ ”}\rangle$ *where* $\mathscr{A}_0 = \mathscr{A}[p \mapsto \mathcal{A}']$ *with* $\langle \mathcal{A}', q\rangle = Norm(\mathcal{A}, t\vec{\sigma})$ *and for all* $i \in [1,n]$, $\mathscr{A}_i = \mathscr{A}_{i-1}[p_i \mapsto \mathcal{A}_i]$ *with* $\langle \mathcal{A}_i, q_i\rangle = Norm(\mathscr{A}_{i-1}[p_i], t_i\vec{\sigma}_i)$;

We write $\mathscr{A}[p \mapsto \mathcal{A}]$ for the new working context derived from $\mathscr{A}$ where p is now associated with $\mathcal{A}$. If I is a set of ICE constraints, then the set of SMT-constraints associated with I (along with the updated working context), $\Gamma(\mathscr{A}, I)$, is simply defined by successive applications of $\Gamma(\mathscr{A}, s)$ for each constraint s of I.

Definition 6.4.11 (Determinism Constraints)**.** *$\Psi(\mathscr{A})$ is the smallest set such that for all $p \in \mathscr{R}$ such that $f(q_1, \ldots, q_n) \to q \in \mathscr{A}(p)$ and $f(q'_1, \ldots, q'_n) \to q \in \mathscr{A}(p)$ then*

$$q_1 \neq q'_1 \vee \ldots \vee q_n \neq q'_n \vee q = q' \in \Psi(\mathscr{A})$$

A solution ϕ to an SMT constraint system gives a renaming (or interpretation) of the working automata states into a new set of states $\mathcal{Q}$ for which the constraints are respected. It also gives $\phi(p)$, the set of $\mathcal{Q}$-states that belong to p according to the SMT constraints. For each predicate p, the automaton $\phi(\mathscr{A}(p))$ is $\langle \Sigma, \mathcal{Q}, \phi(p), \phi(\Delta)\rangle$ where Δ is the TRS defining $\mathscr{A}(p)$, and $\phi(\Delta)$ is obtained by renaming the states in Δ using ϕ.

Example 6.4.2. *Consider the following CHC system specifying the equality predicate eq on natural numbers:*

$$\begin{aligned} & eq(0,0) \\ \forall \underline{y}.\ & \neg eq(0, s(\underline{y})) \\ \forall \underline{x}.\ & \neg eq(s(\underline{x}), 0) \\ \forall \underline{x}, \underline{y}.\ & eq(\underline{x}, \underline{y}) \iff eq(s(\underline{x}), s(\underline{y})) \end{aligned}$$

Suppose that the teacher already issued to the learner the following ICE constraints I about the eq predicate: one positive example $eq(0,0)$, one negative constraint $\{\ eq(0, s(0))\ \}$ and one implication constraint $\langle\{\ eq(0,0)\ \}, eq(s(0), s(0))\rangle$. We start by building the working context $\mathscr{A}$ along with the SMT constraints γ derived from the ICE constraints. This is done by computing $\langle \mathscr{A}, \gamma\rangle = \Gamma(\mathscr{A}_0, I)$ where $\mathcal{A}_0$ is the empty working context. Let us begin with the positive constraint by computing $\Gamma(\mathscr{A}_0, eq(0,0))$. According to Definition 6.4.10, the positive constraint is translated into the SMT constraint "$eq(q_1)$", where q_1 is the state recognizing the term $(0 \otimes 0)$ in $\mathscr{A}_1(eq)$, where $\mathscr{A}_1$ the new working context derived from $\mathcal{A}_0$, where we have added the necessary states in $\mathscr{A}_1(eq)$ to recognize $(0 \otimes 0)$ in q_1. According to the same definition, The SMT constraint associated with the negative ICE constraint $\{\ eq(0, s(0))\ \}$ is "$\neg eq(q_2)$" where q_2 is the state recognizing the term $(0 \otimes s(0))$ in $\mathscr{A}_2(eq)$ (derived from $\mathscr{A}_1$). Finally the SMT constraint associated with the ICE implication constraint $\langle\{\ eq(0,0)\ \}, eq(s(0), s(0))\rangle$ is "$eq(q_1) \Rightarrow eq(q_3)$" where q_3 the state recognizing $(s(0) \otimes s(0))$ in $\mathscr{A}_3(eq)$ (derived from $\mathscr{A}_2$). The final working context defines the following automaton for eq:

$$\frac{0}{0} \to q_1 \qquad \frac{0}{s}(q_2') \to q_2 \qquad \frac{\cdot}{0} \to q_2' \qquad \frac{s}{s}(q_1) \to q_3$$

associated with the following SMT constraints in γ:

$$eq(q_1) \qquad \neg eq(q_2) \qquad eq(q_1) \Rightarrow eq(q_3)$$

A minimal solution ϕ to this constraint system is composed of two states q and q' such that $\phi(q_1) = \phi(q_3) = q$ and $\phi(q_2) = \phi(q_2') = q'$ with $\phi(eq) = \{\ q\ \}$ which correspond to set final states of the following solution automaton $\phi(\mathscr{A}(eq))$ (once reduced):

$$\frac{0}{0} \to q \qquad \frac{s}{s}(q) \to q$$

This tree automatic relation gives a candidate instantiation of the eq predicate that satisfies the ICE constraints given by the teacher. In fact, this tree automatic relation is a solution to our initial CHC system.

Theorem 16 (Learner is sound)**.** *Let I be a set of ICE constraints. Then we have $Learner(I) \models I$.*

Proof. Let $\mathscr{M} = Learner(I)$. For each $s \in I$ we have $\mathscr{M} \models s$.

- If s is a positive example of the form $\langle p(\vec{t}), \sigma \rangle$ then by definition of $\langle \mathscr{A}, \gamma \rangle = \Gamma(\mathscr{A}, I)$ there exists a unique state q of $\mathscr{A}'(p)$ such that $\vec{t\sigma}_{\otimes} \in \mathcal{L}(\mathscr{A}'(p), q)$, and γ contains the constraint " $p(q)$ ". Then by definition of $S(I)$ and ϕ_I, there exists a unique state q of $\mathscr{M}(p)$ such that $\vec{t\sigma}_{\otimes} \in \mathcal{L}(\mathscr{M}(p), q)$ and q is a final state $\mathscr{M}(p)$. Then by definition, $\mathscr{M} \models s$.

- If s is a negative constraint of the form $\{ \langle p_1(\vec{t_1}), \sigma_1 \rangle, \ldots, \langle p_n(\vec{t_n}), \sigma_n \rangle \}$ then by definition of $\langle \mathscr{A}', \gamma \rangle = \Gamma(\mathscr{A}, I)$ for each $i \in [1, n]$ there exists a unique state q_i of $\mathscr{A}'(p_i)$ such that $t_i\vec{\sigma}_{i\otimes} \in \mathcal{L}(\mathscr{A}'(p_i), q_i)$. And in addition, γ contains the constraint " $\neg p_1(q_1) \vee \cdots \vee \neg p_n(q_n)$ ". Then by definition of $S(I)$ and ϕ_I, for each $i \in [1, n]$ there exists a unique state q_i of $\mathscr{M}(p_i)$ such that $t_i\vec{\sigma}_{i\otimes} \in \mathcal{L}(\mathscr{M}(p_i), q_i)$, and there exists one i for which q_i is *not* a final state of $\mathscr{M}(p_i)$. Then by definition, $\mathscr{M} \models s$.

- If s is an implication constraint of the form $\{ \langle p_1(\vec{t_1}), \sigma_1 \rangle, \ldots, \langle p_n(\vec{t_n}), \sigma_n \rangle \}$ then by definition of $\langle \mathscr{A}', \gamma \rangle = \Gamma(\mathscr{A}, I)$ for each $i \in [1, n]$ there exists a unique state q_i of $\mathscr{A}'(p_i)$ such that $t_i\vec{\sigma}_{i\otimes} \in \mathcal{L}(\mathscr{A}'(p_i), q_i)$ and there exists a unique state q of $\mathscr{A}'(p)$ such that $\vec{t\sigma}_{\otimes} \in \mathcal{L}(\mathscr{A}'(p), q)$. And in addition, γ contains the constraint " $\neg p_1(q_1) \vee \cdots \vee \neg p_n(q_n) \vee p(q)$ ". Then by definition of $S(I)$ and ϕ_I, there exists a unique state q of $\mathscr{M}(p)$ such that $\vec{t\sigma}_{\otimes} \in \mathcal{L}(\mathscr{M}(p), q)$ and for each $i \in [1, n]$ there exists a unique state q_i of $\mathscr{M}(p_i)$ such that $t_i\vec{\sigma}_{i\otimes} \in \mathcal{L}(\mathscr{M}(p_i), q_i)$. And either there exists one i for which q_i is *not* a final state of $\mathscr{M}(p_i)$ or q *is* a final state of $\mathscr{M}(p)$. Then by definition, $\mathscr{M} \models s$.

□

Note the the learner does not depend on the actual definition of the convolution operator. In fact, we can design the interaction between the teacher and the learner in such a way that learning constraint are already expressed as convoluted terms. This way, only the teacher needs to reason about convolutions. The learner only has to reason about terms in simple tree automata.

6.4.5 Soundness and Completeness

We now show some important properties about the whole procedure combining the defined teacher and learner. We show that it is sound and complete on regular models. Remember however that we are still working under Conjecture 6.4.1.

Theorem 17 (Soundness)**.** *Let S be a CHC system and i such that $\mathscr{M}_{i+1} = \mathscr{M}_i$. Then $\mathscr{M}_i \models S$.*

Proof. If $\mathscr{M}_{i+1} = \mathscr{M}_i$ then by definition $(Teacher)(\mathscr{M}_i) = \emptyset$. This means that for each CHC in S, the teacher has not generated any ICE learning constraint. From Theorem 14 we deduce this is because $\mathscr{M}_i \models S$. □

Theorem 18 (Completeness)**.** *Let S be a CHC system. If there exists some regular model $\mathscr{M}$ (mapping each abstract predicate to a tree automatic relation) that verifies S, then this procedure will eventually find it.*

Proof. If there exists some regular model $\mathscr{M}$ such that $\mathscr{M} \models S$, then it must verifies every ICE constraint generated by the teacher and can be represented using a finite number of automata states N. By definition, the learner always build a minimal model (in the number of tree automata states) that verifies the ICE constraints

generated by the teacher. Furthermore, because new constraints are introduced after each iteration i if $\mathscr{M}_i \not\models S$ and old constraints are preserved, then $\mathscr{M}_{i+1}$ is different than every previously generated model. And since there are a finite number of models with less than N automata states, we know the procedure will eventually produce the solution with N states (or an equivalent solution with N states). □

6.5 Experiments

We have implemented the tree automatic relation learning procedure described in this chapter in our own Regular CHC Solver [Hau19] written in Rust and using CVC4 [BCD+11] as our SMT solver backend for the learner (cf. Section 6.4.4). For now, only the standard convolution operator has been implemented, but it is enough to explore the practicability of the procedure. To our knowledge, this is the only existing CHC solver designed to handle regular relations on algebraic data types. Hence, the purpose of this section is not to compare our implementation with others, but to simply witness the advantages and limits of our technique.

6.5.1 Test Suite

This time the test suite is different from the one used in the previous chapters. The test suite used in this section, available on the solver's repository [Hau19], is limited to positive first-order regular relational problems. Each problem is expressed as a CHC system written in the SMT-LIBv2 specification language format. We consider here two categories of problems: relation inference, and relational safety property verification.

Relation inference Relation inference problems consist in the exact inference of a desired relation described using CHC constraints. It is the direct application of what have been described in this chapter. For instance, one of the considered problems consists in inferring the relation $\{\ (x, y) \mid x \in \mathbb{N}, y \in \mathbb{N},\ x \leq y\ \}$ using the following CHC system:

$$\begin{aligned} &\forall n.\ 0 \leq n \\ &\forall n.\ \neg(s + 1 \leq 0) \\ \forall n, m.\ &n \leq m \iff n + 1 \leq m + 1 \end{aligned}$$

This relation is regular and can be exactly inferred using our CHC solver.

Relational safety property verification In the context of functional program verification, most functions cannot be represented as regular relations. However we can still use our regular CHC solver to infer regular over-approximations of a given function and verify relational properties over it. For instance, consider the *add* function defined by the following TRS:

$$add(0, y) \to y \qquad\qquad add(s(x), y) \to s(add(x, y))$$

This function can be represented by the ternary relation associating its inputs to the correct output: $\mathcal{R} = \{\ (x, y, z) \mid add(x, y) \to^* z\ \}$. This relation is *not* regular, and cannot be exactly inferred by our CHC solver. However it is possible to infer a

regular over-approximation of the relation using the following CHC system:

$$\forall y.\ add(0, y, y)$$
$$\forall x, y.\ add(x, y, z) \Rightarrow add(s(x), y, s(z))$$

This CHC can easily be derived from the rewriting system. A solution to this problem is a regular relation that over-approximates $\mathcal{R}$. It can be used to verify safety properties over the rewriting system by adding supplementary clauses to the system. For instance we can verify that $add(s(x), s(y))$ never rewrites to 0 by adding the following clause:

$$\forall x, y.\ \neg add(s(x), s(y), 0)$$

By providing an over-approximation of $\mathcal{R}$, the solution model proves that *add* respects the last constraint even though this relation is not regular.

Lists alignment We have seen that with the standard convolution, it is not directly possible to relate the length of a list with an integer because of alignment considerations. As showed in Figure 6.2 it is possible to solve this problem by encoding the lists tail first. We have used this solution in our test suite to verify some properties such as `length (make-list n) = n`.

6.5.2 Experimental Results

Table 6.1 shows the results of our experiments. The first column gives a short description of the inferred relation (first half of the table) or the regular safety property verified (second half of the table). The second column shows the execution time averaged over 10 runs on a Intel® i7-7600U CPU, 4 2.80GHz cores, with the detailed contribution of the teacher and the learner. We observe that our regular

	Time (s)		
Relation/Property	Learning	Teaching	Total
$\{\ x \mid x \in \mathbb{N},\ x\ mod\ 2 = 0\ \}$	0.01±0.0	0.0±0.0	0.01±0.0
$\{\ (x, x) \mid x \in \mathbb{N}\ \}$	0.0±0.0	0.0±0.0	0.0±0.0
$\{\ (x, y) \mid x \in \mathbb{N}, y \in \mathbb{N},\ x \leq y\ \}$	0.0±0.0	0.0±0.0	0.0±0.0
`length (make-list n) = n`	0.0±0.0	0.01±0.001	0.02±0.0
`length (insert-sort l) = length l`	0.04±0.008	0.27±0.048	0.31±0.05
`x + 0 = x`	0.08±0.027	0.02±0.006	0.11±0.03
`x + 0 = 0 + x`	0.03±0.007	0.01±0.002	0.04±0.0
`(cons x l) != l`	0.0±0.0	0.0±0.0	0.0±0.0
`length (rev l) = length l`	0.04±0.008	0.77±0.183	0.82±0.18
`length (tail l) = (length l) - 1`	0.01±0.001	0.01±0.002	0.03±0.0
`(length l = 0)` $\iff$ `l = nil`	0.0±0.0	0.0±0.0	0.01±0.0
`(length l > 0)` $\iff$ `l != nil`	0.0±0.0	0.01±0.012	0.02±0.01
`append l nil = l`	0.04±0.012	1.99±0.094	2.03±0.1

Table 6.1: Experimental Results

CHC solver is able to verify regular relational properties that were out of the scope of previous techniques with satisfying performances. For now, since only the standard

convolution has been implemented, only relations over linear structures such as lists and integers can be tested. However even if a degradation of the performances is to be expected with the full convolution, these preliminary results are promising.

6.6 Conclusion

In this chapter we have built a Constrained Horn Clauses solver able to automatically infer regular models. This regular CHC solver is built upon tree automatic relations, an extension of string automatic relations that consists in representing relations as regular languages of convoluted terms. We have seen that in the case of tree automatic relations, multiple definitions of the convolution operator exist. In particular, we have defined a general convolution operator that extends the standard definition and can be used to represent strictly more tree relations as regular languages. Our regular CHC solver is an instance of the ICE [GLMN14, GNMR16, CCKS18] procedure, a robust framework for learning invariant using implication counter examples by conceptually separating a teacher and a learner. In our case the teacher part, in charge of validating candidate model, is reduced to the problem of finding synchronized convoluted runs in multiple automata. We provide a constructive definition of the set of synchronized convoluted runs for a given problem. On the other side, the learner, in charge of generating new models using the teacher generated constraints, is defined as an SMT-based tree automaton abstraction procedure similar to the one defined in Chapter 5 (Section 5.3.4). We have implemented our CHC solving technique in Rust, and used it to both automatically infer regular relations and automatically verify relational safety properties on tree-processing functional programs by over-approximating functions. For now, only the standard convolution has been implemented, but preliminary results are encouraging as it can verify relational properties on non-trivial function (such as the insertion sort) that were out of the scope of previous chapters techniques in a few millisecond. Some work remains to formally define the synchronized convoluted runs finding algorithm used by the teacher to (in)validate candidate models, and to adapt it to support the full convolution definition. Some more work remains to modularize this technique which could allow it scale and would allow us to define a more advanced regular language type system with relations.

Chapter 7

Conclusion and Future Work

In this thesis we have seen in details how regular tree languages and rewriting systems can be used to automatically verify regular safety properties on higher-order tree-processing functional programs. We showed that (1) it is possible to design complete regular abstraction procedures to find an over-approximation of a higher-order program's reachable states suitable to verify a target property (Chapter 4 and 5), (2) it is possible to modularize the procedure to analyze functions separately and handle more complex problems by stating the problem as a type inference procedure (Chapter 5), (3) it is possible to extend our abstraction procedure to automatically verify relational properties on functional programs without leaving the realm of regular languages using tree automatic relations (Chapter 6).

Equational Abstractions In Chapter 4 we pursued the work on the state of the art equational abstraction method based on the Tree Automata Completion algorithm and extended it to handle higher-order programs while preserving termination guarantees. We formally defined an automatic abstraction procedure based on this algorithm and proved that it is complete on a subclass of regular properties and complete in refutation. However we also showed that it is not possible to achieve full regular completeness with equational abstractions.

Regular Type Inference For this reason in Chapter 5 we defined a new verification technique based on the regular abstract interpretation of term rewriting systems. Instead of equations, this new technique uses an SMT-based regular language learning procedure to abstract the program execution, and can build non-functional regular abstractions. This allows the procedure to tackle any regular safety problem. By using this procedure as a component of a regular language types inference procedure, the analysis can be modularized to scale better when facing more complex problems. This technique has been implemented as the fourth version of the Timbuk [Tbk4] verification tool that outperform the previous version on our test suite.

Regular Relations Finally in Chapter 6 we explored how regular languages can be used to automatically verify relational properties. To do that we adapted our SMT-based regular language learning procedure to learn *regular tree relations*, a family of relations that can be encoded with regular languages by applying a convolution operator over each element of the relation. We extended the standard convolution operator into a full convolution operator that can be used to encode more relations with regular languages. We then defined a regular relation inference procedure based on the ICE framework dedicated to CHC solving and implemented it in our own

Regular CHC solver [Hau19]. To our knowledge this is the first formalization and implementation of a complete regular tree relation inference procedure.

Future Work In order to focus on the core issues related to regular language inference, we focused on the verification of regular safety properties on terminating programs already expressed as term rewriting systems and without polymorphism. This offers plenty of future research directions to find out how the our techniques can be extended to handle non-safety properties on non-terminating programs, on actual functional programming languages with type polymorphism. In the following we give some leads on how to approach these issues.

7.1 Non-Terminating Programs

The strong limitation imposed by our verification procedure so far is to require the TRS encoding the program to be *terminating*. This is required by the Tree Automata Completion algorithm used in our SMT-based iterative inference procedure in Section 5.3.4. At each iteration, this algorithm is used to compute the set terms reachable from a finite subset of the initial states of the program. Since the computation is exact, termination of the program is needed to guarantee the termination of the TAC algorithm. One possible way to lift this requirement is to perform a finite number of TAC completion steps (cf. Definition 4.2.3) at each iteration. If a term considered in the procedure has a normal form, it will eventually be found at some iteration. By definition, no constraints (other than to ensure determinism) will be generated over states of the completed automaton recognizing terms without normal forms (or for which the normal form is not yet known). This means that the resulting abstraction is not impacted by those states. This would allow us to keep the same completeness properties without requiring the program to terminate. However this may slow down the number of iterations necessary for the procedure to converge to a valid abstraction.

7.2 Non-Safety Problems

For now we only have discussed the verification of safety regular properties of the form $(\mu x.\neg\phi \wedge \Box x)$ (cf. Definition 2.4.2). As stated in Section 2.4.2, safety properties can be verified using a simpler family of abstraction: collapsing abstractions (cf. Definition 2.4.9). For a program defined by a term rewriting system $\mathcal{R}$, a collapsing abstraction is an $\mathcal{R}$-closed abstraction. In Chapter 4, the equation set $E_{\mathcal{R}}$ is responsible for the generation of collapsing abstractions, and it is the foundation of the termination guarantees of the equational TAC algorithm. In Chapter 5, our SMT-based regular language learning procedure is also designed to find collapsing abstractions. However it can be adapted to find non-collapsing abstractions. Remember that in this iterative procedure, a candidate abstraction is built from a finite REFD automaton $\mathcal{A}\langle\Sigma, \mathcal{Q}, \mathcal{Q}_f, \Delta\rangle$ representing an $\mathcal{R}$-closed fragment of $\mathcal{R}^*(I)$ where I is an initial language (the set of initial states), and a target type partition T encoding the safety property. The candidate abstraction is generated by solving the constraint system $S(\mathcal{A}, T)$ (cf. Definition 5.3.10). The most important constraints of $S(\mathcal{A}, T)$ required to solve the safety problem are of the form

$$q \neq q' \text{ when } \exists \tau, \tau' \in T.\ q \in T_{\mathcal{A}}(\tau) \wedge q' \in T_{\mathcal{A}}(\tau') \wedge \tau \neq \tau'$$

where $T_{\mathcal{A}}(\tau) = \{\ q \in \mathcal{Q}_f \mid \exists t \in \gamma(\tau).\ t \to_{\mathcal{R}}^* q\ \}$. To simplify, consider $T = \{true^{\#}, false^{\#}\}$. This constraint will separate final states that rewrites to *true* from the ones that rewrite to *false*. From a model checking point of view, $\mathcal{A}$ defines a model $M_{\mathcal{A}} = \langle \mathcal{Q}, \Delta_\epsilon, V \rangle$ using T as the set of propositions where $\Delta_\epsilon = \{\ q' \to q \mid q \to q' \in \Delta\ \}$ (the direction of the arrow is changed to represent the rewriting relation) and $V(\tau) = \{\ q \in \mathcal{Q} \parallel \gamma(\tau) \cap \mathcal{L}(\mathcal{A}, q) \neq \emptyset\ \}$. From this point of view, the constraints in $S(\mathcal{A}, T)$ will separate states that verifies the following property from those who do not:

$$\neg(\mu x.\ (false^{\#} \vee \Diamond x))$$

One can recognize the shape of a safety property, which explains why this procedure can verify safety properties. A possible way to extend the abstraction procedure to any property is to replace $S(\mathcal{A}, T)$ by an SMT-constraints system $S(\mathcal{A}, \phi)$ directly defined from the modal μ-calculus formula ϕ representing the property to verify. As with $S(\mathcal{A}, T)$, the constraints of $S(\mathcal{A}, \phi)$ must ensure determinism of the final abstraction automaton, and the separation of the abstraction of the final states for which ϕ is verified from those for which it is not. Note however that the generated abstraction is still a *subterm-collapsing* abstraction of the model $M_{\mathcal{R}} = \langle \mathcal{R}^*(I), \to_{\mathcal{R}}, V' \rangle$ (cf. Definition 4.2.4). This is because the automaton $\mathcal{A}$ from which the constraints are generated is already a *subterm-collapsing* abstraction of $M_{\mathcal{R}}$. One state of $\mathcal{Q}$ does not recognize only one term of $\mathcal{R}^*(I)$: if $q \in \mathcal{Q}$ recognizes the term $f(t_1, \ldots, t_n)$, it will also recognize every term of the form $f(u_1, \ldots, u_n)$ where $t_i \to_{\mathcal{R}}^* u_i$ for each i. This is a property inherited from the Tree Automata Completion algorithm that is also passed on to the final abstraction. This means that even if generalized, this procedure can only verify regular properties that can be verified with subterm-collapsing abstractions. This is the case of regular safety properties. It is also the case of any other regular property as long as the TRS encoding the program is expressed in continuation passing style (CPS).

Example 7.2.1 (Resource usage problem)**.** *In this example we use the generalization described above to verify a (non-safety) resource usage problem on a program. Consider the following program first used in Section 3.2.1:*

```
let rec g x = if _ then close x else (read x; g x)
```

The `_` *expression stands for a non-deterministic boolean value. We want to verify that from the initial state* `(g file)` *(where* `file` *is an opened file) no read occurs after a close. This program can be translated into a CPS term rewriting system as follows:*

$$\begin{aligned}
g\ \underline{x}\ \underline{k} &\to br(close\ \underline{x}\ \underline{k},\ read\ \underline{x}\ (h\ \underline{x}\ \underline{k})) & h\ \underline{x}\ \underline{k}\ () &\to g\ \underline{x}\ \underline{k}\\
read\ \underline{x}\ \underline{k} &\to \underline{k}\ () & br(\underline{x}, \underline{y}) &\to \underline{x}\\
close\ \underline{x}\ \underline{k} &\to \underline{k}\ () & br(\underline{x}, \underline{y}) &\to \underline{y}\\
end\ \underline{x} &\to \underline{x}
\end{aligned}$$

Each function now takes a new parameter $\underline{k}$ *representing the continuation of the program. The non-deterministic* `if-then-else` *control structure has been replaced by the non-deterministic* br *rules. In this representation,* $read$ *and* $close$ *have no actual effect we only want to track their use. The initial state of the program is represented by the term* $(g\ file\ end)$ *where* end *is the final continuation of the program (the end). It is the only member of the initial set of states* I*. The model represented by* $\mathcal{R}$ *and* I *is* $M_{\mathcal{R}} = \langle \mathcal{R}^*(I), \to_{\mathcal{R}}, V \rangle$ *with the two propositions* $read$ *and* $close$ *such that* $V(read) = \{\ (read\ f\ k) \mid f, k \in \mathcal{T}(\Sigma)\ \}$ *and* $V(close) = \{\ (close\ f\ k) \mid f, k \in \mathcal{T}(\Sigma)\ \}$.

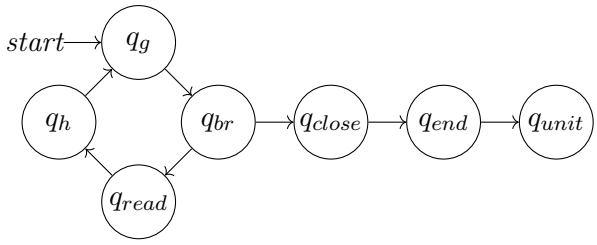
Our property can then be expressed as the following modal μ-calculus formula which must hold for any final state:

$$\phi = \neg\mu x.\ (\underbrace{close \wedge \Diamond\ \overbrace{(\mu y.\ read \vee \Diamond y)}^{\text{a read eventually occurs}}}_{\text{close then eventually read}}) \vee \Diamond x \qquad \text{no read occurs after a close}$$

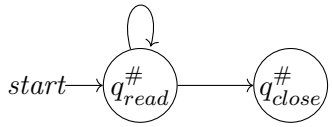
To verify it we use the generalization of the regular language learning procedure described above. The principle is the same: we iterate through growing fragments of $\mathcal{R}^*(I)$ recognized by the automata $\mathcal{A}_0, \mathcal{A}_1, \ldots$ to find candidate abstractions, starting from the initial state $(g\ file\ end)$ recognized by $\mathcal{A}_0$ in q_g. Since $\mathcal{R}$ is not terminating here, we use the method described in the previous section and only perform one step of Tree Automata Completion at each iteration. After 4 iterations, $\mathcal{A}_4$ will be as follows:

iteration 1:	$g\ q_{file}\ q_{end} \to q_g$	$file \to q_{file}$
		$end \to q_{end}$
iteration 2:	$br(q_{close}, q_{read}) \to q_{br}$	$q_{br} \to q_g$
	$close\ q_{file}\ q_{end} \to q_{close}$	
	$read\ q_{file}\ q_h \to q_{read}$	$h\ q_{file}\ q_{end} \to q_h$
iteration 3:	$q_{close} \to q_{br}$	$q_{read} \to q_{br}$
	$q_{end}\ q_{unit} \to q_2$	$q_2 \to q_{close}$
	$q_h\ q_{unit} \to q_h$	$q_h \to q_{read}$
	$() \to q_{unit}$	
iteration 4:	$q_{unit} \to q_2$	$q_g \to q_h$

Since this automaton is quite big, it is easier to look at the model $M_{\mathcal{A}_4} = \langle \mathcal{Q}, \Delta_\epsilon, V \rangle$ encoded by $\mathcal{A}_4$:



In this model we have in particular: $[\![read]\!] = \{\ q_{read}\ \}$, $[\![close]\!] = \{\ q_{close}\ \}$ and $[\![\phi]\!] = \mathcal{Q} \supseteq \mathcal{Q}_f$. The valid abstractions of this model that preserves ϕ for the final state (starting state in the model) are constrained by $S(\mathcal{A}, \phi)$, which gives that the final abstract automaton must have at least two states to separate read from close. The smallest solution is pictured as follows:



where $q_g, q_{br}, q_{read}, q_h$ are abstracted into $q^{\#}_{read}$ and where $q_{close}, q_{end}, q_{unit}$ are abstracted into $q^{\#}_{close}$. The final abstract automaton is $\mathcal{R}$-closed and IRR-complete, so the procedure stops with an correct abstraction of $\mathcal{R}^*(I)$ that verifies our property. Note that this abstraction is not a collapsing abstraction and preserve parts of the control flow of the program execution.

Of course a lot of work remains to properly formalize this concept and prove its correctness and completeness. It is also worth noticing that as it is defined above, this cannot be directly used in a type oriented verification procedure such as the one defined in Chapter 5, which can only express safety properties. Some more work remains to modularize this idea to have a chance to handle larger programs.

7.3 Integration in Higher-Order Functional Languages

The abstraction procedures presented in Chapters 4 and 5 are based on a representation of programs as term rewriting systems. Our goal however is not to impose this representation to programmers but instead use it as an intermediate representation for regular verification. This means that we must provide front-ends to real-world higher-order functional programming languages such as OCaml or Haskell in charge of translating from the original source code to TRS. There are multiple practicable difficulties arising from the translation into TRS:

- Primitive data types and operations must be encoded using terms. For instance, (positive) integers can be encoded using Peano's number representation. The expression $\underline{x}+1$ is translated into $s(\underline{x})$.
- It is not possible to use anonymous functions in a rewriting system. Every anonymous function must first be given a name so that rewriting rules can be created for it. Local function definitions must also be given global names. For instance, to translate the following OCaml expression `(function x -> x + 1)`, we must first give a name, f, to the anonymous function and produce the following rewriting rule for it: $@(f, \underline{x}) \to s(\underline{x})$. This is commonly referred to as a *Lambda-lifting* operation.
- Since local closures are lambda-lifted, we must find a way to preserve local bindings. One way is to use function symbols with non-zero arity, where the arity of a functional symbol f correspond to the number of locally bound variables used by the function. For instance we can encode the expression `(function y -> x + y)` (where `x` is a locally bound variable) by lambda-lifting the anonymous function with the symbol $f(\underline{x})$ and the rewriting rule $@(f(\underline{x}), \underline{y}) \to add(\underline{x}, \underline{y})$.
- In a pattern matching control structure, each matching case can be translated as a rewriting rule. However it must first be disambiguated w.r.t. the other cases since all the rules of a TRS have the same priority, whereas languages such as OCaml gives more to priority to some cases depending on the order of declaration in the pattern matching structure. For instance, the expression `match l with [] -> true | _ -> false` can be translated into

  $$m(nil) \to true \qquad\qquad m(\underline{x}) \to false$$

  However this is not a deterministic TRS: there is an ambiguity since both rules can be applied on $m(nil)$. A more accurate encoding of the OCaml semantics needs to disambiguate the patterns by generating the following rules:

  $$m(nil) \to true \qquad\qquad m(cons(\underline{x}, \underline{y})) \to false$$

  Such disambiguation can be done, for instance, using Krauss's pattern disambiguation method [Kra08].

Having this translation into TRS would allow us to extend our test suite with real-world programs and extensively test its scalability. Of course only purely functional features can be translated into TRS. As a first step toward this, we wish to integrate our verification technique in the OCaml interpreter. The idea is to add a new top level expression of the form `forall <variables>. <expression>` that automatically checks that the given OCaml expression always return `true` for all instances of the input variables. Another less straightforward idea would be to allow users to directly or indirectly specify regular language types in the signatures of functions. This could be more flexible for the user and could be used to give hints to the regular language type inference procedure. For instance, the signature of the `sort` function could be given as follows:

```
sort : t list -> { l | sorted l }
```

Here `{ l | sorted l }` indirectly refers to the regular language type of sorted lists of type `t list`. It can be easily computed by our inference procedure using the definition of `sorted`, as long as the type of list elements (`t`) is finite.

7.4 Polymorphic Lifting

Polymorphic types can be used in programming languages such as OCaml to define functions working with arbitrary types. For instance, the signature of the higher-order sort function can be generalized into

```
sort : ('a -> 'a -> bool) -> 'a list -> 'a list
```

where `'a -> 'a -> bool` is the signature of the comparison function used to sort elements of type `'a`, a type variable universally quantified. Such type variables are not currently supported by our verification procedure, every type must be known when the analysis starts. This means that polymorphic functions can only be verified on a finite set of type instances. Furthermore, depending on the considered property, the verification problem may or may not be regular depending on the instantiation of `'a`. In this example, if we consider the property `forall cmp, l. sorted cmp (sort cmp l) : true`, we have seen that the verification problem stays regular as long as `'a` is replaced by a finite type but becomes irregular when replaced with an infinite type such as `int`. Since *sort* works on polymorphic lists, its definition is independent of the actual structure of the manipulated type. It is never constructed nor destructed inside the function itself, so the correctness of *sort* and our ability to prove it should not depend on the type variable instantiation. In other words, by proving the above property on some (carefully chosen) finite instantiation `T` of `'a`, we should be able to deduce a proof for any type (even infinite ones):

```
forall cmp, l:T. sorted cmp (sort cmp l) : true
                              ⇓
forall cmp. l:'a. sorted cmp (sort cmp l) : true
```

We can formally state our polymorphic problem in the following way. Let Σ be a ranked alphabet and $\mathcal{X}$ a set of variables with p a pattern of $\mathcal{T}(\Sigma, \mathcal{X})$. We want to show that $p\sigma \not\to_{\mathcal{R}}^{*} false$ for every substitution $\sigma : \mathcal{X} \to \mathcal{T}(\Sigma')$ where Σ' can be any ranked alphabet (it is universally quantified to encode polymorphism). We can safely ignore well-typedness considerations here and assume that $p\sigma$ is always well-typed (if not it would not rewrite to $false$). This proof cannot be directly

handled by our technique since Σ' is universally quantified. However we propose here an indirect 3-steps method that works for such polymorphic problem where we (1) choose a finite domain instantiation of each type variable (2) prove that property is verified in this finite domain, (3) show that if the polymorphic property is not verified ($p\sigma \rightarrow^*_{\mathcal{R}} false$), then there exists an abstraction of the counter-example into the chosen finite domain such that the property is not verified on this abstract counter-example either. Since no such counter-example exists thanks to (2), the polymorphic property must be verified.

Choosing a finite domain instantiation In the first step, we instantiate each type variable with a finite domain type. For that we extend Σ with a finite set of constants $\Sigma^\#$ populating the finite domain. For a given set of variables, We write $T_x \subseteq \Sigma^\#$ the monomorphic regular language type associated to the variable x of polymorphic type `'a`. For example, in the *sort* example we can instantiate the type variable `'a` with the finite language type $T_x = \{A, B\} \subseteq \Sigma^\#$. Ideally we want to choose the smallest number of constants that allows the verification of the given polymorphic property. Note that we also need to populate polymorphic arrow types such as `'a -> 'a -> bool` (the comparison function's type). This is done by again adding new constants $T_{cmp} = \{cmp_0, \ldots, cmp_f\} \subseteq \Sigma^\#$ where each cmp_i has type $\{A, B\} \rightarrow \{A, B\} \rightarrow bool$. We then add the necessary rules in $\mathcal{R}$ so that $(cmp_i\ x\ y)$ can be evaluated (into a Boolean). To be complete, we must represent every possible function of this type. Thankfully since we only deal with finite languages this can be easily constructed by enumerating every possible combination of inputs and outputs. In this example, we need to generate the following rules for $\mathcal{R}$:

$$
\begin{array}{cccc}
cmp_0\ A\ A \rightarrow false & cmp_0\ A\ B \rightarrow false & cmp_0\ B\ A \rightarrow false & cmp_0\ B\ B \rightarrow false \\
cmp_1\ A\ A \rightarrow false & cmp_1\ A\ B \rightarrow false & cmp_1\ B\ A \rightarrow false & cmp_1\ B\ B \rightarrow true \\
cmp_2\ A\ A \rightarrow false & cmp_2\ A\ B \rightarrow false & cmp_2\ B\ A \rightarrow true & cmp_2\ B\ B \rightarrow false \\
\ldots & \ldots & \ldots & \ldots \\
cmp_d\ A\ A \rightarrow true & cmp_d\ A\ B \rightarrow true & cmp_d\ B\ A \rightarrow false & cmp_d\ B\ B \rightarrow true \\
cmp_e\ A\ A \rightarrow true & cmp_e\ A\ B \rightarrow true & cmp_e\ B\ A \rightarrow true & cmp_e\ B\ B \rightarrow false \\
cmp_f\ A\ A \rightarrow true & cmp_f\ A\ B \rightarrow true & cmp_f\ B\ A \rightarrow true & cmp_f\ B\ B \rightarrow true
\end{array}
$$

Note that this can reduce the number of generated rules by removing symmetries.

Finite domain proof The next step is to show that our polymorphic property is verified on this particular domain.

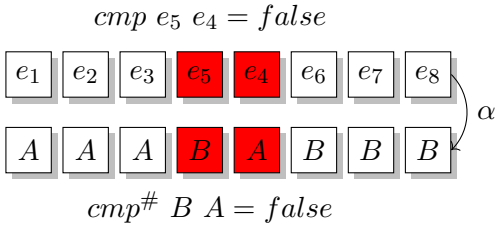
Property 1: For all substitution $\sigma^\# : \mathcal{X} \mapsto \Sigma^\#$ such that $\sigma^\#(x) \in T_x$, $p\sigma^\# \not\rightarrow^*_{\mathcal{R}} false$.

This can be verified using the technique described in Chapter 5.

Polymorphic lifting Once we have shown that $p\sigma^\# \not\rightarrow^*_{\mathcal{R}} false$ for each $\sigma^\#$ in our finite domain, the last step is to lift this proof to the polymorphic property. The trick is to first show the following intermediate property over p and $\mathcal{R}$.

Property 2: For all ranked alphabet Σ' and substitution $\sigma : \mathcal{X} \rightarrow \mathcal{T}(\Sigma')$, if $p\sigma \rightarrow^*_{\mathcal{R}} false$ then there exists an abstraction $\Lambda = \langle \Sigma^\#, \Delta^\# \rangle$ of Σ' such that $\alpha(\sigma(x)) \in T_x$ and $p\sigma \rightarrow^*_{\mathcal{R} \cup \Delta^\#} false$.

For instance, consider that the safety property on *sort* we want to verify is not correct. The *sort* function preserves the elements of the list, but the output is not sorted. Then there exists a type domain instance for `'a` and an initial state of the form `sorted (sort l)` such that `sort l` returns an unsorted list of the form $e_1, \ldots, e_i, e_j, \ldots, e_n$ where $(cmp\ e_i\ e_j = false)$ (which is why the list is not sorted). Then we can abstract every element of the domain into the finite abstract domain $\{\ A, B\ \}$ and the comparison function into $cmp^\#$ in such a way that $(cmp^\#\ \alpha(e_i)\ \alpha(e_j) = false)$ so that the result remains unsorted as pictured below.



If we can show this property then we can show by contradiction that $\forall\sigma^\#.\ p\sigma^\# \not\rightarrow^*_{\mathcal{R}} false$ can be lifted for every Σ' into the proof that $\forall\sigma.p\sigma \not\rightarrow^*_{\mathcal{R}} false$. The idea of the proof is as follows: if there indeed exists Σ' and σ such that $p\sigma \rightarrow^*_{\mathcal{R}} false$, then there exists an abstraction such that $p\sigma' \rightarrow^*_{\mathcal{R}^\# \cup \Delta^\#} false$. Since $\mathcal{R}$ has been extended with the appropriate rules, this means there exists $\sigma^\#$ such that $p\sigma \rightarrow^*_{\Delta^\#} p\sigma^\# \rightarrow^*_{\mathcal{R}} false$. But we already know by hypothesis that $p\sigma^\# \not\rightarrow^*_{\mathcal{R}} false$. This contradiction proves that $p\sigma \not\rightarrow^*_{\mathcal{R}} false$. Property 2 cannot be proved with our procedure because Σ' is universally quantified. Finding a way to show this property is the main difficulty of this future work. We believe that it can be done with a relatively simple static analysis of the program's TRS. To be able to go through the last two steps, it may be needed to iteratively increase the number of constants used in $\Sigma^\#$ to abstract the polymorphic type variables.

7.5 Regular Relations and Higher-Order

In Chapter 6 we have seen how first-order functions could be encoded as predicates into Horn clauses systems in order to be verified. However this idea only works for first-order functions. Another research direction could consist in finding ways to adapt our relational verification technique to handle higher-order programs.

Bibliography

[ADLO10] T. Altenkirch, N. A. Danielsson, A. Löh, and N. Oury. $\pi\sigma$: Dependent types without the sugar. In M. Blume, N. Kobayashi, and G. Vidal, editors, *Functional and Logic Programming*, pages 40–55, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[AF20] I. Amit and D. G. Feitelson. The corrective commit probability code quality metric. *CoRR*, abs/2007.10912, 2020, 2007.10912. URL https://arxiv.org/abs/2007.10912.

[Aug98] L. Augustsson. Cayenne - a language with dependent types. In M. Felleisen, P. Hudak, and C. Queinnec, editors, *Proceedings of the third ACM SIGPLAN International Conference on Functional Programming (ICFP '98), Baltimore, Maryland, USA, September 27-29, 1998*, pages 239–250. ACM, 1998. doi:10.1145/289423.289451.

[BCD83] H. Barendregt, M. Coppo, and M. Dezani-Ciancaglini. A filter lambda model and the completeness of type assignment. *J. Symb. Log.*, 48(4):931–940, 1983. doi:10.2307/2273659.

[BCD+11] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli. CVC4. In G. Gopalakrishnan and S. Qadeer, editors, *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV '11)*, volume 6806 of *Lecture Notes in Computer Science*, pages 171–177. Springer, July 2011. URL http://www.cs.stanford.edu/~barrett/pubs/BCD+11.pdf. Snowbird, Utah.

[BCF03] V. Benzaken, G. Castagna, and A. Frisch. Cduce: an XML-centric general-purpose language. In C. Runciman and O. Shivers, editors, *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming, ICFP 2003, Uppsala, Sweden, August 25-29, 2003*, pages 51–63. ACM, 2003. doi:10.1145/944705.944711.

[BG00] A. Blumensath and E. Grädel. Automatic structures. In *15th Annual IEEE Symposium on Logic in Computer Science, Santa Barbara, California, USA, June 26-29, 2000*, pages 51–62. IEEE Computer Society, 2000. doi:10.1109/LICS.2000.855755.

[BGJ08] B. Boyer, T. Genet, and T. Jensen. Certifying a tree automata completion checker. In A. Armando, P. Baumgartner, and G. Dowek, editors, *Automated Reasoning*, pages 523–538, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[BK13] C. H. Broadbent and N. Kobayashi. Saturation-based model checking of higher-order recursion schemes. In S. R. D. Rocca, editor, *Computer Science Logic 2013 (CSL 2013), CSL 2013, September 2-5, 2013, Torino, Italy*, volume 23 of *LIPIcs*, pages 129–148. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2013. doi:10.4230/LIPIcs.CSL.2013.129.

[BN98] F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

[BN10] J. C. Blanchette and T. Nipkow. Nitpick: A counterexample generator for higher-order logic based on a relational model finder. In M. Kaufmann and L. C. Paulson, editors, *Interactive Theorem Proving*, pages 131–146, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[Bra13] E. Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *J. Funct. Program.*, 23(5):552–593, 2013. doi:10.1017/S095679681300018X.

[CCKS18] A. Champion, T. Chiba, N. Kobayashi, and R. Sato. Ice-based refinement type discovery for higher-order functional programs. In D. Beyer and M. Huisman, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 365–384, Cham, 2018. Springer International Publishing.

[CD80] M. Coppo and M. Dezani-Ciancaglini. An extension of the basic functionality theory for the λ-calculus. *Notre Dame Journal of Formal Logic*, 21(4):685–693, 1980. doi:10.1305/ndjfl/1093883253.

[CDC78] M. Coppo and M. Dezani-Ciancaglini. A new type assignment for λ-terms. *Archiv für mathematische Logik und Grundlagenforschung*, 19(1):139–156, Dec 1978. doi:10.1007/BF02011875.

[CDCS79] M. Coppo, M. Dezani-Ciancaglini, and P. Salle'. Functional characterization of some semantic equalities inside λ-calculus. In H. A. Maurer, editor, *Automata, Languages and Programming*, pages 133–146, Berlin, Heidelberg, 1979. Springer Berlin Heidelberg.

[CDV81] M. Coppo, M. Dezani-Ciancaglini, and B. Venneri. Functional characters of solvable terms. *Math. Log. Q.*, 27(2-6):45–58, 1981. doi:10.1002/malq.19810270205.

[CE82a] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In D. Kozen, editor, *Logics of Programs*, pages 52–71, Berlin, Heidelberg, 1982. Springer Berlin Heidelberg.

[CE82b] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In D. Kozen, editor, *Logics of Programs*, pages 52–71, Berlin, Heidelberg, 1982. Springer Berlin Heidelberg.

[CGJ+00] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In E. A. Emerson and A. P. Sistla, editors, *Computer Aided Verification*, pages 154–169, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

[Cho56] N. Chomsky. Three models for the description of language. *IRE Trans. Inf. Theory*, 2(3):113–124, 1956. doi:10.1109/TIT.1956.1056813.

[CJRS15] K. Claessen, M. Johansson, D. Rosén, and N. Smallbone. Tip: Tons of inductive problems. In M. Kerber, J. Carette, C. Kaliszyk, F. Rabe, and V. Sorge, editors, *Intelligent Computer Mathematics*, pages 333–337, Cham, 2015. Springer International Publishing.

[CKS18] A. Champion, N. Kobayashi, and R. Sato. HoIce: An ICE-based non-linear horn clause solver. In S. Ryu, editor, *Programming Languages and Systems*, pages 146–156, Cham, 2018. Springer International Publishing.

[CNX+14] G. Castagna, K. Nguyen, Z. Xu, H. Im, S. Lenglet, and L. Padovani. Polymorphic functions with set-theoretic types: part 1: syntax, semantics, and evaluation. In S. Jagannathan and P. Sewell, editors, *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, pages 5–18. ACM, 2014. doi:10.1145/2535838.2535840.

[Com00] H. Comon. Sequentiality, monadic second-order logic and tree automata. *Inf. Comput.*, 157(1-2):25–51, 2000. doi:10.1006/inco.1999.2838.

[CPN16] G. Castagna, T. Petrucciani, and K. Nguyen. Set-theoretic types for polymorphic variants. In J. Garrigue, G. Keller, and E. Sumii, editors, *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*, pages 378–391. ACM, 2016. doi:10.1145/2951913.2951928.

[DCM84] M. Dezani-Ciancaglini and I. Margaria. F-semantics for intersection type discipline. In G. Kahn, D. B. MacQueen, and G. Plotkin, editors, *Semantics of Data Types*, pages 279–300, Berlin, Heidelberg, 1984. Springer Berlin Heidelberg.

[dMB08] L. de Moura and N. Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and J. Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[EH83] E. A. Emerson and J. Y. Halpern. "sometimes" and "not never" revisited: On branching versus linear time. In J. R. Wright, L. Landweber, A. J. Demers, and T. Teitelbaum, editors, *Conference Record of the Tenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas, USA, January 1983*, pages 127–140. ACM Press, 1983. doi:10.1145/567067.567081.

[Exp3] T. Genet and T. Haudebourg. Experiments with timbuk 3, 2018. URL https://people.irisa.fr/Thomas.Genet/timbuk/funExperiments/index.html.

[Exp4] T. Genet and T. Haudebourg. Experiments with timbuk 4, 2020. URL https://people.irisa.fr/Thomas.Genet/timbuk/timbuk4/experiments.html.

[FCB02] A. Frisch, G. Castagna, and V. Benzaken. Semantic subtyping. In *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings*, pages 137–146. IEEE Computer Society, 2002. doi:10.1109/LICS.2002.1029823.

[FP91] T. S. Freeman and F. Pfenning. Refinement types for ML. In D. S. Wise, editor, *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation (PLDI), Toronto, Ontario, Canada, June 26-28, 1991*, pages 268–277. ACM, 1991. doi:10.1145/113445.113468.

[Gen98] T. Genet. Decidable approximations of sets of descendants and sets of normal forms. In T. Nipkow, editor, *Rewriting Techniques and Applications*, pages 151–165, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.

[Gen16] T. Genet. Termination criteria for tree automata completion. *J. Log. Algebraic Methods Program.*, 85(1):3–33, 2016. doi:10.1016/j.jlamp.2015.05.003.

[Gen18] T. Genet. Completeness of tree automata completion. In H. Kirchner, editor, *3rd International Conference on Formal Structures for Computation and Deduction, FSCD 2018, July 9-12, 2018, Oxford, UK*, volume 108 of *LIPIcs*, pages 16:1–16:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018. doi:10.4230/LIPIcs.FSCD.2018.16.

[GLGLM13] T. Genet, T. Le Gall, A. Legay, and V. Murat. A completion algorithm for lattice tree automata. In S. Konstantinidis, editor, *Implementation and Application of Automata*, pages 134–145, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[GLMN14] P. Garg, C. Löding, P. Madhusudan, and D. Neider. Ice: a robust framework for learning invariants. In A. Biere and R. Bloem, editors, *Computer Aided Verification*, pages 69–87, Cham, 2014. Springer International Publishing.

[GNMR16] P. Garg, D. Neider, P. Madhusudan, and D. Roth. Learning invariants using decision trees and implication counterexamples. In R. Bodík and R. Majumdar, editors, *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 499–512. ACM, 2016. doi:10.1145/2837614.2837664.

[GR10] T. Genet and V. Rusu. Equational approximations for tree automata completion. *J. Symb. Comput.*, 45(5):574–597, 2010. doi:10.1016/j.jsc.2010.01.009.

[Hau19] T. Haudebourg. Regular chc solver. https://github.com/regular-pv/rchc, commit 571346e, 2019.

[Hin69] R. Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, 1969. URL http://www.jstor.org/stable/1995158.

[Hin82] J. R. Hindley. The simple semantics for coppo-dezani-sallé types. In M. Dezani-Ciancaglini and U. Montanari, editors, *International Symposium on Programming*, pages 212–226, Berlin, Heidelberg, 1982. Springer Berlin Heidelberg.

[HP01] H. Hosoya and B. C. Pierce. Regular expression pattern matching for XML. In C. Hankin and D. Schmidt, editors, *Conference Record of POPL 2001: The 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, London, UK, January 17-19, 2001*, pages 67–80. ACM, 2001. doi:10.1145/360204.360209.

[HP03] H. Hosoya and B. C. Pierce. Xduce: A statically typed XML processing language. *ACM Trans. Internet Techn.*, 3(2):117–148, 2003. doi:10.1145/767193.767195.

[HVP00] H. Hosoya, J. Vouillon, and B. C. Pierce. Regular expression types for XML. In M. Odersky and P. Wadler, editors, *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00), Montreal, Canada, September 18-21, 2000*, pages 11–22. ACM, 2000. doi:10.1145/351240.351242.

[Inr05] Inria. Compcert, 2005. URL https://compcert.inria.fr.

[Inr16] Inria. The coq proof assistant, 2016. URL https://coq.inria.fr.

[JA07] N. D. Jones and N. Andersen. Flow analysis of lazy higher-order functional programs. *Theor. Comput. Sci.*, 375(1-3):120–136, 2007. doi:10.1016/j.tcs.2006.12.030.

[JM79] N. D. Jones and S. S. Muchnick. Flow analysis and optimization of lisp-like structures. In A. V. Aho, S. N. Zilles, and B. K. Rosen, editors, *Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages, San Antonio, Texas, USA, January 1979*, pages 244–256. ACM Press, 1979. doi:10.1145/567752.567776.

[KEH+09] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. sel4: formal verification of an OS kernel. In J. N. Matthews and T. E. Anderson, editors, *Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, USA, October 11-14, 2009*, pages 207–220. ACM, 2009. doi:10.1145/1629575.1629596.

[KN95] B. Khoussainov and A. Nerode. Automatic presentations of structures. In D. Leivant, editor, *Logic and Computational Complexity*, pages 367–392, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg.

[Kob09a] N. Kobayashi. Model-checking higher-order functions. In A. Porto and F. J. López-Fraguas, editors, *Proceedings of the 11th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, September 7-9, 2009, Coimbra, Portugal*, pages 25–36. ACM, 2009. doi:10.1145/1599410.1599415.

[Kob09b] N. Kobayashi. Types and higher-order recursion schemes for verification of higher-order programs. In Z. Shao and B. C. Pierce, editors, *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, pages 416–428. ACM, 2009. doi:10.1145/1480881.1480933.

[Kob13] N. Kobayashi. Model checking higher-order programs. *J. ACM*, 60(3):20:1–20:62, 2013. doi:10.1145/2487241.2487246.

[Kou92] E. Kounalis. Testing for the ground (co-)reducibility property in term-rewriting systems. *Theor. Comput. Sci.*, 106(1):87–117, 1992. doi:10.1016/0304-3975(92)90279-O.

[Kra08] A. Krauss. Pattern minimization problems over recursive data types. In J. Hook and P. Thiemann, editors, *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming, ICFP 2008, Victoria, BC, Canada, September 20-28, 2008*, pages 267–274. ACM, 2008. doi:10.1145/1411204.1411242.

[KRJ09] M. Kawaguchi, P. M. Rondon, and R. Jhala. Type-based data structure verification. In M. Hind and A. Diwan, editors, *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009*, pages 304–315. ACM, 2009. doi:10.1145/1542476.1542510.

[KSU11a] N. Kobayashi, R. Sato, and H. Unno. Mochi: Model checker for higher-order programs, 2011. URL http://www-kb.is.s.u-tokyo.ac.jp/~ryosuke/mochi/.

[KSU11b] N. Kobayashi, R. Sato, and H. Unno. Predicate abstraction and CEGAR for higher-order model checking. In M. W. Hall and D. A. Padua, editors, *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, pages 222–233. ACM, 2011. doi:10.1145/1993498.1993525.

[KTU10] N. Kobayashi, N. Tabuchi, and H. Unno. Higher-order multi-parameter tree transducers and recursion schemes for program verification. In M. V. Hermenegildo and J. Palsberg, editors, *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*, pages 495–508. ACM, 2010. doi:10.1145/1706299.1706355.

[Lei10] K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In E. M. Clarke and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 348–370, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[LGJ07] T. Le Gall and B. Jeannet. Lattice automata: A representation for languages on infinite alphabets, and some applications to verification. In H. R. Nielson and G. Filé, editors, *Static Analysis*, pages 52–68, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[Lin12] A. W. Lin. Accelerating tree-automatic relations. In D. D'Souza, T. Kavitha, and J. Radhakrishnan, editors, *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2012, December 15-17, 2012, Hyderabad, India*, volume 18 of *LIPIcs*, pages 313–324. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2012. doi:10.4230/LIPIcs.FSTTCS.2012.313.

[LST78] B. P. Lientz, E. B. Swanson, and G. E. Tompkins. Characteristics of applications software maintenance. *Commun. ACM*, 21(6):466–471, 1978. doi:10.1145/359511.359522.

[McK06] J. McKinna. Why dependent types matter. In J. G. Morrisett and S. L. P. Jones, editors, *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA, January 11-13, 2006*, page 1. ACM, 2006. doi:10.1145/1111037.1111038.

[MI13] Microsoft Research and Inria. F*, 2013. URL https://www.fstar-lang.org.

[Mil78] R. Milner. A theory of type polymorphism in programming. *J. Comput. Syst. Sci.*, 17(3):348–375, 1978. doi:10.1016/0022-0000(78)90014-4.

[MKU15] Y. Matsumoto, N. Kobayashi, and H. Unno. Automata-based abstraction for automated verification of higher-order tree-processing programs. In X. Feng and S. Park, editors, *Programming Languages and Systems*, pages 295–312, Cham, 2015. Springer International Publishing.

[Nec97] G. C. Necula. Proof-carrying code. In P. Lee, F. Henglein, and N. D. Jones, editors, *Conference Record of POPL'97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, Paris, France, 15-17 January 1997*, pages 106–119. ACM Press, 1997. doi:10.1145/263699.263712.

[Nor09] U. Norell. *Dependently Typed Programming in Agda*, pages 230–266. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009. doi:10.1007/978-3-642-04652-0_5.

[NPW02] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[Ong06] C. L. Ong. On model-checking trees generated by higher-order recursion schemes. In *21th IEEE Symposium on Logic in Computer Science (LICS 2006), 12-15 August 2006, Seattle, WA, USA, Proceedings*, pages 81–90. IEEE Computer Society, 2006. doi:10.1109/LICS.2006.38.

[OR11] C. L. Ong and S. J. Ramsay. Verifying higher-order functional programs with pattern-matching algebraic data types. In T. Ball and M. Sagiv, editors, *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, pages 587–598. ACM, 2011. doi:10.1145/1926385.1926453.

[OTMW04] X. Ou, G. Tan, Y. Mandelbaum, and D. Walker. Dynamic typing with dependent types. In J.-J. Levy, E. W. Mayr, and J. C. Mitchell, editors, *Exploring New Frontiers of Theoretical Informatics*, pages 437–450, Boston, MA, 2004. Springer US.

[Rey69] J. Reynolds. Automatic computation of data set definitions. *Information Processing*, 68:456–461, 1969.

[Rey91] J. C. Reynolds. The coherence of languages with intersection types. In T. Ito and A. R. Meyer, editors, *Theoretical Aspects of Computer Software*, pages 675–700, Berlin, Heidelberg, 1991. Springer Berlin Heidelberg.

[Rey96] J. C. Reynolds. Design of the programming language forsythe. Technical report, 1996.

[RKJ08] P. M. Rondon, M. Kawaguchi, and R. Jhala. Liquid types. In R. Gupta and S. P. Amarasinghe, editors, *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, pages 159–169. ACM, 2008. doi:10.1145/1375581.1375602.

[RNO14] S. J. Ramsay, R. P. Neatherway, and C. L. Ong. A type-directed abstraction refinement approach to higher-order model checking. In S. Jagannathan and P. Sewell, editors, *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, pages 61–72. ACM, 2014. doi:10.1145/2535838.2535873.

[Roc88] S. R. D. Rocca. Principal type scheme and unification for intersection type discipline. *Theor. Comput. Sci.*, 59:181–209, 1988. doi:10.1016/0304-3975(88)90101-6.

[RTI02] Research Triangle Institute. The economic impacts of inadequate infrastructure for software testing. Technical Report Planning Report 02-3, National Institute of Standards and Technology, Acquisition and Assistance Division, Gaithersburg, 2002.

[RV84] S. R. D. Rocca and B. Venneri. Principal type schemes for an extended type theory. *Theor. Comput. Sci.*, 28:151–169, 1984. doi:10.1016/0304-3975(83)90069-5.

[SdB69] D. Scott and J. W. de Bakker. A theory of programs. *Unpublished manuscript, IBM, Vienna*, 1969.

[SK17] R. Sato and N. Kobayashi. Modular verification of higher-order functional programs. In H. Yang, editor, *Programming Languages and Systems*, pages 831–854, Berlin, Heidelberg, 2017. Springer Berlin Heidelberg.

[Tata] H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. Available on: http://www.grappa.univ-lille3.fr/tata, 2007. release October, 12th 2007.

[Tbk3] T. Genet, Y. Boichut, B. Boyer, V. Murat, and Y. Salmon. Reachability Analysis and Tree Automata Calculations. IRISA / Université de Rennes 1, 2001. URL http://people.irisa.fr/Thomas.Genet/timbuk/.

[Tbk4] T. Haudebourg. Timbuk 4: Regular verification framework based on tree automata and term rewriting systems. https://gitlab.inria.fr/regular-pv/timbuk/timbuk, commit 11c84f54, 2019.

[TeReSe] Terese. *Term Rewriting Systems*. Cambridge University Press, 2003.

[TK14] T. Terao and N. Kobayashi. A zdd-based efficient higher-order model checking algorithm. In J. Garrigue, editor, *Programming Languages and Systems*, pages 354–371, Cham, 2014. Springer International Publishing.

[UTK10] H. Unno, N. Tabuchi, and N. Kobayashi. Verification of tree-processing programs via higher-order model checking. In K. Ueda, editor, *Programming Languages and Systems*, pages 312–327, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[Vaz16] N. Vazou. *Liquid Haskell: Haskell as a Theorem Prover*. PhD thesis, University of California, San Diego, USA, 2016.

[VBJ15] N. Vazou, A. Bakst, and R. Jhala. Bounded refinement types. In K. Fisher and J. H. Reppy, editors, *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015*, pages 48–61. ACM, 2015. doi:10.1145/2784731.2784745.

[VJ16a] N. Vazou and R. Jhala. Refinement reflection (or, how to turn your favorite language into a proof assistant using SMT). *CoRR*, abs/1610.04641, 2016, 1610.04641. URL http://arxiv.org/abs/1610.04641.

[VJ16b] N. Vazou and R. Jhala. Refinement reflection (or, how to turn your favorite language into a proof assistant using SMT). *CoRR*, abs/1610.04641, 2016, 1610.04641. URL http://arxiv.org/abs/1610.04641.

[VJCR13] D. Vytiniotis, S. L. P. Jones, K. Claessen, and D. Rosén. HALO: haskell to logic through denotational semantics. In R. Giacobazzi and R. Cousot, editors, *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, pages 431–442. ACM, 2013. doi:10.1145/2429069.2429121.

[VRJ13] N. Vazou, P. M. Rondon, and R. Jhala. Abstract refinement types. In M. Felleisen and P. Gardner, editors, *Programming Languages and Systems*, pages 209–228, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[VSJ14] N. Vazou, E. L. Seidel, and R. Jhala. Liquidhaskell: experience with refinement types in the real world. In W. Swierstra, editor, *Proceedings of the 2014 ACM SIGPLAN symposium on Haskell, Gothenburg, Sweden, September 4-5, 2014*, pages 39–51. ACM, 2014. doi:10.1145/2633357.2633366.

[XP98] H. Xi and F. Pfenning. Eliminating array bound checking through dependent types. In J. W. Davidson, K. D. Cooper, and A. M. Berman, editors, *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation (PLDI), Montreal, Canada, June 17-19, 1998*, pages 249–257. ACM, 1998. doi:10.1145/277650.277732.

UNIVERSITÉ DE RENNES 1

Titre : Verification automatique de programmes fonctionnels d'ordre supérieur à l'aide de languages réguliers d'arbres

Mot clés : Vérification, Ordre-supérieur, Languages fonctionnels, Languages réguliers d'arbres

Résumé : Nous étudions comment les *langages réguliers d'arbres* peuvent être utilisés pour *vérifier automatiquement* des propriétés sur des *programmes fonctionnels d'ordre supérieur*. Notre but est de développer de nouvelles techniques et outils pour les programmeurs permettant de développer des programmes plus sûrs tout en réduisant le temps et l'expertise nécessaire pour les vérifier. Cette thèse se concentre sur la vérification de *propriétés régulières*, famille pour laquelle nous montrons qu'une vérification complète et automatique est possible. Notre méthode de vérification est construite sur une procédure d'abstraction capable d'apprendre des langages réguliers sur-approchant les états atteignables d'un programme. En utilisant les langages réguliers en tant que types, nous montrons comment modulariser cette procédure pour vérifier des propriétés complexes en les formulant en tant que problèmes d'inférence des types. Nous étudions ses performances au travers de notre implémentation OCaml, Timbuk 4, sur plus de 80 problèmes de vérification. Nous montrons ensuite que notre procédure d'abstraction peut être utilisée pour vérifier des propriété relationnelles qui semblaient hors de portée des langages réguliers. Pour cela, nous utilisons et étendons un opérateur de convolution sur les arbres pour représenter une relation par langage régulier. Nous étendons ensuite notre procédure d'apprentissage de langages pour inférer automatiquement ces relations. Nous proposons une implémentation de cette idée en Rust en tant solveur de systèmes de clauses de Horn contraintes et étudions ses performances sur de multiples problèmes relationnels.

Title: Automatic Verification of Higher-Order Functional Programs using Regular Tree Languages

Keywords: Program Verification, Higher-order, Functional languages, Regular tree languages

Abstract: This thesis studies how *regular tree languages* can be used to *automatically verify* properties on *higher-order functional programs*. Our goal is to develop new techniques and tools for the programmers to develop safer programs while reducing the time and expertize needed to verify them. In particular, we focus on the automatic verification of *regular safety properties*, a family of properties for which we show that completely and fully automatic verification can be achieved. Our verification method is build upon a regular abstraction procedure that can automatically learn regular tree languages that over-approximates of the reachable states of a program, allowing the verification of a target property. By using regular languages as types we modularize this procedure to verify complex properties by stating them as type inference problems. In addition we study the performances of the overall technique in our prototype OCaml implementation in the Timbuk 4 verification framework over a test suite of more than 80 verification problems. We then show how our abstraction procedure can be used to verify *relational properties* that seemed out of the scope of regular tree languages. To do that, we use and extend a convolution operator on trees to represent every element of a relation into a regular tree language. We can then extend our previously defied regular language learning procedure to automatically infer such regular relations. We propose a Rust implementation of this idea as a regular solver for Constrained Horn Clauses systems and study its performance on several relational verification problems.