N° d'ordre : h252 ANNÉE 2010

THÈSE / UNIVERSITÉ DE RENNES 1
sous le sceau de l'Université Européenne de Bretagne

pour le grade de

DOCTEUR DE L'UNIVERSITÉ DE RENNES 1

Mention : Informatique

Ecole doctorale Matisse

présentée par

Benoît Boyer

préparée au sein de l'équipe Celtique à l'IRISA (UMR n° 6074)
Institut de recherche en informatique et systèmes aléatoires
UFR Informatique et Electronique (ISTIC)

Réécriture d'automates certifiée pour la vérification de modèle

Thèse soutenue à (lieu)
le (date)

devant le jury composé de :

Olivier RIDOUX
Professeur à l'université Rennes 1 — président

Ahmed BOUAJJANI
Professeur à l'université Paris 7 — rapporteur

Jean GOUBAULT-LARRECQ
Professeur à l'ENS Cachan — rapporteur

Pierpaolo DEGANO
Professeur à l'Università di Pisa — examinateur

Thomas JENSEN
Directeur de Recherche à l'INRIA — directeur de thèse

Thomas GENET
Maître de Conférence à l'université Rennes 1
co-directeur de thèse

Réécriture d'automates certifiée pour la vérification de modèle

Benoît Boyer

28 octobre 2010

Table des matières

Table des matières **i**

1 Introduction **1**
1.1 Pourquoi la vérification et la preuve de programmes ? 1
1.2 La réécriture pour modéliser les programmes 3
1.3 Vérification de systèmes de réécriture 4
1.4 Présentation des Contributions de cette thèse 6

2 Prérequis **9**
2.1 Termes et réécriture 9
2.2 Les automates d'arbres 12
2.3 Le Model-Checking régulier 13
2.4 La complétion d'automates d'arbres 14

3 Détection de contre-exemples et Raffinement **23**
3.1 Fausses alarmes et contre-exemples 23
3.2 Principe de l'approche à la CEGAR appliqué au model-checking régulier . 24
3.3 Contre-exemples et automates d'arbres 26
3.4 L'intuition initiale 27
3.5 Définition d'un $\mathcal{R}_{/E}$-automate 28
3.6 Construire un $\mathcal{R}_{/E}$-Automate pour le problème d'atteignabilité 31
3.7 L'étape de $\mathcal{R}_{/E}$-complétion $\mathtt{C}$ 32
3.8 La normalisation 33
3.9 Résoudre le problème du filtrage 38
3.10 L'étape de $\mathcal{R}_{/E}$-complétion est correcte 41
3.11 L'accélération du calcul W 45
3.12 Vérification de propriétés régulières 47
3.13 Test de vacuité de l'intersection 47
3.14 Raffiner l'approximation par élagage 51
3.15 $\mathcal{R}_{/E}$-complétion et le calcul exact de termes atteignables 54
3.16 Conclusion 55

4 Preuves de propriétés temporelles sur TRS **57**

4.1 Du Bytecode Java à la réécriture 57
4.2 Completion d'automates d'arbres et $\dashrightarrow_{\mathcal{R}_p/E}$ 61
4.3 Extraction d'une structure de Kripke 63
4.4 Une logique temporelle portant des automates d'arbres 65
4.5 Conclusion, discussion 70

5 Certification de la complétion d'automates d'arbres **73**
5.1 Un validateur de résultat pour la complétion d'automates d'arbres 73
5.2 Formalisation de la réécriture 75
5.3 Formalisation des automates d'arbres 78
5.4 L'inclusion efficace d'automates 80
5.5 Formalisation de la clôture par réécriture 91
5.6 Conclusion 99

Bibliographie générale **101**

Chapitre 1

Introduction

1.1 Pourquoi la vérification et la preuve de programmes ?

L'informatique était vue comme la solution pour résoudre les problèmes de fiabilité dans les domaines où toute défaillance opérationnelle peut conduire à des catastrophes. L'étymologie du mot *ordinateur* est un bel exemple des espoirs que l'on a placés dans le domaine. Introduit par Jacques Perret en 1955, il est issu du latin *ordinator* qui signifie celui qui met en ordre, qui règle [1]. Cette définition omet de préciser que si l'ordinateur sait résoudre des problèmes, c'est grâce à l'ordonnateur (plus communément appelé programmeur ou développeur). L'ordinateur exécute les directives (le programme) du programmeur susceptibles de résoudre les problèmes auxquels la machine est assignée. Si la machine est incontestablement meilleure que toute personne humaine dans la répétition intensive de calcul, l'introduction de l'informatique déplace simplement le problème de la fiabilité : le programme est-il suffisamment bien conçu pour permettre à l'ordinateur de toujours réussir dans sa tâche ?

Si l'on se concentre sur la tâche du programmeur, celle-ci peut se résumer essentiellement en deux étapes. Analyser le problème, puis si une méthode de résolution (au sens algorithmique du terme) existe, la traduire en une spécification opérationnelle, le programme. Le *bug* [2] intervient lorsqu'une faute de raisonnement conduit à une spécification erronée ou incomplète : conduisant l'ordinateur dans des comportements inattendus qui constituent les manifestations cliniques du bug. Plus le problème à résoudre est complexe, plus il y a de chance de commettre une faute de conception conduisant à un bug lors de l'exécution du programme. De l'analyse du problème on peut aussi aboutir à une spécification dénotationnelle et formalisée à l'aide d'outils mathématiques. Cette spécification "haut-niveau" est univoque et constituée de définitions et de propriétés simples : la spécification dénotationnelle est souvent plus facile à comprendre que la spécification opérationnelle puisque débarrassée des aspects calculatoires. On oblige les programmeurs à commen-

1. Extrait de la définition d'*ordinator* dans le dictionnaire Gaffiot
2. La terminologie française exacte est *bogue*

ter leur code source pour une bonne raison. La spécification opérationnelle étant peu compréhensible, il est nécessaire de donner au moins une spécification informelle pour qu'une personne autre le programmeur soit capable de comprendre le programme.

Une partie de la science informatique a pour unique but de rendre la conception de programmes plus sûre. L'idée sous-jascente pour mener à bien cet objectif est de rapprocher la spécification opérationnelle d'une spécification dénotationnelle, pour simplifier le plus possible la phase de conception que doit effectuer le programmeur. C'est en partie le but recherché par les langages de programmation. Ils se basent sur des logiques ou des paradigmes particuliers, qui peuvent se révéler plus ou moins adaptés à certains types de problèmes algorithmiques. Au milieu des années 90, on recensait plus de 2000 langages [24]. Un langage de programmation repose sur un générateur (compilateur) ou un interprète (machine virtuelle) assurant la traduction automatique du code source "haut-niveau" vers le code machine "bas-niveau".

Idéalement, on pourrait imaginer que la formalisation dénotationnelle soit suffisante pour exprimer le programme à réaliser : par exemple un programme de tri est un programme qui, à partir d'une liste, fournit une permutation de cette liste où les éléments sont ordonnés. Cela reviendrait à ne donner que des définitions extensionnelles des programmes [24]. Cependant, du point de vue du programmeur celles-ci ne sont pas assez expressives. Par exemple, alors qu'ils produisent le même résultat le tri a bulles et le tri rapide ne sont pas équivalent d'un point de vue opérationnel. A la place, on préfère utiliser une spécification dénotationnelle et vérifier à posteriori que le programme est correct vis-à-vis de cette spécification. Un exemple commun est le système de types dans les langages de programmation. Lorsque le programme est correct, *i.e.* si la vérification des types réussit, on a l'assurance que le programme est sûr : les manipulations et les calculs qu'il effectue sont toujours réalisés sur des données bien interprétées. Il n'y a pas de risque de confusion entre un entier et une adresse mémoire (Un chou est un chou ! [37]). Dans le cas des systèmes de types, la spécification dénotationnelle est facile à construire car il s'agit d'annotations intégrées directement au langage. Pour exprimer des propriétés plus fortes, on a souvent besoin d'un langage de spécification beaucoup plus évolué qui prend place à côté du code source en commentaire (ex : JML pour Java) ou dans un fichier séparé.

Il est aussi possible de se passer d'une partie des annotations dans certains langages modernes celles-ci pouvant être inférées automatiquement pour certaine vérification. Dans ce cas, une analyse automatique et statique (sans exécution) du programme déduit automatiquement une spécification particulière, à partir de laquelle s'effectue la vérification. Certains types de propriétés peuvent être vérifiées de manière complètement automatique. Parmi les domaines qui fournissent des outils à l'analyse statique, on peut citer les analyses utilisant la génération de contraintes sur les variables du programme, la théorie de l'interprétation abstraite [22] ou le model-checking [25]. L'interprétation abstraite génère automatiquement une abstraction de l'ensemble des comportements possibles du programme analysé pour des domaines particuliers. Le model-checking fournit aussi des outils d'analyses entièrement automatiques pour la vérification de propriétés temporelles mais suppose qu'une spécification opérationnelle "haut-niveau" mais correcte du programme est fournie. Enfin, lorsque les propriétés sont trop difficiles à prouver automatiquement, on peut faire

appel à des assistants de preuves [19, 39, 2] qui permettent de construire interactivement des preuves correctes de programmes : le langage de l'assistant est assez expressif pour exprimer le programme et ses propriétés. Des outils comme KeY [41] pour Java, Frama-C [27] pour le C génèrent à partir d'une spécification formelle et du code source d'un programme des obligations de preuves que l'utilisateur doit prouver (soit à la main soit automatiquement) afin d'assurer la correction du programme vis-à-vis de la spécification.

1.2 La réécriture pour modéliser les programmes

La réécriture est un modèle de calcul communément utilisé en informatique. Elle formalise le calcul par des règles de transformation syntaxique pour des objets tels que des mots ou des termes par exemple. Une règle de réécriture est un objet de la forme $l \to r$. La règle de réécriture s'applique à un objet t si celui-ci contient une instance compatible avec l le membre gauche de la règle. Lorsque l'application est possible, on dit que t se réécrit en un nouvel objet t' que l'on obtient en remplaçant l'instance du membre gauche l par l'instance du membre droit r correspondante. Mais on s'intéresse plus particulièrement à la réécriture de termes. Un exemple courant est la définition des opérations élémentaires sur les entiers naturels suivant l'axiomatisation de Peano : les termes sont alors définis par les opérateurs usuels $\_ * \_$, $\_ + \_$ sur les entiers naturels représentés par $0 \in \mathbb{N}$ et $Sn \in \mathbb{N}$ qui représentent le successeur de l'entier $n \in \mathbb{N}$. Ainsi $SSS0$ que l'on peut abréger par la notation $S^3 0$ correspond à l'entier naturel que l'on note plus volontiers 3. On peut définir la calcul de l'addition et de la multiplication, chacune par deux règles de la façon suivante :

$$\text{Addition} : \begin{cases} 0 + x \to x \\ Sx + y \to x + Sy \end{cases} \qquad \text{Multiplication} : \begin{cases} 0 * x \to 0 \\ Sx + y \to (x * y) + x \end{cases}$$

Les variables x et y peuvent être remplacées par n'importe quel terme, pour constituer l'instance du membre gauche d'une règle de réécriture. Par exemple le terme $t = SS0 + SS0$ constitue une instance du membre gauche de la règle $Sx + y \to x + Sy$ en posant $x = S0$ et $y = SS0$: on peut donc réécrire t en $t' = S0 + SSS0$. A contrario, t et t' ne contiennent pas d'instance pour appliquer la règle $0 + x \to x$. En répétant successivement les étapes de réécriture, on obtient la chaîne de réécriture suivante :

$$SS0 + SS0 \longrightarrow S0 + SSS0 \longrightarrow 0 + SSSS0 \longrightarrow SSSS0$$

La transformation syntaxique de $SS0 + SS0$ en $SSSS0$ correspond au résultat attendu pour l'opération 2 "plus" 2 qui est égal à 4.

L'expressivité des systèmes de réécriture permet de modéliser plus que l'arithmétique. En particulier tout système calculatoire peut-être modélisé, dont les programmes informatiques par exemple. En fait, certains langages de programmation comme OCaml [43] ou Tom [50] intègrent nativement la réécriture. L'exemple suivant donne un codage de la célèbre[3] fonction de Fibonacci. On note par $fib(n)$ le nombre de Fibonacci de rang n. En

3. au moins chez les mathématiciens et les cuniculiculteurs

reprenant et complétant la formalisation des entiers naturels précédente, on peut définir la fonction $fib(n)$ de la manière suivante :

$$\begin{cases} 0 \leq y \rightarrow true \\ Sx \leq Sy \rightarrow x \leq y \\ Sx \leq 0 \rightarrow false \end{cases} \qquad \begin{cases} x - 0 \rightarrow x \\ Sx - Sy \rightarrow x - y \\ 0 - Sy \rightarrow \bot \end{cases}$$

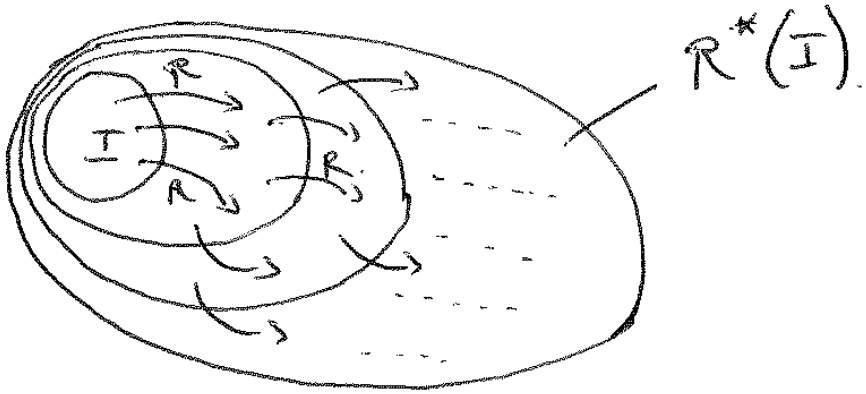
$$\begin{cases} if(true, x, y) \rightarrow x \\ if(false, x, y) \rightarrow y \end{cases}$$

$$fib(x) \rightarrow if(x \leq S0, S0, fib(x - S0) + fib(x - SS0))$$

1.3 Vérification de systèmes de réécriture

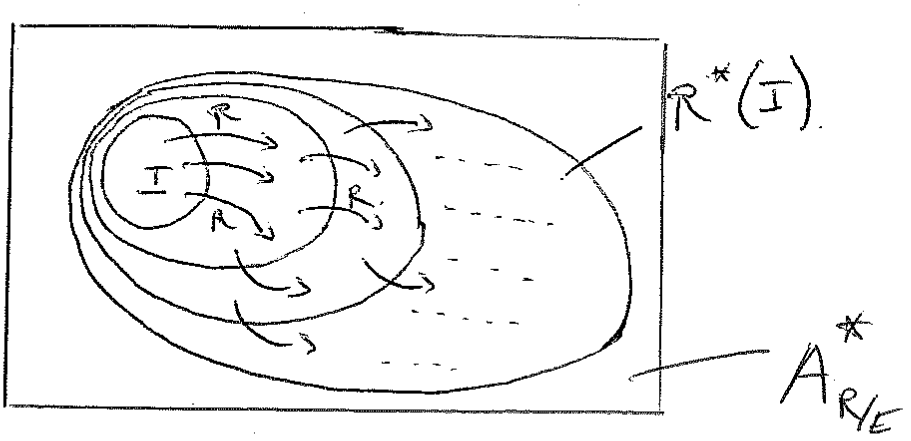
On peut remarquer que la définition de la fonction utilise la définition de la soustraction qui n'est que partiellement définie, ce que l'on symbolise ici par la réécriture en $\bot$. On peut donc se demander si la fonction *fib* est correctement définie pour tout $n \in \mathbb{N}$, c'est à dire qu'elle ne produit jamais le terme $fib(\bot)$ correspondant simplement à un plantage du programme.

Une manière de vérifier que le système de réécriture n'engendre pas le terme $fib(\bot)$ est de générer tous les termes que peut produire le système de réécriture. Pour cela on peut décider de représenter l'ensemble des termes initiaux $fib(n)$, correspondant à l'ensemble des valeurs puis de calculer l'ensemble des termes que l'on peut obtenir, qui correspondent finalement à l'ensemble des valeurs possibles que peut prendre le programme. La technique consiste à compléter itérativement l'ensemble I, par les nouvelles valeurs que peut prendre la fonction jusqu'à ce l'ensemble soit saturé. On note cet ensemble $\mathcal{R}^*(I)$ c'est à dire l'ensemble des termes atteignables par réécriture des termes initiaux.



En supposant que l'on sait calculer $\mathcal{R}^*(I)$, on est alors à même de vérifier que le terme $\mathit{fib}(\perp)$ n'est jamais produit. C'est ce que l'on appelle communément le problème d'atteignabilité.

Mais plusieurs problèmes doivent être résolus pour être capable de calculer $\mathcal{R}^*(I)$. L'ensemble des termes initiaux peut être infini, comme c'est d'ailleurs le cas pour la fonction de Fibonacci. Il faut donc fournir une représentation finie d'un ensemble infini. Les automates d'arbres sont une approche tout à fait indiquée pour résoudre ce problème sur les termes. Cependant, tout ensemble de termes n'est pas représentable par un automate d'arbres. Et c'est souvent le cas de l'ensemble des états accessibles d'un programme, et des termes atteignables d'un systèmes de réécriture qui le modélise. On peut en déduire qu'il n'existe pas toujours d'automates d'arbres capables de représenter l'ensemble des termes atteignables. En fait, de manière plus générale, il n'est pas possible de calculer (et de représenter) l'ensemble des termes atteignables pour un système de réécriture donné. Par contre, il est possible d'en calculer une sur-approximation. C'est l'objectif de la complétion d'automates d'arbres : compléter le langage [4] d'un automate d'arbres en ajoutant successivement les termes atteignables. C'est un semi-algorithme introduit par Th. Genet [30], qui calcule un automate $A^*_{\mathcal{R},E}$ caractérisant une sur-approximation d'un ensemble $\mathcal{R}^*(I)$. Lorsqu'il n'est plus possible de compléter l'automate, ce dernier est clos par réécriture, c'est à dire qu'il contient tous les termes atteignables. L'approximation est paramétrée par E un ensemble d'égalités (ou équations) qui définissent des classes d'équivalence qui permettent des raccourcis syntaxiques. Par exemple, on peut définir les équations $x+y = y+x$ pour définir la commutativité de l'addition ou $Sx = SSx$ pour considérer tout entier (sauf 0) équivalent à son successeur : on partitionne alors les entiers en deux classes $\dot{0}$ et $\dot{1}$. La complétion est un semi-algorithme, car si l'ensemble d'équations ne construit pas des classes d'équivalence assez fortes, alors le calcul diverge.



Du point de vue du problème d'atteignabilité, calculer une sur-approximation reste correct : en effet, si le terme $\mathit{fib}(\perp)$ n'est pas dans la sur-approximation de $\mathcal{R}^*(I)$ alors il n'est

4. l'ensemble des termes représenté par l'automate

clairement pas dans $\mathcal{R}^*(I)$ non plus. Par contre, si la vérification échoue on ne peut rien dire. L'outil Timbuk [34] implémente l'algorithme de complétion et permet de vérifier des propriétés de sûreté sous forme de problème d'atteignabilité. Il est utilisé principalement dans la vérification des protocoles cryptographiques. Il se base sur une spécification symbolique et opérationnelle du système à vérifier, c'est à dire un système de réécriture où les termes sont les états possibles du protocole, et les règles de réécriture modélisent les transitions du protocole. Il fonctionne suivant le principe illustré ci-dessus. Plus récemment, il a été montré qu'il est possible de modéliser un sous-ensemble important du ByteCode Java par un système de règles de réécriture [7]. Les termes du systèmes représentent les états de la machine virtuelle Java, et les règles de réécriture la sémantique de Java. L'outil Copster [5] génère à partir d'un fichier `.class` le système de réécriture : l'exécution du système de réécriture correspond exactement à l'exécution du programme Java.

1.4 Présentation des Contributions de cette thèse

Les contributions de cette thèse trouvent leur place dans le cadre de vérification de programmes Java compilés en systèmes de réécriture et dont l'analyse se base sur la complétion d'automates d'arbres. La complétion a pour ambition de proposer une technique intermédiaire entre la preuve de programme entièrement réalisée à la main, et les outils d'analyse statique. Ces derniers, bien qu'ils soient complètement automatiques, présentent le défaut de ne traiter que certains types de propriétés reposant sur une théorie qui est décidable, comme l'interprétation abstraite et le model-checking. La complétion permet de définir un cadre de vérification semi-automatique grâce à une abstraction régulière paramétrable en fonction de l'analyse et des propriétés à vérifier. La seule intervention dans le processus d'analyse reste la construction de l'ensemble d'équations qui peuvent être générées automatiquement ou fournies directement à la main. Dans ce dernier cas, le degré d'expertise requis est toujours moins important que celui nécessaire à la preuve de programme entièrement manuelle. D'autre part, l'approximation calculée est toujours correcte, quelle que soit l'abstraction choisie. Les travaux de cette thèse ont pris place dans le projet de l'ANR Ravaj [45]. Les objectifs du projet portaient notamment sur l'adaptation de la complétion à l'analyse d'applications Java et la certification des analyses effectuées. En cours de projet, deux autres objectifs se sont dégagés comme le passage à l'échelle, mais aussi le besoin d'étendre les propriétés vérifiables à la logique temporelle. Dans le cas des programmes en ByteCode Java, la taille des systèmes de réécriture est trop importante pour espérer construire à la main une bonne abstraction. La première contribution de cette thèse est de proposer une technique de raffinement automatique d'approximation afin de réduire l'expertise nécessaire à la définition des équations. La deuxième contribution étend les propriétés vérifiables par la complétion aux propriétés temporelles. Enfin la troisième contribution est de certifier les analyses effectuées.

Le raffinement automatique

L'objectif du chapitre 3 de cette thèse est de montrer comment étendre la complétion aux techniques de raffinement automatique de l'abstraction. Fournir une abstraction permettant à la complétion de terminer est faisable même pour les programmes Java. Le problème est de fournir une abstraction qui soit assez fine pour vérifier la propriété attendue, tout en assurant la terminaison. Sous ces contraintes, quelque soit le niveau d'expertise, il est à peu près impossible de définir à la main la bonne abstraction. Le raffinement automatique de l'abstraction permet de simplifier cette tâche : on peut alors définir une abstraction peu précise mais qui assure la terminaison du calcul, l'abstraction étant affaiblie automatiquement lorsque c'est nécessaire. La contribution de ce chapitre est double. Tout d'abord, elle offre à la complétion la possibilité de construire des contre-exemples. Cette fonctionnalité, bien qu'indispensable, était absente et difficile à mettre en place dans le formalisme d'origine. Ensuite, on propose un algorithme de raffinement, non pas basé sur une tabulation des étapes de calculs et sur des calculs arrières qui peuvent s'avérer coûteux, mais sur un élagage original de l'automate construit.

Ce chapitre est actuellement en cours de soumission.

La vérification comportementale

Le chapitre 4 montre comment la complétion d'automates d'arbres peut être utilisée afin de vérifier des propriétés temporelles sur le graphe d'appels des méthodes d'un programme Java. Il difficile de montrer des propriétés avec une composante temporelle en se basant uniquement sur la vérification du problème d'atteignabilité. Une propriété comme *si l'utilisateur clique sur le bouton Non, alors la requête ne sera pas émise sur le réseau* est un exemple des propriétés que l'on souhaiterait prouver sur des programmes Java. Dans ce chapitre, on montre comment tirer partie de l'information contenue dans l'automate. Cette abstraction de la réécriture correspond à une abstraction de l'exécution du programme qui induit notamment la construction explicite du graphe d'appels des méthodes Java. On montre comment extraire cette information de l'automate complété, et comment utiliser les techniques standard de vérification issues du model-checking, ce qui permet de vérifier des propriétés en logique temporelle LTL.

Les travaux du chapitres 4 ont été présentés lors du workshop Rule'09 [13].

La certification de la complétion

Le chapitre 5 décrit les travaux réalisés dans l'assistant de preuves Coq pour certifier la complétion d'automates d'arbres. L'outil Timbuk a été notablement optimisé, et par ailleurs, différentes implémentations ont vu le jour. Certaines de ces implémentations n'utilisent plus les automates d'arbres au niveau algorithmique mais uniquement comme format d'entrée du problème et comme format de sortie pour le résultat. Par conséquence, ces outils sont de plus en plus éloignés de la spécification originale de l'algorithme. Les

optimisations sont une source de bugs supplémentaires, généralement difficiles à repérer. D'autre part, les implémentations sont capables de traiter des problèmes de taille beaucoup plus importante. Comme le résultat est non vérifiable à la main, il est difficile de se convaincre de sa correction. L'ensemble de ces constats a permis de conclure qu'il était nécessaire de vérifier les résultats de ces outils, de manière à obtenir une preuve Coq de la validité des résultats. C'est le rôle que joue le vérificateur certifié en Coq. A partir d'une spécification de la réécriture et des automates d'arbres en Coq, on formalise la correction d'un automate point fixe par rapport au problème d'atteignabilité. On montre ensuite comment vérifier la propriété en Coq, ce qui définit le vérificateur. Ensuite, il est possible d'extraire puis de compiler un vérificateur conforme à la spécification Coq. Celui-ci permet de vérifier la validité de chaque résultat obtenu par un des outils qui implémentent la complétion.

Le chapitre 5 a fait aussi l'objet d'une publication lors de la conférence IJCAR'08 [14].

Chapitre 2

Prérequis

Ce chapitre a pour ambition de rappeler les bases nécessaires pour la théorie de la réécriture, des automates d'arbres, et présentera le cadre de la vérification de modèles symoblique auquel nous nous intéresserons dans cette thèse. Pour plus détails, le lecteur pourra trouver plus d'informations sur la réécriture dans [3] et dans [18] pour les automates d'arbres.

2.1 Termes et réécriture

Définition 2.1.1. *Une* ***signature*** $\mathcal{F}$ *est un ensemble de symboles de fonctions dont l'arité est fixée par une fonction* $\phi : \mathcal{F} \to \mathbb{N}$. *Par* $\mathcal{F}_i$, *on distingue l'ensemble des symboles* $f \in \mathcal{F}$ *d'arité* $\phi(f) = i$. *Dès lors,* $\mathcal{F}_0$ *caractérise l'ensemble des* ***constantes****.*

Définition 2.1.2. *Soit* $\mathcal{F} = \{f_2, g_1, h_2, \ldots\}$ *une signature*[1]. *Soit* $\mathcal{X}$ *un ensemble de* ***variables*** *tel que* $\mathcal{X} \cap \mathcal{F} = \emptyset$. *L'ensemble* $\mathcal{T}(\mathcal{F}, \mathcal{X})$ *dénote l'****algèbre de termes*** *engendrée par la signature* $\mathcal{F}$ *à partir de* $\mathcal{X}$. *On définit* $\mathcal{T}(\mathcal{F}, \mathcal{X})$ *comme le plus petit ensemble tel que :*

- $\mathcal{X} \subseteq \mathcal{T}(\mathcal{F}, \mathcal{X})$
- *pour tout* $n \in \mathbb{N}$ *et* $f \in \mathcal{F}_n$, *si* $t_1, \ldots, t_n \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ *alors* $f(t_1, \ldots, t_n) \in \mathcal{T}(\mathcal{F}, \mathcal{X})$

Les éléments de $\mathcal{T}(\mathcal{F}, \mathcal{X})$ sont appelés des **termes**. On note $\mathcal{T}(\mathcal{F})$, l'algèbre des **termes clos** – *i.e.* sans variable – lorsque $\mathcal{T}(\mathcal{F}, \mathcal{X})$ est engendrée à partit de $\mathcal{X} = \emptyset$. Cette définition inductive nous indique que toute variable est un terme, et que l'application d'un symbole de fonction à des termes engendre de nouveaux termes. Cette construction permet de voir les termes comme des arbres étiquetés. Considérons $\mathcal{F} = \{f_3, g_2, h_1\}$ et $\mathcal{X} = \{x, y, z\}$. On peut alors construire le terme $f(g(x, h(y)), g(x, g(y, z)), h(x))$ que l'on peut représenter par l'arbre de la figure 2.1.

La représentation d'un terme sous forme d'arbre permet de voir facilement que le terme $f(g(x, h(y)), g(x, g(y, z)), h(x))$ est constitué des **sous-termes** représentés

1. Pour définir rapidement $\mathcal{F}$, Chaque symbole est indicé par son arité

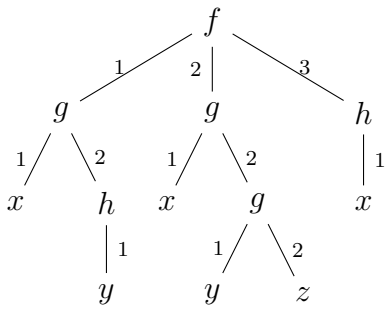


FIGURE 2.1: le terme $f(g(x,h(y)),g(x,g(y,z)),h(x))$ vu comme un arbre étiqueté

par les sous-arbres dans la figure 2.1. Ainsi $g(x,g(y,z))$ est un sous-terme de $f(g(x,h(y)),g(x,g(y,z)),h(x))$. Un sous-terme sont caractérisé par sa position à partir de la racine du terme qui le contient. Une **position** p est définie comme un **mot de** $\mathbb{N}^*$ qui correspond à un chemin dans l'arbre. Le mot vide ϵ correspond à la racine du terme.

Définition 2.1.3. *Soit un terme $t \in \mathcal{T}(\mathcal{F},\mathcal{X})$. L'ensemble $\mathcal{P}os(t)$ des positions de t se définit inductivement comme :*

- $\mathcal{P}os(t) = \{\epsilon\}$ *si* $t \in \mathcal{X}$
- $\mathcal{P}os(f(t_1,\ldots,t_n)) = \{\epsilon\} \cup \{i.p \mid 1 \leq i \leq n \text{ et } p \in \mathcal{P}os(t_i)\}$

Si $p \in \mathcal{P}os(t)$, alors $t|_p$ dénote le **sous-terme de t à la position** p et $t[s]_p$ dénote le terme obtenu après le **remplacement** du sous-terme $t|_p$ par le terme s. On peut les définir par induction sur $p \in \mathcal{P}os(t)$:

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

et

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

Définition 2.1.4. *L'ensemble des variables d'un terme $t \in \mathcal{T}(\mathcal{F},\mathcal{X})$ est dénoté par $\mathcal{V}(t)$ et se définit simplement comme $\mathcal{V}(t) = \{x \in \mathcal{X} \mid \exists p \in \mathcal{P}os(t) \text{ t.q. } x = t|_p\}$*

On remarque pour tout terme $t \in \mathcal{T}(\mathcal{F},\mathcal{X})$, $\mathcal{V}(t) = \emptyset$ alors t est un terme clos, ce qui est équivalent à $t \in \mathcal{T}(\mathcal{F})$.

Définition 2.1.5. *Une substitution est une fonction σ de $\mathcal{X}$ vers $\mathcal{T}(\mathcal{F},\mathcal{X})$, telle que pour un certain nombre de variables $x \in X$ on a $x \neq \sigma(x)$. L'ensemble des ces variables est noté $\mathcal{D}om(\sigma)$. On étend les substitutions à des endomorphismes de $\mathcal{T}(\mathcal{F},\mathcal{X})$. On note alors $t\sigma$ (ou $t.\sigma$) l'application de la substitution σ au terme t qui remplace toute occurence des variables par leur image respective via σ :*

- $t\sigma = \sigma(t)$ *lorsque* $t \in \mathcal{X}$
- $f(t_1,\ldots,t_n)\sigma = f(t_1\sigma,\ldots,t_n\sigma)$

On appelle id, la substitution dont le domaine $\mathcal{D}om(\sigma)$ est vide. Pour tout terme $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, on a $t.id = t$.

On dit qu'un terme t est une **instance** d'un terme s lorsqu'il existe une substitution σ telle que $t = s\sigma$.

Un système de règles de réécriture – TRS en abrégé [2] – $\mathcal{R}$ est un ensemble de règles de réécriture.

Définition 2.1.6. *Une **règle de réécriture** est une paire, notée $l \to r$, composée de deux termes $l, r \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, telle que $l \notin \mathcal{X}$, et $\mathcal{V}ar(l) \supseteq \mathcal{V}ar(r)$. On appelle l et r respectivement le membre gauche et le membre droit de la règle. Une règle de réévriture est **linéaire à gauche** (ou à droite) si chaque variable de $\mathcal{X}$, n'apparait pas plus d'une fois dans son membre gauche (droit resp.).*

Un système de règles de réécriture est linéaire à gauche (ou à droite) si chacune de ses règles est linéaire à gauche (à droite *r*esp.). De même, le système est linéaire si il est linéaire à gauche et à droite.

Définition 2.1.7. *Si $\mathcal{R}$ est un TRS, alors on note $\to_{\mathcal{R}}$ la relation de réécriture qu'il induit sur les termes. On dit que le terme s se réécrit en t par la règle $l \to r \in \mathcal{R}$ si et seulement si l'un des sous-termes de s est une instance de l et t est le terme s où l'instance de l est remplacée par l'instance de r correspondante :*
$s \to_{\mathcal{R}} t \iff$

$$\exists\ l \to r \in \mathcal{R},\ p \in Pos(s),\ \sigma : \mathcal{X} \to \mathcal{T}(\mathcal{F}, \mathcal{X}) \text{ tels que } \qquad s|_p = l\sigma \quad \text{et} \quad t = s[r\sigma]_p$$

Ainsi, la règle $g(x, y) \to y$ permet de réécrire le terme $h(g(x, g(y, z)))$ soit en $h(g(x, z))$ ou $h(g(y, z))$, suivant que l'on choisisse de réécrire le sous-terme à la position 1.2 ou à la position 1.

La clôture transitive et réflexive de $\to_{\mathcal{R}}$ se note $\to^*_{\mathcal{R}}$. Si A est un ensemble de termes clos, on définit l'ensemble des $\mathcal{R}$-descendants de A comme $\mathcal{R}^*(A) = \{t \in \mathcal{T}(\mathcal{F}) \mid \exists s \in A \text{ t.q. } s \to^*_{\mathcal{R}} t\}$.

Définition 2.1.8. *Une **équation** est une paire, notée $s = t$, composée de deux termes $s, t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$. Si e est l'équation $s = t$, alors deux termes u et v sont égaux modulo e si et seulement si*
$u =_e v \iff$

$$\exists\ p \in Pos(s),\ \sigma : \mathcal{X} \to \mathcal{T}(\mathcal{F}, \mathcal{X}) \text{ tels que } \quad s|_p = l\sigma \quad \text{et} \quad t = s[r\sigma]_p$$

Si E est un ensemble d'équations, on note $=_E$ la plus petite relation d'équivalence qui contienne la relation $\{u =_E v \mid \exists e \in E,\ u =_e v\}$

2. abréviation issue de la formulation anglaise équivalente *term rewriting system*

Définition 2.1.9. *Soient $\mathcal{R}$ un TRS, et E un ensemble d'équations. On définit la* ***réécriture modulo*** *comme la relation $\rightarrow_{\mathcal{R}/E}$ définie comme :*

$$\forall\ s,t \in \mathcal{T}(\mathcal{F},\mathcal{X}),\quad s \rightarrow_{\mathcal{R}/E} t \equiv \exists u,v \in \mathcal{T}(\mathcal{F},\mathcal{X}),\quad s =_E u \rightarrow_{\mathcal{R}} v =_E t$$

De la même manière que pour $\mathcal{R}$, note $\rightarrow^*_{\mathcal{R}/E}$ la clôture transitive et réflexive de $\rightarrow_{\mathcal{R}/E}$, et on définit l'ensemble des $\mathcal{R}/E$-descendants pour un ensemble A de termes clos par ${\mathcal{R}/E}^*(A) = \{t \in \mathcal{T}(\mathcal{F}) \mid \exists s \in A \text{ t.q. } s \rightarrow^*_{\mathcal{R}/E} t\}$.

2.2 Les automates d'arbres

Soit Q un ensemble fini de constantes appelées **états** tel que $Q \cap \mathcal{F} = \emptyset$. On définit $\mathcal{T}(\mathcal{F} \cup Q)$ l'algèbre engendrée par $\mathcal{F}$ et Q, que l'on appelle l'ensemble des **configurations**.

Définition 2.2.1. *Dans un automate d'arbres, une* ***transition*** *est une règle de réécriture de la forme $c \rightarrow q$, où c est une configuration i.e. $c \in \mathcal{T}(\mathcal{F} \cup Q)$ et $q \in Q$ est un état. On distingue deux sortes de transitions dans un automate d'arbres :*

- *Une* ***transition normalisée*** *est une transition de la forme $f(q_1,\ldots,q_n) \rightarrow q$ avec $f \in \mathcal{F}_n$, et $q_1,\ldots,q_n,q \in Q$ sont des états*
- *Une* ***ε-transition*** *est une transition de la forme $q' \rightarrow q$ où $q',q \in Q$ sont des états*

Définition 2.2.2. *Un automate d'arbres ascendant fini non-déterministe – ou plus simplement automate d'arbres – pour la signature $\mathcal{F}$ est un quadruplet $A = \langle \mathcal{F}, Q, Q_F, \Delta\rangle$, avec*

- *Q un ensemble d'états,*
- *$Q_F \subseteq Q$ un ensemble d'états finals et,*
- *Δ un ensemble fini de transitions normalisées et de ε-transitions.*

Un automate d'arbres A reconnaît des termes clos de $\mathcal{T}(\mathcal{F})$. L'exécution d'un automate d'arbres débute en réécrivant les feuilles du terme au moyen des transitions, remplacant progressivement chaque sous-terme par des états jusqu'à la racine du terme. La sémantique d'une exécution repose sur le principe qu'un terme t (et donc un sous-terme) est reconnu par un certain état q de l'automate lorsqu'on a réduit t en q par réécriture, ce que l'on note $t \rightarrow^*_\Delta q$ ou plus généralement $t \rightarrow^*_A q$. De plus, si l'état q est final (*i.e.* $q \in Q_f$), alors t est reconnu ou accepté par l'automate.

Définition 2.2.3. *L'ensemble des termes reconnus par l'automate A en l'état q est défini par $\mathcal{L}(A,q) = \{t \in \mathcal{T}(\mathcal{F}) \mid t \rightarrow^*_A q\}$. Le langage reconnu par A est $\mathcal{L}(A) = \bigcup_{q_f \in Q_f} \mathcal{L}(A,q_f)$.*

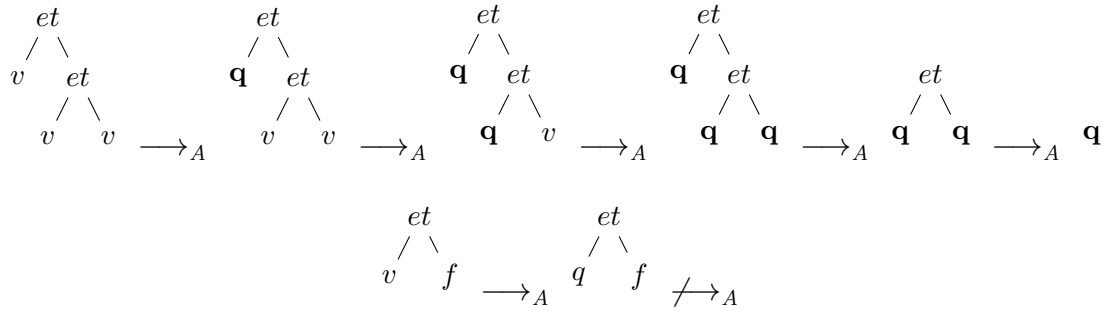
Sans perte de généralité, on considèrera dans la suite que **tout automate A est toujours nettoyé** c'est à dire que A ne posssède pas d'état dont le langage est vide : $\forall q \in Q, \mathcal{L}(A,q) \neq \emptyset$.

Exemple 2.2.4. *Soit l'automate d'arbres $A = \langle \mathcal{F}, Q, Q_F, \Delta\rangle$ avec*

- *$\mathcal{F} = \{et_2, v_0, f_0\}$*

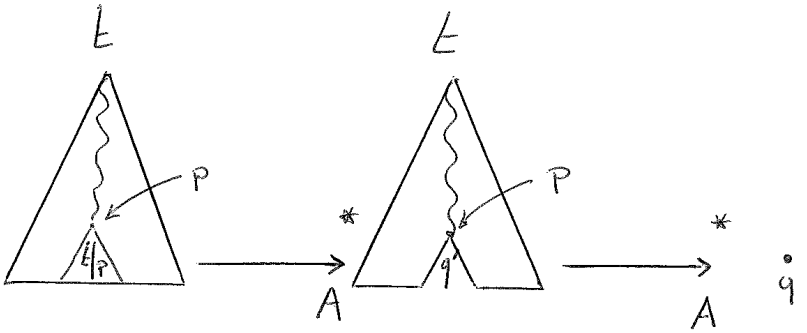
- $Q = \{q\}$
- $Q_F = \{q\}$
- $\Delta = \{v \to q, et(q,q) \to q\}$

Cet automate ne reconnait que les conjonctions s'évaluant à vrai (V).



Propriété 2.2.5. *Soit un terme $t \in \mathcal{T}(\mathcal{F})$ reconnu par l'état $q \in Q$ d'un automate A. On peut décomposer autant de fois que nécessaire l'exécution de l'automate pour le terme t. Tout sous-terme de t est aussi reconnu par l'automate A.*

$$t \to_A^* q \implies \forall p \in \mathcal{P}os(t), \exists q' \in Q,\ t.q.\ t|_p \to_A^* q' \text{ et } t[q']_p \to_A^* q$$



Soit $L \subseteq \mathcal{T}(\mathcal{F})$ un ensemble de ensembles de termes clos. Le langage L est régulier si et seulement si il existe un automate d'arbres A tel que $L = \mathcal{L}(A)$. Les langages réguliers de termes sont clos par union, intersection, et complémentation. Cependant, il est important de noter que la complémentation nécessite de déterminiser l'automate d'arbres ce qui peut entraîner une explosion combinatoire de l'exponentielle.

2.3 Le Model-Checking régulier

Dans cette thèse, on s'intéresse à la vérification de propriétés de systèmes, et en particulier aux systèmes pour lesquels l'ensemble des états accessibles est infini mais peut-être modélisé de manière finie [51]. Il est à noté que l'approche envisagée fonctionne aussi dans le cas de systèmes à ensemble fini d'états.

Le premier problème est de fournir une représentation symbolique pour exprimer et manipuler des ensembles potentiellement infinis d'états. C'est une question récurrente en vérification qui est malheureusement indécidable, et pour laquelle il n'existe que des solutions partielles. Ici, l'angle d'attaque retenu est celui de la vérification de modèles à base de langage régulier de termes plus communément appelée *Tree Regular Model Checking (TRMC)* [1]. Cette approche est une extension de l'approche qui fut initiallement introduite pour calculer l'ensemble des états atteignables en utilisant les langages réguliers de mots [12, 8].

En TRMC, on modélise un système de transitions (tel qu'un programme ou un protocole) par un triplet $(\mathcal{F}, I, Rel)$, avec

- $\mathcal{F}$ une signature permettant de construire $\mathcal{T}(\mathcal{F})$ l'ensemble des termes représentants les différentes configurations ou états du système ;
- $I \subseteq \mathcal{T}(\mathcal{F})$ un ensemble régulier de configurations donc représentable par un automate d'arbres A, *i.e.* $\mathcal{L}(A) = I$;
- Rel est une relation de transition représentée par un ensemble de règles de réécriture [3]. On impose à $\mathcal{R}$ d'être linéaire à gauche.

Dans ce contexte, un programme sera alors representé par un triplet $(\mathcal{F}, A, \mathcal{R})$. Ce type de modélisation est très expressif, d'une part parce que les systèmes de réécriture linéaires à gauche sont Turing-complet [38], et d'autre part il est facile de modéliser les protocoles cryptographiques [32], ainsi que pour exprimer un sous-ensemble important du ByteCode Java [7]. On considère le problème d'atteignabilité pour ce modèle.

Définition 2.3.1 (Problème d'atteignabilité)**.** *Soit un programme $(\mathcal{F}, A, \mathcal{R})$ et un ensemble Bad de termes correspondant à des états interdits que le programme ne doit jamais atteindre. Le problème d'atteignabilité consiste à vérifier qu'il n'existe aucun terme de $\mathcal{R}^*(\mathcal{L}(A))$ qui soit dans Bad.*

Le problème d'atteignabilité permet de vérifier des propriétés de sûreté pour le système considéré. L'ensemble Bad décrit des configurations invalides qui ne doivent pas être atteignables pour le système.

Dans le cas d'un protocole cryptographique Bad peut décrire des configurations résultant d'une attaque du protocole ex : le message chiffré est révélé en clair à un protagoniste externe à la transaction. . ..

Pour les programmes Java, Bad peut décrire par des états où l'on s'apprête à accéder à une zone mémoire non définie ex : pointeur null,. . .

2.4 La complétion d'automates d'arbres

Dans le cas des systèmes à espace d'états fini (et de taille raisonnable), le calcul de l'ensemble des termes atteignables ($\mathcal{R}^*(\mathcal{L}(A))$) peut se réduire à une simple énumération

3. A l'origine, *Rel* est modélisée par un transducteur d'arbres. Mais comme les automates d'arbres peuvent être vus comme une classe particulière de systèmes règles de réécriture, on choisit ici de généraliser

des termes que l'on peut atteindre à partir des termes de l'ensemble initial. Mais il n'est pas possible d'utiliser une telle technique sur les systèmes à espace infini d'états (ou trop important). En effet, lorsque $\mathcal{R}$ ne termine pas ou si $\mathcal{L}(A)$ est infini, l'ensemble $\mathcal{R}^*(\mathcal{L}(A))$ est alors infini et n'est généralement pas calculable [36]. Dans ces cas, il est nécessaire d'*accélérer* l'exploration de l'espace d'états afin d'atteindre notamment les termes situés à une *distance infinie* des termes initiaux, pour se ramener à un *temps de calcul* raisonnable. C'est dans ce contexte que se place la **complétion d'automates d'arbres** décrite dans [30, 26]. C'est un semi-algorithme qui calcule un automate d'arbres, *i.e.* une représentation régulière (et donc finie) des ensembles infinis, qui est en général une sur-approximation de l'ensemble des termes atteignables. Elle peut être utilisée pour résoudre certaines instances du problème d'atteignabilité : la complétion d'automates d'arbres est facilement paramétrable afin de définir la sur-approximation souhaitée [30, 26, 49] pour le problème considéré.

2.4.1 Principe général

Ainsi, en utilisant l'algorithme de complétion on peut construire un automate d'arbres B dont le langage $\mathcal{L}(B) \supseteq \mathcal{R}^*(\mathcal{L}(A))$ contient au moins tous les termes atteignables. Il est alors suffisant de vérifier que l'automate B ne reconnait pas de termes de Bad – *i.e.* $\mathcal{L}(B) \cap Bad = \emptyset$ – pour montrer que le système n'atteint jamais aucune des configurations de Bad soit $\mathcal{R}^*(\mathcal{L}(A)) \cap Bad = \emptyset$.

La complétion d'automates d'arbres termine lorsque le calcule arrive sur un point fixe, lorsque l'automate est $\mathcal{R}$-clos.

Définition 2.4.2. *Un automate d'arbres B est dit $\mathcal{R}$-clos si pour tous termes s, t tels que $s \to_{\mathcal{R}} t$ avec s reconnu par B dans un état q, alors t est aussi reconnu par B dans l'état q. La propriété de clôture est illustrée par le diagramme de la figure 2.2.*

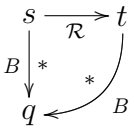


FIGURE 2.2: Propriété de $\mathcal{R}$-clôture

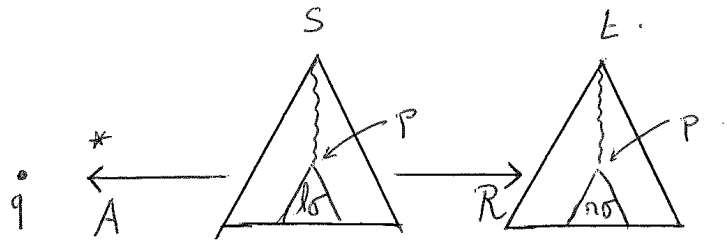
Il est évident de constater que l'on a $\mathcal{L}(B) \supseteq \mathcal{R}^*(\mathcal{L}(A))$ si B est $\mathcal{R}$-clos et si $\mathcal{L}(B) \supseteq \mathcal{L}(A)$ [14].

D'un point de vue algorithmique, la construction d'un automate $\mathcal{R}$-clos, que l'on notera $A^*_{\mathcal{R}}$, à partir de l'automate A consiste à *compléter* l'automate A par de nouvelles transitions pour étendre son langage. L'algorithme de complétion calcule une succession d'automates $A^1_{\mathcal{R}}, A^2_{\mathcal{R}}, \ldots$ en applicant à chaque fois l'ensemble des règles de réécriture sur $A^i_{\mathcal{R}}$ pour produire $A^{i+1}_{\mathcal{R}}$. Le calcul peut s'arrêter lorsque l'automate l'on obtient un automate $\mathcal{R}$-clos $A^k_{\mathcal{R}}$ *i.e.* tel que $\mathcal{L}(A^{k+1}_{\mathcal{R}}) \supseteq \mathcal{L}(A^k_{\mathcal{R}})$.

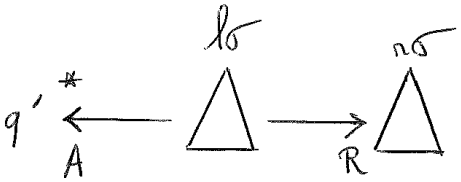
2.4.3 Etape de complétion : calcul exact des $\mathcal{R}$-descendants

Chaque application du TRS $\mathcal{R}$ est appelée **étape de completion** et consiste à rechercher parmi les termes reconnus par l'automate qui peuvent être réécrits dont les descendants ne sont pas encore reconnus par l'automate. Cela revient à rechercher les **paires critiques** $\langle q, t\rangle$ pour lesquelles le diagramme de la figure 2.2 n'est pas clos, *i.e.* on a $s \to_{\mathcal{R}} t$ et $s \to_A^* q$ mais pas $t \not\to_A^* q$.

On ne considère que les paires critiques dont l'origine est localisée à la racine du terme c'est à dire que la position p à laquelle le terme est réécrit par $\mathcal{R}$ est $p = \epsilon$. En effet, quelle que soit la position $p \in \mathcal{P}os(s)$ à laquelle on décide de réécrire le terme s, on peut toujours revenir au cas d'une paire critique où la réécriture n'a lieu qu'à la racine du terme. Il suffit de considérer l'exécution $s \to_A^* q$ par l'automate A. On sait que le terme s se réécrit à la position $p \in \mathcal{P}os(s)$ par une règle $l \to r$ de $\mathcal{R}$. Ce qui signifie que le sous-terme à la position p est une instance de l donc de la forme $l\tau$, où τ est une substitution $\mathcal{X} \to \mathcal{T}(\mathcal{F})$. Or d'après la propriété 2.2.5, comme s est reconnu par l'automate A, c'est aussi le cas pour chacun de ses sous-termes et donc en particulier il existe un état q' tel que $l\tau \to_A^* q'$. On peut donc simplifier le problème de la paire critique $\langle q, t\rangle$ à la position p en considérant la paire critique $\langle q', r\tau\rangle$ à la position ϵ.



Ce qui revient à



Une étape de complétion résout toutes les paires critiques en construisant à partir de l'automate A, un nouvel automate $A^1_{\mathcal{R}}$ avec des transitions supplémentaires de façon à obtenir $t \to_{A^1_{\mathcal{R}}}^* q$. Cela correspond au résultat de l'application $\mathcal{R}$ sur le terme s. Ensuite

on réapplique la procédure pour résoudre les paires critiques de $A^1_{\mathcal{R}}$ pour construire $A^2_{\mathcal{R}}$ jusqu'à obtenir obtenir un automate $A^*_{\mathcal{R}}$ qui soit exempt de paires critiques. En construisant l'automate $A^*_{\mathcal{R}}$, on a atteint un point fixe, et $A^*_{\mathcal{R}}$ est $\mathcal{R}$-clos.

Comme le langage reconnu par l'automate $A^i_{\mathcal{R}}$ peut-être infini, l'ensemble des paires critiques $\langle q, r\tau\rangle$ pour la règle $l \to r \in \mathcal{R}$ peut lui aussi être infini : l'ensemble des termes $l\tau$ reconnu dans l'état q peut-être infini. Chacun des termes $l\tau$ pouvant se réécrire en $r\tau$, on a bien une infinité de paires critiques, qu'il n'est pas envisageable d'énumérer. La solution retenue dans [30] pour contourner ce problème consiste à construire des ensembles de substitutions $\sigma : \mathcal{X} \mapsto Q$ – appelées Q-substitutions – associant à chaque variable des règles de réécriture un état de l'automate. L'ensemble de substitutions à considérer est fini. En effet le domaine des substitutions est restreint à l'ensemble des variables apparaissant dans les règles de réécriture, qui est un ensemble fini. De même, l'ensemble Q des états de l'automate est fini. Cependant, un état peut reconnaître un ensemble infini de termes, et donc l'instance $l\sigma$ peut représenter une infinité de termes. On peut construire l'ensemble des termes dénotés par $l\sigma$ par un ensemble de substitutions $\eta : Q \to \mathcal{T}(\mathcal{F})$ qui associe à chaque état q terme $\eta(q)$ reconnu par l'automate A en l'état q. Ce qui donne

$$\forall \eta \in \{\eta : Q \to \mathcal{T}(\mathcal{F}) \mid \eta(q) \to^*_A q\}, \quad l\sigma.\eta \to^*_A l\sigma$$

Ainsi, réécrire $l\sigma$ en $r\sigma$, revient donc à réécrire tous les termes $l\sigma.\eta$ en $r\sigma.\eta$ du point de vue de l'automate.

Recherche des paires critiques

Définition 2.4.4. *Pour $A^i_{\mathcal{R}}$ un automate d'arbres donné et $l \to r \in \mathcal{R}$ une règle de réécriture, on appelle le* ***problème de filtrage****, le problème qui consiste à retrouver toutes les paires critiques de $l \to r$ avec $A^i_{\mathcal{R}}$. On utilise $l \trianglelefteq q$ pour dénoter plus précisément le problème de filtrage pour tout état q de $A^i_{\mathcal{R}}$.*

La complétion utilise un **algorithme de filtrage** [26] pour résoudre chaque problème $l \trianglelefteq q$ dans l'automate $A^i_{\mathcal{R}}$. Cet algorithme produit un ensemble de Q-subsitutions $\sigma : \mathcal{X} \mapsto Q$ formant des paires critiques $\langle q, r\sigma\rangle$ telles que $l\sigma \to^*_{A^i_{\mathcal{R}}} q$. Le problème de filtrage doit être résolu pour toutes les règles $l \to r$ de $\mathcal{R}$ avec chaque état q de l'automate $A^i_{\mathcal{R}}$.

Définition 2.4.5. *Une paire critique $\langle q, r\sigma\rangle$ est non résolue entre l'automate $A^i_{\mathcal{R}}$ et $\mathcal{R}$ si $r\sigma \not\to^*_{A^i_{\mathcal{R}}} q$.*

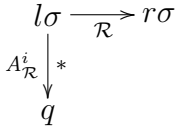


FIGURE 2.3: Paire critique non résolue

La résolution des paires critiques consiste simplement à ajouter la nouvelle transition $r\sigma \to q$. Il y a deux variantes pour ajouter cette transition. L'ajout immédiat de la transition $r\sigma \to q$ ou l'ajout des transitions $r\sigma \to q'$ et $q' \to q$. La deuxième version plus récente, permet de gagner en information : la transition $q' \to q$ permet de distinguer les termes reconnus en q', comme un sous-ensemble des termes q, conséquence de l'étape de complétion. Sans l'utilisation de la ε-transition, il n'est pas possible à postériori de distinguer les successeurs $r\sigma$. Dans la suite, on ne considère plus que la variante $r\sigma \to q'$ et $q' \to q$ pour résoudre les paires critiques, sauf au chapitre 5 où l'on considère un automate sans ε-transitions qu'il est toujours possible d'obtenir par nettoyage des ε-transitions.

En général, les transitions $r\sigma \to q'$ ne respectent pas la définition 2.2.1 et doivent être normalisées avant d'être ajoutées à l'automate.

Définition 2.4.6. *La **normalisation** d'une transition $r\sigma \to q'$ transforme cette tansition en un ensemble Π de transitions normalisées 2.2.1 telles que $r\sigma \to_\Pi^* q'$. Les transitions sont ensuite ajoutées aux transitions de $A^i_\mathcal{R}$.*

On notera que la définition normalisation [30] nécessite souvent l'utilisation de nouveaux états dans l'automate pour construire les nouvelles transitions. C'est d'ailleurs le critère le plus important à considérer du point de vue de la terminaison. Il est important de limiter le plus possible la création de nouveaux états, pour limiter la divergence.

Définition 2.4.7. *On note $\mathtt{C}(A^i_\mathcal{R})$, l'**étape de complétion** qui consiste à rechercher les paires critiques présentes dans $A^i_\mathcal{R}$ pour les résoudre dans l'automate $A^{i+1}_\mathcal{R} = \mathtt{C}(A^i_\mathcal{R})$.*

Propriété 2.4.8. *Soient $A^i_\mathcal{R}$ un automate, et $A^{i+1}_\mathcal{R} = \mathtt{C}(A^i_\mathcal{R})$ l'automate obtenu après une étape de complétion par $\mathcal{R}$, alors*
- *$\mathcal{L}(A^i_\mathcal{R}) \subseteq \mathcal{L}(A^{i+1}_\mathcal{R})$*
- *pour tous termes $s \in \mathcal{L}(A^i_\mathcal{R})$ et $t \in \mathcal{T}(\mathcal{F})$,*

$$s \to_\mathcal{R} t \implies t \in \mathcal{L}(A^{i+1}_\mathcal{R})$$

Sémantique liée aux ε-transitions Lors de la résolution d'une paire critique, la transition $r\sigma \to q'$ reliée à l'état q par le biais de la transition $q' \to q$. Cette ε-transition permet de créer une relation d'ordre entre les termes dans l'automate. En effet, si l'on se place du point de vue de l'état q, on a bien étendu le langage en lui ajoutant $r\sigma.\eta$ les termes représentés par la configuration $r\sigma$. Or si l'on se place du point de vue de l'état q', on ne verra dans $\mathcal{L}(A, q')$ que les termes $r\sigma.\eta$ et leurs $\mathcal{R}$-descendants. Donc à moins que certains des termes dèjà présents avant la résolution de la paire critique $\langle q, r\sigma\rangle$ ne soient des $\mathcal{R}$-descendants de termes reconnus par q', ils ne seront pas dans $\mathcal{L}(A, q')$. En fait, il est possible de définir plus formellement cette relation d'ordre.

Définition 2.4.9. *Soit A un automate d'arbres. On définit par $\to_A^{\not\varepsilon}$ la relation induite par les transitions normalisées de A. On appelle **représentants** de l'état q, tous les termes clos de l'ensemble $\mathcal{R}ep(q) = \{t \in \mathcal{L}(A, q) \mid t \to_A^{\not\varepsilon} q\}$.*

Par définition l'automate $A^0_{\mathcal{R}} = A$ représente l'ensemble initial et n'a jamais été complété : on impose alors qu'il ne contienne pas de ε-transitions. Seule la résolution des paires critiques ajoute des ε-transitions à l'automate. La complétion d'un automate d'arbres par $\mathcal{R}$ assure alors la propriété suivante :

Propriété 2.4.10. *Soit l'automate $A^i_{\mathcal{R}}$ obtenu par complétion. Pour tout état q, chaque terme reconnu en l'état q est le successeur par réécriture d'un représentant de l'état q :*

$$\forall t \in \mathcal{L}(A, q), \exists s \in Rep(q) \text{ } t.q. \text{ } s \rightarrow^*_{\mathcal{R}} t$$

Vers une approximation de $\mathcal{R}^*(\mathcal{L}(A))$ Cependant, excepté pour des classes particulières de TRS [26, 31], l'automate représentant l'ensemble des termes atteignables ne peut pas être obtenu à partir de A en applicant un nombre fini de fois l'étape de complétion...

Exemple 2.4.11. *Soit $\mathcal{R} = \{0 \rightarrow S(0),\ S(x) \rightarrow S(S(X))\}$ qui engendre l'ensemble des entiers naturel à partir de l'automate A_0 consitué de l'unique transition $0 \rightarrow q_0$ avec l'état q_0 final. Le tableau suivant résume quelles paires critiques sont résolues et les transitions ajoutées par chaque étape de complétion pour obtenir l'automate A_i :*

A_0 :	$0 \rightarrow q_0$	-	-
A_1 :	$Sq_0 \rightarrow q_1$	$q_1 \rightarrow q_0$	$\langle q_0, S(0)\rangle$
A_2 :	$Sq_1 \rightarrow q_2$	$q_2 \rightarrow q_1$	$\langle q_1, S(S(q_0))\rangle$
A_3 :	$Sq_2 \rightarrow q_3$	$q_3 \rightarrow q_2$	$\langle q_2, S(S(q_1))\rangle$
A_4 :	$Sq_3 \rightarrow q_4$	$q_4 \rightarrow q_3$	$\langle q_3, S(S(q_2))\rangle$
	$\vdots$	$\vdots$	$\vdots$
	$\vdots$	$\vdots$	$\vdots$

On peut voir que le calcul diverge, car le nouvel état introduit à chaqe étape de complétion, fournit une nouvelle paire critique avec la règle $S(x) \rightarrow S(S(X))$.

On peut aussi remarquer qu'une fois que la paire critique est résolue en l'état q_i, il est aussi résolue pour tous les états q' qui sont accessibles par des ε-transitions. Lorsque l'on a $l\sigma \rightarrow^*_{A_i} q_i$ et $r\sigma \rightarrow^*_{A_i} q_i$ alors pour tout état q tel que $q_i \rightarrow^*_{A_i} q$ alors la paire critique $\langle q, r\sigma\rangle$ est aussi résolue. On utilise cette propriété pour optimiser la recherche des paires critiques.

Le processus de complétion requiert alors une accélération pour terminer. Pour cela on peut utiliser une technique d'approximation basée sur un ensemble d'équations E et produire une sur-approximation de l'ensemble des termes atteignables, *i.e.* un automate d'arbres $A^*_{\mathcal{R},E}$ tels que $\mathcal{L}(A^*_{\mathcal{R},E}) \supseteq \mathcal{R}^*(\mathcal{L}(A))$.

Le principe de l'approximation repose sur la relation $=_E$ induite par les equations E liée à la notion des représentants (*c.f.* définition 2.4.9). L'accélération consiste à définir un nouvel automate $A^i_{\mathcal{R},E}$ à partir de l'automate $A^i_{\mathcal{R}}$:

Elle repose sur la **fusion d'états** dirigées par les équations de E.

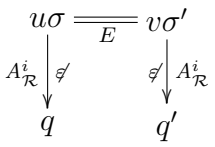


FIGURE 2.4: Fusion d'états par l'équation $u = v$

Définition 2.4.12. *Deux états distincts q, q' de l'automate $A^i_{\mathcal{R}}$ sont **fusionnés** si pour une équation $u = v \in E$, il existe deux Q-substitutions $\sigma, \sigma' : \mathcal{X} \to Q$ telle que :*

Alors on fusionne les états q et q' dans l'automate $A^i_{\mathcal{R},E}$. Les ensembles des substitutions σ et sigma$'$ peuvent être calculées respectivement par résolution des problèmes de filtrage $u \trianglelefteq q$ et $v \trianglelefteq q'$. Il suffit de vérifier si il existe un couple (σ, σ') tel que pour toute variable $x \in \mathcal{V}(u) \cap \mathcal{V}(v)$ $\sigma(x) = \sigma(x')$. La fusion consiste alors en un simple renommage de q en q', ou inversement.

Définition 2.4.13. *On définit $\mathtt{W}$, la fonction d'accélération paramétrée par E un ensemble d'équations, que l'on applique après chaque étape de complétion sur l'automate $A^i_{\mathcal{R}}$. Elle construit un nouvel automate $A^i_{\mathcal{R},E} = \mathtt{W}(A^i_{\mathcal{R}})$, en fusionnant tous les couples d'états de $A^i_{\mathcal{R}}$ qui peuvent l'être.*

Exemple 2.4.14. *En reprenant l'exemple précédent qui diverge, et en appliquant l'équation $x = S(S(x))$ sur l'automate A_2 par exemple (en fait on ne peut pas l'appliquer plus tôt). L'automate A_2 est composé des transitions $\{0 \to q_0,\ S(q_0) \to q_1,\ S(q_1) \to q_2\}$ et $\{q_2 \to q_1,\ q_1 \to q_0\}$. On trouve alors la substitution $x \mapsto q_0$, ce qui donne $q_0 = S(S(q_0))$ et $S(S(q_0)) \to^*_{A_2} q_2$. On fusionne donc les états q_2 et q_0, et obtient le nouvel automate A'_2 dans lequel on remplace q_2 par q_0, ce dont on déduit les transitions $\{0 \to q_0,\ S(q_0) \to q_1,\ S(q_1) \to q_0\}$ et $\{q_0 \to q_1,\ q_1 \to q_0\}$.*

On cherche les nouvelles paires critiques en q_0 et q_1 pour les règles de $\mathcal{R} = \{0 \to S0,\ S(x) \to S(S(X))\}$. on trouve $\langle q_0, S(0)\rangle$ qui est déjà résolue, puis les paires $\langle q_0, S(S(q_0))\rangle$ et $\langle q_1, S(S(q_1))\rangle$ qui sont aussi résolues : l'automate A'_2 est clos par $\mathcal{R}$, on a atteint un point fixe.

On relie alors les étapes de complétion et d'accélération en posant $A^0_{\mathcal{R},E} = A$, puis en combinant complétion et accélération $A^{i+1}_{\mathcal{R},E} = \mathtt{W}(\mathtt{C}(A^i_{\mathcal{R},E}))$, jusqu'à l'obtention d'un point fixe $A^*_{\mathcal{R},E}$, automate $\mathcal{R}$-clos.

Dans [33], il a été montré que si l'automate initial respecte la $\mathcal{R}_{/E}$-cohérence, $A^*_{\mathcal{R},E}$ est un automate $\mathcal{R}_{/E}$-coherent qui ne reconnaît pas plus de termes que les termes atteignables par réécriture de $\mathcal{R}$ modulo E.

Définition 2.4.15. *Un automate A est $\mathcal{R}_{/E}$-cohérent [33] si et seulement si pour tout état q, il existe $s \in \mathcal{R}ep(q)$ un représentant de l'état q tel que :*

- $\forall t \in \mathcal{R}ep(q)$, $s =_E t$
- $\forall t \in \mathcal{L}(A, q)$, $s \to^*_{\mathcal{R}_{/E}} t$.

Cette technique d'approximation et cette méthodologie est comparable à celle employée par les abstractions équationnelles définies dans [44].

Chapitre 3

Détection de contre-exemples et Raffinement

3.1 Fausses alarmes et contre-exemples

Dans ce chapitre, on s'intéresse aux problèmes que l'on rencontre lors de la vérification de systèmes où l'ensemble des états atteignables ne peut être calculé que par sur-approximation.

Dans la méthode de vérification basée sur la complétion d'automates d'arbres telle que décrite précédemment (*cf.* sec. 2.4), on ne s'est intéressé qu'à la correction de l'approche : dans le cas où la vérification réussit, on peut systématiquement conclure à la préservation de la propriété. En revanche on ne peut absolument rien conclure lorsque la vérification a échoué. En effet, à cause de la sur-approximation l'échec peut-être dû à une fausse alarme. Les *fausses alarmes* sont le résultat d'une approximation trop grossière qui contient des états supplémentaires qui n'appartiennent pas au système mais violent la propriété. En pratique, construire une « bonne »approximation demande un bon niveau d'expertise du système à vérifier, ce qui est d'autant plus difficile lorsque le système est complexe. Ainsi, il est rare de trouver immédiatement une approximation qui contienne l'ensemble des états du système et qui en même temps soit assez fine pour éviter les fausses alarmes. On se propose donc d'améliorer la technique de vérification pour qu'elle soit plus informative en cas d'échec. En résumé, la propriété peut être violée pour deux raisons :

- L'exécution du système peut conduire à un état (atteignable) qui ne respecte pas la propriété : l'état constitue un **contre-exemple**.
- Le système respecte la propriété mais l'approximation trop grossière introduit des états (non atteignables) qui violent la propriété : c'est une **fausse alarme**.

Comme cette méthode de vérification d'une propriété par non-atteignabilité est semi-décidable, on sait qu'il n'est généralement pas possible de distinguer un contre-exemple d'une fausse alarme, et donc de savoir si le système viole la propriété. Cependant on peut

repérer les états dont l'appartenance au système est sûre. Un tel marquage est réalisable itérativement : il sûr de considérer qu'un état appartient au système si il peut être produit à partir d'un état pour lequel on a aussi l'assurance qu'il appartient au système. Dualement, on perd l'assurance pour tout état introduit par l'approximation ou si l'état est obtenu par des transitions issues d'états pour lesquels il n'existe aucune assurance qu'ils appartiennent au système. Ainsi dans la figure 3.1, si les états noirs ont été introduits avant de considérer

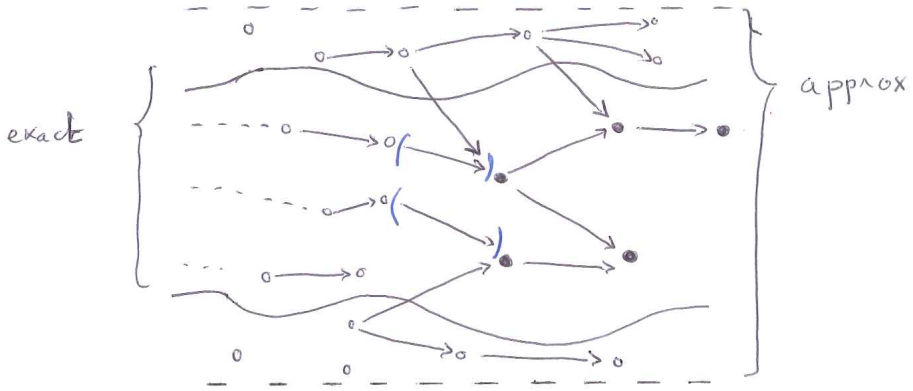


FIGURE 3.1:

les transitions parenthèsées, on ne peut être sûr que ces états et leur successeurs sont atteignables tant que ce transitions n'ont pas été considérées par l'exploration.

Cette technique permet de réduire considérablement la zone d'incertitude en cas d'échec. Lorsque la vérification de la propriété échoue à cause d'un état s dont on est sûr qu'il appartienne au système, on peut conclure que le système viole la propriété : l'état s constitue un *contre-exemple*. Dans le cas contraire on ne sait toujours rien dire, on est soit dans le cas d'une fausse alarme soit d'un contre-exemple.

3.2 Principe de l'approche à la CEGAR appliqué au model-checking régulier

Il est possible d'améliorer l'approche précédente, en exploitant l'information lorsque la vérification échoue et que l'on ne peut pas conclure à un contre-exemple ou une fausse alarme. En effet, si l'on n'est pas sûr qu'un état $s_?$ est atteignable, alors il a été introduit par une étape d'accélération. Pour cet état $s_?$, on peut envisager les deux situations suivantes :

- Soit $s_?$ n'est pas réellement atteignable : raffiner l'approximation de façon à supprimer $s_?$ est correct, puisqu'il s'agit d'une fausse alarme.
- Soit $s_?$ est atteignable : il existe au moins une exécution du système qui passe par cet état : cela signifie qu'au moment où la propriété a été violée, la trace correspondant à l'exécution n'a pas encore été considérée, mais l'état est la conséquence d'une étape d'accélération. Si même si on retire $s_?$ de l'approximation, comme il est atteignable, on finira toujours par produire le terme par des étapes de calcul exact.

On peut donc envisager de supprimer l'état $s_?$ de l'approximation, si l'on tente de compléter à nouveau l'ensemble des états atteignables. C'est le principe de l'approche à la CEGAR (*Counter-Example Guided Abstraction Refinement*) [17] : on retire successivement tous les états $s_?$ qui conduisent à la mise en échec de la propriété s'il n'est pas possible de déterminer s'ils sont des contre-exemples ou non. Chaque phase de étape de raffinement conduit alors à la reprise du calcul de l'ensemble des atteignables privé des états $s_?$. Cependant, raffiner approximation de l'ensemble des états peut entraîner une divergence du calcul puisque la raffinement affaiblit l'approximation : cela impose de raffiner avec parcimonie, pour éviter d'accroître ce risque de divergence.

Dans [11], Bouajjani et al. ont introduits les premières techniques à la CEGAR pour le model checking régulier, avant de l'étendre aux termes dans [9, 10]. Le principe de ces travaux repose sur une pile qui contient tous les automates d'arbres intermédiaires produits par le calcul depuis l'automate initial. La relation de transition est définie par T un transducteur d'automates d'arbres [18]. L'abstraction est construite en utilisant des prédicats qui paramètrent les étapes d'accélération. Les prédicats sont représentés par des automates d'arbres.

Le processus de calcul est basée sur l'alternance étape de transduction suivie d'une étape d'abstraction. Pour déterminer si un terme est un contre-exemple ou un terme suspicieux [1], l'approche repose sur un calcul arrière qui impose de construire la relation de transition inverse T^{-1}. À partir d'un ensemble X_n de termes reconnus dans l'automate M_n tels que X_n viole une certaine propriété, on cherche dans les automates précédents quels termes ont permis par T de produire les termes de X_n. Pour cela, on calcule un sur-ensemble des prédesseurs possibles, et on recherche ceux qui étaient effectivement reconnus par l'automate M_{n-1}, (calcul d'intersection) ce qui constitue l'ensemble de termes X_{n-1}. On recommence le processus jusqu'à :

- obtenir l'ensemble $X_0 \neq \emptyset$ dont l'un des termes est reconnu par l'automate initial, permet de produire l'un des termes de X_n : l'ensemble X_n contient un contre-exemple.
- obtenir $X^k = \emptyset$ pour $k \leq n$, on peut alors en déduire que les termes de X_{k+1} n'ont pas été produits par l'application du transducteur T sur l'automate M_k. Ils sont donc le résultat d'une étape d'accélération réalisée sur l'automate M_{k+1}. Donc pour éviter les termes M_n, il faut retirer les termes de M_{k+1} qui sont à l'origine de leur introduction en raffinant l'abstraction réalisée sur M_{k+1}.

Cette opération, est d'autant plus coûteuse que la propriété est violée tardivement. Plus n est grand, plus la recherche en arrière coûte cher, les opérations booléennes sur les automates d'arbres ayant un coût non négligeable.

Dans [6], Boichut et al. ont tenté une approche similaire mais en utilisant des règles de réécriture à la place des transducteurs et définissent l'abstraction des règles d'approximation directement pour l'automate. De la même manière, le processus est basé sur la sauvegarde des automates obtenus à chaque pas de calcul. Néanmoins, cette technique nécessite que les règles de réécriture soient linéaires à gauche et à droite, pour effectuer le calcul en arrière : la complétion d'automates d'arbres nécessite de modéliser la relation

1. un terme dont on ne peut déterminer si il est atteignable ou non

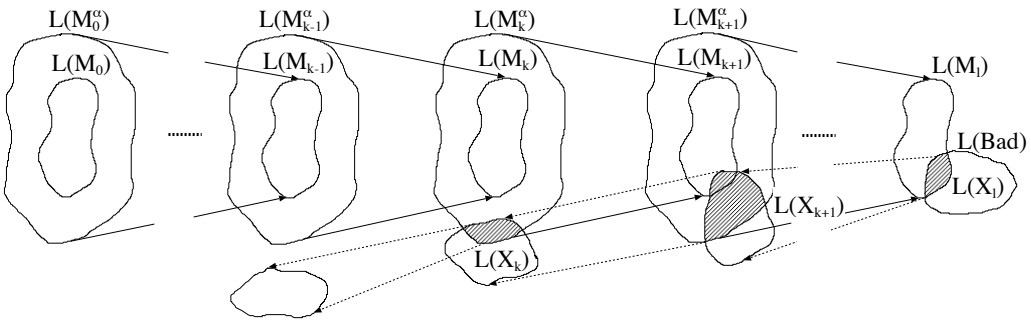


FIGURE 3.2: Le calcul en arrière en model-checking régulier (fig. extraite de [11])

de transition Rel par un système de réécriture linéaire à gauche, et le calcul arrière – lui aussi réalisé par la complétion – nécessite la relation Rel^{-1} définie en retournant chaque règle $l \to r \in \mathcal{R}$ en $r \to l \in \mathcal{R}^{-1}$, imposant aux membres droits dans $\mathcal{R}$ d'être linéaires. Si la modélisation de systèmes par un ensemble de règles de réécriture linéaires est toujours possible, elle peut s'avérer moins immédiate qu'avec des règles de réécritures linéaires à gauches.

3.3 Contre-exemples et automates d'arbres

La contribution de ce chapitre est de proposer une solution pour améliorer la complétion en cas d'échec et de proposer une procédure de raffinement à la CEGAR pour l'algorithme de complétion d'automates d'arbres sans avoir recours au calcul en arrière pour déterminer si un terme est un contre-exemple ou non. On espère ainsi gagner en performances par rapport aux techniques existantes. Pour cela, on définit des automates d'arbres instrumentés que l'on appelle $\mathcal{R}_{/E}$**-automates** et dont la sémantique nous informe sur la manière de réécrire les termes. En résumé l'automate caractérise les termes ajoutés par réécriture de $\mathcal{R}$ ou par réécriture $\mathcal{R}$ modulo E en gardant une trace des équations utilisées pendant l'accélération. On propose alors une procédure de complétion d'un $\mathcal{R}_{/E}$-automate pour $\mathcal{R}$ un système de règles linéaires à gauche et un ensemble d'équations E. L'information nécessaire étant maintenue à jour $\mathcal{R}_{/E}$-automate courant, il n'est plus nécessaire de revenir sur les calculs antérieurs pour connaître la nature d'un terme : l'exécution de l'automate pour un terme t permet d'obtenir des informations sur l'atteignabilité. L'automate constitue en lui-même la procédure de semi-décision. Pour un terme t reconnu par le $\mathcal{R}_{/E}$-automate, la sémantique informe des différentes étapes d'accélération qui ont été nécessaire à sa contruction. Lorsque l'automate indique qu'il n'y a pas eu d'abstraction pour obtenir t, le terme est atteignable par $\mathcal{R}$. Dans le cas contraire, on dispose de l'information nécessaire pour raffiner efficacement l'approximation par la procédure d'élagage du $\mathcal{R}_{/E}$-automate que l'on propose. De plus, nous verrons que la technique de raffinement développée ici est plus précise que celle présentée dans [6], ce qui lui permet converger

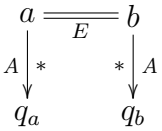
plus fréquemment. Enfin, ce travail étend les abstractions équationnelles [44, 49] pour la détection de contre-exemples et le raffinement.

3.4 L'intuition initiale

L'idée qui a conduit à la l'introduction des $\mathcal{R}_{/E}$-automates peut-être illustrée à l'aide de l'exemple 3.4.1.

Exemple 3.4.1. *Soit A un automate défini comme :*
- $Q_f = \{q_g, q_f\}$
- $\Delta = \{f(q_a) \to q_f,\ g(q_b) \to q_g,\ a \to q_a,\ b \to q_b\}$

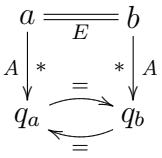
L'automate reconnaît les termes $f(a)$ et $g(b)$. On applique à cet automate l'équation $a = b$ ce qui permet de fusionner les états q_a et q_b



On obtient l'automate A' formé par les transitions $\Delta' = \{f(q_a) \to q_f,\ g(q_a) \to q_g,\ a \to q_a,\ b \to q_a\}$ où q_b est remplacé par q_a. Ainsi les termes a et b sont dans la même classe d'équivalence, ce qui a pour conséquence d'ajouter les termes $f(b) \in \mathcal{L}(A', q_f)$, et $g(a) \in \mathcal{L}(A', q_g)$.

La fusion produit bien l'effet escompté sur le langage reconnu. En revanche, lorsque se place du point de vue de l'état q_f, la fusion ne permet pas de comprendre à posteriori que $f(b)$ est obtenu par l'abstraction, alors que $f(a)$ était déjà reconnu par A. Dualement, on peut remarquer le même problème à l'état q_g avec les termes $g(b)$ et $g(a)$.

Une manière de palier à cette limitation consiste à rendre explicites les opérations de fusion. La solution retenue consiste à remplacer le renommage par des ε-transitions particulières :



Maintenant, on peut savoir si un terme a été ajouté au langage grâce à l'abstraction ou non, juste en considérant les transitions utilisées pour le reconnaître :

Exemple 3.4.2. *$f(a) \longrightarrow f(q_a) \longrightarrow q_f$ indique $f(a)$ ne vient pas de l'abstraction, $f(b) \longrightarrow f(q_b) \xrightarrow{=} f(q_a) \longrightarrow q_f$ est obtenu par l'abstraction.*

Cela est suffisant pour caractériser la relation de causalité liée aux étapes d'abstraction qui peuvent avoir lieu lors de la complétion de l'automate. C'est sur ce principe que repose les $\mathcal{R}_{/E}$-automates introduits dans la section 3.5.

3.5 Définition d'un $\mathcal{R}_{/E}$-automate

Un $\mathcal{R}_{/E}$-automate est une extension de la définition standard des automates d'arbres, dont la sémantique donne des informations sur les termes reconnus par cet automate notamment pour un problème d'atteignabilité. Elle permet en particulier de distinguer les termes pour lesquels l'atteignabilité est sûre des autres termes. Définir un $\mathcal{R}_{/E}$-automate n'a donc de sens que dans le cadre d'un problème d'atteignabilité donné $(I, \mathcal{R}, Bad)$ et d'une abstraction définie par un ensemble E d'équations. On notera que le cas particulier du calcul exact dénoté par $E = \emptyset$ est couvert par l'approche, mais ne se révèle pas intéressant puisque le problème soulevé n'existe que dans le cas d'une sur-approximation.

Soit un problème d'atteignabilité défini dans l'ensemble de termes $\mathcal{T}(\mathcal{F})$ avec I comme ensemble fini de termes initiaux $\mathcal{R}$ un ensemble de règles de réécriture et E un ensemble d'équations.

Définition 3.5.1. *Un $\mathcal{R}_{/E}$-**Automate** est un automate d'arbres défini comme un quadruplé de la forme $A = \langle \mathcal{F}, Q, Q_F, \Delta \cup \varepsilon_{\mathcal{R}} \cup \varepsilon_{=} \rangle$, avec $Q_F \subseteq Q$ et composé de 3 ensembles de transitions distinctes :*

- Δ *est un ensemble de transitions normalisées (c.f. définition 2.2.1),*
- $\varepsilon_{=}$ *est un ensemble de ε-transitions,*
- $\varepsilon_{\mathcal{R}}$ *est un ensemble de ε-transitions étiquetées par $\top$ ou par une conjonction de prédicats de la forme $Eq(q, q')$ avec $q, q' \in Q$, et $q \to q' \in \varepsilon_{=}$.*

Un $\mathcal{R}_{/E}$-automate est un automate d'arbre dans lequel les transitions sont classées en trois catégories. D'une part les transitions normalisées et deux ensembles distincts de ε-transitions :

- les premières caractérisent la relation de réécriture qui peut exister entre les classes d'équivalence de l'automate, comme les ε-transitions produites par la complétion 2.4, mais étiquetées par des formules logiques.
- les secondes définissent la relation d'abstraction qui peut exister entre deux classes d'équivalence comme illustré en section 3.4.

Exemple 3.5.2. *Soit A un $\mathcal{R}_{/E}$-automate reconnaissant dans l'état final $\mathbf{q_f}$ le système $(\mathcal{F}, I, \mathcal{R})$ défini par*

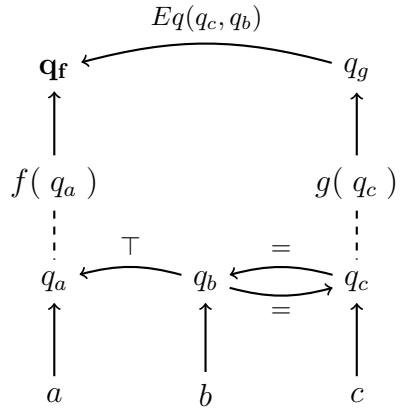
$I = f(a)$, $\mathcal{R} = \{f(c) \to g(c),\ a \to b\}$, et un ensemble d'équations $E = \{b = c\}$.

Dans A, l'égalité $b = c$ est symbolisée par les 2 transitions $q_c \to q_b$ et $q_b \to q_c$ de $\varepsilon_{=}$, en tenant compte que b et c sont respectivement reconnus dans les états q_b et q_c.

Ainsi du point de vue de l'état q_c, la transition $q_b \to q_c$ nous indique que le terme b est obtenu à partir du terme c par égalité. On peut remarquer constater que la transition $q_c \to q_b$ nous permet de conclure réciproquement que le terme c est obtenu à partir du terme b.

Les transitions $q_g \to q_f$ et $q_b \to q_a$ denotent des étapes de réécriture. En fait, elles nous permettent de déduire $f(a) \to^*_{\mathcal{R}_{/E}} g(c)$ et, $a \to^*_{\mathcal{R}_{/E}} b$. En dépliant la relation $f(a) \to^*_{\mathcal{R}_{/E}} g(c)$, on obtient la chaîne $f(a) \to_{\mathcal{R}} f(b) =_E f(c) \to_{\mathcal{R}} g(c)$, qui utilise l'égalité $b = c$ pour

passer de $g(b)$ à $g(c)$, relation que l'on peut déduire de la transition $q_c \to q_b$. Ainsi, on étiquette la transition $q_g \to q_f$ avec la formule $Eq(q_c, q_b)$ pour conserver cette information. En revanche, la transition $q_b \to q_a$ est étiquetée avec $\top$ ce qui signifie qu'aucune égalité n'est utilisée pour réécrire a en b : de cette transition, on déduit $a \to^*_{\mathcal{R}} b$.



Intuitivement, dans un $\mathcal{R}_{/E}$-automate, les termes sont reconnus en utilisant les transitions de Δ, les transitions de $\varepsilon_{\mathcal{R}}$ dénotent la relation de réécriture entre ces termes, et les transitions ε-transitions de $\varepsilon_=$ dénotent les étapes d'abstraction. Chaque transition de $\varepsilon_{\mathcal{R}}$ est étiquetée par une formule logique correspondant aux approximations réalisées pour avoir l'étape de réécriture. Plus exactement, la formule indique quelles sont les étapes d'abstraction qui ont été nécessaires pour produire le terme qui a pu être réécrit. Ainsi, lorsque la formule est $\top$, aucune approximation n'est nécessaire et le terme est atteignable simplement par réécriture, sinon le terme a été produit par réécriture modulo E. Dans [33], les auteurs établissent déjà le lien entre la réécriture $\mathcal{R}_{/E}$ et les automates produit par la complétion.

3.5.3 Sémantique

Dans la suite, on utilisera $\to_\Delta$ pour dénoter la clôture transitive et réflexive de de la relation induite par les transitions de tout ensemble Δ. Dans le cas où Δ est l'ensemble de toutes les transitions normalisées on utilisera $\to^{\not\varepsilon}$ pour $\to_\Delta$. Pour Δ un l'ensemble donné de transitions normalisées d'un $\mathcal{R}_{/E}$-automate, on défini l'ensemble des représentants d'un état comme $Rep(q) = \{t \in \mathcal{T}(\mathcal{F}) | t \to^{\not\varepsilon} q\}$. Maintenant, on défini formellement les exécutions d'un $\mathcal{R}_{/E}$-automate.

Définition 3.5.4. *On définit l'**exécution du $\mathcal{R}_{/E}$-automate** A sur le terme $t \in \mathcal{T}(\mathcal{F} \cup Q)$ reconnu en l'état q par la relation $t \xrightarrow{\alpha}_A q$ définie comme :*

- $t|_p = f(q_1, \ldots, q_n)$ *et* $f(q_1, \ldots, q_n) \to q \in \Delta$ *alors* $t \xrightarrow{\top}_A t[q]_p$
- $t|_p = q$ *et* $q \to q' \in \varepsilon_=$ *alors* $t \xrightarrow{Eq(q,q')}_A t[q']_p$
- $t|_p = q$ *et* $q \xrightarrow{\alpha} q' \in \varepsilon_{\mathcal{R}}$ *alors* $t \xrightarrow{\alpha}_A t[q']_p$
- $u \xrightarrow{\alpha}_A v$ *et* $v \xrightarrow{\alpha'}_A w$ *alors* $u \xrightarrow{\alpha \wedge \alpha'}_A w$

Une exécution $\xrightarrow{\alpha}$ abstrait une séquence de réécriture de $\rightarrow_{\mathcal{R}/E}$. Si $t \xrightarrow{\alpha} q$, alors il existe un terme $s \in Rep(q)$ tel que $s \rightarrow^*_{\mathcal{R}/E} t$. La formule α dénote le sous-ensemble de transitions de $\varepsilon_=$ utilisé pour reconnaître t dans q.

Exemple 3.5.5. *On considère le $\mathcal{R}_{/E}$-automate A de l'exemple 3.5.2, puis la transition $g(b) \xrightarrow{Eq(q_b,q_c)\wedge Eq(q_c,q_b)} q$: on sait qu'il existe une séquence de réécriture $\rightarrow_{\mathcal{R}/E}$ à partir de $f(a)$, l'unique terme de l'ensemble des représentants de $Rep(q)$, pour produire $g(b)$. La formule indique que cette séquence de réécriture utilise la relation d'équivalence induite par $b = c$ dans les deux sens (transitions $q_b \rightarrow q_c$ et $q_c \rightarrow q_b$).*

La relation $\xrightarrow{\alpha}$ correspond à la relation standard de réécriture (*c.f.* la section 2.2) d'un automate d'arbres instrumenté avec des formules logiques.

Théorème 3.5.6. $\forall t \in \mathcal{T}(\mathcal{F} \cup Q),\ q \in Q,\ t \xrightarrow{\alpha}_A q \iff t \rightarrow^*_A q$

Démonstration. La preuve est simple et peut se réaliser par induction en montrant qu'il suffit d'ignorer les formules qui sont manipulées par la définition 3.5.4, pour retrouver la relation originale $\rightarrow_A$. □

Ce théorème est rassurant puisque les $\mathcal{R}_{/E}$-automates ont la même expressivité que les automates d'arbres, ce qui permet de rester dans la classe des langages réguliers de termes. Ce point est important puisqu'on ne perd pas d'expressivité pour représenter les approximations. D'autre part, puisqu'on ne gagne pas en expressivité non plus, on évite le coût du passage à des classes de langages non réguliers comme c'est le cas par exemple pour des automates à contraintes [18].

On s'intéresse maintenant aux $\mathcal{R}_{/E}$-automates *bien-définis*. La propriété de bonne définition permet d'assurer que l'information qu'apporte les $\mathcal{R}_{/E}$-automates est correcte par rapport à la réécriture par $\mathcal{R}$ et E. La propriété de bonne définition est nécessaire pour s'assurer lors du raffinement que l'on distinguera correctement les contre-exemples des faux-positifs.

Définition 3.5.7 (Un $\mathcal{R}_{/E}$-automate bien-défini)**.** *A est un $\mathcal{R}_{/E}$-automate* bien-défini*, si :*

- *Pour tout état q de A, et tout terme v tel que $v \xrightarrow{\top}_A q$, il existe u un terme représentatif de q tel que $u \rightarrow^*_{\mathcal{R}} v$*
- *Si $q \xrightarrow{\phi} q'$ est une transition de $\varepsilon_{\mathcal{R}}$, alors il existe deux termes $s, t \in \mathcal{T}(\mathcal{F})$ tels que $s \xrightarrow{\phi}_A q$, $t \xrightarrow{\top}_A q'$ et $t \rightarrow_{\mathcal{R}} s$.*

Le premier point de la définition 3.5.7 assure que tout terme reconnu en utilisant uniquement des transitions étiquetées par la formule $\top$ est atteignable à partir de l'ensemble initial. Le second point est utilisé pour raffiner l'automate. Une séquence de réécriture $\rightarrow_{\mathcal{R}/E}$ dénotée par $q \xrightarrow{\phi} q'$ est possible grâce à certaines transitions de $\varepsilon_=$ caractérisées par ϕ. Si on supprime ces transitions de $\varepsilon_=$ de telle sorte que ϕ devienne fausse, alors la transition $q \xrightarrow{\phi} q'$ sera aussi supprimée.

A partir de cette définition, un terme t qui est reconnu en utilisant au moins une transition étiquetée avec une formule différente de $\top$ peut être enlevée du $\mathcal{R}_{/E}$-automate en supprimant des transitions de $\varepsilon_{=}$. Cette opération d'"élagage" est illustrée ci-dessous.

Exemple 3.5.8. *On reprend le $\mathcal{R}_{/E}$-automate A de l'exemple 3.5.2. Cet automate reconnaît le terme $g(c)$. En effet, d'après la définition 3.5.4, on a $g(c) \xrightarrow{Eq(q_c,q_b)} q_f$. On Considère maintenant la séquence de réécriture $f(a) \to_{\mathcal{R}} f(b) =_E f(c) \to_{\mathcal{R}} g(c)$. On peut voir que si l'étape $f(b) =_E f(c)$ dénotée par la transition $q_c \to q_b$ est supprimée, alors $g(c)$ devient non-atteignable. La première étape de l'élagage de A consiste donc à retirer cette transition. Dans un deuxième temps, on propage cette information en retirant toutes les transitions de $\varepsilon_{\mathcal{R}}$ étiquetées par une formule construite avec le prédicat $Eq(q_c, q_b)$. Cela permet de supprimer tous les termes obtenus par réécriture en utilisant l'équivalence $b =_E c$. Suite à l'élagage, on constate que A ne caractérise plus que l'ensemble des termes $\{f(a), f(b)\}$.*

3.6 Construire un $\mathcal{R}_{/E}$-Automate pour le problème d'atteignabilité

Dans cette section, on étend les principes de complétion et d'élargissement présentés dans la section 2.4 aux $\mathcal{R}_{/E}$-automates. On parlera alors de $\mathcal{R}_{/E}$**-completion**. Pour cela, on considère dans toute la section, un ensemble $\mathcal{R}$ de règles de réécriture et une approximation définie par E l'ensemble d'équations. On veut produire un $\mathcal{R}_{/E}$-automate $A^*_{\mathcal{R},E}$ clos par réécriture à partir de I, l'ensemble des termes initiaux caractérisé par l'automate $\mathcal{L}(A) = I$. Le $\mathcal{R}_{/E}$-automate $A^*_{\mathcal{R},E}$ est obtenu à partir de $A^0_{\mathcal{R},E}$ en calculant successivement les approximations $A^i_{\mathcal{R},E}$, jusqu'à obtenir un point fixe $\mathcal{L}(A^{k+1}_{\mathcal{R},E}) \supseteq \mathcal{L}(A^k_{\mathcal{R},E})$. Le $\mathcal{R}_{/E}$-automate $A^{i+1}_{\mathcal{R},E} = \mathtt{W}(\mathtt{C}(A^i_{\mathcal{R},E}))$ est $A^i_{\mathcal{R},E}$ après une phase de complétion $\mathtt{C}$ et suivie d'une phase d'accélération $\mathtt{W}$.

3.6.1 Construction de $A^0_{\mathcal{R}}$

On définit le $\mathcal{R}_{/E}$-automate $A^0_{\mathcal{R},E} = \langle \mathcal{F}, Q^0, Q_F, \Delta^0 \rangle$ tel que le langage de $A^0_{\mathcal{R},E}$ est exactement l'ensemble I. Cela revient à prendre l'automate A auquel on ajoute les deux ensembles vides pour $\varepsilon^0_{\mathcal{R}}$ et $\varepsilon^0_{=}$, ce qui est raisonnable puisque les termes de I sont obtenus sans récriture ni approximation. Par construction, $A^0_{\mathcal{R},E}$ est bien-défini.

Propriété 3.6.2. *$A^0_{\mathcal{R},E}$ est bien-défini.*

Démonstration. $A^0_{\mathcal{R},E} = \langle \mathcal{F}, Q^0, Q_f, \Delta^0 \cup \emptyset \cup \emptyset \rangle$ respecte de la définition 3.5.7, seulement si les deux points de la définition 3.5.7 sont vérifiés. On sait que $A^0_{\mathcal{R}}$ ne possède aucune ε-transitions. Tous les termes sont reconnus au seul moyen des transitions de Δ^0. Cela signifie que pour tout état q l'ensemble des termes se définit comme $\{t \in \mathcal{T}(\mathcal{F}) | t \to_{\Delta^0} q\}$. qui est égal à $Rep(q)$, l'ensemble des représentants de l'état q. On constate aussi que pour tout

terme t, $t \to_{\Delta_0} q$ implique $t \xrightarrow{\top}_{A^0_{\mathcal{R},E}} q$: le second et le troisième point de la définition 3.5.4, ne sont pas utilisés, puisqu'il n'y a aucune ε-transition dans $A^0_{\mathcal{R},E}$.

Le premier point de la définition 3.5.7 est assuré : pour tout état q, et tout terme $t \xrightarrow{\top} q$, on a $t \in Rep(q)$, et $t \to^*_{\mathcal{R}} t$ par réflexivité. En particulier, si $q \in Q_f$ est un état final, alors le terme t est un terme initial, donc atteignable.

Quant au second point de la définition 3.5.7, il est assuré par $A^0_{\mathcal{R}}$ puisque $\varepsilon^0_{\mathcal{R}}$ est vide. □

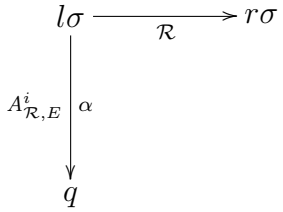
On s'intéresse maintenant à la construction de $A^{i+1}_{\mathcal{R},E}$ soit à la $\mathcal{R}_{/E}$-complétion à proprement parler, ce qui requiert la redéfinition des fonctions `C` et `W`. Cela est dû à la prise en compte des formules qui doivent être ajoutées pour chaque transition de $\varepsilon_{\mathcal{R}}$ ajoutée au $\mathcal{R}_{/E}$-automate.

3.7 L'étape de $\mathcal{R}_{/E}$-complétion `C`

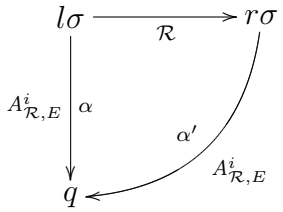
Une étape de $\mathcal{R}_{/E}$-complétion consiste à construire le $\mathcal{R}_{/E}$-automate $\mathtt{C}(A^i_{\mathcal{R},E})$ que l'on obtient à partir de $A^i_{\mathcal{R},E}$ et $\mathcal{R}$. Toute étape de $\mathcal{R}_{/E}$-complétion assure que :

- $\mathcal{L}(\mathtt{C}(A^i_{\mathcal{R},E})) \supseteq \mathcal{L}(A^i_{\mathcal{R},E})$
- $\forall t \in \mathcal{L}(A^i_{\mathcal{R},E})$, si $t \to_{\mathcal{R}} t'$ alors $t \in \mathcal{L}(\mathtt{C}(A^i_{\mathcal{R},E}))$
- si $A^i_{\mathcal{R},E}$ est bien-défini, $\mathtt{C}(A^i_{\mathcal{R},E})$ l'est aussi.

Comme expliqué précédemment dans la section 2.4, la complétion est basée sur la recherche et la résolution des paires critiques qui existent entre les transitions de l'automate et $\mathcal{R}$. Dans le cas de la $\mathcal{R}_{/E}$-completion, le principe reste le même. Cependant, on considère que les paires critique pour un $\mathcal{R}_{/E}$-automate sont caractérisées par un triplet $\langle q, r\sigma, \alpha\rangle$ où α est une formule telle que :



On considère qu'une telle paire critique n'est pas résolue si il n'existe pas de formule α' telle que le diagramme ci-dessous commute :



Pour résoudre une telle paire critique, on ajoute au $\mathcal{R}_{/E}$-automate $\mathtt{C}(A^i_{\mathcal{R},E})$ de nouvelles transitions pour avoir $r\sigma \xrightarrow{\alpha}_{\mathtt{C}(A^i_{\mathcal{R},E})} q$. Plus formellement, on peut définir la résolution d'une paire critique de la manière suivante.

Définition 3.7.1. *Résolution d'une paire critique On considère la paire critique* $p = \langle r\sigma, \alpha, q\rangle$ *qui est non résolue dans* $A^i_{\mathcal{R},E}\langle \mathcal{F}, Q, Q_f, \Delta \cup \varepsilon_{\mathcal{R}} \cup \varepsilon_{=}\rangle$. *La paire critique* p *est résolue* p *dans le* $\mathcal{R}_{/E}$*-automate* $\mathtt{C}(A^i_{\mathcal{R},E})$ *si* $\mathtt{C}(A^i_{\mathcal{R},E}) = \langle \mathcal{F}, Q', Q_f, \Delta' \cup \varepsilon'_{\mathcal{R}} \cup \varepsilon_{=}\rangle$ *avec :*

- $\Delta' \supseteq \Delta \cup \mathtt{Norm}(r\sigma, \Delta \setminus \Delta^0)$
- $\varepsilon'_{\mathcal{R}} \supseteq \varepsilon_{\mathcal{R}} \cup \{q' \xrightarrow{\alpha} q\}$ *avec* q' *un état tel que* $r\sigma \rightarrow^*_{\Delta'} q'$
- $Q' \supseteq Q$ *contient les nouveaux états de* Δ' *produits par la normalisation.*

On ajoute alors les transitions telles qu'il existe un état q' avec $r\sigma \xrightarrow{\top}_{\mathtt{C}(A^i_{\mathcal{R},E})} q'$ et $q' \xrightarrow{\alpha} q \in \varepsilon'_{\mathcal{R}}$.

On remarque que Δ^0, l'ensemble des transitions initiales $A^0_{\mathcal{R}}$, n'est jamais utilisé pour la normalisation de $r\sigma$. Ceci est nécessaire pour que la normalisation construise un ensemble de transitions qui préserve la bonne définition du $\mathcal{R}_{/E}$-automate. En effet, on a doit imposer certaines contraintes aux transitions normalisées. Or ses contraintes sont trop restrictives pour $A^0_{\mathcal{R},E}$, qui ne pourrait plus représenter des ensembles infinis de termes initiaux. Donc on exclut les transitions de Δ^0 pour la normalisation. Le $\mathcal{R}_{/E}$-automate $\mathtt{C}(A^i_{\mathcal{R},E})$ est obtenu en itérant ce principe de résolution à toutes les paires critiques p qui existent entre $\mathcal{R}$ et $A^i_{\mathcal{R},E}$. Cet ensemble de paires critiques peut être découvert en résolvant le *problème de filtrage* $l \trianglelefteq q$ pour chaque règle de réécriture $l \rightarrow r \in \mathcal{R}$ et chaque état $q \in A^i_{\mathcal{R},E}$. L'algorithme de filtrage est détaillé plus tard dans la section 3.9.

3.8 La normalisation

Comme dans le cas de la complétion, il n'est pas possible d'ajouter directement la transition $r\sigma \xrightarrow{\top}_{\mathtt{C}(A^i_{\mathcal{R},E})} q'$, elle peut ne pas être normalisée. Une étape importante du calcul passe par la normalisation, qui assure notamment que la bonne-définition du futur $\mathcal{R}_{/E}$-automate $\mathtt{C}(A^i_{\mathcal{R},E})$. On s'intéresse donc tout d'abord à la normalisation et aux propriétés qu'elle doit assurer.

Définition 3.8.1 (Normalisation)**.** *La normalisation est réalisée en 2 étapes mutuellement récursives paramétrées par la configuration* c *à reconnaître, et par* Δ *un ensemble de transitions à étendre.*

$$\begin{cases} \mathtt{Norm}(c, \Delta) = \mathtt{Slice}(d, \Delta) & c \rightarrow^!_{\Delta} d,\ \text{et } c,\ d \in \mathcal{T}(\mathcal{F} \cup Q) \\ \mathtt{Slice}(q, \Delta) = \Delta & q \in Q \\ \mathtt{Slice}(f(q_1, \ldots, q_n), \Delta) = \Delta \cup \{f(q_1, \ldots, q_n) \rightarrow q\} & q_i \in Q \text{ et } q \in Q_{new} \\ \mathtt{Slice}(f(t_1, \ldots, t_n), \Delta) = \mathtt{Norm}(f(t_1, \ldots, t_n), \mathtt{Slice}(t_i, \Delta)) & t_i \in \mathcal{T}(\mathcal{F} \cup Q) \setminus Q \end{cases}$$

Pour simplifier la définition, on ne manipule pas explicitement la mise à jour de l'ensemble des états Q de l'automate qui s'agrandit à chaque ajout de transitions. D'autre part, l'ensemble des nouveaux états Q_{new} est toujours considéré comme disjoint de Q, y compris lors de l'accroissement de Q.

La définition de la normalisation est similaire à celle introduite dans la complétion d'automates d'arbres. Cependant, l'utilisation et les propriétés que l'on en attend sont différentes.

Lorsque l'on normalise, on suppose que Δ – le second argument de `Norm` – est **déterministe**.

Définition 3.8.2. *L'ensemble de transitions Δ est **déterministe**, si pour toutes transitions normalisées de la forme $f(q_1, \ldots, q_n) \to q$ et $f(q_1, \ldots, q_n) \to q'$, alors on a $q = q'$.*

Comme la relation de réécriture induite par Δ est bien-fondée, imposer le déterminisme à Δ permet d'assurer à tout terme $s \in \mathcal{T}(\mathcal{F} \cup Q)$ de posséder une **unique forme normale** t ce que l'on dénotera par la relation $s \to_\Delta^! t$. Cette propriété requise par la première étape de `Norm` est aussi préservée par le calcul. Lors de la première étape de complétion pour normaliser la première transition ajoutée au $\mathcal{R}_{/E}$-automate $A^0_{\mathcal{R},E}$, on prend $\Delta = \Delta^0 \setminus \Delta^0$ qui est égal à l'ensemble vide, et est trivialement déterministe.

Définition 3.8.3. *On définit Δ comme **injectif**, si pour tout état q, il existe une seule transition de la forme $f(q_1, \ldots, q_n) \to q$.*

Cette propriété d'injectivité est importante pour assurer le premier point de la bonne-définition d'un $\mathcal{R}_{/E}$-automate. L'exemple 3.8.4 illustre le problème posé lorsque cette contrainte n'est pas respectée. Comme pour le déterminisme, l'injectivité est une propriété préservée par la fonction `Norm`. De plus, pour $A^0_{\mathcal{R},E}$, l'ensemble $\Delta = \emptyset$ est évidemment injectif.

Exemple 3.8.4. *Soit le $\mathcal{R}_{/E}$-automate $A^0_{\mathcal{R},E}$ défini par l'ensemble de transitions $\Delta^0 = \{a \to q,\ f(q) \to q,\ g(q) \to q_g\}$ et d'état final q_g. Le $\mathcal{R}_{/E}$-automate $A^0_{\mathcal{R},E}$ reconnaît un ensemble infini de termes initiaux grâce à Δ^0, en contrepartie Δ^0 n'est pas injectif. On considère la règle de réécriture $g(x) \to h(f(x))$. On va considérer la normalisation avec et sans les transitions de Δ^0. On résoud la paire critique $\langle q_g,\ h(f(q)),\ \top \rangle$ par les transitions $h(f(q)) \xrightarrow{\top} q'$ et $q' \xrightarrow{\top} q$. On tente de calculer $\texttt{Norm}(h(f(q)), \Delta^0)$ alors que Δ^0 n'est clairement pas injectif en q :*

$$\texttt{Norm}(h(f(q))), \Delta^0) = \texttt{Slice}(h(q), \Delta^0) = \Delta^0 \cup h(q) \to q'$$

On obtient alors le nouveau $\mathcal{R}_{/E}$-automate $A^1_{\mathcal{R},E}$ tel que

$$\Delta^1 = \{a \to q,\ f(q) \to q,\ g(q) \to q_g,\ h(q) \to q'\} \text{ et } \varepsilon^1_{\mathcal{R}} = \{q' \to q_g\}$$

Pour le terme $h(a)$, on peut voir que $h(a) \xrightarrow{\top} q_g$, ce qui signifie que $f(a)$ atteignable à partir de $\mathcal{L}(A^0_{\mathcal{R},E})$ par réécriture alors qu'il ne l'est clairement pas. Maintenant si on normalise sans tenir sans les transitions de Δ^0 :

$\texttt{Norm}(h(f(q))), \emptyset) = \texttt{Slice}(h(f(q)), \emptyset) = \texttt{Norm}(h(f(q)), \texttt{Slice}(f(q), \emptyset))$

$\texttt{Slice}(f(q), \emptyset) = \{f(q) \to q_f\}$

$\texttt{Norm}(h(f(q)), \{f(q) \to q_f\}) = \texttt{Slice}(h(q_f), \{f(q) \to q_f\}) = \{f(q) \to q_f,\ h(q_f) \to q'\}$

Dans ce cas, on a $A^1_{\mathcal{R},E}$ *de la forme*

$$\Delta^1 = \{a \to q,\ f(q) \to q,\ g(q) \to q_g,\ f(q) \to q_f,\ h(q_f) \to q'\} \text{ et } \varepsilon^1_{\mathcal{R}} = \{q' \to q_g\}$$

Alors on peut voir que tous les termes reconnus en q' *sont de la forme* $h(f(x))$ *et sont atteignables.*

Remarque 3.8.5. *Le calcul de la fonction* $\texttt{Norm}(c, \Delta)$ *termine. Pour caractériser la terminaison, on utilise une mesure* $\mu : \mathcal{T}(\mathcal{F} \cup Q) \to \mathbb{N}$ *qui compte le nombre d'occurrences de symboles de* $\mathcal{F}$ *de la configuration* c*. Exemple :* $\mu(f(q_1, g(q_2), a)) = 3$*. Toutes les preuves présentées dans la suite sont basées sur la décroissance stricte de* $\mu(c)$ *à chaque appel de* `Norm` *ou de* `Slice`*. On définit inductivement cette mesure par :*

- $\mu(q) = 0$ *si* $q \in Q$,
- $\mu(f(t_1, \ldots, t_n)) = 1 + \sum_1^n \mu(t_i)$.

Propriétés sur la normalisation

Propriété 3.8.6. *On pose* $A = \langle \mathcal{F}, Q, Q_f, \Delta \cup \varepsilon_{\mathcal{R}} \cup \varepsilon_= \rangle$*, un* $\mathcal{R}_{/E}$*-automate tel que* $\Delta \setminus \Delta^0$ *détermine l'ensemble déterministe de ses transitions, et* $c \in \mathcal{T}(\mathcal{F} \cup Q)$ *une configuration. Si on a* $\Delta' = \texttt{Norm}(c, \Delta \setminus \Delta^0)$*, alors il existe un état* q *tel que* $c \to^!_{\Delta'} q$.

Démonstration. On définit $A = \langle \mathcal{F}, Q, Q_f, \Delta \cup \varepsilon_{\mathcal{R}} \cup \varepsilon_= \rangle$ et $c \in \mathcal{T}(\mathcal{F} \cup Q)$. On suppose également que $\Delta^1 = \Delta \setminus \Delta^0$ est déterministe.

La première étape de calcul $\texttt{Norm}(c, \Delta^1)$ consiste à réécrire c par Δ^1 en sa forme normale notée d. La seconde étape $\texttt{Slice}(d, \Delta^1)$ doit retourner Δ^2 sur-ensemble de Δ^1 tel qu'il existe un unique état q qui soit la forme normale de $d \to^!_{\Delta^2} q$, et donc par conséquent la nouvelle forme normale de c. La preuve est construite par induction sur la décroissance de $\mu(d)$. On considère les 3 cas de $\texttt{Slice}(d, \Delta^1)$ d'après la définition 3.8.1 :

1. $\texttt{Slice}(q, \Delta^1) = \Delta^1$. Cela signifie que d est déjà un état q. Il n'y a rien à faire, q est déjà la forme normale attendue, et $\Delta^2 = \Delta^1$ est déterministe.
2. $\texttt{Slice}(f(q_1, \ldots, q_n), \Delta^1) = \Delta^1 \cup \{f(q_1, \ldots, q_n) \to q \mid q \in Q_{new}\}$. Chaque q_i est un état. La configuration $f(q_1, \ldots, q_n)$ peut être utilisé pour former le membre gauche d'une transition normalisée. On construit la nouvelle transition $f(q_1, \ldots, q_n) \to q$ en utilisant un nouvel état $q \in Q_{new}$. Le déterminisme est automatiquement préservé par l'ajout de cette transition à Δ^1. On sait en effet que $d = f(q_1, \ldots, q_n)$ est en forme normale, donc impossible à réécrire davantage par les transitions de Δ^1 : la nouvelle transition $f(q_1, \ldots, q_n) \to q$ est bien l'unique moyen de réécrire d. On a donc construit $\Delta^2 = \Delta^1 \cup \{f(q_1, \ldots, q_n) \to q \mid q \in Q_{new}\}$ qui est déterministe, et $d \to^!_{\Delta^2} q$.
3. $\texttt{Slice}(f(t_1, \ldots, t_n), \Delta^1) = \texttt{Norm}(f(t_1, \ldots, t_n), \texttt{Slice}(t_i, \Delta^1)\ ),\ t_i \in \mathcal{T}(\mathcal{F} \cup Q) \setminus Q$. Dans ce cas, on a choisi t_i un sous-terme immédiat de d qui n'est pas un état. Il en existe au moins un, sinon on est dans le cas précédent. Comme t_i est un sous-terme de d on peut en déduire $\mu(t_i) < \mu(d)$ d'après la définition de μ. Par induction, Δ^1 est étendu par $\texttt{Slice}(t_i, \Delta^1)$ pour obtenir Δ^3 déterministe pour lequel il existe

un état q tel que $t_i \to^!_{\Delta^3} q$. En utilisant ce nouvel ensemble de transitions Δ^3, on déplie $\texttt{Norm}(f(t_1,\ldots,t_n),\Delta^3)$ qui consiste dans un premier temps à réécrire le terme $f(t_1,\ldots,t_n)$ par Δ^3. On obtient une nouvelle configuration $f(t'_1,\ldots,t'_n)$ où l'on sait au moins que le sous-terme t'_i est égal à q puisque le sous-terme t_i peut être réécrit en q par Δ^3. On peut remarquer aussi que si certains des sous-termes de t_i sont communs aux autres t_j (les autres sous-termes immédiats de d), chaque t_j sera aussi réécrits par Δ^3 pour produire t'_j la forme normale correspondante. Chaque étape de réécriture par Δ^3 remplace un symbole de $\mathcal{F}$ par un état de Q par définition de la transition normalisée. Cette remarque permet de montrer $\mu(f(t_1,\ldots,t_n) > \mu(f(t'_1,\ldots,t'_n)$. Pour le sous-terme immédiat t_i, on sait que $\mu(t_i) > 0$ (t_i n'est pas un état), et $\mu(t'_i) = 0$ (t'_i est l'état q). Pour les autres sous-termes immédiats t_j avec $j <> i$, on sait seulement $\mu(t_j) \geq \mu(t'_j)$ puisque $t_j \to^!_{\Delta^3} t'_j$ il peut ne pas y avoir eu réécriture nécessairement et donc $t_j = t'j$. On a $\mu(f(t_1,\ldots,t_n) > \mu(f(t'_1,\ldots,t'_n)$ par définition de μ, et $f(t'_1,\ldots,t'_n)$ est réécrit en sa forme normale par l'ensemble Δ^3 déterministe par hypothèse d'induction. Alors, on peut utiliser à nouveau l'hypothèse d'induction pour déduire que $\Delta^2 = \texttt{Slice}(f(t'_1,\ldots,t'_n),\Delta^3)$ étend Δ^2 de façon à avoir l'état q comme forme normale de $f(t'_1,\ldots,t'_n) \to^!_{\Delta^2} q$. Par transitivité, on a $d \to^!_{\Delta^2} q$ en utilisant l'ensemble déterministe Δ^2 pour réécrire d qui est égal à $f(t_1,\ldots,t_n)$.

Finalement, on a montré que $\Delta^2 = \texttt{Slice}(d,\Delta^1)$ étend Δ^1 en préservant le déterminisme tel que il existe un état q pour lequel $d \to^!_{\Delta^2} q$. On sait aussi que $c \to^!_{\Delta^2} d$. On peut conclure que $\Delta^2 = \texttt{Norm}(c,\Delta^1)$ est déterministe, et qu'il existe un état q tel que $c \to^!_{\Delta^2} q$. □

Propriété 3.8.7 (Normalisation et injectivité)**.** *Si Δ^1 est injectif, alors $\texttt{Norm}(c \mid \Delta^1)$ l'est aussi.*

Démonstration. On pose le $\mathcal{R}_{/E}$-automate $A = \langle \mathcal{F}, Q, Q_f, \Delta \cup \varepsilon_{\mathcal{R}} \cup \varepsilon_{=} \rangle$, la configuration $c \in \mathcal{T}(\mathcal{F} \cup Q)$, et l'ensemble de transitions normalisées $\Delta^1 = \Delta \setminus \Delta^0$ qui est injectif.

En fait, d'après la définition de $\texttt{Norm}(c,\Delta^1)$, l'ajout des transitions est uniquement réalisé lorsque l'on se trouve dans le cas $\texttt{Slice}(f(q_1,\ldots,q_n),\Delta^1)$. La preuve se fait par induction comme pour la propriété précédente, à partir de l'étude de cas sur $\texttt{Slice}(d,\Delta^1)$:

1. $\texttt{Slice}(q,\Delta^1) = \Delta^1$. Cela signifie que d est déjà un état q. L'injectivité est préservée.
2. $\texttt{Slice}(f(q_1,\ldots,q_n),\Delta^1) = \Delta^1 \cup \{f(q_1,\ldots,q_n) \to q \mid q \in Q_{new}\}$. On utilise un nouvel état $q' \in Q_{new}$, ce qui signifie que l'état n'est présent dans aucune transition de Δ^1. La transition $f(q_1,\ldots,q_n) \to q'$ est donc l'unique transition de la forme $c \to q'$, donc l'ensemble $\Delta^1 \cup \{c \to q'\}$ est injectif.
3. $\texttt{Slice}(f(t_1,\ldots,t_n),\Delta^1) = \texttt{Norm}(f(t_1,\ldots,t_n),\texttt{Slice}(t_i,\Delta^1)\ ),\ t_i \in \mathcal{T}(\mathcal{F} \cup Q) \setminus Q$ Ici, on utilise l'hypothèse d'induction pour $\texttt{Slice}(t_i,\Delta^1)$ puisque $\mu(t_i) < \mu(f(t_1,\ldots,t_n))$. Ce qui signifie que l'ensemble $\Delta' = \texttt{Slice}(t_i,\Delta^1)$ calculé est injectif. On regarde ensuite $\texttt{Norm}(f(t_1,\ldots,t_n),\Delta')$. On sait d'après la propriété précédente, qu'il existe un état q_i tel que $t_i \to^*_{\Delta'} q_i$. On a alors $\texttt{Norm}(f(t_1,\ldots,t_n),\Delta') = \texttt{Slice}(t,\Delta')$ avec t le terme $f(t_1,\ldots,t_n)$ réécrit par Δ'. On a $\mu(t) < \mu(f(t_1,\ldots,t_n))$, donc en appliquant à nouveau l'hypothèse d'induction pour $\texttt{Slice}(t,\Delta^1)$ on obtient bien l'ensemble final Δ^2 qui est injectif.

□

De cette propriété, on peut alors en déduire la propriété suivante :

Propriété 3.8.8. *Si la transition $r\sigma \to_{\Delta^2} q'$ résulte de la normalisation $\Delta^2 = \texttt{Norm}(r\sigma, \Delta^1)$ avec Δ^1 injectif, alors tous les représentants de l'état q' sont de la forme $r\tau$, avec $\tau : \mathcal{X} \to \mathcal{T}(\mathcal{F})$.*

Démonstration. On pose le $\mathcal{R}_{/E}$-automate $A = \langle \mathcal{F}, Q, Q_f, \Delta \cup \varepsilon_{\mathcal{R}} \cup \varepsilon_{=} \rangle$, tel que l'ensemble de transitions normalisées $\Delta^1 = \Delta \setminus \Delta^0$ qui est injectif.

On considère que la transition $r\sigma \to_{\Delta^2} q'$ résulte de la normalisation $\Delta^2 = \texttt{Norm}(r\sigma, \Delta^1)$. En fait pour toutes les transitions $c \to q'$ de Δ^2, l'état q' a toujours été introduit par la normalisation(*c.f.* le cas 2 de `Slice` dans la définition 3.8.1). Cette remarque implique qu'il n'est pas possible de réécrire $r\sigma$ en q' au moyen de Δ^0 puisque cela contredirait la remarque. Donc la relation $r\sigma \to_{\Delta^2 \cup \Delta^0} q'$ implique nécessairement $r\sigma \to_{\Delta^2} q'$. D'autre part, on définit les représentants $\mathcal{R}ep(q')$ comme l'ensemble des termes $t \in \mathcal{T}(\mathcal{F})$ tels que $t \to^*_{\Delta^2 \cup \Delta^0} q'$. On peut alors en déduire que les termes sont de la forme $r\tau$ avec $\tau : \mathcal{X} \to \mathcal{T}(\mathcal{F})$. En effet, on a $r\tau \to^*_{\Delta^2 \cup \Delta^0} q'$ qui se décompose en $r\tau \to^*_{\Delta^2 \cup \Delta^0} r\sigma$ et $r\sigma \to^*_{\Delta^2 \cup \Delta^0} q'$. D'après la remarque, on a nécessairement $r\sigma \to^*_{\Delta^2} q'$. Comme Δ^2 est injectif, si l'un des représentants t est réductible en q' sans passer par la configuration $r\sigma$, alors il utilise une réduction différente.

On pose $t = r'\sigma$ tel que le terme r' diffère du terme r à la position $p \in \mathcal{P}os(r)$. Par induction sur la décroissance de la longueur de la position p, on montre que les termes sont nécessairement reconnus dans des états différents :

- Soit $p = \epsilon$, dans ce cas les termes diffèrent dès la racine. On a donc $r'|_p \to^*_{\Delta^2} q_p$ et $r|_p \to^*_{\Delta^2} q'_p$. Comme r et r' diffèrent à leur racine, il existe deux transitions $f(q_1, \ldots, q_n) \to q_p$ et $g(q_1, \ldots, q_m) \to q'_p$ de Δ^2. Quelle que soit la position, les états q_p et q'_p sont différents sinon Δ^2 n'est pas injectif.
- On examine la position $i.p$. D'après l'hypothèse d'induction, les termes $r|_{i.p}$ et $r'|_{i.p}$ diffèrent, donc ils sont reconnus respectivement dans les états q et q' qui diffèrent. En considérant les sous-termes à la position p, ils ne diffèrent que par leur sous-terme à la position i : on a donc $r|_p = f(t_1, \ldots, r|_{i.p}, \ldots, t_n)$ et $r'|_p = f(t_1, \ldots, r'|_{i.p}, \ldots, t_n)$. En utilisant Δ^2, on réduit chacun des termes en $f(q_1, \ldots, q, \ldots, q_n)$ et $f(q_1, \ldots, q', \ldots, q_n)$ respectivement. Pour respecter, l'injectivité de Δ^2, les deux configurations doivent être envoyer dans deux états différents.

Ce qui permet de conclure que tous les représentants reconnus en q' sont bien de la forme $r\tau$. □

La dernière propriété concerne les nouveaux états introduits par la normalisation : pour chacun d'eux, le premier point de la bonne-définition est respectée.

Propriété 3.8.9. *Si Q' est l'ensemble des nouveaux états introduits par la normalisation alors pour tout état de $q' \in Q'$, on a pour tout terme v, si $v \xrightarrow{\top} q'$ alors il existe un représentant $u \in \mathcal{R}ep(q')$ tel que $u \to^*_{\mathcal{R}} v$.*

Démonstration. On se place dans le cas d'un $\mathcal{R}_{/E}$-automate $A = \langle \mathcal{F}, Q, Q_f, \Delta \cup \varepsilon_{\mathcal{R}} \cup \varepsilon_{=} \rangle$, et d'un système de réécriture $\mathcal{R}$, et qui est bien-défini. On montre que pour tout nouvel état introduit q' par le calcul de $\texttt{Norm}(c, \Delta^1)$ tel que $\Delta^1 = \Delta \setminus \Delta^0$ pour tout terme $v \xrightarrow{\top} q'$, il existe un représentant $u \in \mathcal{R}ep(q')$ tel que $u \to^*_{\mathcal{R}} v$.

A partir de $\texttt{Norm}(c, \Delta^1)$, on a $\Delta^2 = \texttt{Slice}(d, \Delta^1)$ tel que $c \to^!_{\Delta^1} d$. On considère alors tous les termes clos tels que $t \xrightarrow{\top}_A d$. Comme t est réécrit par les transitions l'automate A, les états qui consituent la configuration d sont nécessairement des états de l'ensemble Q, pour lesquels la propriété est vérifiée.

La preuve se fait par induction, à partir de l'étude de cas sur $\texttt{Slice}(d, \Delta^1)$.

- $\texttt{Slice}(q, \Delta^1) = \Delta^1$. Comme $d \in Q$, la propriété est vérifiée.
- $\texttt{Slice}(f(q_1, \ldots, q_n), \Delta^1) = \Delta^1 \cup \{f(q_1, \ldots, q_n) \to q \mid q \in Q_{new}\}$. Pour chaque état q_i, la propriété est vérifiée : si $t_i \xrightarrow{\top}_A q_i$ alors il existe $u_i \in Rep(q_i)$ tel que $u_i \to^*_{\mathcal{R}} t_i$. Grâce à la nouvelle transition, on obtient le terme $t = f(t_1, \ldots, t_n) \xrightarrow{\top}_A q$ par composition. De la même manière, on peut définir le terme $u = f(u_1, \ldots, u_n)$ qui est un représentant du nouvel état q. De plus, et par réécriture successive des différents u_i en t_i, on a bien $u \to^*_{\mathcal{R}} t$.
- $\texttt{Slice}(f(t_1, \ldots, t_n), \Delta^1) = \texttt{Norm}(f(t_1, \ldots, t_n), \texttt{Slice}(t_i, \Delta^1)\ ),\ t_i \in \mathcal{T}(\mathcal{F} \cup Q) \setminus Q$ Ici, on utilise l'hypothèse d'induction pour $\texttt{Slice}(t_i, \Delta^1)$ puisque $\mu(t_i) < \mu(f(t_1, \ldots, t_n))$. Ce qui signifie que l'ensemble $\Delta' = \texttt{Slice}(t_i, \Delta^1)$, et pour les nouveaux états q introduits par $\texttt{Slice}(t_i, \Delta^1)$ la propriétés est vérifiée. On considère $\texttt{Norm}(f(t_1, \ldots, t_n), \Delta')$. Comme il existe un état q_i tel que $t_i \to^*_{\Delta'} q_i$. on peut réécrire $f(t_1, \ldots, t_n)$ en t par Δ' pour lequel on a $\mu(t) < \mu(f(t_1, \ldots, t_n))$. Comme le calcul donne $\texttt{Norm}(f(t_1, \ldots, t_n), \Delta') = \texttt{Slice}(t, \Delta')$, d'après l'hypothèse d'induction pour $\texttt{Slice}(t_i, \Delta^1)$ on obtient bien l'ensemble final Δ^2 injectif.

□

3.9 Résoudre le problème du filtrage

L'algorithme est assez similaire à celui de la complétion, bien qu'il associe en plus à chaque substitution trouvée la formule (conjonction) permettant de caractériser la paire critique. Résoudre le problème de filtrage $l \trianglelefteq q$ revient donc à calculer S, l'ensemble de tous les couples (α, σ) tels que α est une formule, σ est une substitution $\mathcal{X} \mapsto Q^i$, et $l\sigma \xrightarrow{\alpha} q$. Chaque couple (α, σ) dénote donc une paire critique $\langle q, r\sigma, \alpha \rangle$.

La définition 3.9.1 introduit l'algorithme de filtrage pour calculer S solution de $l \trianglelefteq q$ le problème donné, ce que l'on note $l \trianglelefteq q \vdash_{A^i_{\mathcal{R},E}} S$. Lorsque S est vide, on remarque que cela signifie qu'aucun des termes reconnus par q ne peut être réécrit par la règle $l \to r$.

Définition 3.9.1 (Algorithme de filtrage)**.** *En posant le problème de filtrage $l \trianglelefteq q$ pour le $\mathcal{R}_{/E}$-automate $A^i_{\mathcal{R},E}$, S est la solution du problème, si il existe une dérivation pour $l \trianglelefteq q \vdash_{A^i_{\mathcal{R},E}} S$ en utilisant les règles suivantes :*

$$\textit{(Var)}\ \frac{}{x \trianglelefteq q \vdash_A \{(\alpha_k, \{x \mapsto q_k\}) \mid q_k \xrightarrow{\alpha_k}_A q\}} (x \in \mathcal{X})$$

$$\textit{(Delta)}\ \frac{t_1 \trianglelefteq q_1 \vdash_A S_1 \quad \dots \quad t_n \trianglelefteq q_n \vdash_A S_n}{f(t_1,\dots,t_n) \lhd q \vdash_A \bigotimes_1^n S_k}\ (\ f(q_1,\dots,q_n) \to q \in \Delta\ )$$

$$\textit{(Epsilon)}\ \frac{t \lhd q \vdash_A S_0 \quad t \lhd q_1 \vdash_A S_1 \quad \dots \quad t \lhd q_n \vdash_A S_n}{t \trianglelefteq q \vdash_A S_0 \cup \bigcup_{k=1}^n \{(\phi \wedge \alpha_k, \sigma) \mid (\phi,\sigma) \in S_k\}} \begin{pmatrix} \{(q_k,\alpha_k) \mid q_k \xrightarrow{\alpha_k} q\}_1^n \\ t \notin \mathcal{X} \end{pmatrix}$$

On définit l'opérateur ensembliste $\bigotimes$ comme un opérateur qui construit tous les couples formule-substitution à partir du produit cartésien de n ensembles de couples formule-substitution.

$$\bigotimes_1^n S_j = \{(\top, id) \oplus (\phi_1,\sigma_1) \oplus \dots \oplus (\phi_n,\sigma_n) \mid (\phi_j,\sigma_j) \in S_j\}$$

Il est définit à partir de l'opérateur $\oplus$ qui définit la réunion des couples formule-substitution par $(\phi,\sigma) \oplus (\phi',\sigma') = (\phi \wedge \phi', \sigma \cup \sigma')$.

En fait, cet algorithme ne contient que deux règles de dérivation. Par soucis de clarté, l'une des règles est séparée en deux : la règle (Delta) ne peut être suivie que de la règle (Epsilon) à cause du symbole $\lhd$. On remarque aussi que l'opérateur $\oplus$ utilise l'union de substitutions, ce qui n'a de sens que lorsque les domaines des substitutions sont bien disjoints. C'est la raison pour laquelle cet algorithme ne résoud le problème $l \trianglelefteq q$ que si l est linéaire.

3.9.2 Terminaison et précision de l'algorithme

On observe que, par définition, l'algorithme de filtrage considère possiblement une infinité d'éxecution de la forme $l\sigma \xrightarrow{\alpha} q$. En effet, les ε-transitions de $\varepsilon_{\mathcal{R}}^i \cup \varepsilon_{=}^i$ peuvent former des boucles. Cela sous-entend que l'on exclut les boucles pour que l'algorithme fonctionne. En fait, retirer les chemins basés sur des boucles n'enlève pas d'information critique, c'est même le contraire. Compte-tenu de l'information sémantique que donnent les ε-transitions, on constate que les boucles ajoutent de la redondance. En effet, si un terme est reconnu dans un certain état q sans ε-transitions, alors on sait que le terme est immédiatement atteignable du point de vue de q. Ajouter une boucle composée de transition de $\varepsilon_{\mathcal{R}}$ ou $\varepsilon_{=}$ indique simplement que le terme est accessible par étape de réécriture ou d'égalité à partir de lui-même ! Le principe peut être illustré, à partir du $\mathcal{R}_{/E}$-automate A de l'exemple 3.5.2. On peut remarquer que $f(b) \xrightarrow{Eq(q_b,q_c) \wedge Eq(q_c,q_b)}_A q_f$ utilise la boucle qui correspond à $f(b) =_E f(c) =_E f(b)$. Cette boucle est d'autant plus inutile que l'on préfère déduire que $f(a) \to_{\mathcal{R}}^* f(b)$ qui correspond l'exécution $f(b) \xrightarrow{\top}_A q_f$ sans boucle.

3.9.3 Propriétés de l'algorithme de Filtrage

On montre que l'on calcule bien l'ensemble des couples (α,σ) correspondant à une paire critique pour le problème $l \trianglelefteq q$ dans l'automate $A^i_{\mathcal{R},E}$.

Propriété 3.9.4 (Complétude de l'algorithme de filtrage)**.** *Soient un $\mathcal{R}_{/E}-$automate A, q un de ses états, $l \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ le membre gauche et linéaire d'une règle de réécriture et σ une Q-substitution avec un domaine restreint à $\mathcal{V}(l)$. Si S est l'ensemble obtenu par résoution du problème de filtrage, alors on a pour tout (α, σ) tel que $l\sigma \xrightarrow{\alpha}_A q$ est une exécution sans boucle, alors on a $(\alpha, \sigma) \in S$.*

Démonstration. On définit $A = \langle \mathcal{F}, Q, Q_f, \Delta \cup \varepsilon_{\mathcal{R}} \cup \varepsilon_{=} \rangle$; $l \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ et $q \in Q$; $\sigma : \mathcal{V}ar(l) \to Q$ et $\alpha = \bigwedge_1^n Eq(q_k, q'_k)$ tel que $l\sigma \xrightarrow{\alpha}_A q$.

La preuve est faite par induction structurelle sur le terme l.

- l est une variable. Dans ce cas, σ doit être une Q-substitution de la forme $\sigma = \{l \mapsto q'\}$. En utilisant cette observation et l'hypothèse, on a $q' \xrightarrow{\alpha}_A q$. Le problème de filtrage $l \trianglelefteq q$ est résolu en utilisant la règle (Var). Cela signifie que $S = \{(\alpha_k, \{l \mapsto q_k\}) \mid q_k \xrightarrow{\alpha_k}_A q\}$. Par définition de S on voit que S contient (α, σ).
- on suppose maintenant que l'on a l qui est un terme linéaire de la forme $f(t_1, \ldots, t_n)$. On va alors décomposer $f(t_1, \ldots, t_n)\sigma \xrightarrow{\alpha}_A q$ en séquence de transitions. Premièrement, on va découper σ en plusieurs Q-substitutions $\sigma_1 \ldots \sigma_n$, telles que l'on ait l'égalité $f(t_1, \ldots, t_n)\sigma = f(t_1\sigma_1, \ldots, t_n\sigma_n)$. On pose $\sigma = \sigma_1 \sqcup \cdots \sqcup \sigma_n$ avec $dom(\sigma_i) = \mathcal{V}(t_i)$ et $\forall x \in dom(\sigma_i),\ \sigma_i(x) = \sigma(x)$. Comme l est un terme linéaire, chaque variable de X n'apparaît au plus qu'une seule fois dans l. Cela signifie que les ensembles $\mathcal{V}(t_i)$ sont disjoints et donc que les domaines de σ_i sont aussi disjoints. Cela assure que σ puisse être divisée correctement de la sorte.
  Maintenant, on étudie la décomposition de la transition $f(t_1\sigma_1, \ldots, t_n\sigma_n) \xrightarrow{\alpha}_A q$ pour montrer que les transitions de A utilisées pour reconnaître le terme $f(t_1\sigma_1, \ldots, t_n\sigma_n)$ sont considerées par les règles correspondantes de l'algorithme de filtrage.
  On observe que le terme $f(t_1\sigma_1, \ldots, t_n\sigma_n)$ est reconnu dans l'état q. En effet, on a $f(q_1, \ldots, q_n) \to q' \in \Delta$, et chaque sous-terme $t_i\sigma_i$ est reconnu dans un état q_i tel que $t_i\sigma_i \xrightarrow{\alpha_i} q_i$. Par décomposition suivant la reconnaissance de chaque sous-terme, on obtient la séquence suivante :

$$f(t_1, \ldots, t_n) \xrightarrow{\alpha_1} f(q_1, t_2, \ldots, t_n) \xrightarrow{\bigwedge_1^2 \alpha_i} f(q_1, q_2, t_3, \ldots, t_n) \xrightarrow{\bigwedge_1^3 \alpha_i} \ldots \ldots \xrightarrow{\bigwedge_1^n \alpha_i} f(q_1, \ldots, q_n) \xrightarrow{\bigwedge_1^n \alpha_i \wedge \top} q'$$

  Il y a alors deux cas à prendre en compte : (1) $q = q'$ et (2) $q \neq q'$. (1) Si $q = q'$, la décomposition est complète et si $f(t_1\sigma_1, \ldots, t_n\sigma_n) \xrightarrow{\alpha}_A q$ avec $\alpha = \bigwedge_1^n \alpha_i$.

$$f(t_1\sigma_1, \ldots, t_n\sigma_n) \xrightarrow{\bigwedge_{i=1}^n \alpha_i} f(q_1, \ldots, q_n) \xrightarrow{\bigwedge_{i=1}^n \alpha_i} q$$

  (2) $q \neq q'$: $f(t_1\sigma_1, \ldots, t_n\sigma_n) \xrightarrow{\alpha}_A q$ est vrai seulement si on a une transition $q' \xrightarrow{\alpha'} q$ telle que $\alpha = \bigwedge_1^n \alpha_i \wedge \alpha'$.

$$f(t_1\sigma_1, \ldots, t_n\sigma_n) \xrightarrow{\bigwedge_{i=1}^n \alpha_i} f(q_1, \ldots, q_n) \xrightarrow{\top} q' \xrightarrow{\alpha'} q$$

Par induction, on sait que pour chaque séquence $t_i\sigma_i \xrightarrow{\alpha_i} q_i$, le problème de filtrage est résolu *i.e.* $t_i \trianglelefteq q_i \vdash S_i$ avec S_i qui contient (α_i, σ_i). La règle (Delta) est alors appliquée à l'ensemble des prémisses $t_i \trianglelefteq q_i \vdash_A S_i$ pour la transition $f(q_1, \ldots, q_n) \to q' \in \Delta$. A partir de cela, on obtient un ensemble $S' = \bigotimes_1^n S_i$. En dépliant la définition de $\bigotimes$, on a $S = \{(\top, id) \oplus (a^1, s^1) \oplus \ldots (a^n, s^n) \mid (a^i, s^i) \in S_i\}$. Comme chaque S_i contient (α_i, σ_i), S' contient $(\top, id) \oplus (\alpha_1, \sigma_1) \oplus \ldots (\alpha_n, \sigma_n)$ lequel est, par définition de $\oplus$ égal à $(\bigwedge_1^n \alpha_i,\ \sigma)$. Ainsi, on obtient le résultat intermédiare $f(t_1, \ldots, t_n) \triangleleft q' \vdash_A S'$ tel que $f(t_1, \ldots, t_n)\sigma \xrightarrow{\bigwedge_1^n \alpha_i} q'$, avec $(\bigwedge_1^n \alpha_i, \sigma) \in S'$.
Ce résultat doit correspondre à une des prémisses de la règle (Epsilon) pour produire le résultat attendu $f(t_1, \ldots, t_n) \trianglelefteq q \vdash_A S$. Là encore, il y a deux cas à considérer :$q = q'$ et $q \neq q'$.
Si $f(q_1, \ldots, q_n) \to q' \in \Delta$ est la dernière transition utilisée pour avoir $f(t_1, \ldots, t_n)\sigma \xrightarrow{\alpha}_A q$ alors on a $\alpha = \bigwedge_1^n \alpha_i$ et on est dans le cas $q = q'$: ce cas correspond à la prémisse 0 de la règle (Epsilon) et donc $S' = S_0$. Par définition de la règle (Epsilon), S' est inclus dans S. Ce qui signifie que $(\alpha, \sigma) \in S$.

Si on a $q \neq q'$, alors il reste une séquence de transitions $q' \xrightarrow{\alpha'} q$ pour obtenir $f(t_1, \ldots, t_n)\sigma \xrightarrow{\alpha}_A q$. Le couple (α', q') est dans l'ensemble $\{(q_k, \alpha_k) \mid q_k \xrightarrow{\alpha_k} q\}$.Cela implique que le résultat $f(t_1, \ldots, t_n) \trianglelefteq q \vdash_A S'$ est une des prémisses restantes (de 1 à m). Par définition de la règle (Epsilon), S contient tous les couples $(a \wedge \alpha', s)$ où $(a, s) \in S'$. En particulier, S contient $(\bigwedge_1^n \alpha_i \wedge \alpha', \sigma)$ ce qui conclut la preuve.

□

On a aussi la propriété de correction, permettant de conclure que tous les couples (α, σ) définissent bien une paire critique.

Propriété 3.9.5 (Correction de l'algorithme de filtrage)**.** *Soient un $\mathcal{R}_{/E}-$automate A, q un de ses états, $l \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ le membre gauche et linéaire d'une règle de réécriture et σ une Q-substitution avec un domaine restreint à $\mathcal{V}(l)$. Si S est l'ensemble obtenu par du problème de filtrage, alors on a pour tout $(\alpha, \sigma) \in S$ on a $l\sigma \xrightarrow{\alpha}_A q$*

Démonstration. La preuve plus simple que celle de la complétude du filtrage suit le même schéma, par induction structurelle sur le terme filtré l. □

3.10 L'étape de $\mathcal{R}_{/E}$-complétion est correcte

On commence par montrer grâce aux différentes propriétés que l'on a montré sur la normalisation et le filtrage que la résolution d'une paire critique préserve la bonne-définition.

Lemme 3.10.1 (La résolution d'une paire critique préserve la *bonne définition*)**.** *Soient A et A' deux $\mathcal{R}_{/E}-$automates tels que A' est obtenu à partir de A en résolvant la paire critique $\langle q_c, r\sigma, \alpha\rangle$ dans A. Si A est bien-défini, alors A' l'est aussi.*

Démonstration. Soient les $\mathcal{R}_{/E}$-automates $A = \langle \mathcal{F}, Q, Q_f, \Delta \cup \varepsilon_{\mathcal{R}} \cup \varepsilon_{=} \rangle$ $A' = \langle \mathcal{F}, Q', Q_f, \Delta' \cup \varepsilon'_{\mathcal{R}} \cup \varepsilon'_{=} \rangle$. Conformément à la définition 3.7.1, on a $\Delta' = \Delta \cup \texttt{Norm}(r\sigma, \Delta \setminus \Delta^0)$, $\varepsilon'_{\mathcal{R}} = \{q'_c \xrightarrow{\alpha} q_c\} \cup \varepsilon_{\mathcal{R}}$ et $\varepsilon'_{=} = \varepsilon_{=}$. On suppose bien-sûr que l'ensemble $\Delta \setminus \Delta^0$ est déterministe et injectif, ce qui implique d'après les propriétés 3.8.6 et 3.8.7 que Δ' est aussi déterministe et injectif. Suivant la définition 3.5.7, on montre en (1) que pour tout état q de A', et tout terme v tel que $v \xrightarrow{\top}_{A'} q$, il existe $u \in \mathcal{R}ep(q)$ un représentant de q tel que $u \rightarrow^*_{\mathcal{R}} v$. Dans (2) on montre ensuite que si $q_1 \xrightarrow{\phi} q_2$ est une transition de $\varepsilon'_{\mathcal{R}}$, alors il existe deux termes $s, t \in \mathcal{T}(\mathcal{F})$ tels que $s \xrightarrow{\phi}_{A'} q_1$, $t \xrightarrow{\top}_{A'} q_2$ et $t \rightarrow_{\mathcal{R}} s$.

1. On montre la propriété par induction sur la hauteur du terme t. On suppose donc que pout tout terme t' de hauteur inférieure à celle de t et pour tout état $q \in Q'$, on a $t' \xrightarrow{\top}_{A'} q \implies \exists u \in Rep(q) : u \rightarrow^*_{\mathcal{R}} t'$. Maintenant, on montre que la propriété est vraie pour t, par étude de cas portant sur $q \in Q$ et $t \xrightarrow{\top}_{A} q$.
   - Si $q \in Q$ et $t \xrightarrow{\top}_{A} q$ alors comme A est bien-défini, on sait qu'il existe un representant $u \in Rep(q)$ tel que $u \rightarrow^*_{\mathcal{R}} t$ puisque A est bien-défini. Comme toutes les transitions de A sont présentes dans A', la propriété est immédiatement transférée sur l'automate A'.
   - Si $q \in Q$, $t \not\xrightarrow{\top}_{A} q$ et $t \xrightarrow{\top}_{A'} q$. On prouve la propriété en utilisant l'induction induction sur la taille de t. On considère donc le terme t. Comme t est reconnu par A' mais pas par A, Cela signifie que l'exécution $t' \xrightarrow{\top}_{A'} q$ utilise les transitions qui ont été ajoutées par la résolution de la paire critique.
     Par conséquent, il existe une règle de réécriture $l \rightarrow r$, une substitution $\sigma : \mathcal{X} \mapsto Q$, une formule α et un état q_c tel que $l\sigma \xrightarrow{\alpha}_{A} q_c$ et $\langle r\sigma, \alpha, q_c \rangle$ est la paire critique. De plus, la résolution de cette paire critique crée $\Delta' = \Delta \cup \Delta^1$ avec $\Delta^1 = \texttt{Norm}(r\sigma, \Delta \setminus \Delta^0)$ et $\varepsilon'_{\mathcal{R}} = \varepsilon_{\mathcal{R}} \cup \{q'_c \xrightarrow{\top} q_c\}$. D'après la propriété 3.8.6, on a $r\sigma \rightarrow_{\Delta^1} q'_c$. Pour rappel, $t' \xrightarrow{\top}_{A'} q$ peut utiliser des transitions qui ne sont pas dans A. Cependant, toutes les *nouvelles* transitions produites par $\texttt{Norm}(r\sigma, \Delta_A \setminus \Delta_0)$ utilisent un *nouvel* état comme membre droit, *i.e.* des états qui ne sont pas dans Q. Par conséquent, ces transitions ne peuvent pas être utilisées directement pour $t' \xrightarrow{\top}_{A'} q$ sans passer par ces nouveaux états. Cela signifie que l'exécution $t \xrightarrow{\top}_{A'} q$ utilise au moins une fois $q'_c \xrightarrow{\top} q_c$ puisque la totalité de l'exécution considérée est étiquetée par la formule $\top$. En résumé, on sait qu'il existe un contexte clos $C[\,]$ tel que

     $$t = C[t'] \xrightarrow{\top}_{A'} C[q'_c] \xrightarrow{\top}_{A'} C[q_c] \xrightarrow{\top}_{A} q$$

     On débute le raisonnement sur l'occurrence de la transition à l'état q'_c. Maintenant, l'objectif consiste à exhiber le terme $u \in \mathcal{R}ep(q'_c)$ tel que $u \rightarrow^*_{\mathcal{R}} t'$. On sait que on a $r\sigma \rightarrow'_{\Delta} q'_c$ à cause du résultat de la normalisation $\Delta^1 = \texttt{Norm}(r\sigma, \Delta_A \setminus \Delta_0)$. D'autre part, en combinant l'hyptothèse d'induction (les sous-termes de t' sont bien de taille inférieure à t) avec la propriété 3.8.9, on peut en déduire que il existe $u \in \mathcal{R}ep(q'_c)$ un représentant de l'état q'_c tel que $u \rightarrow^*_{\mathcal{R}} t'$. Et grâce à la propriété 3.8.8, on sait

que ce terme u est nécessairement de la forme $r\tau$ où $\tau : \mathcal{X} \to \mathcal{T}(\mathcal{F})$. On a donc

$$C[r\tau] \xrightarrow{\top}_{A'} C[q'_c] \xrightarrow{\top}_{A'} C[q_c] \xrightarrow{\top}_{A} q \text{ et } C[r\tau] \to^*_{\mathcal{R}} t$$

Par décomposition, on a $r\tau \to_{\Delta'} r\sigma \to_{\Delta'} q_c$. Ce qui signifie que pour toute variable $x \in \mathcal{V}(r)$, on a $\tau(x) \to_{\Delta'} \sigma(x)$. On sait que la subsitution σ est définie pour les variables de l, puisque qu'elle permet de définir la configuration $l\sigma$. Quant à la substitution τ, elle n'est définie que sur les variables de r. Comme les termes l et r forment la règle $l \to r$, on sait que $\mathcal{V}(r) \subseteq \mathcal{V}(l)$. On a alors besoin d'étendre la subsitution τ aux variables de l. On définit alors τ' telle que

$$\tau'(x) = \begin{cases} \tau(x) \text{ si } x \in \mathcal{V}(r) \\ t \text{ tel que } t \in \mathcal{R}ep(\sigma(x)) \end{cases}$$

On obtient alors un terme $l\tau'$ qui se réécrit en $r\tau' = r\tau$, ce qui nous donne :

$$C[l\tau'] \xrightarrow{\top}_{A'} C[q_c] \xrightarrow{\top}_{A} q \text{ et } C[l\tau'] \to^*_{\mathcal{R}} t$$

On se focalise maintenant sur $l\tau' \xrightarrow{\top}_{A'} q_c$, et on montre que l'on a nécessairement $l\tau' \xrightarrow{\top}_{A'} q_c$. On décompose $l\tau' \xrightarrow{\top}_{A'} q_c$ en $l\tau' \xrightarrow{\top}_{A'} l\sigma \xrightarrow{\top}_{A'} q_c$. A cause de la paire critique, on sait que l'on a $l\sigma \xrightarrow{\top}_{A} q_c$. De plus, pour chaque sous-terme $\tau'(x)$ est un représentant de $\sigma(x)$. Or on ne peut avoir $\tau'(x) \to'_{\Delta} \sigma(x)$ si on utilise des transitions de la normalisation, sinon $\sigma(x)$ serait un nouvel état introduit par la normalisation ce qui signifie $\sigma(x) \notin Q$. Cela contredit l'hypothèse issue de la paire critique qui pose $\sigma : \mathcal{X} \to Q$. $\tau'(x) \to_{\Delta} \sigma(x)$ et $l\tau' \xrightarrow{\top}_{A} q_c$. Ce qui donne finalement $C[l\tau'] \xrightarrow{\top}_{A} C[q_c] \xrightarrow{\top}_{A} q$. Grâce à la bonne définition de A, il est possible de fournir v un représentant de l'état q tel que $v \to^*_{\mathcal{R}} C[l\tau']$. Comme $C[l\tau'] \to^*_{\mathcal{R}} t$, on a $v \to^*_{\mathcal{R}} t$ par transitivité, ce qui termine la preuve pour ce cas.

– Si $q \notin Q$ alors clairement $q \in Q' \setminus Q$, ce qui implique que l'état q est un état introduit par la normalisation. On a donc $t \not\xrightarrow{\top}_{A} q$ alors que $t \xrightarrow{\top}_{A'} q$. Comme précédemment, on considère que la règle de réécriture $l \to r$, forme une paire critique $\langle q_c, r\sigma, \alpha \rangle$. Grâce à la propriété 3.8.9 appliquée à la l'hypothèse d'induction, on peut en conclure que la bonne définition est respectée pour état q.

2. Comme A est bien-défini, la propriété est vraie pour toutes les transitions $q \to q' \in \varepsilon_{\mathcal{R}}$. On considère la transition $q'_c \xrightarrow{\alpha} q_c$ résultant de la résolution de la paire critique $\langle q_c, r\sigma, \alpha \rangle$. L'unique transition ajoutée à $\varepsilon_{\mathcal{R}}$ pour former $\varepsilon'_{\mathcal{R}}$. Il suffit de montrer que la transition $q'_c \xrightarrow{\alpha} q_c$ valide le deuxième point de la définition 3.5.7. Grâce à la propriéte 3.8.6, on a $\Delta^1 = \texttt{Norm}(r\sigma, \Delta \setminus \Delta^0)$ dont on déduit $r\sigma \to_{\Delta^1 \cup \Delta} q'_c$, et donc $r\sigma \xrightarrow{\top}_{A'} q'_c$. Pour chaque variable $x \in \mathcal{V}ar(l)$, on peut définir la substitution $\tau : \mathcal{X} \to \mathcal{T}(\mathcal{F})$ telle que le terme $\tau(x)$ est un représentant de l'état $\sigma(x)$ dans l'automate A. Pour chacun des termes, on a donc $\tau(x) \xrightarrow{\top} \sigma(x)$. Par composition de la relation $\xrightarrow{\alpha}$, on en déduit la dérivation $l\tau \xrightarrow{\alpha}_{A} q$ Comme $l \to r$ est une règle de

réécriture bien formée, on a $\mathcal{V}(r) \subseteq \mathcal{V}(l)$, et le terme $r\tau$ est bien un terme clos. De même par composition de $\xrightarrow{\alpha}$, on construit la réduction $r\tau \xrightarrow{\top} q'_c$. On peut donc bien conclure à la préservation de la propriété En posant $s = l\tau$ et $t = r\tau$, on a $s \xrightarrow{\alpha} q$, $t \xrightarrow{\alpha} q'$ et $s \to_{\mathcal{R}} t$.

En conclusion, A' est bien-défini. □

Théorème 3.10.2. *Si $A^i_{\mathcal{R},E}$ est un $\mathcal{R}_{/E}$-automate obtenu après i étapes de complétion à partir $A^0_{\mathcal{R},E}$. Pour l'étape de complétion suivante, on a :*
- $\mathcal{L}(\mathtt{C}(A^i_{\mathcal{R},E})) \supseteq \mathcal{L}(A^i_{\mathcal{R},E})$
- $\forall t \in \mathcal{L}(A^i_{\mathcal{R},E})$, *si* $t \to_{\mathcal{R}} t'$ *alors* $t \in \mathcal{L}(\mathtt{C}(A^i_{\mathcal{R},E}))$
- *si $A^i_{\mathcal{R},E}$ est bien-défini, $\mathtt{C}(A^i_{\mathcal{R},E})$ l'est aussi.*

Démonstration. On pose C_p, l'ensemble des paires critiques de la forme $\langle q_c, r\sigma, \alpha\rangle$ pour toute règle $l \to r \in \mathcal{R}$ et tout état $q_c \in Q$. On peut calculer cet ensemble en résolvant le problème de filtrage pour chaque état et membre gauche des règles de réécriture $l \trianglelefteq q \vdash S$. D'après la propriété 3.9.4, on sait que pour toute configuration de la forme $l\sigma \xrightarrow{\alpha} q$ sans boucle, alors S contient le couple (α, σ) qui correspond bien à la paire critique $\langle q, r\sigma, \alpha\rangle$ de C_p. On obtient $\mathtt{C}(A^i_{\mathcal{R},E})$ en résolvant successivement les différentes paires critiques de C_p, qui ne le sont pas encore. On pose $A^0_{\mathcal{R},E} = A_0$ et A_{i+1} est obtenu en résolvant la $i+1^{\text{ème}}$ paire critique. On montre la préservation des trois points suivants par induction sur le nombre de paires critiques résolues :
- $\mathcal{L}(A_i) \supseteq \mathcal{L}(A_0)$
- Pour les i premières paires critiques résolues de la forme $\langle q_i, r_i\sigma, \alpha_i\rangle$, alors pour tout terme $t \in \mathcal{L}(A_i, q_i)$ de la forme $l_i\tau$ alors $r_i\tau \in \mathcal{L}(A_i)$
- si A_i est bien-défini.
- Pour A_0, les trois points sont trivialement vérifiés.
- On pose $A_i = \langle \mathcal{F}, Q, Q_f, \Delta \cup \varepsilon_{\mathcal{R}} \cup \varepsilon_{=}\rangle$ qui vérifie les trois propriétés. On considère la $i+1^{\text{ème}}$ paire critique, de la forme $\langle q_c, r\sigma, \alpha\rangle$. Si la paire critique est déjà résolue *i.e.* il existe α' tel que $r_i\sigma \xrightarrow{\alpha'} q_i$, alors $A_{i+1} = A_i$ respecte les trois proprétés naturellement. Sinon la paire critique est résolue en suivant la procédure 3.7.1 : on a alors $r\sigma \to_{\Delta'} q'_c$ et $q'_c \to q_c$ où $\Delta' = \Delta \cup \mathtt{Norm}(r\sigma, \Delta \setminus \Delta^0)$. D'après le théorème 3.10.1 le $\mathcal{R}_{/E}$-automate A_{i+1} obtenu par la résolution de la paire critique est bien défini. D'autre part le premier point est trivial puisque la résolution d'une paire critique ne fait qu'ajouter des transistions, donc on a $\mathcal{L}(A_{i+1}) \supseteq \mathcal{L}(A_i)$ ce qui implique bien $\mathcal{L}(A_{i+1}) \supseteq \mathcal{L}(A_0)$. Enfin pour n'importe quelle des $i+1$ paires critiques $\langle q, r\sigma, \alpha\rangle$, dans l'automate A_{i+1} on a $l\sigma \xrightarrow{\alpha}_{A_{i+1}} q$ qui implique $l\sigma \to^*_{A_{i+1}} q$. De même et comme la paire est résolue, on a $r\sigma \to^*_{A_{i+1}} q$. On construit tout terme clos de la forme $l\tau \to^*_{A_{i+1}} q$ se construit en définissant une substitution $\tau : \mathcal{X} \to \mathcal{T}(\mathcal{F})$ telle que $\tau(x) \to^*_{A_{i+1}} \sigma(x)$, se qui permet définir le terme $r\tau \to^*_{A_{i+1}} q$.

Au final on obtient le $\mathcal{R}_{/E}$-automate $\mathtt{C}(A^i_{\mathcal{R},E})$ qui est bien-défini et tel que $\mathtt{C}(A^i_{\mathcal{R},E}) \supseteq A^i_{\mathcal{R},E}$. Par reformulation du deuxième point ci-dessus, on sait que pour tout terme clos de la forme $t \in \mathcal{L}(\mathtt{C}(A^i_{\mathcal{R},E}), q)$ si t se réécrit à la position ϵ en t' alors on a $t' \in \mathcal{L}(\mathtt{C}(A^i_{\mathcal{R},E}), q)$.

On généralise la propriété par induction sur la position p. Le cas $p = \epsilon$ est déjà couvert. On considère le terme $t \in \mathcal{L}(\mathtt{C}(A^i_{\mathcal{R},E}), q)$, réécrit à la postion $i.p$. Le terme t est forcément de la forme $f(t_1, \ldots, t_n)$ avec t_i qui se réécrit à la position p. On décompose l'exécution du $\mathcal{R}_{/E}$-automate $\mathtt{C}(A^i_{\mathcal{R},E})$ pour t :

$$f(t_1, \ldots, t_i, \ldots, t_n) \rightarrow^*_{\mathtt{C}(A^i_{\mathcal{R},E})} f(q_1, \ldots, q_i, \ldots, q_n) \rightarrow^*_{\mathtt{C}(A^i_{\mathcal{R},E})} q$$

On considère la réduction du sous-terme $t_i \rightarrow^*_{\mathtt{C}(A^i_{\mathcal{R},E})} q_i$. D'après l'hypothèse d'induction, on sait que si t_i se réécrit en t'_i à la position p, alors $t'_i \rightarrow^*_{\mathtt{C}(A^i_{\mathcal{R},E})} q_i$. On en déduit la réduction :

$$f(t_1, \ldots, t'_i, \ldots, t_n) \rightarrow^*_{\mathtt{C}(A^i_{\mathcal{R},E})} f(q_1, \ldots, q_i, \ldots, q_n) \rightarrow^*_{\mathtt{C}(A^i_{\mathcal{R},E})} q$$

Ce qui termine la preuve. □

3.11 L'accélération du calcul W

On Considère un $\mathcal{R}_{/E}$-automate $A = \langle \mathcal{F}, Q, Q_f, \Delta \cup \varepsilon_{\mathcal{R}} \cup \varepsilon_= \rangle$, l'opérateur d'élargissement consiste à calculer un $\mathcal{R}_{/E}$-automate $\mathtt{W}(A)$ qui est obtenu à partir de A en utilisant E l'ensemble d'équations qui définit l'abstraction.

Pour chaque équation $l = r$ de E, on considère tout couple (q, q') d'états distincts de Q telle qu'il existe une substitution σ de façon à avoir le diagramme suivant. On utilise $\rightarrow^=_A$ la clôture transitive et réflexive de la relation de réécriture induite par $\Delta \cup \varepsilon_=$.

$$\begin{array}{ccc} l\sigma & \overset{E}{=\!=} & r\sigma \\ {\scriptstyle A}\downarrow & = \quad = & \downarrow{\scriptstyle A} \\ q & & q' \end{array}$$

Intuitivement, si on a $u \rightarrow^=_A q$, alors on sait qu'il existe un terme t de $Rep(q)$ tel que $t =_E u$. Le $\mathcal{R}_{/E}$-automate $\mathtt{W}(A)$ est défini comme $\langle \mathcal{F}, Q, Q_f, \Delta \cup \varepsilon_{\mathcal{R}} \cup \varepsilon'_= \rangle$, avec $\varepsilon'_=$ qui est obtenu à partir de A en ajoutant les transitions $q \rightarrow q'$ and $q' \rightarrow q$ à l'ensemble $\varepsilon_=$, pour chaque paire (q, q'). On maintient aussi *la clôture transitive* de $\varepsilon'_=$, mais seulement pour les couples d'états distincts. Intuitivement, la clôture transitive de $\varepsilon'_=$ correspond à expliciter la clôture transitive des termes qui sont équivalents par $=_E$. Comme montré dans la section 3.14, cet aspect est important pour raffiner avec précision.

Remarque 3.11.1. *La fonction* W *termine. La relation $\varepsilon_=$ est un sous-ensemble de $Q \times Q$. L'ajout de nouvelle transition sature progressivement $\varepsilon_=$. Quand $\varepsilon_= = Q \times Q$, il n'est tout simplement plus possible d'ajouter de transitions à $\varepsilon_=$.*

Lemme 3.11.2. *Pour tout $\mathcal{R}_{/E}$-automate A, $\mathcal{L}(\mathtt{W}(A)) \supseteq \mathcal{L}(A)$.*

Démonstration. Comme pour C, la preuve est triviale : la fonction W ne fait qu'ajouter des transitions, ce qui ne restreint en aucun cas le langage reconnu. Par conséquent, un critère syntaxique sur l'inclusion des ensembles d'états et de transitions est suffisant : toute exécution de A peut être rejouée par $\mathtt{W}(A)$ □

Lemme 3.11.3 ($\mathtt{W}$ preserve la bonne-définition)**.** *Soit A un $\mathcal{R}_{/E}$-automate. Si A est bien-défini, alors $\mathtt{W}(A)$ l'est aussi.*

Démonstration. On pose $A = \langle \mathcal{F}, Q, Q_f, \Delta \cup \varepsilon_{\mathcal{R}} \cup \varepsilon_{=} \rangle$ un $\mathcal{R}_{/E}$-automate bien-défini et E un ensemble d'équations qui définit l'abstraction. On construit $\mathtt{W}(A) = \langle \mathcal{F}, Q, Q_f, \Delta \cup \varepsilon_{\mathcal{R}} \cup \varepsilon'_{=} \rangle$, avec $\varepsilon_{\mathcal{R}} \supseteq \varepsilon'_{\mathcal{R}}$. puisque $\mathtt{W}$ ne fait qu'ajouter des transitions à $\varepsilon_{\mathcal{R}}$. On doit prouver que les deux items de la définition 3.5.7 sont respectés par $\mathtt{W}(A)$.

- Les transitions de $\varepsilon'_{=}$ ne sont jamais utilisés pour une exécution de $\xrightarrow{\alpha}$ avec $\alpha = \top$, d'après le second point de la définition 3.5.4. Cela signifie que pour tout terme t et tout état q, $t \xrightarrow{\top}_{\mathtt{W}(A)} q$ est équivalent à $t \xrightarrow{\top}_{A} q$. Comme A est bien-défini, on sait quil existe un représentant $u \in Rep(q)$ tel que $u \to^*_{\mathcal{R}} t$. u est aussi un représentant de $\mathtt{W}(A)$, et donc on peut en déduire and que le $\mathcal{R}_{/E}$-automate $\mathtt{W}(A)$ respecte le premier point de la définition 3.5.7.
- La fonction $\mathtt{W}$ ne fait qu'ajouter des transitions à $\varepsilon'_{=}$ et ne supprime aucune transition de A. Pour toutes les transitions $q \xrightarrow{\alpha} q' \in \varepsilon'_{\mathcal{R}}$ on a donc $q \xrightarrow{\alpha} q' \in Drw$. Comme A est bien-défini, on sait qu'il existe deux termes $s, t \in \mathcal{T}(\mathcal{F})$ tels que $s \xrightarrow{\phi}_{A} q$, $t \xrightarrow{\top}_{A} q'$ et $t \to_{\mathcal{R}} s$. On a donc $s \xrightarrow{\phi}_{\mathtt{W}(A)} q$, $t \xrightarrow{\top}_{\mathtt{W}(A)} q'$ et $t \to_{\mathcal{R}} s$, ce qui montre le second point de la définition 3.5.7.

□

On peut donc conclure que $\mathtt{W}$ préserve la bonne définition tout en élargissant le langage.

Propriété 3.11.4. *Soit A un $\mathcal{R}_{/E}$-automate bien-défini, on a $\mathcal{L}(A) \supseteq \mathtt{W}(A)$, et $\mathtt{W}(A)$ est bien-défini.*

Remarque 3.11.5. *Comme $\mathtt{W}$ ne modifie pas l'ensemble Δ de transitions normalisées, l'injectivité et le déterminisme de l'ensemble $\Delta \setminus \Delta^0$ sont naturellement préservés.*

Exemple 3.11.6. *Soit $\mathcal{R} = \{f(x) \to f(s(s(x)))\}$ et le $\mathcal{R}_{/E}$-automate $A^0_{\mathcal{R},E} = \langle \mathcal{F}, Q, Q_F, \Delta^0 \rangle$ tel que $Q_F = \{q_0\}$ et $\Delta_0 = \{a \to q_1,\ f(q_1) \to q_0\}$. D'après la définition 3.9.1, pour $f(x) \trianglelefteq q_0$ on obtient l'ensemble $S = \{(\phi, \sigma)\}$ avec $\sigma = \{x \to q_1\}$ et $\phi = \top$. Ensuite, $\langle q_0, f(s(s(q1))), \top, q_0 \rangle$ est la seule paire critique à résoudre, puisque $f(s(s(q1))) \not\xrightarrow{\top}_{A^0_{\mathcal{R},E}} q_0$. Donc, $\mathtt{C}(A^0_{\mathcal{R},E}))$ est un $\mathcal{R}_{/E}$-automate défini comme $\mathtt{C}(A^0_{\mathcal{R},E}) = \langle \mathcal{F},\ Q_1, Q_F, \Delta_1 \cup \varepsilon^1_{\mathcal{R}} \cup \varepsilon^1_{=} \rangle$ avec :*

- $\Delta_1 = \mathtt{Norm}(f(s(s(q_1))), \emptyset) \cup \Delta^0 = \{s(q_1) \to q_2,\ s(q_2) \to q_3,\ f(q_3) \to q_4\} \cup \{a \to q_1,\ f(q_1) \to q_0\}$,
- $\varepsilon^1_{\mathcal{R}} = \{q_4 \to q_0\}$, *car* $f(s(s(q_1))) \to_{\Delta^1} q_4$,
- $\varepsilon^1_{=} = \emptyset$ *et* $Q_1 = \{q_0, q_1, q_2, q_3, q_4\}$.

Maintenant, on calcule $A^1_{\mathcal{R},E} = \mathtt{W}(\mathtt{C}(A^0_{\mathcal{R},E}))$ en utilisant l'équation $s(s(x)) = s(x)$. On a $\sigma = \{x \mapsto q_1\}$ ainsi que le diagramme ci-contre. On obtient alors le $\mathcal{R}_{/E}$-automate $A^1_{\mathcal{R},E} = \langle \mathcal{F}, Q^1, Q_f, \Delta^1 \cup \varepsilon^1_{\mathcal{R}} \cup \varepsilon^1_{=} \rangle$, avec $\varepsilon^1_{=} = \varepsilon^0_{=} \cup \{q_3 \to q_2,\ q_2 \to q_3\}$ et $\varepsilon^0_{=} = \emptyset$. On

observe que $A^1_{\mathcal{R},E}$ *est un point fixe :* $\mathtt{C}(A^1_{\mathcal{R},E}) = A^1_{\mathcal{R},E}$.

$$\begin{array}{ccc} s(s(q_1)) & \underset{E}{=\!=} & s(q_1) \\ {\scriptstyle \mathtt{C}(A^0_{\mathcal{R},E})}\downarrow = & & = \downarrow{\scriptstyle \mathtt{C}(A^0_{\mathcal{R},E})} \\ q_3 & & q_2 \end{array}$$

3.12 Vérification de propriétés régulières

On se place à nouveau dans le cadre de la vérification de systèmes comme définie dans la section 2.3.1. On va montrer comment utiliser la $\mathcal{R}_{/E}$-complétion pour dans le cadre du model-checking régulier avec raffinement d'abstraction pour de vérifier des propriétés sur des systèmes.

Soient I un ensemble de termes initiaux caractérisés par le $\mathcal{R}_{/E}-$automate $A^0_{\mathcal{R},E}$, $\mathcal{R}$ un système de règles de réécriture, et Bad un ensemble de termes interdits représentés par A_{Bad} un automate d'arbres. Le problème d'atteignabilité se réduit donc à vérifier $\mathcal{R}^*(I) \cap Bad \stackrel{?}{=} \emptyset$. Il y a des classes de systèmes pour lesquelles $\mathcal{R}^*(I)$ est régulier et peut être calculé dans un temps fini (*c.f.* section 3.15) mais en général, le calcul ne termine pas. Dans de tel cas, on peut espérer obtenir un résultat en se reposant sur une approche "à la CEGAR", *Counterexample-Guided Abstraction Refinement [17]*, qui calcule une série d'approximations successivement raffinées jusqu'à ce que la propriété puisse être prouvée correcte ou non. Grâce à la définition $\mathcal{R}_{/E}$-complétion définie dans les sections précédentes de ce chapitre, on possède une procédure de construction de l'approximation $\mathcal{R}^*(I)$, *i.e.* par étapes successives de complétion jusqu'à l'obtention d'un $\mathcal{R}_{/E}$-automate $A^*_{\mathcal{R},E}$ point-fixe qui est clos par réécriture. Comme les $\mathcal{R}_{/E}$-automates produits par complétion sont bien-définis 3.5.7, on peut déterminer si un terme de l'approximation qui viole la propriété est un contre-exemple ou un terme suspicieux. La figure 3.12, illustre l'approche que l'on va mettre en oeuvre. Il n'est pas nécessaire d'atteindre le point-fixe pour vérifier que la propriété est violée. Si on tente de vérifier la propriété sur le $\mathcal{R}_{/E}$-automate obtenu après $k+1$ étapes tel que l'intersection soit non-vide. Si l'intersection contient un terme atteignable alors la propriété est violée, sinon on raffine l'approximation en produisant l'automate $A^{k+2}_{\mathcal{R},E}$ et en reprenant le processus de $\mathcal{R}_{/E}$-complétion.

Pour appliquer cette technique, il faut donc être capable de calculer l'intersection entre le $\mathcal{R}_{/E}$-automate $A^{k+1}_{\mathcal{R},E}$ et l'automate A_{Bad}, puis être capcable d'en déterminer le contenu, et enfin d'en extraire l'information nécessaire au raffinement de l'approximation.

3.13 Test de vacuité de l'intersection

Le produit d'un $\mathcal{R}_{/E}$-automate avec un automate d'arbres classique n'a de sens que si l'on considère aussi le $\mathcal{R}_{/E}$-automate comme un automate classique, mais on perd alors l'information accumulée pendant le calcul. On ne peut pas non plus définir par un $\mathcal{R}_{/E}$-automate le produit d'un $\mathcal{R}_{/E}-$*automate* par un automate classique. En effet, les formules

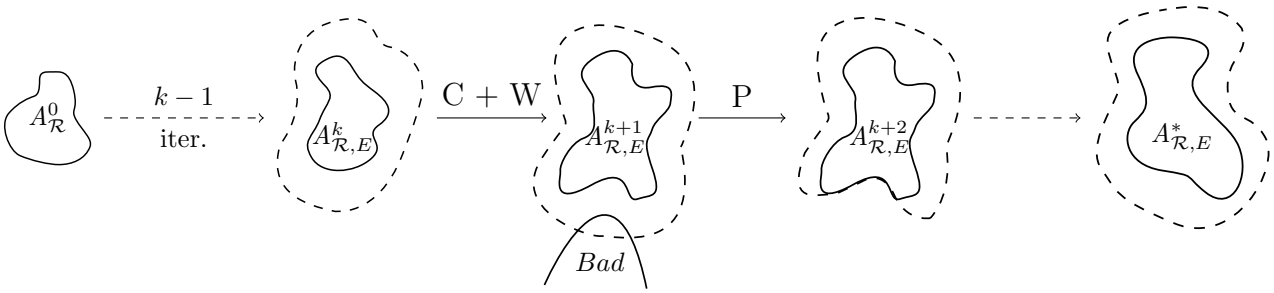


FIGURE 3.3: Etape de raffinement durant la complétion

de ce $\mathcal{R}_{/E}$-automate devraient porter sur celui a servi à construire le produit et non lui-même : le $\mathcal{R}_{/E}$-automate produit n'est pas cohérent avec la définition 3.5.1. En partant de ce constat, la solution retenue consiste à "simuler" l'exécution du produit entre le $\mathcal{R}_{/E}$-automate et repérer sous quelles conditions l'intersection n'est pas vide. Les conditions sont définies par des formules logiques issues de l'information du $\mathcal{R}_{/E}$-automate.

On donc introduit un algorithme spécifique qui construit l'ensemble S des états accessibles pour le produit d'un $\mathcal{R}_{/E}$-automate A par un automate d'arbres B où chaque produit d'état est étiqueté par une formule sur les états de A. On définit d'arbord un ordre $>$ sur les formules, tel que $\phi_1 > \phi_2$ dénote ϕ_1 comme une formule plus "permissive" que phi_2.

Définition 3.13.1. *Soient ϕ_1 et ϕ_2 deux formules, $\phi_1 > \phi_2$ si et seulement si $\phi_2 \models \phi_1$ et $\phi_1 \not\models \phi_2$.*

Définition 3.13.2 (Etats accessibles du produit d'un $\mathcal{R}_{/E}$-automate par un automate d'arbres)**.** *Soit $A = \langle \mathcal{F}, Q^A, Q^A_f, \Delta^A, \varepsilon_{\mathcal{R}}, \varepsilon_= \rangle$ un $\mathcal{R}_{/E}$-automate et $B = \langle \mathcal{F}, Q^B, Q^B_f, \Delta^B \rangle$ un automate d'arbres sans ε-transition. L'ensemble S des états accessibles de $A \times B$ est l'ensemble des triplets (q, q', ϕ) tels que $q \in Q^A, q' \in Q^B$ et ϕ est une formule. En commençant à partir de l'ensemble $Q^A \times Q^B \times \{\bot\}$, la valeur de S peut être calculée en utilisant les deux règles de déduction suivantes :*

$$\frac{\{(q_1, q'_1, \phi_1), \ldots, (q_n, q'_n, \phi_n)\} \cup \{(q, q', \phi)\} \cup P}{\{(q_1, q'_1, \phi_1), \ldots, (q_n, q'_n, \phi_n)\} \cup \{(q, q', \phi \vee \bigwedge_{i=1}^n \phi_i)\} \cup P} \quad \begin{array}{ll} si & f(q_1, \ldots, q_n) \to q \in \Delta^A \\ & et\ f(q'_1, \ldots, q'_n) \to q' \in \Delta^B \\ & et\ (\phi \vee \bigwedge_{i=1}^n \phi_i) > \phi \end{array}$$

$$\frac{\{(q_1, q, \phi_1), (q_2, q, \phi_2)\} \cup P}{\{(q_1, q, \phi_1), (q_2, q, (\phi_1 \wedge \phi) \vee \phi_2)\} \cup P} \quad \begin{array}{ll} si\ q_1 \xrightarrow{\phi} q_2 \in \varepsilon_{\mathcal{R}} & ou \quad si\ q_1 \to q_2 \in \varepsilon_= \\ et\ ((\phi_1 \wedge \phi) \vee \phi_2) > \phi_2 & \quad\ et\ \phi = Eq(q_1, q_2) \\ & \quad\ et\ ((\phi_1 \wedge \phi) \vee \phi_2) > \phi_2 \end{array}$$

Concernant le problème d'atteignabilité, cette définition, fournit un moyen de distinguer les contre-exemples réels des termes qui peuvent être éliminés par raffinement de l'abstraction. En effet, pour tout triplet $(q, q', \phi) \in S$ avec q final dans A et q' final dans B, si $\phi \models \top$ alors certains des termes reconnus par q' dans B sont atteignables. Sinon,

ϕ est une formule à invalider, *i*.e. dont certains atomes doivent être invalidés pour que ϕ devienne $\bot$.

Lemme 3.13.3 (décision de la vacuité du produit d'un $\mathcal{R}_{/E}$-automate par un automate d'arbres)**.** *Soient A un $\mathcal{R}_{/E}$-automate et B un automate d'arbres. Soit S l'ensemble des états accessibles de $A \times B$ construit comme expliqué dans la definition 3.13.2. Pour tout état final q de A, tout état final q' de B, pour toutes formules $\phi_S \neq \bot$, $\phi \neq \bot$ et tout terme $t \in \mathcal{T}(\mathcal{F})$, on a $t \xrightarrow{\phi}{}^*_A q$ et $t \to^*_B q'$ (i.e. $\mathcal{L}(A) \cap \mathcal{L}(B) \neq \emptyset$) si et seulement si il existe un triplet $(q, q', \phi_S) \in S$ tel que $\phi \models \phi_S$.*

Démonstration. Soient $A = \langle \mathcal{F}, Q^A, Q^A_f, \Delta^A, \varepsilon_{\mathcal{R}}, \varepsilon_= \rangle$ un $\mathcal{R}_{/E}$-automate et $B = \langle \mathcal{F}, Q^B, Q^B_f, \Delta^B \rangle$ un automate d'arbres.

On prouve une propriété plus forte sur tous les états q de A et q' de B (et pas seulement sur les états finals).

Premièrement, on montre la partie "seulement si" de l'équivalence. On suppose que il existe un terme $t \in \mathcal{T}(\mathcal{F})$ tel que $t \xrightarrow{\phi}{}^*_A q$, $t \to^*_B q'$. Par induction sur la taille de t, on a :

- Si t est une constante, comme B est un automate sans ε-transition, le seul moyen d'avoir $t \to^*_B q'$ est une transition $t \to q' \in B$. Concernant A, par la définition 3.5.4, $t \xrightarrow{\phi}{}^*_A q$ signifie qu'il existe des états $q_0, q_1, \ldots, q_n$ et formules $\phi_1, \ldots, \phi_n$ tels que $t \to_{\Delta_A} q_0 \xrightarrow{\phi_1} q_1 \xrightarrow{\phi_2} \ldots q_n$ avec $q = q_n$ et $\phi = \phi_1 \wedge \ldots \wedge \phi_n$.Les transitions $q_i \xrightarrow{\phi_i} q_{i+1}$ sont soit des transitions de $\varepsilon_{\mathcal{R}}$ ou des transitions de $\varepsilon_=$ avec $\phi_i = \top$. A cause des transitions $t \to q_0 \in \Delta_A$ et $t \to q' \in \Delta_B$, en utilisant le premier cas de la définition 3.13.2, on obtient que $(q_0, q', \top) \in S$. De la même manière, en utilisant le second cas de la définition, on obtient qu'il existe des formules ϕ'_i avec $i = 1 \ldots n$ telles que $(q_1, q', \phi_1 \vee \phi'_1), (q_2, q', (\phi_1 \wedge \phi_2) \vee \phi'_2), \ldots (q_n, q', (\phi_1 \wedge \ldots \wedge \phi_n) \vee \phi'_n)$ appartiennent à S. Finalement, puisque $q_n = q$ et $\phi = \phi_1 \wedge \ldots \wedge \phi_n$, on a que $(q, q', \phi \vee \phi'_n) \in S$. De plus, on a aussi $\phi_S = \phi \vee \phi'_n$ et $\phi \models \phi_S$.
- On suppose que pour tout terme de hauteur inférieure ou égale à $n \in \mathbb{N}^*$, la propriété est vraie. On va alors prouver que c'est aussi le cas pour le terme $f(t_1, \ldots, t_n)$ avec $t_1, \ldots, t_n$ de hauteur inférieur ou égale à n. Comme $f(t_1, \ldots, t_n) \to^*_B q'$ et B est un automate sans ε-transitions, on obtient que $\exists q'_1, \ldots, q'_n \in Q^B$ tel que $\forall i = 1 \ldots n : t_i \to^*_B q'_i$ et $f(q'_1, \ldots, q'_n) \to q' \in \Delta_B$. Concernant A, par la définition 3.5.4, $f(t_1, \ldots, t_n) \xrightarrow{\phi}{}^*_A q$ signifie que il existe des états $q_0, q_1, \ldots, q_m, q''_1, \ldots, q''_n$ et des formules $\phi_1, \ldots, \phi_m, \phi'_1, \ldots, \phi'_n$ tels que $\forall i = 1 \ldots n : t_i \xrightarrow{\phi'_i}{}^*_A q''_i$, $f(q''_1, \ldots, q''_n) \to_{\Delta_A} q_0$ et $q_0 \xrightarrow{\phi_1} q_1 \xrightarrow{\phi_2} \ldots q_n$, $q = q_n$. De plus, on a aussi $\phi = \bigwedge_{i=1}^n \phi'_i \wedge \bigwedge_{i=1}^m \phi_i$. Comme les termes t_i sont de hauteur inférieure ou égale à n, $\forall i = 1 \ldots n : t_i \to^*_B q_i$ et $\forall i = 1 \ldots n : t_i \xrightarrow{\phi'_i}{}^*_A q''_i$, on peut appliquer l'hypothèse d'induction et on obtient que $\forall i = 1 \ldots n : (q_i, q''_i, \phi''_i) \in S$ avec $\phi'_i \models \phi''_i$. D'ailleurs, en utilisant le cas 1 de la définition 3.5.4 sur $f(q_1, \ldots, q_n) \to q' \in \Delta_B$, $f(q''_1, \ldots, q''_n) \to q_0 \in \Delta_A$, et $\forall i = 1 \ldots n : (q_i, q''_i, \phi''_i) \in S$, on obtient qu'il existe une formule ϕ' telle que $(q_0, q', (\bigwedge_{i=1}^n \phi''_i) \vee \phi') \in S$. Alors, comme pour le cas de base, puisque

$q_0 \xrightarrow{\phi_1} q_1 \xrightarrow{\phi_2} \dots q_n$, $q = q_n$, on peut déduire qu'il existe une formule ϕ'' telle que $(q, q', (\bigwedge_{i=1}^{n} \phi_i'' \wedge \bigwedge_{i=1}^{m} \phi_i) \vee \phi'') \in S$. Soit $\phi_S = (\bigwedge_{i=1}^{n} \phi_i'' \wedge \bigwedge_{i=1}^{m} \phi_i) \vee \phi''$. Puisqu'on sait à partir de ci-dessus que $\phi = \bigwedge_{i=1}^{n} \phi_i' \wedge \bigwedge_{i=1}^{m} \phi_i$ et $\forall i = 1 \dots n : \phi_i' \models \phi_i''$, on a $\phi \models \phi_S$.

Maintenant, on montre le sens "si" de l'équivalence : si $(q, q', \phi_S) \in S$ et $\phi_S \neq \bot$ alors il existe un terme t et une formule $\phi \neq \bot$ tels que $\phi \models \phi_S$, $t \xrightarrow{\phi}{}^*_A q$ et $t \to_B^* q'$. La preuve est réalisée par induction sur le nombre d'aplications des deux règles de la définition 3.13.2, nécessairement pour prouver que (q, q', ϕ_S) dans S.

- Si le nombre d'étapes est 0 alors, comme le calcul de S débute à partir de $Q^A \times Q^B \times \bot$, alors tout triplet (q, q', ϕ_S) est tel que $\phi_S = \bot$, ce qui est une contradiction.
- On suppose que la propriété est vraie pour tout triplet (q, q', ϕ_S) qui peut être déduit après n ou moins applications des règles de la définition 3.13.2. Ensuit, on considère le cas d'un triplet (q, q', ϕ_S) qui est déduit par la $n+1^{\text{ième}}$ application d'une des deux règles de déduction.
  - Si la première règle est concernée, cela signifie qu'il existe des triplets $(q_1, q_1', \phi_1), \dots, (q_n, q_n', \phi_n)$ et (q, q', ϕ) de S déduit avant la $n+1$-th étape, ainsi que des transitions $f(q_1, \dots, q_n) \to q \in \Delta_A$ et $f(q_1', \dots, q_n') \to q' \in \Delta_B$. De plus, on sait que $\phi_S = \phi \vee \bigwedge_{i=1}^{n} \phi_i$. Si $\phi \neq \bot$ alors, comme il a été montré que (q, q', ϕ) appartient S avant la $n+1$-th étape, on peut appliquer l'hypothèse d'induction et directement obtenir qu'il existe un terme t et une formule ϕ' tels que $\phi' \models \phi$, $t \xrightarrow{\phi'}{}^*_A q$ et $t \to_B^* q'$. On remarque $\phi' \models \phi$ implique $\phi' \models \phi_S$. Sinon, si $\phi = \bot$, alors on peut appliquer l'hypothèse d'induction sur les triplets (q_i, q_i', ϕ_i), $i = 1 \dots n$ et obtenir que $\forall i = 1 \dots n : \exists \phi_i' : \exists t_i \in \mathcal{T}(\mathcal{F}) : \phi_i' \models \phi_i$, $t_i \xrightarrow{\phi_i'}{}^*_A q_i$ et $t_i \to_B^* q_i'$. Finallement, à cause des deux transitions $f(q_1, \dots, q_n) \to q \in \Delta_A$ et $f(q_1', \dots, q_n') \to q' \in \Delta_B$, on a $f(t_1, \dots, t_n) \xrightarrow{\phi'}{}^*_A f(q_1, \dots, q_n) \to_A^* q$ avec $\phi' = \bigwedge_{i=1}^{n} \phi_i'$ d'une part, et $f(t_1, \dots, t_n) \to_B f(q_1', \dots, q_n') \to_B^* q$ d'autre part. De plus, puisque $\forall i = 1 \dots n : \phi_i' \models \phi_i$, on a $\bigwedge_{i=1}^{n} \phi_i' \models \bigwedge_{i=1}^{n} \phi_i$. On rappelle que $\phi' = \bigwedge_{i=1}^{n} \phi_i'$ et $\phi_S = \phi \vee \bigwedge_{i=1}^{n} \phi_i$. Ainsi, $\phi' \models \phi_S$.
  - Si la seconde règle est concernée, cela signifie qu'il existe des triplets (q_1, q', ϕ_1) et (q, q', ϕ_2) dans S déduits avant la $n+1^{\text{ième}}$ étape. De plus, on sait que $\phi_S = (\phi_1 \wedge \phi) \vee \phi_2$. Comme précédement, si $\phi_2 \neq \bot$ alors on peut appliquer l'hypothèse d'induction sur (q, q', ϕ_2) et triviallement obtenir le résultat. Autrement, si $\phi_2 = \bot$ alors on peut utiliser à nouveau l'hypothèse d'induction pour le triplet (q_1, q', ϕ_1) et obtenir qu'il existe une formule ϕ_1' et un terme t_1 tels que $t_1 \xrightarrow{\phi_1'}{}^*_A q_1$, $t_1 \to_B^* q'$ et $\phi_1' \models \phi_1$. Alors, par cas sur la ε-transition utilisée pour la deéuction sur S, on prouve que $t_1 \xrightarrow{\phi_1' \wedge \phi}{}^*_A q$:
    - On suppose que $q_1 \xrightarrow{\phi} q \in \varepsilon_{\mathcal{R}}$. Alors, d'après la définition 3.5.4, on déduit que $t_1 \xrightarrow{\phi_1' \wedge \phi}{}^*_A q$. De plus, puisque $\phi_1' \models \phi_1$, on a que $\phi_1' \wedge \phi \models \phi_1 \wedge \phi$ et, finalement, que $\phi_1' \wedge \phi \models \phi_S$.
    - On suppose que $q_1 \to q \in \varepsilon_{=}$. Par la définition 3.5.4, on obtient que $t \xrightarrow{\phi_1 \vee Eq(q_1, q)}{}^*_A$

q. Finalement, comme précédemment, on peut déduire que $\phi'_1 \wedge Eq(q_1, q) \models \phi_1 \wedge Eq(q_1, q)$ et ainsi $\phi'_1 \wedge Eq(q_1, q) \models \phi_S$.

□

3.14 Raffiner l'approximation par élagage

Soit $A^k_{\mathcal{R},E} = \langle \mathcal{F}, Q^k, Q_f, \Delta^k \cup \varepsilon^k_{\mathcal{R}} \cup \varepsilon^k_{=} \rangle$ le $\mathcal{R}_{/E}$-automate obtenu après k étapes de complétion et d'élargissement à partir de $A^0_{\mathcal{R},E}$ et on suppose que $A^k_{\mathcal{R},E}$ reconnaît des termes de Bad, ce qui signifie que $\mathcal{L}(A^k_{\mathcal{R},E}) \cap Bad \neq \emptyset$.

On commence par détailler la procédure d'élagage pour un terme, mais la procédure fonctionne de la même manière pour retirer la totalité des termes de l'approximation contenus dans l'intersection, à partir des formules obtenues par le test de vacuité avec l'automate A_{Bad}.

Soit $t \in \mathcal{L}(A^k_{\mathcal{R},E}) \cap Bad$ l'un de ses termes. Puisque le terme est reconnu par l'automate $A^k_{\mathcal{R},E}$, il existe une exécution $t \xrightarrow{\phi}_{A^k_{\mathcal{R},E}} q_f$ où $q_f \in Q_f$ est un état final. On sait que $A^k_{\mathcal{R},E}$ est bien défini par construction.

Cela implique que si la formule ϕ est égale à $\top$, on sait que $t \in \mathcal{R}^*(I)$ est atteignable. On en déduit donc que $t \in \mathcal{R}^*(I) \cap Bad$, c'est un contre-exemple, et d'un point de vue vérification le propriété est violée comme formulée à la section 2.4.

Sinon, on a une formule $\phi = \bigwedge_1^n Eq(q_j, q'_j)$, et t est peut-être un contre-exemple fallacieux que l'on souhaite retirer de l'approximation. Pour raffiner l'approximation, on procède à l'*élagage* du $\mathcal{R}_{/E}$-automate déjà introduit informellement dans l'exemple 3.5.8. On définit l'étape d'élagage `P` appliquée à un $\mathcal{R}_{/E}$-automate et paramétrée par une formule $\phi = \bigwedge_1^n Eq(q_j, q'_j)$.

3.14.1 Procédure d'élagage `P`.

Comme présenté dans l'exemple 3.5.8, on élague le $\mathcal{R}_{/E}$-automate $\mathtt{P}(A^k_{\mathcal{R},E}, \phi)$ en deux étapes. La première étape consiste à retirer des transitions de $\varepsilon^k_{=}$ jusqu'à ce que ϕ ne soit plus vraie *i.e.* $\phi = \bot$.

Si on considère la formule ϕ contenant le prédicat $Eq(q, q')$: on remplace ce prédicat par $\bot$ si on décide de retirer la transition $q \to q'$ de $\varepsilon^k_{=}$. La seconde étape consiste à propager l'information. En effet, on doit aussi retirer toutes les transitions $q \xrightarrow{\alpha} q' \in \varepsilon^k_{\mathcal{R}}$, où la conjonction α contient des atomes correspondants à transitions retirées de $\varepsilon^k_{=}$. La procédure est répétée pour chaque chemin d'exécution qui permet à l'automate $A^k_{\mathcal{R},E}$ de reconnaître le terme t. Finalement on obtient le $\mathcal{R}_{/E}$-automate $A^{k+1}_{\mathcal{R},E}$. Il est facile de voir que, pour toute formule ϕ, il n'existe plus aucune exécution $t \xrightarrow{\phi}_{A^{k+1}_{\mathcal{R},E}} q_f$. On remarque que cette procédure permet aussi retirer d'autres termes engendrés par ϕ.

Propriété 3.14.2. `P` *préserve la bonne-définition.*

Démonstration. L'élagage du $\mathcal{R}_{/E}$-automate $A^k_{\mathcal{R},E}$ ne supprime pas de transitions de Δ^k ni les transitions de la forme $q' \xrightarrow{\top} q$. Donc pour tout terme t avec $t \xrightarrow{\top}_{A^k_{\mathcal{R},E}} q$, on a aussi $t \xrightarrow{\top}_{A^{k+1}_{\mathcal{R},E}} q$. Comme il en est de même avec les représentants de chaque état, on peut en conclure que $A^{k+1}_{\mathcal{R},E}$ respecte le premier point de la bonne définition si $A^k_{\mathcal{R},E}$ est bien défini.

Pour le deuxième point, on considère une transition $q' \xrightarrow{\alpha} q \in \varepsilon^{k+1}_{\mathcal{R}}$ du $\mathcal{R}_{/E}$-automate élagué $A^{k+1}_{\mathcal{R},E}$. Comme pour le point précédent, on sait qu'il existe un terme t tel que $t \xrightarrow{\top}_{A^{k+1}_{\mathcal{R},E}} q'$. Comme $A^k_{\mathcal{R},E}$ est bien-défini, on sait aussi qu'il existe un terme s tel que $s \xrightarrow{\alpha}_{A^k_{\mathcal{R},E}} q$. D'après la transition $s \xrightarrow{\alpha}_{A^k_{\mathcal{R},E}} q$, on peut en déduire que s est réduit en l'état q au moyen de transitions de $\varepsilon^k_{\mathcal{R}}$ et de $\varepsilon^k_{=}$ qui ont pu être supprimées par élagage :

- Si la transition $q_1 \to q_2 \in \varepsilon^k_{=}$ a été supprimée dans $A^{k+1}_{\mathcal{R},E}$, alors comme l'atome $Eq(q_1, q_2)$ devient faux : comme on a utilisé la transition $q_1 \to q_2$ pour avoir $s \xrightarrow{\alpha}_{A^k_{\mathcal{R},E}} q$, on sait que l'atome est présent dans la formule α : la procédure d'élagage prévoit de supprimer toute transition de $q_a \xrightarrow{\phi} q_b \in \varepsilon^k_{\mathcal{R}}$ où la conjonction ϕ contient $Eq(q_1, q_2)$. On en déduit que la transition $q' \xrightarrow{\alpha} q$ ne peut exister dans $\varepsilon^{k+1}_{\mathcal{R}}$ puisqu'elle devait être supprimée, ce qui est contradictoire avec l'hypothèse.
- Si c'est une transition $q_1 \xrightarrow{\phi} q_2 \in \varepsilon^k_{\mathcal{R}}$ qui a été supprimée, on suit un raisonnement similaire. Par définition de la réduction $s \xrightarrow{\alpha}_{A^k_{\mathcal{R},E}} q$, on sait que les atomes de ϕ sont aussi dans α. On sait que si $q_1 \xrightarrow{\phi} q_2$ est supprimée par l'élagage, c'est au moins à cause d'un atome de ϕ. Comme cet atome est présent dans α, la transition $q' \xrightarrow{\alpha} q$, doit être supprimée dans $\varepsilon^{k+1}_{\mathcal{R}}$: on arrive à la même conclusion que précédemment.

On en déduit donc que le s est aussi réduit dans le $\mathcal{R}_{/E}$-automate $A^{k+1}_{\mathcal{R},E}$ tel que $s \xrightarrow{\alpha}_{A^k_{\mathcal{R},E}} q$, ce qui termine la preuve que $A^{k+1}_{\mathcal{R},E}$ est bien-défini. □

Remarque 3.14.3. P *preserve l'injectivité et le déterminisme de* $\Delta \setminus \Delta^0$*, puisque la procédure d'élagage ne modifie pas les transitions de* Δ*.*

Si maintenant, on considère les formules obtenues obtenues pour chaque triplet (q, q', ϕ), où q est un état final de $A^k_{\mathcal{R},E}$, q' est un état final de A_{Bad} et $\phi = \top$, alors certains termes reconnus par q' dans A_{Bad} sont atteignables. Sinon, ϕ est la formule à invalider, *i.e.* dont il faut rendre faux certains atomes pour que ϕ deviennent $\bot$. A partir de ϕ, on peut élaguer l'automate en utilisant la procédure d'élagage ci-dessus.

3.14.4 Exemple

On reprend le $\mathcal{R}_{/E}$-automate $A^1_{\mathcal{R},E}$ donné dans l'exmple 3.11.6. Pour illustrer la procédure de vérification, on définit l'automate A_{Bad} dont l'état final est q'_0 et dont les transitions sont $a \to q'_1, s(q'_1) \to q'_2, s(q'_2) \to q'_1$ et $f(q'_2) \to q'_0$. Les termes interdits sont donc les termes appartenant au langage $\mathcal{L}(A_{Bad})$ *i.e.* les termes de la forme $f(s^{2k+1}(a))$. En fait le système de réécriture ne peut pas produire ces termes à partir de $A^0_{\mathcal{R},E}$. Cepen-

dant, l'équation $s(s(x)) = s(x)$ utilisée pour calculer $A^1_{\mathcal{R},E}$ produit une approximation trop grossière qui inclut des termes reconnus par A_{Bad}.

On peut donc remarquer que l'intersection $\mathcal{L}(A^1_{\mathcal{R},E}) \cap \mathcal{L}(A_{Bad})$ n'est pas vide. En utilisant l'algorithme proposé en section 3.13, on obtient un triplet (q_0, q'_0, ϕ) pour les états finals de $A^1_{\mathcal{R},E}$ et A_{Bad} où ϕ est la formule $(Eq(q_2, q_3) \wedge Eq(q_3, q_2)) \vee Eq(q_2, q_3) \vee Eq(q_3, q_2)$. On peut remarquer que la première sous-formule de la disjonction implique les deux autres. Par contraposée, il suffit d'appliquer la procédure sur les deux dernières formules pour raffiner suffisamment l'approximation, *i.e.* que l'intersection avec $\mathcal{L}(A_{Bad})$ soit vide.

- $\mathtt{P}(A^1_{\mathcal{R},E}, Eq(q_2, q_3))$: on retire la transition $q_2 \to q_3 \in \varepsilon^1_=$ pour invalider la formule.
- $\mathtt{P}(A', Eq(q_3, q_2))$: on retire la transition $q_3 \to q_2 \in \varepsilon^1_=$ pour invalider la formule.

On obtient alors l'automate $A^2_{\mathcal{R},E}$ avec les ensembles de transitions $\Delta^2 = \Delta^1$, $\varepsilon^2_{\mathcal{R}} = \varepsilon^1_{\mathcal{R}}$ et $\varepsilon^2_= = \emptyset$. On remarque que cette étape d'élagage n'a supprimé aucune transition de $\varepsilon^1_{\mathcal{R}}$, car elle sont toutes étiquetées par la formule $\top$.

Le $\mathcal{R}_{/E}$-automate élagué n'est plus $\mathcal{R}$-clos, il y a de nouveau des paires critiques qui ne sont pas résolues : on relance la procédure de complétion à partir de $A^2_{\mathcal{R},E}$. On construit l'automate $A^3_{\mathcal{R},E} = \mathtt{W}(\mathtt{C}(A^2_{\mathcal{R},E}))$.

La règle de réécriture $f(x) \to f(s(s(x)))$ une paire critique $\langle q_4, f(s(s(q_3))), \top \rangle$ dont la résolution donne les ensembles de transitions $\Delta^3 = \Delta^2 \cup \{s(q_3) \to q_5, s(q_5) \to q_6, f(q_6) \to q_7$, $\varepsilon^3_{\mathcal{R}} = \varepsilon^2_{\mathcal{R}} \cup \{q_7 \xrightarrow{\top} q_4\}$ et $\varepsilon^3_= = \varepsilon^2_=$.

L'étape de complétion terminée, on applique la fonction d'élargissement $\mathtt{W}$ sur $A^3_{\mathcal{R},E}$. On peut voir l'équation $s(x) = s(s(x))$ ajoute directement à $\varepsilon^3_=$ les deux transitions entre les paires d'états (q_2, q_3), (q_3, q_5) et (q_5, q_6). Mais le calcul de la clôture transitive de $\varepsilon^3_=$ permet de produire de nouvelles transitions entre les états (q_2, q_5), (q_3, q_6) et (q_2, q_6). Au total, on se retrouve avec un ensemble $\varepsilon^3_=$ composé de douze transitions.

Le test de vacuité de $\mathcal{L}(A^3_{\mathcal{R},E}) \cap \mathcal{L}(A_{Bad})$ indique que l'intersection n'est pas vide et renvoie un nouveau triplet de la forme (q_0, q'_0, ϕ). On ne donne pas explicitement la formule construite qui est trop importante mais reste calculable. L'étape d'élagage pour cette formule conduit à la suppression de toutes les transitions de $\varepsilon^3_=$ exceptées $q_3 \to q_6$ et $q_6 \to q_3$ qui forment le nouvel ensemble $\varepsilon^4_=$ de l'automate $A^4_{\mathcal{R},E}$. On peut remarquer que ce sont des transitions issues de la clôture qui ne sont pas élaguées. Autrement, comme toutes les transitions de $\varepsilon^3_{\mathcal{R}}$ sont étiquetées par $\top$, l'élagage n'a eu aucun impact sur $\varepsilon^4_{\mathcal{R}} = \varepsilon^3_{\mathcal{R}}$. On tente de reprendre le processus de complétion à partir de l'automate $A^4_{\mathcal{R},E}$, on constate que l'on a atteint un point fixe : $A^4_{\mathcal{R},E} = \mathtt{C}(A^4_{\mathcal{R},E})$. De plus, comme on a $\mathcal{L}(A^4_{\mathcal{R},E}) \cap \mathcal{L}(A_{Bad}) = \emptyset$, avec $\mathcal{L}(A^4_{\mathcal{R},E}) \supseteq \mathcal{R}^*(I)$, on peut conclure que $\mathcal{R}^*(I) \cap Bad = \emptyset$. Pour cet exemple, le raffinement est très fin puisqu'il permet d'inférer l'approximation qui est exactement l'ensemble des termes atteignables *i.e.* $\mathcal{L}(A^4_{\mathcal{R},E}) = f(s^{2*k}(a))$.

Remarque 3.14.5. *Cet exemple ne peut pas être traité par l'approche contenue dans [6] mais peut-être traité par [10]. En effet dans [6], la technique utilisée ne permet pas d'inférer de nouvelles abstractions, seulement retarder l'étape d'abstraction : elle permet seulement de prouver la propriété pour une borne donnée $Bad_k = \{f(s^{2*i+1}(a))sepi < k\}$, sinon la procédure diverge pour la totalité de Bad.*

Remarque 3.14.6. *On peut aussi remarquer que l'algorithme n'est optimal dans la manière de construire l'approximation après l'élagage : on peut remarquer que les transitions entre les états (q_2, q_3) sont ajoutées une deuxième fois lors de la deuxième étape d'accélération à $A^4_{\mathcal{R},E}$ alors que la première étape d'élagage venait de les supprimer dans l'automate $A^2_{\mathcal{R},E}$. C'est inefficace car les transitions sont à nouveau supprimer par l'étape d'élagage, mais pourtant utile, pour construire lors de la clôture pour $\mathtt{W}(A^3_{\mathcal{R},E})$ les transitions entre les couples d'états (q_2, q_5) et (q_2, q_6). On peut quand améliorer la situation en sauvegardant dans un ensemble $\varepsilon'_=$ les transitions de $\varepsilon_=$ supprimés par l'élagage. Lors de l'étape d'accélération suivante, les états en relation par $\varepsilon'_=$ sont alors tout simplement à ignorer lors de l'application des équations dans le $\mathcal{R}_{/E}$-automate. Par contre, il faut tenir des transitions contenues dans ε et ε' pour construire la clôture car si la transition $q_2 \to q_3 \notin \varepsilon_=$ à cause de l'élagage, on veut quand même en déduire $q_2 \to q_5$ à partir de la transition $q_3 \to q_5 \in \varepsilon_=$.*

3.15 $\mathcal{R}_{/E}$-complétion et le calcul exact de termes atteignables

La $\mathcal{R}_{/E}$-completion peut être utilisée pour vérifier des propriétés pour un bon nombre des classes de systèmes de réécriture $\mathcal{R}$ pour lesquels $\mathcal{R}^*(\mathcal{L}(A))$ est un langage régulier. Pour ces classes, la complétion termine toujours avec un $\mathcal{R}_{/E}$-automate $A^*_{\mathcal{R}}$ pour lequel $\varepsilon_=$ est vide. Comme tous les termes reconnus par $A^*_{\mathcal{R}}$ sont atteignables, tous les triplets $(q, q', \phi) \in S$ calculés par le l'algorithme de filtrage are such that $\phi = \top$.

Théorème 3.15.1 (Calcul exact par complétion)**.** *Si $E = \emptyset$ et $\mathcal{R}$ est clos [23, 15], linéaire à droite et monadique [48], linéaire et semi-monadique [21], linéaire et inversement croissant [40], de la constructor [46] ou linear generalized finite path overlapping [49], alors la complétion d'un $\mathcal{R}_{/E}$-automate A termine pour $A^*_{\mathcal{R},\emptyset}$ et $\mathcal{L}(A^*_{\mathcal{R},\emptyset}) = \mathcal{R}^*(\mathcal{L}(A))$.*

Démonstration. Quand $E = \emptyset$, l'algorithme de completion ne produit aucune transition pour l'ensemble $\varepsilon_=$ et, donc, chaque transition de $\varepsilon_{\mathcal{R}}$ ne peut être étiquetée que par la formule $\top$. En d'autres mots, la $\mathcal{R}_{/E}$-completion produit un automate d'arbres ordinaire au lieu d'un $\mathcal{R}_{/E}$-automate. Comme résultat, lorsque l'on n'a pas d'abstraction *i.e.* $E = \emptyset$, l'algorithme proposé dans ce papier coincide totalement avec l'algorithme original [33] pour la complétion d'automates d'arbres. Dans [31], le théorème 114 montre que la complétion d'automates d'arbres [33] termine avec $E = \emptyset$, pour les classes de systèmes listés ci-dessus. De plus, le théorème 45 et 49 toujours dans [33] assurent que dans ces cas-là, $A_{\mathcal{R},\emptyset}$ l'automate obtenu reconnaît exactement l'ensemble des termes atteignables $\mathcal{L}(A^*_{\mathcal{R},\emptyset}) = \mathcal{R}^*(\mathcal{L}(A))$. □

3.16 Conclusion

Cette nouvelle procédure de complétion est en cours d'intégration dans l'outil Timbuk. C'est une étape incontournable pour espérer valider l'approche principalement d'un point de vue des performances, même si l'on suppose que l'absence de calcul en arrière soit en elle-même une nette amélioration. A côté de cela, un certain nombre de points restent encore à considérer. Tout d'abord l'aspect heuristique de l'élagage mérite d'être d'être regardé. En effet, il est précisé que la fonction `P` retire des transitions de l'automate de telle sorte qu'une certaine formule ϕ devienne fausse. A partir du moment où la formule à invalider est une conjonction $\bigwedge_1^n Eq(q_i, q_i')$, retirer n'importe laquelle des transitions $q_i \to q_i'$ est suffisant pour invalider toute la conjonction. Cela laisse une certaine latitude dans la manière d'élaguer l'automate et d'autant plus que l'impact sur le langage n'est pas du tout le même suivant la transition que l'on choisit de supprimer.

Pour donner une idée des conséquences, on peut considérer une chaîne de ε-transitions de $\varepsilon_{\mathcal{R}} \cup \varepsilon_{=}$ dans le $\mathcal{R}_{/E}$-automate A de la forme

$$q_n \xrightarrow{\alpha_n}_A q_{n-1} \xrightarrow{\alpha_{n-1}} \quad \dots \quad \xrightarrow{\alpha_2} q_1 \xrightarrow{\alpha_1} q_0$$

Si l'on raisonne uniquement sur les langages, on sait qu'une ε-transition peut être vue comme la relation d'ordre $\mathcal{L}(A, q_n) \subseteq \mathcal{L}(A, q_{n-1}) \cdots \subseteq \mathcal{L}(A, q_0)$. Du point de vue de q_0, supprimer la transition $q_n \xrightarrow{\alpha_n}_A q_{n-1}$ plutôt que la transition $q_1 \xrightarrow{\alpha_1}_A q_0$ n'élimine moins de termes de q_0. D'autre part, comme les ε-transitions sont le résultat de `C` ou `W`, $q_1 \xrightarrow{\alpha_1}_A q_0$ revient à oublier plus d'étapes de calcul que lorsqu'on supprime $q_n \xrightarrow{\alpha_n}_A q_{n-1}$. On peut donc en déduire que la profondeur de la transition que l'on choisit de supprimer peut avoir un impact sur la puissance ou dualement sur la précision de l'élagage. De même si on considère la transition $f(q_2, q_4) \to q_5$ et les ε-transitions $q_1 \xrightarrow{\alpha_1} q_2$, $q_3 \xrightarrow{\alpha_3} q_4$ et $q_5 \xrightarrow{\alpha_5} q_6$, le choix de la transition à supprimer peut jouer lorsqu'on cherche à invalider la formule $\alpha_1 \wedge \alpha_3 \wedge \alpha_5$. Vaut-il mieux élaguer le $\mathcal{R}_{/E}$-automate en retirant de l'approximation des sous-termes du membre gauche ou du membre droit ? Ou Alors supprimer de la transition $q_5 \xrightarrow{\alpha_5} q_6$, ce qui contribue à élaguer plus fortement le $\mathcal{R}_{/E}$-automate. A partir de la future implémentation, il serait intéressant de monter un protocole expérimental pour évaluer ces premières idées. Cela permettra la mise en place de stratégies d'élagage pour Timbuk. On peut imaginer que certaines stratégies sont plus favorables que d'autres pour aider la complétion avec raffinement à converger. A ce titre, on peut aussi se poser la question de la complétude partielle, *i.e.* sous quelles conditions la procédure est-elle capable de fournir un résultat. On sait déjà que si le système ne valide pas la propriété, alors il existe un terme atteignable qui viole la propriété. Dans ce cas, il suffit de compléter l'automate jusqu'à atteindre le contre-exemple. A l'opposé, lorsqu'il existe un point fixe permettant de vérifier la propriété, la procédure est-elle toujours en mesure de le construire ? Un tel résultat donnerait une précision supplémentaire sur le degré de convergence de la complétion avec raffinement.

Chapitre 4

Preuves de propriétés temporelles sur TRS

Dans sa version originale, la complétion d'automates d'arbres permet de vérifier des propriétés de sûreté par non-atteignabilité. A chaque étape de complétion, on place une ε-transition qui permet de distinguer un terme de ses successeurs. Ici l'idée est d'exploiter cette relation d'ordre pour vérifier des propriétés temporelles du système de réécriture. Il est possible d'extraire de l'automate completé un graphe orienté dénotant une relation de réécriture particulière sur laquelle on peut vérifier des propriétés temporelles en utilisant des techniques de model-checking standard. En particulier, on exploitera l'approche afin de réaliser une analyse interprocédurale sur des programmes Java.

4.1 Du Bytecode Java à la réécriture

Dans des travaux antérieurs [7], il a été montré qu'il est possible de tranduire un programme en ByteCode Java en un jeu de règles de réécriture. Cette transformation est assurée par l'outil *Copster* [5].

Le système de réécriture est généré de telle sorte que les termes modélisent les états de la machine virtuelle Java et l'exécution des instructions ByteCode est représentée par les règles de réécriture. Principalement, un état de la machine virtuelle Java contient le contexte de la méthode courante, le tas, la pile d'appels, les entrées sorties. La figure 4.1 donne un aperçu de la forme des termes qui modélisent les états. On peut voir que le contexte d'appel d'une méthode est un quadruplet $\langle m_i, pc_i, st_i, lv_i, heap\rangle$ où m_i représente la méthode en cours d'exécution, pc_i le point programme courant dans la méthode m_i, st_i la pile d'opérandes et lv_i la liste des variables locales. On joint au contexte de la méthode courante le tas *heap*. On ne rentrera pas plus dans les détails de la constitution du tas, mais intuitivement il s'agit d'un tableau qui contient les différents objets et tableaux stockés au cours de l'exécution du programme. La pile d'appels est une simple liste dont les éléments sont les contextes des méthodes appelantes qui sont en attente. On notera que la modélisation des états prend en compte les différentes entrées sorties avec lesquelles le

programme courant est en mesure d'intéragir, mais on ne s'attardera pas plus sur ce point détaillé dans la documentation de *Copster* [5].

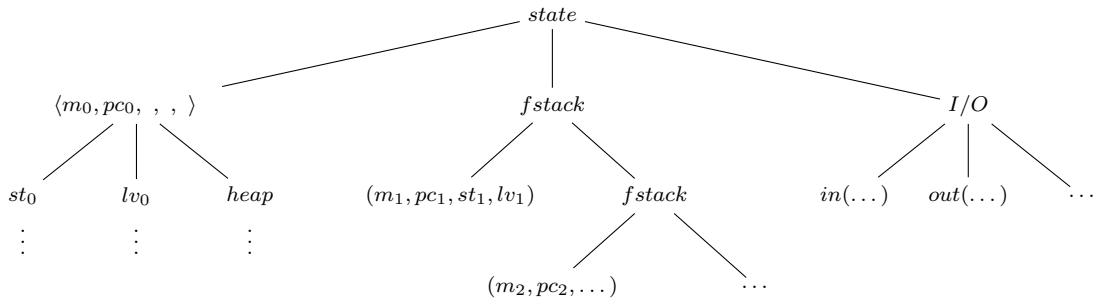


FIGURE 4.1: Un état de la JVM représenté comme un terme

L'outil *Copster* produit deux types de règles de réécriture à partir du programme donné en entrée. Une partie des règles est propre au programme : ces règles permettent d'établir le séquencement des instructions de chaque méthode. Le seconde catégorie de règles est propre à la sémantique opérationnelle de Java qui modélisent les différentes instructions du langage, et d'autre part un ensemble de règle modélisant le programme. Elle constitue une base fixe et commune à tous les programmes modélisés sous forme par les systèmes de réécriture générés. La figure 4.2 donnent deux exemples de règles sémantiques pour des instructions du ByteCode Java.

$$(pop)\ \frac{(m, pc, x :: s, l)}{(m, pc+1, s, l)} \qquad (store_i)\ \frac{(m, pc, x :: s, l)}{(m, pc+1, s, x \rightarrow_i l)}$$

FIGURE 4.2: Extrait de la sémantique opérationnelle du bytecode Java

On peut noter qu'en fonction de la nature des instructions, les règles de la sémantique n'ont d'impact que sur une partie de l'état de la machine virtuelle. Ainsi, la majorité des instructions (`store`$_i$, `aload_0`, `goto`, `if_icmpe`, ...) du ByteCode, ne portent que sur le contexte de la méthode courante et/ou sur le tas (`get_field`, `put_field`. Cependant, les instructions dédiées aux appels et retour de méthodes (`invokeVirtual`, `return`, ...) modifient la pile d'appel en plus du contexte de la méthode courante et concerne une plus grande partie de l'état de la machine virtuelle.

Le système de réécriture généré par *Copster* pour un programme donné assure un certain nombre de bonnes propriétés issues de la sémantique. Principalement, le système de réécriture produit est toujours linéaire à gauche, déterministe, et préserve la séquentialité des instructions de telle sorte qu'il n'existe qu'une seule manière de réécrire chaque terme atteignable. Le comportement de chaque instruction est simulé par un ensemble de règles équivalent à la règle sémantique. On peut voir dans la figure 4.3 qu'il suffit de trois règles pour exécuter l'instruction au point de programme pc_1 de la méthode `C.m1()`. La règle (2) est la simple traduction de la règle sémantique de `pop`, alors que les règles (1) et (3)

sont dédiées à la méthode `C.m1()`, et donc propres au programme. La règle (1) permet de lier au point de programme pc_1 de `C.m1()` l'instruction correspondante : `pop`. On voit que la règle (1) convertit le contexte en un contexte intermédiaire $xframe(\dots)$ surlequel est défini la sémantique des instructions. Une fois l'exécution terminée, un nouveau contexte $frame(\dots)$ est produit. La règle (3) incrémente simplement l'index de l'instruction à exécuter.

1	$frame(name(m1, C), pc1, st, lv, h)$	$\longrightarrow xframe(pop, name(m1, C), pc_1, st, lv, h)$
2	$xframe(pop, m, pc, stack(x, s), l, h)$	$\longrightarrow frame(m, next(pc), s, l, h)$
3	$next(pc_1)$	$\longrightarrow pc_2$

FIGURE 4.3: Règles pour l'instruction `pop` de la méthode `C.m1()`

Un point crucial pour la suite est la possibilité de déterminer une classification des règles. En effet, la position à laquelle peut réécrire une règle de réécriture dans un terme détermine la nature sémantique de la règle. En se focalisant sur une position donnée du terme, on peut alors analyser les séquences de réécriture correspondant aux graphes d'appels des méthodes Java, les opérations réalisées sur le tas *etc...* On va donc montrer comment exploiter cette propriété du langage pour réaliser une analyse inter-procédurale du flot de contrôle pour des programmes Java.

4.1.1 Chaîne de réécriture à la position p

On considère $\mathcal{R}$ le système de règles de réécriture, et un ensemble d'équations E. On va définir $\dashrightarrow_{\mathcal{R}_p/E}$ une sous relation de $\rightarrow^*_{\mathcal{R}/E}$ la relation de réécriture induite par $\mathcal{R}_{/E}$. Le but de la relation $\dashrightarrow_{\mathcal{R}_p/E}$ est de caractériser la réécriture uniquement par les étapes de réécriture qui ont lieu à la position p.

Définition 4.1.2. *Soient $s,\ t \in \mathcal{T}(\mathcal{F})$ deux termes tels que $s \rightarrow^*_{\mathcal{R}/E} t$. Soit $p \in \mathcal{P}os(s)$ une position de s. Alors on a $s \dashrightarrow_{\mathcal{R}_p/E} t$ si et seulement si il existe un terme w qui satisfait les conditions suivantes :*

- $s \rightarrow^*_{\mathcal{R}/E} w$
- *w doit être obtenu par réécriture uniquement de sous-termes stricts de $s|_p$.*
- $t =_E w[r\sigma]_p$ *avec* $l \rightarrow r \in \mathcal{R}$ *et* $w|_p = l\sigma$

Dans le cas où l'ensemble des équations est vide, on note par $\dashrightarrow_{\mathcal{R}_p}$ la relation $\dashrightarrow_{\mathcal{R}_p/\emptyset}$. De la même manière on utilise la notation $\dashrightarrow_{\mathcal{R}/E}$ pour $p = \epsilon$. Ce qui nous donne au final, la notation $\dashrightarrow_{\mathcal{R}}$ dans le cas $p = \epsilon$ et $E = \emptyset$. On voit par l'intermédiaire de l'exemple 4.1.3, que la relation $\dashrightarrow_{\mathcal{R}_p}$ permet d'abstraire la séquence de réécriture : les différents étapes de réécriture nécessaires pour arriver à l'étape de réécriture à la position p sont simplement oubliées. De ce fait, on ne conserve que les termes qui sont produits par la réécriture à la position p.

Exemple 4.1.3. *On considère la séquence de réécriture suivante. A chaque étape, le sous-terme réécrit est souligné, et le sous-terme produit est en gras.*

$$f(g(\underline{a}), h(b)) \longrightarrow_{\mathcal{R}} f(\underline{g(\mathbf{b})}, h(b)) \longrightarrow_{\mathcal{R}} f(\mathbf{h(\underline{b})}, h(b)) \longrightarrow_{\mathcal{R}}$$

$$f(\mathbf{a}, h(\underline{b})) \longrightarrow_{\mathcal{R}} f(a, \underline{h(\mathbf{c})}) \longrightarrow_{\mathcal{R}} \underline{f(a, \mathbf{a})} \longrightarrow_{\mathcal{R}} \mathbf{g(f(a,\ a))}$$

conformément à la définition 4.1.2 on peut exhiber les relations suivantes :

- $f(g(a), h(b)) \dashrightarrow_{\mathcal{R}_1} f(h(b), h(b))$
- $f(a, h(b)) \dashrightarrow_{\mathcal{R}_2} f(a, a)$
- $f(g(a), h(b)) \dashrightarrow_{\mathcal{R}} g(f(a, a))$

Dans le cas où le système de réécriture modélise l'exéction d'un programme Java, la position de réécriture permet de caractériser la nature sémantique l'étape de réécriture. Si l'on considère s un état conforme à la figure 4.1 aux positions suivantes, on peut considérer :

- Le sous-terme $s|_1 = \langle m_0, pc_0, \dots \rangle$ est le contexte de la méthode en cours. La réécriture à la position 1 correspond au début de l'exécution des instructions du byteCode.
- La réécriture aux positions $s|_{13}$ et $s|_{14}$ correspond aux opérations élémentaires comme la modification de la pile ou des variables locales...
- Plus intéressant, le sous-terme $s|_{15}$ correspond au tas : les règles de réécriture appliquées à cette position concerne la consultation, la création ou la modification des objets ou tableaux stockés en mémoire.
- Le sous-terme $s|_\epsilon$ soit s lui-même, est réécrit par les règles qui correspondent aux appels et retours de méthodes.
- ...

On considère maintenant un état de la machine virtuelle correspondant au début de l'exécution d'une méthode m donné par le contexte $f_0 = \langle m, pc_0, \epsilon, lv_0, h_0 \rangle$, une certaine pile d'appels fs et des entrées-sorties de la forme $I/O(\dots)$. Cet état est la forme $s_0 = state(f_0, fs, I/O(\dots))$. Si on déroule l'exécution de la méthode pas-à-pas de l'instruction $\texttt{instr}_i$ au point de programme pc_i on obtient un nouveau contexte f_{i+1} par réécriture. L'exécution de l'instruction débute par la réécriture de l'état s_i à la position 1, ce qui construit un état intermédiaire, sur lequel a lieu toutes les étapes de réécriture nécessaires pour réaliser les différentes manipulations (sur les variables, la pile, le tas) induites par l'exécution de l'instruction $\texttt{instr}_i$. Comme indiqué ci-dessus, ces étapes de réécriture ont lieu uniquement sur les sous-termes de f_i :

$$state(\underline{f_i}, fs, I/O(\dots)) \longrightarrow^*_{\mathcal{R}} state(\mathbf{f_{i+1}}, fs, I/O(\dots))$$

Si le point de programme p_n correspond à un appel de la méthode m' qui va produire un nouvel état de la forme $state(f'_0, fs', I/O(\dots))$ où le contexte est de la forme $f'_0 = \langle m', pc_0, \epsilon, lv'_0, h_n \rangle$ et la pile $fs' = fstack((m, pc_n, st_n, lv_n), fs)$. Au final, on obtient la

séquence de réécriture suivante :

$$state(\underline{f_0}, fs, I/O(\dots)) \longrightarrow^*_{\mathcal{R}} \underline{state(\mathbf{f_n}, fs, I/O(\dots))} \longrightarrow_{\mathcal{R}} \mathbf{state}(\mathbf{f'_0}, \mathbf{fstack}((\mathbf{m}, \mathbf{pc_n}, \mathbf{st_n}, \mathbf{lv_n}), \mathbf{fs}), \mathbf{I/O}(\dots))$$

qui peut être abstraite :

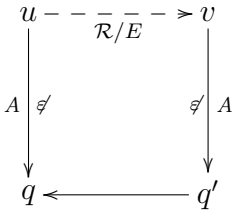
$$state(f_0, fs, I/O(\dots)) \quad \dashrightarrow_{\mathcal{R}} \quad state(f'_0, fstack((m, pc_n, st_n, lv_n), fs), I/O(\dots))$$

Puisque f_0 et f'_0 représentent respectivement les états au point d'entrée des méthodes m et m', la relation ci-dessus dénote bien à l'appel de la méthode m' par m. Ainsi, les traces d'exécutions abstraites par la relation $\dashrightarrow_{\mathcal{R}}$ permettent de construire le graphe d'appels d'un programme Java. Ce graphe interprocédural peut-être obtenu par la complétion d'un automate d'arbres. En fait l'automate étant souvent obtenu par une sur-approximation paramétrée par E un ensemble d'équations, il caractérise un sous-ensemble de la relation plus générale $\dashrightarrow_{\mathcal{R}_p/E}$.

4.2 Completion d'automates d'arbres et $\dashrightarrow_{\mathcal{R}_p/E}$

La relation $\dashrightarrow_{\mathcal{R}_p/E}$ que l'on cherche à caractériser est en fait construite et maintenue naturellement par la complétion. Elle est intrinsèquement liée à la nature des automates d'arbres et finalement à la manière dont sont ajoutées les ε-transitions par les étapes de complétion. En effet, chaque ε-transition de la forme $q' \to q$ met en relation les représentants de q' et q par $\dashrightarrow_{\mathcal{R}_\epsilon/E}$. Ce qui nous donne la propriété suivante :

Lemme 4.2.1. *Soit $A = \langle \mathcal{F}, Q, Q_f, \Delta \cup \varepsilon \rangle$ un automate obtenu par complétion à partir des règles $\mathcal{R}$ et des équations E. Alors pour toute transition $q' \longrightarrow q \in \varepsilon$ de A, et tous les représentants de q et q' :*

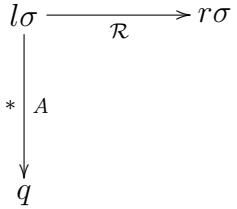


Démonstration. La preuve se base sur la définition 2.4.15 de la $\mathcal{R}_{/E}$-cohérence qui est plus forte. Elle assure que pour tout état q d'un automate produit par complétion, si $s \in \mathcal{R}ep(q)$ un représentant de l'état q alors :

- $\forall t \in \mathcal{R}ep(q)$, $s =_E t$
- $\forall t \in \mathcal{L}(A, q)$, $s \to^*_{\mathcal{R}_{/E}} t$.

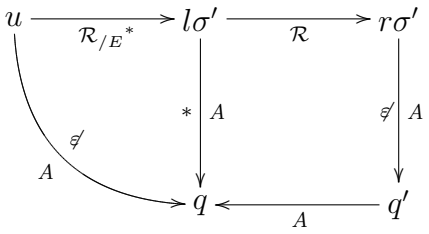
On considère la transition $q \to q' \in \varepsilon$. Par construction, cette ε-transition est ajoutée par la résolution de la paire critique $\langle q, r\sigma \rangle$ formée par l'automate avec une règle $l \to r \in \mathcal{R}$.

On rappelle que la substitution σ associe à chaque variable x du membre gauche l un état $\sigma(x) \in Q$ telle que :



Une telle paire critique a été résolue en ajoutant à l'automate les transitions $r\sigma \to^{\not\epsilon} q'$ et $q' \to q$. Ensuite on considère la subsitution $\sigma' : \mathcal{X} \to \mathcal{T}(\mathcal{F})$ de la forme $\{x \mapsto \sigma'(x) \mid \sigma'(x) \in \mathcal{R}ep(\sigma(x))\}$. On a automatiquement $l\sigma' \to_A^* l\sigma$: l'ensemble des substitutions σ' dénotent les termes $l\sigma'$ reconnus en q qui sont réécrits par la règle $l \to r$ tels que $r\sigma'$ est un représentant de q' puisque l'on a $r\sigma' \to_A^{\not\epsilon} r\sigma$.

Grâce au premier point de la définition 2.4.15 et la résolution de la paire critique on obtient le diagramme suivant :



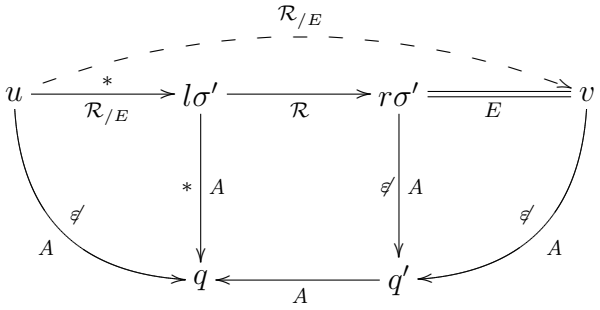
où u est un terme représentant quelconque de q. On obtient donc une chaîne de réécriture entre les termes u et $r\sigma'$ dont la dernière étape de réécriture est réalisée à la position ϵ. De plus, u et $r\sigma'$ sont respectivement des représentants de états q et q'.

Pour établir la relation $u \dashrightarrow_{\mathcal{R}_{/E}} r\sigma'$, il suffit de montrer qu'il n'y a aucune étape de réécriture située à la position ϵ sur $u \to_{\mathcal{R}_{/E}}^* l\sigma'$. Pour cela, il faut regarder une caractéristique de l'algorithme de filtrage : si σ est une solution du problème de filtrage $l \trianglelefteq q$, alors on sait que $l\sigma \to_A^* q$. Cependant, pour des raisons d'optimalité, la dernière transition utilisée pour réécrire $l\sigma$ en q, est une transition normalisée de Δ, donc de la forme $f(q_1, \ldots, q_n) \to q$. Il est suffisant de considérer ce cas pour que la paire critique soit aussi résolue pour tous les états dans lesquels q peut-être réécrit : avoir $r\sigma \to_A^* q$ est suffisant pour obtenir la transition $r\sigma \to_A^* q''$ pour toute paire critique $\langle l\sigma, q'' \rangle$ avec $q \to_A^* q''$ à partir d'une transition de la forme $l\sigma \to_A^* q \to_A^* q''$.

On peut donc en déduire que tout terme $l\sigma'$ est un terme de la forme $f(t_1, \ldots, t_n)$ tels que pour chaque sous-terme on ait $t_i \to_A^* q_i$. D'après le premier point de la $\mathcal{R}_{/E}$-cohérence, on peut construire un terme s_i représentant de q_i tel que $s_i \to_{\mathcal{R}_{/E}}^* t_i$. Par composition, $f(s_1, \ldots, s_n)$ est un représentant de l'état q : on a $f(s_1, \ldots, s_n) \to_A^{\not\epsilon} f(q_1, \ldots, q_n)$ et $f(q_1, \ldots, q_n) \to q \in \Delta$. On peut produire le terme $l\sigma'$ par réécriture des sous-termes stricts de $f(s_1, \ldots, s_n)$.

De plus, comme le terme u, le terme $f(s_1, \ldots, s_n)$ est un représentant de q. En adéquation avec le second point de la $\mathcal{R}_{/E}$-cohérence, on peut en déduire que $u =_E$

$f(s_1, \ldots, s_n)$. De même $r\sigma'$ est un représentant de q' : donc pour tout représentant v de q' on a $r\sigma' =_E v$ Finalement, cela permet de conclure la preuve puisqu'on a $u \to^*_{\mathcal{R}/E} l\sigma' \to_{\mathcal{R}} r\sigma' =_E v$ avec $l\sigma'$ obtenu uniquement par des étapes de réécriture sur des sous-termes stricts :



pour tout $u \in \mathcal{R}ep(q)$ et $v \in \mathcal{R}ep(q')$. □

De cette propriété découle immédiatement la généralisation pour n'importe quelle position. En effet, on considère un terme $t \in \mathcal{L}(A, q)$ tel que $t_p \in \mathcal{R}ep(q_1)$ et $t[q_1]_p \to^*_A q$: pour toute transition $q_2 \to q_1 \in \varepsilon$ alors pour tout $u_1 \in \mathcal{R}ep(q_1)$ on a $t \dashrightarrow_{\mathcal{R}_p/E} t[u_1]_p$. Ainsi à partir de la chaîne de ε-transitions $q_1 \leftarrow q_2 \leftarrow q_3 \leftarrow q_4 \leftarrow \ldots$, on peut déduire la séquence

$$t \dashrightarrow_{\mathcal{R}_p/E} t[u_1]_p \dashrightarrow_{\mathcal{R}_p/E} t[u_2]_p \dashrightarrow_{\mathcal{R}_p/E} t[u_3]_p \dashrightarrow_{\mathcal{R}_p/E} t[u_4]_p \quad \ldots$$

avec $u_i \in \mathcal{R}ep(q_i)$.

4.3 Extraction d'une structure de Kripke

Soit $A^*_{\mathcal{R},E} = \langle \mathcal{T}(\mathcal{F}), Q, Q_F, \Delta \cup \varepsilon \rangle$ un automate d'arbres obtenu par complétion pour $\mathcal{R}$ un système de réécriture donné, avec un ensemble d'équation E, à partir du langage initial I. On suppose que $A^*_{\mathcal{R},E}$ est $\mathcal{R}$-clos soit $A^*_{\mathcal{R},E} \supseteq \mathcal{R}(A^*_{\mathcal{R},E})$.

Une structure de Kripke est un graphe orienté utilisé dans le model-checking pour représenter le comportement d'un système. Les noeuds représentent les états accessibles du système et les arcs représentent les transitions du système. Une fonction d'étiquetage associe à chaque état l'ensemble des propriétés de cet état.

Définition 4.3.1. *Une structure de Kripke est un quadruplet de la forme $K = (S, S_0, R, L)$ tel que*

- *S est un ensemble d'états,*
- *$S_0 \subseteq S$ est l'ensemble des états initiaux,*
- *$R \subseteq S \times S$ est la relation de transition qui doit être totale à gauche,*
- *L est une fonction qui étiquette chaque état s avec un ensemble de prédicats qui sont vrais dans l'état s.*

On veut construire un tel modèle pour le sous-ensemble de la relation $\dashrightarrow_{\mathcal{R}_p/E}$ contenu par l'automate $A^*_{\mathcal{R},E}$ à partir d'un ensemble de termes $t_0 \in \mathcal{L}(A^*_{\mathcal{R},E})$. Cependant à cause du lemme 4.2.1, les sous-termes initiaux $t_0|_p$ concernés par $\dashrightarrow_{\mathcal{R}_p/E}$ doivent être des représentants de l'états qui les accepte :

$$t_0 \to^{\not\varepsilon}_{A^*_{\mathcal{R},E}} t_0[q]_p \to^*_{A^*_{\mathcal{R},E}} q_f, \quad \text{avec } q_f \in Q_f$$

On peut alors ramener le problème d'analyse de $\dashrightarrow_{\mathcal{R}_p/E}$ à l'analyse de $\dashrightarrow_{\mathcal{R}_p/E}$ pour les sous-termes $t_0|_p$. Ces sous-termes sont des représentants pour l'état q, et l'acceptation de $t_0|_p$ par l'état q est une étape dans la reconnaissance de t_0 par l'automate. On note Q_0 l'ensemble de ces états q.

Pour construire le modèle de Kripke, on prend les états de l'automate comme états de la structure de Kripke, la fonction de transition R est définie au moyen des ε-transitions. Enfin l'ensemble des prédicats attachés à chaque état est défini par l'ensemble des représentants associés à l'état. On définit alors la fonction de *labelling* L simplement par une fonction qui associe un automate d'arbres à chaque état.

Définition 4.3.2. *On définit la fonction de labelling $L : q \mapsto \langle \mathcal{F}, Q, \{q\}, \Delta \rangle$ comme la fonction qui associe à un état q, l'automate $L(q)$ issu de l'automate $A^*_{\mathcal{R},E}$ privé des ε-transitions et pour lequel q est l'unique état final. Par construction, on peut montrer que les termes reconnus par l'automate $L(p)$ sont exactement les représentants de l'états q dans l'automate complet $A^*_{\mathcal{R},E}$.*

$$\forall t \in \mathcal{L}(L(q)), \quad t \to^{\not\varepsilon}_{A^*_{\mathcal{R},E}} q$$

Ce qui permet alors de construire la structure de Kripke correspondant à la relation $\dashrightarrow_{\mathcal{R}_p/E}$ pour l'ensemble des termes t_0.

Définition 4.3.3. *On construit le quadruplet (S, S_0, R, L) à partir de l'automate $A^*_{\mathcal{R},E}$. Chaque composante du quadruplet se définit par :*

- $S = Q$,
- $S_0 = Q_0$,
- $R(q, q')$ *si* $q' \to q \in \varepsilon$
- *la fonction de labelling L telle que définie ci-dessus.*

Cependant, une structure de Kripke doit être constituée d'une relation R qui soit totale à gauche. Ainsi pour n'importe quel état q qui n'a pas de successeur par R, on le fait boucler sur lui-même de façon à avoir $R(q, q)$. Cette transformation relativement classique provient du fait que la logique temporelle CTL* utilisée pour formaliser les propriétés reposent sur des modèles dont les traces d'exécution sont infinies.

Dans notre analyse inter-procédurale, ce sont les étapes de réécriture au sommet des termes qui caractérisent les appels et retours de méthodes dans les programmes Java. En supposant que l'ensemble des états finals de $A^*_{\mathcal{R},E}$ est Q_f, on veut donc obtenir un modèle pour la relation de transition induite par la relation $\dashrightarrow_{\mathcal{R}/E}$, pour les termes $t_0 \in I$

l'ensemble initial. Ce qui implique que l'ensemble des états initiaux S_0 correspond simplement à Q_f, l'ensemble des états finals de l'automate. En effet, tout terme initial de I est systématiquement un représentant d'un état final par construction.

La structure de Kripke obtenue modélise la relation de réécriture $\dashrightarrow^*_{\mathcal{R}/E}$ à la position p à partir des sous-termes $t_0|_p$.

Théorème 4.3.4. *Soit $K = (S, S_0, R, L)$ la structure de Kripke extraite à partir de l'automate $A^*_{\mathcal{R},E}$. Pour tous les états s, s' tels que $R(s, s')$ soit vraie, alors pour tous les termes $u \in \mathcal{L}(L(s))$ et $v \in \mathcal{L}(L(s'))$ on a $u \dashrightarrow_{\mathcal{R}/E} v$. De plus, pour tout terme u, $v \in \mathcal{L}(L(s))$ de tout état s, on a $u =_E v$.*

Démonstration. Ce théoréme est la conséquence immédiate du lemme 4.2.1 appliqué à l'automate $A^*_{\mathcal{R},E}$ et de la définition de la relation de transition R. □

Exemple 4.3.5. *Pour illustrer ce résultat, on propose de considérer l'automate d'arbres suivant obtenu par complétion à partir du système de réécriture $\mathcal{R}$ défini par $\{a \to b,\ b \to c\} \cup \{f(c) \to g(a),\ g(c) \to h(a),\ h(c) \to f(a)\}$. On part de l'ensemble initial $I = \{f(a)\}$. Sans approximation, i.e. $E = \emptyset$, on obtient l'automate point fixe suivant :*

$$A^*_{\mathcal{R}} = \left\langle Q_F = \{q_f\}, \quad \Delta = \left\{ \begin{array}{rcl} a & \to & q_a \\ b & \to & q_b \\ c & \to & q_c \\ f(q_a) & \to & q_f \\ g(q_a) & \to & q_g \\ h(q_a) & \to & q_h \end{array} \right\} \varepsilon = \left\{ \begin{array}{rcl} q_b & \to & q_a \\ q_c & \to & q_b \\ q_g & \to & q_f \\ q_h & \to & q_g \\ q_f & \to & q_h \end{array} \right\} \right\rangle$$

Si on regarde la transition $q_h \to q_g$, et les termes $h(a)$ et $g(a)$ représentants de q_h et q_g respectivement, on en déduit $g(a) \dashrightarrow_{\mathcal{R}} h(a)$. Ce qui correspond bien à $g(\underline{a}) \to_{\mathcal{R}} g(\underline{\boldsymbol{b}}) \to_{\mathcal{R}} \underline{g(\boldsymbol{c})} \to_{\mathcal{R}} \boldsymbol{h(a)}$. Les figures 4.4 et 4.5 montrent l'extraction de la structure de Kripke à partir de l'automate pour les relation $\dashrightarrow_{\mathcal{R}_1}$ et$\dashrightarrow_{\mathcal{R}_\epsilon}$ respectivement.

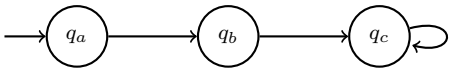


FIGURE 4.4: K_1 modélise $\dashrightarrow_{\mathcal{R}_1}$

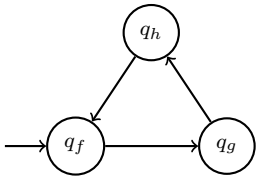


FIGURE 4.5: K_ϵ modélise $\dashrightarrow_{\mathcal{R}_\epsilon}$

4.4 Une logique temporelle portant des automates d'arbres

A partir du moment où l'on obtenu un modèle de Kripke $K = (S, S_0, R, L)$, on a accès à l'ensemble des techniques exploitant ce type de modèle [25]. la logique temporelle

CTL* est une logique permettant de raisonner sur les structures de Kripke, elle constitue donc la base des techniques du model-checking. Cette logique est basée sur le dépliage des traces d'exécution possibles à partir d'un état initial du modèle. Une trace d'exécution est caractérisée par une séquence infinie d'états de la forme $\pi = s_0, s_1, s_2, \ldots$ telle que $(s_i, s_{i+1}) \in R$. On utilise les notations $\pi(i)$ et π^i pour dénoter respectivement le $i^{\text{ème}}$ état s_i de π, et le suffixe de π commençant à l'état s_i. Dans la suite, on va s'intéresser à la logique temporelle linéraire LTL, sous-ensemble de CTL*, pour illustrer l'adaptation des techniques de vérifications sur les modèles extraits à partir des automates complétés.

4.4.1 La logique LTL, et les automates d'arbres

Une des particularité de cette approche est l'étiquetage des états du modèle par des automates d'arbres. Initialement, la logique temporelle est définie sur les différents opérateurs logiques et temporels ($\wedge$, $\vee$, $\neg$, **X**, **F**, **G**, **U**, **R**) à partir de prédicats qui consituent les formules atomiques. Si la sémantique des différents opérateurs est préservée, on modifie celles des formules atomiques représentées les prédicats étant alors représenté par des automates d'arbres. On propose de définir le Regular Linear Temporal Logic (R-LTL). La R-LTL est la logique LTL où les prédicats sont définis par des automates d'arbres. Le langage d'un tel automate d'arbres caractérise l'ensemble des termes qui sont admissibles pour le prédicat.

Ainsi un état q d'une structre de Kripke K valide le prédicat P caractérisé par l'automate d'arbres A_p si et seulement si un terme t reconnu par l'automate $L(q)$ satisfait le prédicat P. Cela implique que le terme t soit lui aussi reconnu par l'automate A_P. Ce qui nous donne la nouvelle règle sémantique :

$$K,\ q \models P \quad \Longleftrightarrow \quad \mathcal{L}(L(q)) \cap \mathcal{L}(A_P) \neq \emptyset$$

L'intuition est que le langage de l'automate A_p décrit une propriété par un ensemble de termes possibles. On peut alors reformuler la satisfiabilité du prédicat P pout t par une disjonction de la forme $\bigvee_{u \in \mathcal{L}(A_p)} t = u$. Cela représente un avantage notable pour exprimer des motifs réguliers sur les termes. En effet, il est facile de construire un automate qui décrit une propriété par tous les termes de la forme $f(a, f(x, y))$. Quant à l'automate $L(q)$, il fournit l'ensemble des termes qui sont accessibles par $\dashrightarrow_{\mathcal{R}/E}$. De ce fait, il est alors raisonnable de considérer que l'état q satisfait le prédicat P si l'un des termes reconnus par $L(q)$ est aussi dans le langage de A_p, ce qui est équivalent à $\mathcal{L}(L(q)) \cap \mathcal{L}(A_P) \neq \emptyset$. La figure 4.6, présente un sous-ensemble des connecteurs logiques de LTL. On peut y distinguer deux types de formules, celles comme f, f_1, f_2 qui portent sur les états et les formules comme g qui portent sur une séquence. Dans le chapitre 3 de [25], on trouve le détail de la sémantique de tous les opérateurs, ainsi que les règles de construction des formules en LTL.

Les transitions de la structure K dénotent les étapes de réécriture abstraite $\dashrightarrow_{\mathcal{R}/E}$ plutôt que sur la relation de réécriture $\rightarrow_{\mathcal{R}/E}$. Comme la sémantique des opérateurs est définie sur la relation de transitions de K, les propriétés que l'on peut exprimer et vérifier portent bien sur la relation $\dashrightarrow_{\mathcal{R}/E}$.

$K, s \models \neg f$	$\iff$	$K, s \not\models f$
$K, s \models f_1 \vee f_2$	$\iff$	$K, s \models f_1$ ou $K, s \models f_2$
$K, s \models f_1 \wedge f_2$	$\iff$	$K, s \models f_1$ et $K, s \models f_2$
	$\ldots$	
$K, \pi \models f$	$\iff$	$K, \pi(0) \models f$
$K, \pi \models \mathbf{X}g$	$\iff$	$K, \pi^1 \models g$
$K, \pi \models \mathbf{F}g$	$\iff$	$\exists k \geq 0,\ K\pi^k \models g$
$K, \pi \models \mathbf{G}g$	$\iff$	$\forall i \geq 0,\ K\pi^i \models g$
	$\ldots$	

FIGURE 4.6: Principaux connecteurs logiques de LTL

Par exemple, la formule [1] $\mathbf{G}(\{f(a)\} \implies \mathbf{X}\{g(a)\})$ sur K_ϵ (*c.f.* figure 4.5) : la formule doit être interprétée comme pour tout q q', si $K_\epsilon,\ q \models \{f(a)\}$ et $R(q, q')$ alors on a $K_\epsilon,\ q' \models \{g(a)\}$. D'un point de vue de la réécriture par $\mathcal{R}$ (exemple 4.3.5), le seul terme u tel que $f(a) \dashrightarrow_{\mathcal{R}_\epsilon} u$ est $u = g(a)$.

4.4.2 Les automates de Büchi

Les propriétés en logique LTL peuvent être vérifiées au moyen de la théorie des automates de Büchi. Les automates de Büchi [16] sont des automates finis qui reconnaissent des mots de taille infinie. Les mots infinis vont permettre de représenter les comportements du modèle dont les traces d'exécution sont infines. La structure des automates de Büchi est équivalente à celle des automates de mots finis.

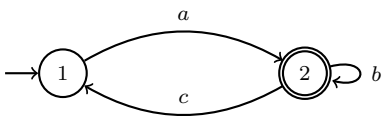
Définition 4.4.3. *Un automate de Büchi se définit comme un quintuplet de la forme* $B = \langle \Sigma, Q, \Delta, Q_0, \mathcal{F} \rangle$ *avec*

- Σ*, un alphabet*
- Q*, un ensemble d'états*
- $\Delta \subseteq Q \times \Sigma \times Q$*, la relation de transition*
- $Q_0 \subseteq Q$ *et* $\mathcal{F} \subseteq Q$*, les états initiaux et acceptants respectivement.*

Un mot infini $w \in \Sigma^\omega$ est un mot qui contient un nombre infini de répétitions [2]. Il est accepté par un automate de Büchi si et seulement si il existe une exécution infinie à partir d'un état initial et qui passe infiniment souvent par un état acceptant.

Exemple 4.4.4. *Exemple d'automate de Büchi :*

1. Par soucis de simplicité dans les petits exemples, plutôt que de représenter chaque prédicat par un automate, on représente directement l'ensemble des termes que devrait accepter l'automate caractérisant le prédicat.
2. La répétition infinie est caractérisée par l'exposant ω



L'automate ci-contre reconnaît le langage ω-régulier $a(b^*ca)^\omega$, soit des mots infinis de la forme acabbbbcabbca... par exemple.

Pour vérifier que le modèle K statisfait la propriété P exprimée en R-LTL, il faut construire deux automates de Büchi B_K et B_P qui acceptent respectivement tous les comportements du modèle, et tous les comportements spécifiés par P. La procédure de vérification revient alors à s'assurer que tous les comportements du modèle K sont couverts par la spécification P, ce qui se traduit par $\mathcal{L}(B_K) \subseteq \mathcal{L}(B_P)$. La manière standard de vérifier l'inclusion revient à vérifier qu'aucun comportement de K ne viole P :

$$\mathcal{L}(B_K) \cap \overline{\mathcal{L}(B_P)} = \emptyset$$

Bien sûr, la vacuité du langage est décidable pour les automates de Büchi, et ils sont clos par intersection et complémentation [16].

Il suffit de construire l'automate B_K et l'automate $\overline{B_P}$ qui reconnaît le langage $\overline{\mathcal{L}(B_P)}$. L'alphabet Σ est constitué par les langages réguliers de termes de $\mathcal{T}(\mathcal{F})$ dénotés par les automates d'arbres.

Définition 4.4.5. *A partir du modèle $K = (S, S_0, R, L)$, on définit l'automate de Büchi qui reconnaît tous les comportements de K comme $B_K = \langle \Sigma, S \cup \{\iota\}, \Delta, \{\iota\}, S \cup \{\iota\} \rangle$ tel que $\Sigma = 2^{\mathcal{T}(\mathcal{F})}$, l'état initial est $\iota \notin S$, et on a les transitions $(s, L(s'), s') \in \Delta$ si et seulement si $(s, s') \in R$. De plus, on ajoute à Δ les transitions $(\iota, L(s), s)$ pour tout état $s \in S_0$.*

Par construction tous les états de B_K sont finals, pour permettre à tout chemin infini de la structure de Kripke d'être un mot reconnu par B_K. Si $\pi = s_0 s_1 s_2 s_3 \ldots$ est une séquence valide d'états dans la structure de Kripke, alors le mot $\pi' = L(s_0)L(s_1)L(s_2)\ldots$ est reconnu par B_K. Ce mot π' dénote l'ensemble de séquence de réécriture de la forme $t_0 \dashrightarrow_{\mathcal{R}/E} t_1 \dashrightarrow_{\mathcal{R}/E} t_2 \dashrightarrow_{\mathcal{R}/E} \ldots$ tels que $t_i \in \mathcal{L}(L(s_i))$.

Exemple 4.4.6. *La figure 4.7 définit l'automate de Büchi qui pour la figure de Kripke K_ϵ :*

FIGURE 4.7: Construction l'automate B_{K_ϵ} pour le modèle K_ϵ

On ne s'étendra pas ici sur les techniques permettant de contruire l'automate de Büchi $\overline{B_P}$ dénotant la négation de la formule LTL P, mais il est possible de trouver les détails de l'algorithme pour construire ce type d'automate dans [35].

Exemple 4.4.7. *Soit la formule* $P = \mathbf{G}(\{f(a)\} \implies \mathbf{X}\{g(a)\})$. *L'automate* $\overline{B_P}$ *(fig. 4.8) reconnaît la négation de la formule* P *exprimée par* $\mathbf{F}(\{f(a)\} \wedge \mathbf{X}\neg\{g(a)\})$ *et* B_K *(fig. 4.7) reconnaît l'ensemble des comportements de la structure de Kripke* K_ϵ *(fig. 4.7). La notation* A_α *dénote l'automate d'arbres tel que son langage est décrit par* α *(*$A_{\neg g(a)}$ *reconnaît la complément du langage* $\mathcal{L}(A_{g(a)})$ *et* A_* *reconnaît tout terme de* $\mathcal{T}(\mathcal{F})$*).*

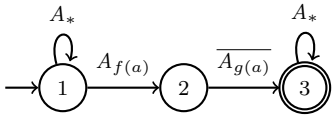


FIGURE 4.8: l'automate $\overline{B_P}$

L'automate intersection $B_\cap$ est obtenu en calculant le produit de automates B_K et $\overline{B_P}$. Comme tous les états sont finals dans l'automate B_K, il est possible d'employer une version plus simple du produit général d'automates de Büchi pour ce cas particulier [25]. On présente une version adaptée de cet algorithme pour les automates de Büchi dont les transitions étiquetées par des automates d'arbres.

Définition 4.4.8 ($B_K \times \overline{B_P}$)**.** *Le produit de* $B_K = \langle \Sigma,\ Q,\ Q_0,\ Q,\ \Delta \rangle$ *par* $\overline{B_P} = \langle \Sigma,\ Q',\ Q'_0,\ \mathcal{F}',\ \Delta' \rangle$ *est défini*

$$\langle \Sigma,\ Q \times Q',\ Q_0 \times Q'_0,\ Q \times F',\ \Delta_\times \rangle$$

où $\Delta_\times$ *est un ensemble de transitions* $(q_K, q_P) \xrightarrow{(A_K \cap A_P)} (q'_K, q'_P)$ *tel que* $q_K \xrightarrow{A_K} q'_K$ *est une transition de* B_K *et* $q_P \xrightarrow{A_P} q'_P$ *est une transition de* $\overline{B_P}$*. De plus, la transition est seulement valide si l'intersection entre les langages de* A_K *et* A_P *est non vide.*

L'intersection d'automates d'arbres calculée sur chaque transition de l'automate intersection se justifie simplement. Une transition de la forme s, A, s' pourrait être vue comme un ensemble de transitions de la forme s, t, s' où $t \in \mathcal{L}(A)$. Ainsi l'intersection placée sur les transitions du produit $B_K \times \overline{B_P}$ correspond simplement à l'ensemble des transitions $((s_1, s_2), t, (s'_1, s'_2))$ tel que $t \in \mathcal{L}(B_K)$ et $t \in \mathcal{L}(B_P)$.

Enfin, la vacuité du langage $\mathcal{L}(B_\cap)$ peut être vérifiée en utilisant l'algorithme basé sur la recherche en profondeur qui assure que toute composante fortement connexe accessible à partir d'un état initial ne contient pas d'états acceptants.

Exemple 4.4.9. *Pour illustrer l'approche, on propose de vérifier pour la formule* $P = \mathbf{G}(\{f(a)\} \implies \mathbf{X}\{g(a)\})$ *de l'exemple 4.3.5. La figure 4.9 montre le résultat de l'intersection* $B_\cap$ *entre* B_K *et* $\overline{B_P}$*. Seul les états atteignables states et les transitions valides*

(étiquetées par une intersection non vide d'automates d'arbres) sont montrées. Comme aucun état atteignable de $B_\cap$ n'est final, sont langage est vide. Cela signifie que toutes les traces d'éxecution de K_ϵ satisfont P : en effet, le seul successeur de $f(a)$ pour $\dashrightarrow_{\mathcal{R}_\epsilon}$ est $g(a)$.

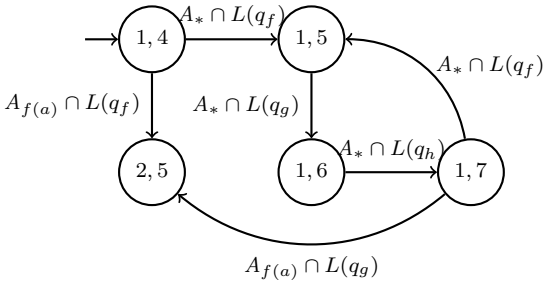


FIGURE 4.9: Automaton $B_\cap$

4.5 Conclusion, discussion

Dans ce chapitre, on a montré comment exploiter le mécanisme de relation d'ordre et préserver la relation d'ordre entre les termes accessibles, grâce aux transitions ε-transitions. On a également montré comment utiliser de la relation induite par ces transitions pour prouver des formules en logique temporelle linéaire sur le système de réécriture. Le chapitre traite de l'analyse dans le cas où le calcul des termes atteignables est réalisé sans approximation. On peut montrer qu'il est possible d'extraire un modèle de Kripke pour la relation $\dashrightarrow_{\mathcal{R}/E}$ dans le cas où la complétion produit un automate d'arbres $A^*_{\mathcal{R},E}$ pour un programme java donné $\mathcal{R}$ dont l'approximation est donnée par un ensemble d'équations E. L'adaptation de la preuve est très simple. Cependant, il n'est pas possible de vérifier toutes les formules aussi simplement.

Approximation et sûreté Les approximations sont toujours correctes par construction dans le cas où l'on vérifie des propriétés de sûreté. Comme les accélérations fusionnent des états, l'ajout d'information est toujours correct quand on veut montrer qu'une propriété n'est pas vraie.

Des pistes pour la vivacité Par contre si l'on souhaite montrer des propriétés de vivacité, il y a deux problèmes majeurs, pour lesquels il est proposé des pistes à explorer pour apporter une solution.

– Le premier est directement lié à la phase d'approximation `W`, il n'est plus possible de fusionner les états aussi simplement. La fusion abusive d'états peut dissimuler

des blocages conduisant à des conclusions fausses pour la vivacité. On considère trois termes $u\tau$, $v\tau$ et t, tels que $u\tau \not\to_{\mathcal{R}}$, $v\tau \dashrightarrow_{\mathcal{R}} t$ et une équation $u = v$ de E. Dans l'automate A, on suppose que les termes $u\tau$ et $v\tau$ sont des représentants respectivement de q_u et q_v, et que $v\tau \dashrightarrow_{\mathcal{R}} t$ est dénotée par la transition $q_t \to q_v$. Comme on a $u\tau = v\tau$, on peut fusionner les états $q_u = q_v$, et on obtient l'automate $\mathtt{W}(A)$ dans lequel les termes $u\tau$ et te $v\tau$ sont les représentants du même état q_v, avec la transition $q_t \to q_v$. Si l'on regarde cette transition à postériori, on peut alors en déduire que pour tout terme $s \in \mathcal{R}ep(q_v)$, et $s' \in \mathcal{R}ep(q_t)$ alors on a $s \dashrightarrow_{\mathcal{R}} s'$, et en particulier $u \dashrightarrow_{\mathcal{R}} t$. On peut donc montrer que le terme t est un successeur du terme u, ce qui est contradictoire avec le cas exact $u \not\to_{\mathcal{R}}$.
La manière simple de contourner le problème consiste à n'autoriser la fusion d'état que lorsqu'on est sûr que les deux termes peuvent être réécrits par $\mathcal{R}$. Ce qui nécessite de modifier la procédure d'accélération $\mathtt{W}$. Si on considère les états q et q' qu'il est possible de fusionner par l'équation $u = v$: il existe une Q-substitution $\sigma : \mathcal{X} \to Q$ telle que :

$$\begin{array}{ccc} u\sigma & \underset{E}{=\!=} & v\sigma \\ {\scriptstyle A^i_{\mathcal{R}}}\downarrow & \not\to & \not\to\downarrow{\scriptstyle A^i_{\mathcal{R}}} \\ q_u & & q_v \end{array}$$

Pour une approximation destinée à la vérification de propriété de vivacité, on n'autorise la fusion des états q_u et q_v seulement si on peut décomposer $u\sigma \to_A^{\not\to} q$ et $v\sigma' \to_A^{\not\to} q'$ aux positions p et q telles que :

$$u\sigma|_p \to_A^{\not\to} q_p \text{ avec } u\sigma[q_p]_p \to_A^{\not\to} q_u \text{ et } q'_p \to q_p$$
$$v\sigma|_q \to_A^{\not\to} q_q \text{ avec } u\sigma[q_q]_q \to_A^{\not\to} q_v \text{ et } q'_q \to q_q$$

Les deux $q'_p \to q_p$, $q'_q \to q_q$ sont suffisantes pour déterminer que les sous-termes de la forme $u|_p$ et $v|_q$ peuvent être réécrits par $\mathcal{R}$, et donc $u\sigma$ et $v\sigma$ ne cachent pas de blocage, la fusion est correcte.

– Un autre problème est lié à l'indéterminisme de la réécriture $\mathcal{R}_{/E}$. En fait, dans le cas exact, on sait que le système de réécriture engendré pour les programmes Java est déterministe. C'est une propriété suffisante pour s'assurer que lorsque l'on considère une transition $q' \to q$ alors la relation $\dashrightarrow_{\mathcal{R}}$ dénotée est aussi déterministe. Ainsi si considère la transtion $q' \to q$ dans un automate complet pour $\mathcal{R}_{/E}$, on sait qu'on a $u \dashrightarrow_{\mathcal{R}_{/E}} v$ où u et v sont des représentants de q et q' respectivement. D'après la définition 4.1.2 on sait qu'il existe une susbtitution σ et une règle de réécriture telle que $u \to^*_{\mathcal{R}_{/E}} l\sigma \to_{\mathcal{R}} r\sigma =_E t$. On peut déjà montrer qu'il n'y a pas de paire critique pour le terme $l\sigma$. En effet, comme $\mathcal{R}$ est déterministe, on sait que la règle $l \to r$ ne peut que réécrire des termes qui sont irréductibles. Ceci est aussi vrai pour chacun des sous-termes de $l\sigma$. Donc en tenant compte de la nouvelle manière de fusionner les états de l'automates, aucun des sous-termes de $l\sigma$ ne peut être le résultat d'une fusion. En effet, ceci ajouterait une réécriture de la forme $l\sigma \to_{\mathcal{R}_{/E}} \ldots$. Donc la seule manière d'introduire du non-déterminisme est dans la relation $u \to^*_{\mathcal{R}_{/E}} l\sigma$. Mais

comme l'approximation ne peut introduire des paires critiques par réécriture à des positions différentes, si il n'existe pas de chaîne de ε-transitions contenant un motif de la forme $q_2 \to q_1$ et $q_3 \to q_1$, alors il n'y a pas de paire critique par réécriture à la même position. Donc en proposant une analyse de l'automate d'arbres basés sur ce dernier critère, on espère étiqueter avec certitude les transitions $q' \to q$ qui dénotent une relation où $\mathcal{R}_{/E}$ est localement déterministe.

Chapitre 5

Certification de la complétion d'automates d'arbres

5.1 Un validateur de résultat pour la complétion d'automates d'arbres

La certification d'un algorithme dans un assistant de preuve, est une tâche difficile dans la mesure où la preuve doit généralememt être entièrement construite à la main. Cela peut devenir d'autant plus ardu que l'algorithme implémente des optimisations complexes basées sur des structures de données avancées où des astuces algorithmiques. Ce qui est souvent requis si l'on cherche à certifier une implémentation réaliste, c'est à dire dont les performances supportent un passage à l'échelle. Dans le cas de la complétion d'automates d'arbres, les difficultés rencontrées lors du passage à l'échelle ont conduit les différents développement de l'outil Timbuk vers une implémentation qui est actuellement bien éloignée du formalisme initial. Bien qu'impromptue, des bugs ont été et peuvent être découverts mais la détection est d'autant plus difficile que les automates obtenus sont de taille importantes. Même si chaque nouveau bug supprimé est l'occasion d'enrichir le jeu de tests, on ne peut que s'interroger quant à l'empirisme d'une telle approche. Peut-on avoir *confiance* en un outil tel que Timbuk ? La question est d'autant plus cruciale, lorsque cet outil sert la validation d'un système, comme dans les chapitres précédents. L'objet de la certification est donc d'augmenter le degré de confiance que l'on peut accorder à Timbuk. On souhaiterait donc prouver que l'implémentation `timbuk` calcule bien un post point-fixe, soit le théorème de correction suivant la propriété :

$$\forall A\ A'\ \mathcal{R},\quad A' = \texttt{completion}(A,\ \mathcal{R}) \implies \mathcal{L}(A') \supseteq \mathcal{R}^*(\mathcal{L}(A))$$

Si la question est légitime, il existe plusieurs manières de répondre à cette question. Allant des différentes techniques d'analyse statique jusqu'aux assistants de preuve, la preuve d'un programme de plus de 11000 lignes de code en OCaml, reste un challenge important. D'autre part, une telle preuve de correction est liée à une implémentation particulière de l'algorithme. Ce qui signifie que tout apport d'une nouvelle optimisation rend caduque la

partie de la preuve concernée par la modification, remettant en cause la correction de toute l'implémentation.

Une solution consiste à contourner le problème en transférant le problème de la certification en un problème de certification du résultat. Plutôt que de montrer que toute exécution de la fonction `completion` donne un résultat correct, on propose de fournir un validateur appelé `checker` dans le reste du chapitre, qui vérifie si le résultat correspond à la spécification attendue. L'avantage est que le `checker` est un programme plus simple, qui ne nécessite pas d'être modifié tant que la spécification de l'outil ne change pas ! On veut alors montrer la propriété suivante :

$$\forall A\ A'\ \mathcal{R},\quad \texttt{checker}(A,\ \mathcal{R},\ A') = \texttt{true} \implies \mathcal{L}(A') \supseteq \mathcal{R}^*(\mathcal{L}(A))$$

On obtient un certificat du résultat dès lors que l'étape de la validation se conclut par un succès. Cependant comme l'étape de validation est requise après chaque exécution, il est nécessaire que le coût de la vérification puisse être négligeable ou au moins raisonnable par rapport au temps de calcul. En effet, si le coût de la vérification est plus important que le gain apporté par les optimisations de l'implémentation, il est évident que les optimisations deviennent inefficaces, dans le cas où l'on veut un certificat.

En transposant le problème de la certification de l'algorithme de la complétion vers l'assistant de preuve Coq, le problème de certification revient alors à montrer le théorème suivant :

```
Theorem sound_checker :
      ∀ A A' R, checker A R A' = true →  AReachable A R A'.
```

où `AReachable` est un prédicat Coq qui décrit la propriété de correction : $\mathcal{L}(A')$ *contient tous les termes atteignables par réécriture des termes de* $\mathcal{L}(A)$ *avec* $\mathcal{R}$, *i.e.* $\mathcal{L}(A') \supseteq \mathcal{R}^*(\mathcal{L}(A))$. Pour établir formellement ce prédicat en Coq, on a besoin de donner une formalisation des systèmes de réécriture et des automates d'arbres en Coq (*cf.* Section 5.2). Pour deux automates A, A' donnés et un système de réécriture $\mathcal{R}$, on veut vérifier que $\mathcal{L}(A') \supseteq \mathcal{R}^*(\mathcal{L}(A))$ soit (`AReachable A R A'`) en Coq. Pour réaliser cela, on a besoin de vérifier les deux propriétés suivantes :

- `Included` : que tous les termes de l'ensemble initial sont présents dans le point fixe : $\mathcal{L}(A) \subseteq \mathcal{L}(A')$.
- `IsClosed` : A' est clos par réécriture avec $\mathcal{R}$: pour toute règle $l \to r \in \mathcal{R}$ et tout terme $t \in \mathcal{L}(A')$, si t peut se réécrire en un terme t' par la règle $l \to r$ alors on a $t' \in \mathcal{L}(A')$.

Pour chacun des items, on procède de la même façon, en fournissant une fonction Coq avec son théorème Coq. La fonction `inclusion` est dédiée à la vérification de l'inclusion et la fonction `closure` vérifie si un automate d'arbres est clos par réécriture.

```
Theorem inclusion_sound:
      ∀ A A', inclusion A A' = true →  Included A A'.

Theorem closure_sound:
      ∀ R A', closure R A' = true →  IsClosed R A'.
```

Le théorème qui permet de déduire `AReachable A R A'` à partir de `Included A A'` et `IsClosed R A'` :

```
Theorem Included_IsClosed_Reachable:
      ∀ A A' R, Included A A' →  IsClosed R A' →  AReachable A R A'.
```

On se concentre sur la preuve de $\mathcal{L}(A') \supseteq \mathcal{R}^*(\mathcal{L}(A))$. Cependant, pour prouver la propriété de non-atteignabilité, la vacuité de l'intersection entre $\mathcal{L}(A')$ et l'ensemble des termes interdits doit aussi être vérifiée. La formalisation en Coq de l'intersection et la décision du vide sont proches de leur definition standard [18], et ils sont déjà traités dans leur implémentation Coq a déjà été traitée dans [47]. D'autre part, cette version du `checker` est destinée à la version de la complétion où la résolution des paires critiques ne faisaient pas intervenir les ε-transitions. On considère donc que le résultat `completion`$(A,\ \mathcal{R})$ est donc un automate d'arbres sans ε-transitions.

5.2 Formalisation de la réécriture

Le but de cette partie est de formaliser en Coq : les termes, la réécriture, les termes atteignables et le problème d'atteignabilité lui-même. Premièrement on utilise les entiers binaires positifs fournis par la librairie standard de Coq pour définir les ensembles de symboles comme les variables ($\mathcal{X}$), les symboles de fonctions ($\mathcal{F}$), ou les ensembles d'états (Q). Pour être plus explicite, on renomme les `positive` en `ident`. L'ensemble des termes se définit inductivement :

```
Definition ident := positive.

Inductive term : Set :=
| Fun : ident →  list term →  term
| State : ident →  tern
| Var : ident →  term.
```

Cet ensemble contient plus de termes que les ensembles $\mathcal{T}(\mathcal{F}, \mathcal{X})$, $\mathcal{T}(\mathcal{F})$ et $\mathcal{T}(\mathcal{F} \cup Q)$. Pour être exact, il définit tous les termes de l'algèbre $\mathcal{T}(\mathcal{F} \cup Q, \mathcal{X})$. Ainsi, on introduit les prédicats `TFX: term → Prop`, `TF: term → Prop` et `TFQ: term → Prop` pour caractériser respectivement chacun des sous-ensembles.

Maintenant, le terme $f(x, a)$ sera construit comme `Fun 0 (Var 0::(Fun 1 nil)::nil)` en supposant que l'on a la correspondance suivante entre les symboles, variables et les entiers binaires $f \mapsto 0$, $a \mapsto 1$ et $x \mapsto 0$ par exemple. On remarque qu'il est possible de d'attacher la valeur 0 à f et x, comme `Fun` et `Var`, les constructeurs du type `term`, permettent de distinguer les variables des symboles de fonction.

Remarques : Une faiblesse de Coq est la génération automatique du principe d'induction `term_rect`. Le théorème généré ne tient pas compte dans le cas *Fun* que le terme est constitué d'une `list term` pour former les sous-termes et ce qui nécessite souvent d'utiliser l'hypothèse d'induction : `term_rect` est trop faible pour prouver quoi que ce soit. Dès lors,

il est nécessaire de définir un second théorème nommé `term_rect'` pour construire les preuves qui nécessitent une étude de cas sur `term`.

```
Lemma term_rect' :
   ∀ (P: term → Type) (P0: list term → Type),
      (∀ x, P (Var x)) → (∀ q, P (State q)) →
         (∀ p l, P0 l → P (Fun p l)) →
         P0 nil →
         (∀ t lt, P t → P0 lt → P0 (t::lt)) →
           ∀ t, P t.
```

Ce principe d'induction repose sur l'imbrication mutuelle de deux prédicats `P` et `P0` portant respectivement l'un sur les termes et l'autre sur les listes de sous-termes. Le but de ce principe d'induction est de montrer la conclusion `∀ t, P t`. Le prédicat `P0` n'est là que pour fournir une aide sur la manière dont on veut raisonner pour transmettre (le pas d'induction) la propriété `P` des sous-termes qui constituent `l` vers un terme complet de la forme `Fun f l`. En fait, dans la majorité des cas, il suffit de considérer une instance particulière de ce principe où `P0` est instancié par `fun l ⇒ ∀x, In x l → P x` :

```
Lemma term_rec_In :
   ∀ (P: term → Set),
      (∀ x, P (Var x)) → (∀ q, P (State q)) →
         (∀ p l, (∀ x, In x l → P x) → P (Fun p l)) →
            ∀ t, P t.
```

En fait, pour construire ce principe d'induction, on a besoin au préalable de montrer la décidabilité de l'égalité sur les termes. Cette propriété ne peut-être montré qu'avec une instance particulière du principe `term_rect'` où le prédicat `P0` exprime la décidabilité de l'égalité sur les listes de termes. De plus, le résultat sur la décision de l'égalité est la propriété indispensable que l'on utilise couramment pour comparer les termes.

```
Theorem term_eq_dec : ∀ u v: term, {u = v} + {u <> v}.
```

Une règle de réécriture $l \to r$ est représentée par un couple de termes avec une preuve que la règle est bien formée, *i.e.* un terme de preuve Coq qui assure que l'ensemble des variables du membre droit r est un sous-ensemble des variables du membre gauche l. A ce titre, la fonction `Fv : term → list ident` construit l'ensemble des variables d'un terme.

```
Inductive rule: Set :=
| Rule(l r: term) (Hsub: subseteq (Fv r)(Fv l))
                  (Hl: TFX l) (Hr: TFX r)       : rule.
```

D'autre part, la complétion manipule des ensembles de règles linéaires à gauches. On a introduit donc une deuxième définition possède une hypothèse supplémentaire assurant la linéarité du membre gauche. Cette dernière est assurée par le prédicat `linear: term → Prop` qui fixe le nombre d'occurrences de chaque variable à 1 au maximum.

```
Inductive l_rule: Set :=
| Lrule(l r: term) (Hsub: subseteq(Fv r)(Fv l))
                   (Hl: TFX l) (Hr: TFX r)
                   (Hlin: linear l)             : lrule.
```

Dans la suite, le type `list rule` représente un système de règles de réécriture et donc `list lrule` un systèmes de règles de réécriture linéaires à gauche. On utilise la notation `(t @ sigma)` pour dénoter le terme résultant de l'application d'une substitution `sigma` à chaque variable qui apparaît dans le terme `t`.

```
Definition substitution := ident → option term.
```

En Coq, la relation de réécriture *"u est réécrit en v par $l \to r$"*, habituellement définie comme $\exists p \in \mathcal{P}os(t), \exists \sigma \ t.q. \ u|_p = l\sigma \ \wedge \ v = u[r\sigma]_p$, est divisée en deux prédicats :

- Le premier `TRew` définit la réécriture d'un terme à la racine du terme (à la position ϵ). En fait, l'ensemble des couples de termes (t, t') qui sont réécrits à la racine par la règle $l \to r$ peuvent être vus comme l'ensemble des termes $(l\sigma, r\sigma)$ pour toute substitution σ.
- Le second prédicat `Rew` définit inductivement la relation de réécriture pour toute position d'un terme t par la règle $l \to r$, par la réécriture à la racine de tout sous-terme de t par $l \to r$.

```
(* Topmost rewriting : *)
Inductive TRew (x : rule) : term → term → Prop :=
| R_Rew :
     ∀ s l r Hsub Hl Hr,
        x = Rule l r Hsub Hl Hr → TRew x (l @ s) (r @ s).

(* Rewriting at any position of any term *)
Inductive Rew (r : rule) : term → term → Prop :=
| RewT : ∀ t t',
     TRew r t t' → Rew r t t'
| RewSub : ∀ f l l',
     RewTerms r l l' → Rew r (Fun f l) (Fun f l')
with RewTerms (r : rule) : list term → list term → Prop :=
| RewNext: ∀ t l l',
     RewTerms r l l' → RewTerms r (t::l) (t::l')
| RewThis: ∀ l t t',
     Rew r t t' → RewTerms r (t::l) (t'::l).
```

Cette définition de la relation de réécriture présente la particularité d'être facilement manipulable. On est tout de même en droit de se demander si cette formalisation est équivalente à la définition standard. Une manière simple de répondre consiste à montrer l'équivalence des deux définitions. On introduit donc la définition standard `StdRew` et on montre l'équivalence :

```
Definition StdRew (x: rule)(u v: term) :=
   ∀ l r Hsub Hl Hr, Rule l r Hsub Hl Hr = x →
      ∃ p: position, ∃ s: substition,
         get u p = Some (l @ s) ∧ v = set u p (r @ s).

Theorem StdRew_Rew:
   ∀ (x: rule)(u v: term), StdRew lr u v ↔ Rew lr u v.
```

La définition standard utilise les fonctions `get` et `set` permettant respectivement d'accéder au de remplacer le sous-terme à la postion p : `get u p = Some t` est équivalent à

$u|_p = u$ et `set u p t` équivaut à $u[t]_p$. On remarquera qu'il n'est pas nécessaire de préciser que la position p appartient à $\mathcal{P}os(u)$. La fonction `get` est une fonction partielle (elle renvoie une valeur de type `option term`) dont l'évaluation est la forme `Some t` si et seulement si $p \in pos(u)$:

```
Lemma Pos_get: ∀ t p, Pos t p ↔ ∃ t', get t p = Some t'.
```

Ensuite, on doit définir $\rightarrow^*_{\mathcal{R}}$. En Coq, on preférera voir cela comme le prédicat `Reachable R u` qui caractérise l'ensemble des termes atteignables à partir de u par $\rightarrow^*_{\mathcal{R}}$.

```
Inductive Reachable(R : list rule)(t : term) : term → Prop:=
| R_refl : Reachable R t t
| R_trans : ∀ u v r, Reachable R t u → In r R → Rew r u v →
    Reachable R t v.
```

5.3 Formalisation des automates d'arbres

Le fait que le checker, qui doit être exécuté, est directement extrait de la formalisation Coq contraint la formalisation des automates d'arbres. Comme les structures de données utilisées dans la formalisation sont celles qui sont réellement utilisées lors de lors de l'exécution, elles doivent être *efficaces* d'un point de vue algorithmique. Pour les automates d'arbres, au lieu d'une représentation naïve, il est donc nécessaire d'utiliser une formalisation de la structure de données proposée dans [47] pour manipuler de façon optimale les automates d'arbres.

Dans la section 5.2, on a représenté les variables $\mathcal{X}$ et les symboles de de fonctions $\mathcal{F}$ par le type `ident`. On fait la même chose pour définir les états Q. On définit un automate comme un couple (Q_F, Δ), où Q_F est l'ensemble fini des états finals, et Δ l'ensemble fini des transitions normalisées comme $f(q_1, \ldots, q_n) \rightarrow q$. En Coq, Q_F est une simple `list ident` alors que Δ est représentée en utilisant les `FMapPositive` de la librairie Coq. Il s'agit d'une implémentation des tables d'associations fonctionelles, où les données sont indexées par `positive`. Les `positive` sont en fait une représentation binaire des entiers strictement positifs. Dans la structure de `FMapPositive`, chaque transition $f(q_1, \ldots, q_n) \rightarrow q$ est encodée par une liste d'états $(q_1, \ldots, q_n)$ indexée par f dans une première table qui est ensuite indexée par l'état q dans une seconde table. Cette représentation est une bonne solution pour manipuler efficacement les ensembles de transtions en Coq.

```
Module Delta : DELTA.
  (* Transition sets : *)
  Definition config := list state.

  Definition t :=
       FMap.t (FMap.t (list config)).
  (* ............... *)
```

Ensuite, on peut définir un prédicat pour caractériser le langage reconnu par un automate d'arbres. En fait, il s'agit de définir l'ensemble des termes clos qui sont réduits (réécrits) dans un état q par les transitions de Δ. Cet ensemble, qui correspond à $\mathcal{L}(\Delta, q)$

si Δ est l'ensemble des transitions de A, peut-être construit inductivement en Coq en utilisant l'unique règle de déduction :

$$\frac{t_1 \in \mathcal{L}(\Delta, q_1) \qquad \ldots\ldots \qquad t_n \in \mathcal{L}(\Delta, q_n)}{f(t_1, \ldots, t_n) \in \mathcal{L}(\Delta, q)} \text{ Si } f(q_1, \ldots, q_n) \to q \in \Delta$$

En Coq, on exprime cette proposition en utilisant le prédicate inductif `IsRec`. Un terme t est reconnu par un automate d'arbre (Q_F, Δ), si le prédicat `IsRec Δ q t` est valide pour $q \in Q_F$.

```
Inductive IsRec (D: Delta.t) : state → term → Prop :=
  Rec_Term : ∀ f lt q,
      IsRec' D (Delta.get q f D) lt → IsRec D q (Fun f lt)

with IsRec' (D: Delta.t) : list config → list term → Prop :=
| Rec_SubTerm : ∀ lt c lc, IsRec'' D c lt → IsRec' D (c::lc) lt
| Rec_SubTerm' : ∀ lt c lc, IsRec' D lc lt → IsRec' D (c::lc) lt

with IsRec'' (D: Delta.t) : config → list term → Prop :=
| Rec_Nil : IsRec'' D nil nil
| Rec_Cons : ∀ t q lt lq, IsRec D q t → IsRec'' D lq lt →
       IsRec'' D (q::lq) (t::lt).
```

Il est encore légitime de se demander quel crédit accorder à une telle spécification qui n'a plus beaucoup de point commun avec la théorie des automates d'arbres classiques. C'est pour cette raison que l'on décide de montrer l'équivalence entre ce formalisme et la définition classique telle que présentée dans le chapitre 2. On définit alors un automate comme un couple composé de l'ensemble des états finaux et et l'ensemble des transitions normalisées. Les ensembles sont représentés par des listes.

```
Inductive transition : Set :=
| Ground (l r: term):
      ∀ f lt q (Hl: l = Fun f lt) (Hr: r = State q)
                  (Hnorm: ∀ t, t ∈ lt → ∃ q', t = State q') : Transition.

(* Conversion  Delta.t vers un ensemble de transitions *)
Definition trans_of: Delta.t → list transitions.
    (* ... Body ... *)

(* L'execution t →_A t' *)
Definition Run (t: transition)(u v: term) :=
   ∀ l r f lt q Hl Hr Hnorm, Ground l r f lt q Hl Hr Hnorm = t →
      ∃ p: position,
         get u p = Some l ∧ v = set u p r.

(* Cloture de Run t →*_A t' *)
Inductive StdRun (l: list transition): term → term → Prop :=
| RunRefl: ∀ u, StdRun l u
| RunStep: ∀ u v w t, StdRun l u v → t ∈ l → Run t v w → StdRun l u w.
```

```
(* Equivalence: *)
Theorem StdRun_IsRec:
   ∀ Δ q t,
      IsRec Δ q t ↔StdRun (trans_of Δ) t (state q).
```

Bien que la formalisation soit plus simple que dans la librairie Coq d'automates déterministes descendants [47], la preuve qu'elle soit équivalente à la formalisation théorique est un argument irréfutable pour se convaincre de la correction de l'approche. Enfin, il existe aussi une librairie récente d'automates d'arbres pour Isabelle[42] basée sur une implémentation similaire à celle-ci. La principale différence est l'utilisation des arbres "rouges et noirs" pour construire les tables d'associations plutôt que des arbres d'association. Cela permet de maintenir des structures avec une bonne complexité pour tout accès notamment pour contruire les unions et intersections d'automates. Pour le checker, le seul algorithme nécessaire est la décision du vide de l'intersection entre deux automates, pour vérifier les propriétés. Or il n'est pas nécessaire de construire explicitement l'intersection, on peut simplement rechercher si il existe une exécution commune entre les deux automates d'arbres considérés.

5.4 L'inclusion efficace d'automates

Dans cette section, on donne la définition formelle de la propriété `Included` ainsi que la fonction `inclusion` utilisée pour vérifier efficacement l'inclusion d'automates d'arbres. En se basant sur les définitions précédentes sur les automates d'arbres, on peut établir le `Included` prédicat de la manière suivante :

```
Definition Included (a b : t_aut) : Prop :=
  ∀ t q, q ∈ a.qf → IsRec a.delta q t →
    ∃ q', q' ∈ b.qf ∧ IsRec b.delta q' t.
```

Ensuite on se concentre sur la fonction `inclusion` elle-même. L'algorithme usuel pour montrer l'inclusion de langages réguliers reconnus par des automates d'arbres non-déterministes et ascendants, par exemple pour montrer $\mathcal{L}(A) \subseteq \mathcal{L}(B)$, consiste à montrer que $\mathcal{L}(A) \cap \mathcal{L}(\overline{B}) = \emptyset$, où $\overline{B}$ est l'automate qui reconnaît le complément le langage de l'automate B. Cependant, l'algorithme pour construire $\overline{B}$ à partir B est EXPTIME-complete [18]. Cela est d'autant peu raisonnable que l'automate B est le point fixe calculé par la complétion, il s'agit d'un automate particulièrement gros. Pour cette raison, on propose de contourner le problème en définissant un critère dont la complexité est meilleure en pratique. Il est basé sur une simple comparaison syntaxique des ensembles de transitions, *i.e.* on vérifie l'inclusion des ensembles de transitions modulo le renommage qui a pu être réalisé par la fonction `W` qui fusionne des états. Cette technique améliore considérablement l'efficacité du `checker`, spécialement la consommation mémoire. Or il est crucial de pouvoir vérifier l'inclusion de gros automates d'arbres (*cf.* la section 5.5.2). Cet algorithme est correct mais, bien sûr, il n'est pas complet en général. Il n'est donc pas toujours capable de montrer $\mathcal{L}(A) \subseteq \mathcal{L}(B)$. Cependant, on montre par la suite, sous certaines hypothèses sur A et B, qui sont satisfaites si B est obtenu par la complétion de l'automate A, cet al-

gorithme est complet et redevient une procédure de décision. En premier lieu, on introduit la notation suivante :

Γ	:	l'ensemble d'hypothèses d'induction
Δ_i	:	l'ensemble de transitions de l'automate d'arbres $\mathcal{A}_i$
$\{c \mid c \to q \in \Delta\}$	:	l'ensemble des membres gauches (configurations) des transitions de Δ qui produisent l'état q
$\{c_i\}_n^m$	:	l'ensemble des configurations c_i pour i compris entre n et m

Le problème d'inclusion se formule alors de la sorte : $\Gamma \vdash_{A,B} q \Subset q'$. Une telle formulation signifie que sous les hypothèses de Γ, il est possible de montrer que $\mathcal{L}(A, q) \subseteq \mathcal{L}(B, q')$.

L'algorithme tente de construire un arbre de preuves permettant de conclure sur le problème formulé au moyen des règles de déduction suivantes.

$$\text{(Induction)}\ \frac{\Gamma \cup \{q \Subset q'\} \vdash_{A,B} \{c \mid c \to_{\Delta_A} q\} \Subset \{c \mid c \to_{\Delta_B} q'\}}{\Gamma \vdash_{A,B} q \Subset q'}\ \text{si}\ (q \Subset q') \notin \Gamma$$

$$\text{(Axiom)}\ \frac{}{\Gamma \cup \{q \Subset q'\} \vdash_{A,B} q \Subset q'} \qquad \text{(Empty)}\ \frac{}{\Gamma \vdash_{A,B} \emptyset \Subset \{c'_j\}_1^m}$$

$$\text{(Split-l)}\ \frac{\Gamma \vdash_{A,B} c_1 \Subset \{c'_j\}_1^m \quad \ldots\ldots \quad \Gamma \vdash_{A,B} c_n \Subset \{c'_j\}_1^m}{\Gamma \vdash_{A,B} \{c_i\}_1^n \Subset \{c'_j\}_1^m}$$

$$\text{(Weak-r)}\ \frac{\Gamma \vdash_{A,B} c \Subset c'_k}{\Gamma \vdash_{A,B} c \Subset \{c'_i\}_1^n}\ \text{si}\ (1 \leq k \leq n) \qquad \text{(Const.)}\ \frac{}{\Gamma \vdash_{A,B} a() \Subset a()}$$

$$\text{(Config)}\ \frac{\Gamma \vdash_{A,B} q_1 \Subset q'_1 \quad \ldots\ldots \quad \Gamma \vdash_{A,B} q_n \Subset q'_n}{\Gamma \vdash_{A,B} f(q_1, \ldots, q_n) \Subset f(q'_1, \ldots, q'_n)}$$

Si on considère Q_{F_A} et Q_{F_B} comme les ensembles d'états finals de A et B, et $\#()$ un symbole particulier d'arité 1 qui n'est pas dans $\mathcal{F}$. Pour prouver $\mathcal{L}(A) \subseteq \mathcal{L}(B)$, il suffit de démarrer le raisonnement à partir de la formulation $\emptyset \vdash_{A,B} \{\#(q) \mid q \in \mathcal{Q}_{F_A}\} \Subset \{\#(q) \mid q \in \mathcal{Q}_{F_B}\}$.

Exemple 5.4.1. *Soient A et B deux automates tels que :*

$$A = \left\{ \begin{array}{rcl} a & \to & q_1 \\ b & \to & q_2 \\ f(q_1, q_2) & \to & \boldsymbol{q} \end{array} \right\} \textit{ with } Q_{F_A} = \{\boldsymbol{q}\} \textit{ and } B = \left\{ \begin{array}{rcl} a & \to & \boldsymbol{q'} \\ b & \to & \boldsymbol{q'} \\ f(q', q') & \to & \boldsymbol{q'} \end{array} \right\} \textit{ with } Q_{F_B} = \{\boldsymbol{q'}\}$$

On voit facilement que $\mathcal{L}(A) \subseteq \mathcal{L}(B)$ et on peut construire un arbre de preuve pour conclure à $\emptyset \vdash_{A,B} \#(q) \Subset \#(q')$ au moyen des règles de déduction :

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\{q \Subset q', q_1 \Subset q'\} \vdash_{A,B} a() \Subset a()}(\textit{Const.})}{\{q \Subset q',\ q_1 \Subset q'\} \vdash_{A,B} a() \Subset \{a(), b(), f(q', q')\}}(\textit{Weark-r})}{\{q \Subset q'\} \vdash_{A,B} q_1 \Subset q'}(\textit{Induction}) \qquad \dfrac{\dfrac{\dfrac{}{\{q \Subset q', q_2 \Subset q'\} \vdash_{A,B} b() \Subset b()}(\textit{Const.})}{\{q \Subset q',\ q_2 \Subset q'\} \vdash_{A,B} b() \Subset \{a(), b(), f(q', q')\}}(\textit{Weark-r})}{\{q \Subset q'\} \vdash_{A,B} q_2 \Subset q'}(\textit{Induction})}{\{q \Subset q'\} \vdash_{A,B} f(q_1, q_2) \Subset f(q', q')}(\textit{Config})}{\{q \Subset q'\} \vdash_{A,B} f(q_1, q_2) \Subset \{a(), b(), f(q', q')\}}(\textit{Weak-r})}{\emptyset \vdash_{A,B} q \Subset q'}(\textit{Induction})}{\emptyset \vdash_{A,B} \#(q) \Subset \#(q')}(\textit{Config})$$

La principale propriété que l'on veut montrer en Coq est que le critère syntaxique implique l'inclusion sémantique c'est à dire l'inclusion de langages tel que formulée dans la section 5.3.

```
Theorem inclusion_sound :
  ∀ A B, inclusion A B = true → Included A B.
```

Avant de prouver ce résultat en Coq, on doit définir plus formellement la fonction `inclusion`. Cette fonction ne peut être définie comme une simple récursion structurelle. Or en Coq, la théorie sous-jacente repose sur la propriété de normalisation forte qui impose à toute fonction définie de terminer. Ainsi, cet algorithme dont la terminaison n'est pas immédiate nécessite de fournir à Coq une preuve de terminaison. Initialement définie avec l'extension `Function` de Coq, à cause d'un certain nombre de limitations, la fonction `inclusion` est actuellement construite en utilisant le langage de tactiques. C'est un théorème qui exprime la décidabilité de l'ensemble des règles de déduction pour tout but de la forme $\Gamma \vdash_{A,B} q \Subset q'$, but dénoté en Coq par le prédicat `Istate A.(Delta) B.(Delta) Gamma q q'`. Comme Coq repose sur une logique constructive, la preuve contient l'algorithme qui permet la décision.

```
(* Decidability of Predicate Istate *)
Lemma inclusion :
  ∀ A B G q q',
    {Istate A.(Delta) B.(Delta) G q q'} + {¬ Istate A.(Delta) B.(Delta) G q q'}.

(* New soundness Theorem *)
Theorem inclusion_sound:
  ∀ A B G q q', Istate A.(Delta) B.(Delta) G q q' → Included A B.
```

5.4.2 Preuve de terminaison

La construction de `inclusion` repose sur la décroissance stricte d'une mesure μ pour chaque application de règle. La fonction de mesure μ est définie sur les formulations $\Gamma \vdash_{A,B} \alpha \Subset \beta$. La relation Γ est un sous-ensemble de $\mathcal{Q}_A \times \mathcal{Q}_B$ qui est un ensemble fini. Tous les automates ont un nombre fini d'états. La mesure $\mu(\Gamma \vdash_{A,B} \alpha \trianglelefteq \beta)$ se définit comme le couple $(\mu_1(\Gamma), \mu_2(\alpha) + \mu_2(\beta))$ où :

$$\left[\begin{array}{lcl} \mu_1(\Gamma) & = & |Q_A \times Q_B| - |\Gamma| \\ \mu_2(x) & = & \begin{cases} (m+1-n) \text{ si } x = \{c_i\}_n^m \\ 1 \text{ si } x = f(q_1, \ldots, q_n), \\ 0 \text{ sinon} \end{cases} \end{array}\right.$$

On peut alors définir la relation d'ordre $\ll$ comme la combinaison lexicographique de l'ordre usuel $<$ sur les entiers naturels pour μ_1 et μ_2. Ce qui donne :

$$(x, y) \ll (x', y') \Longleftrightarrow \bigvee \begin{cases} x < x' \\ x = x' \wedge y < y' \end{cases}$$

Puisque $<$ est une relation bien fondée, il est évident que $\ll$ est aussi une relation bien-fondée.

Théorème 5.4.3. *(Terminaison) A chaque étape de déduction, la mesure décroît strictement :*

$$\frac{\Gamma \vdash_{A,B} \alpha \Subset \beta}{\Gamma' \vdash_{A,B} \alpha' \Subset \beta'} \Longrightarrow \mu(\Gamma \vdash_{A,B} \alpha \Subset \beta) \ll \mu(\Gamma' \vdash_{A,B} \alpha' \Subset \beta')$$

Démonstration. Le tableau suivant résume pour chaque règle de dérivation quelle composante du couple prouve que μ décroît entre la conclusion et les prémisses de la règle :

	μ_1	μ_2
Induction	$\mu_1(\Gamma) < \mu_1(\Gamma')$	$-$
Split-l	$\mu_1(\Gamma) = \mu_1(\Gamma')$	$\mu_2(c_i) = 1 < \mu_2(\{c_i\}_1^n)$
Weak-r	$\mu_1(\Gamma) = \mu_1(\Gamma')$	$\mu_2(c_k) < \mu_2(\{c_i\}_1^n)$
Config	$\mu_1(\Gamma) = \mu_1(\Gamma')$	$\mu_2(f(\dots, q_i, \dots)) = \mu_2(f(\dots, q_i', \dots)) = 1$ $\mu_2(q_i) = \mu_2(q_i') = 0$ donc $2 > 0$

Pour la règle `Split-l` (resp. `Weak-r`), on considère $n > 1$ si l'on a l'ensemble $\alpha = \{c_i\}_1^n$ (resp. β) qui contient au moins deux éléments différents. Sinon, si $(n = 1)$ alors cette règle ne s'applique pas sur $\Gamma \vdash_{A,B} \alpha \Subset \beta$.

□

Théorème 5.4.4. *Lorsque $\mu(\Gamma \vdash_{A,B} \alpha \Subset \beta) = (0, 0)$, on sait que l'on peut directement appliquer la règle Axiom ou la règle Nil : la branche courante de la preuve est complète.*

Démonstration. A partir $\mu(\Gamma \vdash_{A,B} \alpha \Subset \beta) = (0, 0)$ on en déduit immédiatement :

1. $\mu_1(\Gamma) = 0$ on a $\Gamma = Q_A \times Q_B$
2. $\mu_2(\alpha) = \mu_2(\beta) = 0$ induit α et β sont tous les deux soit un état soit l'ensemble vide.

On en déduit que le but que l'on cherche à prouver peut être de la forme :

- $\Gamma \vdash_{A,B} \emptyset \Subset \emptyset$ ce qui correspond au cas de la règle Empty : la dérivation est terminée.
- $\Gamma \vdash_{A,B} q \Subset q'$: on utilise alors le fait que $\Gamma = Q_A \times Q_B$, et que $(q, q') \in Q_A \times Q_B \implies (q \Subset q') \in \Gamma$. Ce cas correspond à la règle Axiom ce qui conclue encore la dérivation courante.

□

En Coq, on ne définit pas la mesure exactement de cette manière. En réalité, on se passe aisément de la mesure μ_2 qui se ramène à de la récurrence structurelle. En particulier, on utilise le type `list` pour représenter les ensembles de configurations, et les configurations sont des listes d'états (`list state`). Comme le parcours des listes est structurellement récursif, c'est alors aussi le cas pour les ensembles et les configurations. Au final, l'implémentation de l'algorithme est basé sur un seul appel récursif de la forme :

```
(* hypothese d'induction *)
(* Γ est l'ensemble des hypotheses du but *)
IH: ∀ Γ', μ₁ Γ' < μ₁ Γ →
      ∀ q q' : state, {I_state Δ₁ Δ₂ Γ' q q'} + {¬ I_state Δ₁ Δ₂ Γ' q q'}.
```

Les ensembles Γ et Γ' sont modélisés par une structure qui contient des couples d'états. Initialement implémentée par une `list` of `state * state` cette structure s'est révélée très vite insuffisante pour de gros automates : en effet la recherche d'un élément dans une liste est peut efficace, puisqu'il faut parcourir toute la liste dont la taille peut aller jusqu'à $|Q_A \times Q_B|$. Le recours aux tables d'associations s'est imposé pour revenir à une complexité plus raisonnable. Or la preuve est plus facile à réaliser sur les listes s'association que sur les tables d'associations, notamment pour raisonner sur μ_1. Pour rendre plus facile la définition d'une mesure pour n'importe quelle implémentation d'ensemble fini (par exemple les listes ou les tables), on utilise l'interface suivante pour abstraire l'implémentation de l'ensemble fini : cette signature de module isole les fonctions nécessaires ainsi que leur propriétés pour construire les différentes instances Γ. En fait, la seule opération nécessaire dans cet algorithme est l'ajout d'un élément à Γ. Le type α abstrait le types des éléments de l'ensemble, et τ correspond à la structure qui implémente l'ensemble. On suppose bien sûr que l'on est en mesure de distinguer les éléments de l'ensemble (α_eq).

```
Module Type FINITE_SET.
  Variable α : Type.
  Hypothesis α_eq: ∀(x y: α), {x = y} + {x <> y}.

  Variable τ : Type.
  Variable In : α → τ → Prop.

  Variable add : α → τ → τ.
  Hypothesis add_x_spec : ∀ E x, In x (add x E).
  Hypothesis add_other_spec : ∀ x y, x <> y →  ∀ E, (In y (add x E) ↔
In y E).
  Hypothesis elements : τ → list α.
  Hypothesis elements_spec : ∀ E a, In a E ↔ List.In a (elements E).
End FINITE_SET.
```

Cette signature suppose aussi que l'on est en mesure d'établir une équivalence entre le type τ et une liste d'éléments α. Cette astuce déjà présente dans la librairie standard Coq [20], se révèle très pratique : il permet de ramener le raisonnement sur les listes, pour lesquelles les preuves sont souvent plus simples. En se basant sur un module coq qui implémente l'interface `FINITE_SET`, on peut construire un nouveau module en utilisant le foncteur `Make_Wf_Measure`.

```
Module Make_Wf_Measure (Fs : FINITE_SET) <: WF_MEASURE.

Module Type WF_MEASURE.
  Variable α: Type.
  Variable τ: Type.
  Variable add: α → τ → τ.
  Variable R: list α → τ → τ → Prop.
  Variable In: α → τ → Prop.

  Hypothesis R_add:
    ∀ top,
      ∀ a m, List.In a top →  ¬In a m →  R top (add a m) m.
```

```
  Hypothesis wf_R: ∀ top, well_founded (R top).
End WF_MEASURE.
```

Le module construit contient une relation `R top` bien-fondée définit comme `R top Γ' Γ` équivaut à $\mu_1(\Gamma') < \mu_1(\Gamma)$. Le premier argument $\top$ correspond à la totalité de l'ensemble que l'on veut parcourir, soit $\top = Q_A \times Q_B$. Au final, le module fournit une preuve que la relation `(R top)` est bien fondée ainsi qu'une preuve que l'ajout d'un nouvel élément est ordonné : $\mu_1(\{q \Subset q'\} \cup \Gamma) < \mu_1(\Gamma)$ si $\{q \Subset q'\} \notin \Gamma$.

L'implémentation de la fonction `inclusion` repose sur uniquement sur un module de type `FINITE_SET` et un module de type `WF_MEASURE` pour le raisonnement et la manipulation de Γ.

5.4.5 Preuve de correction

Pour la correction, on veut assurer que si il existe un arbre de preuve $\prod$ pour $\emptyset \vdash_{A,B} q \Subset q'$ alors on peut en déduire l'inclusion de langages $\mathcal{L}(\Delta_A, q) \subseteq \mathcal{L}(\Delta_B, q')$

Pour construire la démonstration de la correction, on a besoin du résultat intermédiaire suivant.

Lemme 5.4.6. *(Coupures dans les arbres de $\Subset$-preuve) Soient deux automates d'arbres A et B, si il existe $\prod$ un arbre de preuve pour $\Gamma \vdash_{A,B} q \Subset q'$, ainsi qu'un arbre de preuve pour $\Gamma \cup \{q \Subset q'\} \vdash_{A,B} q_a \Subset q_b$ alors il est possible dec construire un arbre de preuve pour $\Gamma \vdash_{A,B} q_a \Subset q_b$.*

Démonstration. On procède par induction sur $\mu(\Gamma)$.
si $\mu(\Gamma) = 0$, on sait alors que $Q_A \times Q_B = \Gamma$. Comme $q_a \Subset q_b \in \Gamma$, on peut donc prouver que $\Gamma \vdash_{A,B} q_a \Subset q_b$ en utilisant la règle Axiom.

Maintenant, comme hypothèse d'induction, on pose que $\forall \Gamma$ *s.t.* $\mu(\Gamma) = n$, $\forall q\ q'$, si il existe un arbre de preuve $\prod$ pour $\Gamma \vdash_{A,B} q \Subset q'$ et si pour tous les états q_a, q_b il existe un arbre de preuve pour $\Gamma \cup \{q \Subset q'\} \vdash_{A,B} q_a \Subset q_b$ alors on a aussi un arbre de preuve pour $\Gamma \vdash_{A,B} q_a \Subset q_b$. On va donc montrer que cette propriété est vraie pour Γ tel que $\mu(\Gamma) = n + 1$.

On considère l'arbre de preuve de la seconde hypothèse $\Gamma \cup \{q \Subset q'\} \vdash_{A,B} q_a \Subset q_b$. Tout d'abord, si l'arbre de preuve est construit en utilisant la règle Axiom, on sait que $(q_a \Subset q_b) \in \Gamma \cup \{(q \Subset q')\}$. Deux cas sont possibles :

- soit $(q_a \Subset q_b) \in \Gamma$, et alors on construit la preuve de $\Gamma \vdash_{A,B} q_a \Subset q_b$ en utilisant la règle Axiom.
- soit $q = q_a$ et $q' = q_b$, et donc $\Gamma \vdash_{A,B} q_a \Subset q_b$ est tout simplement équivalent à $\Gamma \vdash_{A,B} q \Subset q'$ pour lequel l'arbre de preuve est $\prod$.

Ensuite, si l'arbre de preuve pour $\Gamma \cup \{q \Subset q'\} \vdash_{A,B} q_a \Subset q_b$ est construit en utilisant la règle Induction, alors on a :

$$\dfrac{\dfrac{\dfrac{\prod_{c_1}}{\Gamma \cup \{q_a \Subset q_b\} \cup \{q \Subset q'\} \vdash_{A,B} c_1 \Subset c'_{k_1}}}{\Gamma \cup \{q_a \Subset q_b\} \cup \{q \Subset q'\} \vdash_{A,B} c_1 \Subset \{c'_k | c'_k \to_B q_b\}_1^m}\text{(Weark-r)} \quad \ldots\ldots \quad \dfrac{\dfrac{\prod_{c_n}}{\Gamma \cup \{q_a \Subset q_b\} \cup \{q \Subset q'\} \vdash_{A,B} c_n \Subset c'_{k_n}}}{\Gamma \cup \{q_a \Subset q_b\} \cup \{q \Subset q'\} \vdash_{A,B} c_n \Subset \{c'_k | c'_k \to_B q_b\}_1^m}\text{(Weark-r})}{\Gamma \cup \{q_a \Subset q_b\} \cup \{q \Subset q'\} \vdash_{A,B} \{c_i | c_i \to_A q_a\}_1^n \Subset \{c'_i | c'_i \to_B q_b\}_1^m}\text{(Split-l)}}{\Gamma \cup \{q \Subset q'\} \vdash_{A,B} q_a \Subset q_b}\text{(Induction)}$$

Où chaque $\prod_{c_i}$ est construit de la manière suivante (en supposant que $c_i = f(q_{i_1}, \ldots, q_{i_n})$ et $c'_{k_i} = f(q'_{i_1}, \ldots, q'_{i_n})$) :

$$\dfrac{\dfrac{\prod_{i_1}}{\Gamma \cup \{q_a \Subset q_b\} \cup \{q \Subset q'\} \vdash_{A,B} q_{i_1} \Subset q'_{i_1}} \quad \ldots\ldots \quad \dfrac{\prod_{i_n}}{\Gamma \cup \{q_a \Subset q_b\} \cup \{q \Subset q'\} \vdash_{A,B} q_{i_n} \Subset q'_{i_n}}}{\Gamma \cup \{q_a \Subset q_b\} \cup \{q \Subset q'\} \vdash_{A,B} f(q_{i_1}, \ldots, q_{i_n}) \Subset f(q'_{i_1}, \ldots, q'_{i_n})}\text{(Config)}$$

Si on essaye de construire la preuve du but $\Gamma \vdash_{A,B} q_a \Subset q_b$, il commence nécessairement de la même manière, excepté que $\{q \Subset q'\}$ n'apparaîtra pas dans la liste des hypothèses, *i.e.* dans la partie gauche du but. Chaque branche de cet arbre se terminera avec une conclusion de la forme $\Gamma \cup \{q_a \Subset q_b\} \vdash_{A,B} q_{i_j} \Subset q'_{i_j}$.

Pour conclure la preuve, il reste à donner les arbres de preuve $\prod'_{i_j}$ pour chacun des buts correspondants. On sait qu'il existe des arbres de preuve $\prod_{i_j}$ pour chaque $\Gamma \cup \{q \Subset q'\} \cup \{q_a \Subset q_b\} \vdash_{A,B} q_{i_j} \Subset q'_{i_j}$. On peut utiliser l'hypothèse d'induction sur $\prod_{i_j}$ pour obtenir $\prod'_{i_j}$:

- Puisque $\mu(\Gamma) = n + 1$, on a $\mu(\Gamma \cup \{q_a \Subset q_b\}) = n$
- Comme $\prod$ est une preuve de $\Gamma \vdash_{A,B} q \Subset q'$, c'est aussi une preuve de $\Gamma \cup \{q_a \Subset q_b\} \vdash_{A,B} q \Subset q'$.
- Chaque $\prod_{i_j}$ est une preuve de $\Gamma \cup \{q_a \Subset q_b\} \cup \{q \Subset q'\} \vdash_{A,B} q_{i_j} \Subset q'_{i_j}$

En utilisant le principe d'induction, on en déduit que pour tout i, j il existe une arbre de preuve $\prod'_{i_j}$ pour $\Gamma \cup \{q_a \Subset q_b\} \vdash_{A,B} q_{i_j} \Subset q'_{i_j}$. Ce qui termine la construction de l'arbre pour le but $\Gamma \vdash_{A,B} q_a \Subset q_b$. □

Théorème 5.4.7. *(Correction) Soient deux automates d'arbres A et B, si il existe un arbre $\prod$ de preuve pour $\emptyset \vdash_{A,B} q \Subset q'$ alors on a $\mathcal{L}(A, q) \subseteq \mathcal{L}(B, q')$*

Démonstration. On montre que la propriété $\forall t,\ t \in \mathcal{L}(A, q) \implies t \in \mathcal{L}(B, q')$ est vérifiée par induction sur le terme t. On pose $t = f(t_1, \ldots, t_n)$. On suppose alors que la propriété est vraie pour chacun sous-terme t_i, *i.e.* pour tous les états q_i, q'_i tels que si il existe un arbre de preuve $\prod_i$ pour $\emptyset \vdash_{A,B} q_i \Subset q'_i$ alors $t_i \to_A^* q \implies t \to_B^* q'_i$. Comme $t = f(t_1, \ldots, t_n) \in \mathcal{L}(A, q)$, alors pour chaque sous-terme t_i, on sait qu'il existe n états $q_1, \ldots, q_n$ tels que $t_i \in \mathcal{L}(A, q_i)$ et $f(q_1, \ldots, q_n) \to q \in A$. D'autre part, en dépliant l'arbre de preuve $\prod$ de $\emptyset \vdash_{A,B} q \Subset q'$, on peut déduire que pour chaque transition de la forme $f(q_1, \ldots, q_n) \to q \in A$, il existe une transition $f(q'_1, \ldots, q'_n) \to q' \in B$ telle que on ait un arbre de preuve $\prod_i$ pour $\{q \Subset q'\} \vdash_{A,B} q_i \Subset q'_i$. Comme $f(q_1, \ldots, q_n) \to q \in A$, on obtient que $f(q'_1, \ldots, q'_n) \to q' \in B$ et un arbre de preuve $\prod_i$ pour $\{q \Subset q'\} \vdash_{A,B} q_i \Subset q'_i$. Pour conclure que $f(t_1, \ldots, t_n) \in \mathcal{L}(B, q')$ il suffit de montrer que $t_i \in \mathcal{L}(B, q'_i)$. On remarque que l'on a un arbre de preuve $\prod_i$ pour $\{q \Subset q'\} \vdash_{A,B} q_i \Subset q'_i$ et que pour appliquer

l'hypothèse d'induction, on a besoin d'un arbre de preuve pour $\emptyset \vdash_{A,B} q_i \Subset q'_i$. C'est ici que l'on utilise le lemme 5.4.6 sur $\prod$ et $\prod_i$, on peut en déduire l'existence de $\prod'_i$, l'arbre de preuve pour $\emptyset \vdash_{A,B} q_i \Subset q'_i$. Alors en utilisant l'hypothèse d'induction sur t'_i, q_i, q'_i et $\prod'_i$, on obtient que pour chaque sous-terme $t_i \in \mathcal{L}(A, q_i)$, on a aussi $t_i \in \mathcal{L}(B, q'_i)$. Enfin, comme $f(q'_1, \ldots, q'_n) \to q' \in B$, on en déduit que $t = f(t_1, \ldots, t_n) \in \mathcal{L}(B, q')$. □

5.4.8 Preuve de complétude

Comme expliqué précédemment, l'algorithme présenté n'est pas complet en général. Cependant, on va montrer qu'il est complet pour les automates d'arbres produits par la complétion. En effet, si $A^k_{\mathcal{R}}$ est obtenu après k étapes de complétion à partir de l'automate A^0 alors on peut construire une preuve $\prod$ pour le but $\emptyset \vdash_{A^0, A^k_{\mathcal{R}}} \{\#(q) \mid q \in \mathcal{Q}_{F_0}\} \Subset \{\#(q') \mid q' \in \mathcal{Q}_{F_k}\}$. On rappelle que l'automate produit par la $k^{\text{ème}}$ étape de complétion est noté $\mathcal{A}_k = \langle \mathcal{F}, \mathcal{Q}_k, \mathcal{Q}_{F_k}, \Delta_k \rangle$.

Pour rappel, la complétion d'automates d'arbres effectue principalement deux opérations à chaque étape de calcul : la *normalisation* et la *fusion d'états*. Dans le cas de la normalisation, l'inclusion des langages peut être simplement démontrée en utilisant l'inclusion des états de transitions,la normalisation ne produisant que de nouvelles transitions. Pour l'opération de la fusion d'états, l'inclusion des ensembles de transitions n'est plus suffisante, puisque la fusion d'états implique aussi la fusion de transitions. C'est pour cette raison que l'on définit une nouvelle relation d'ordre préservée par chaque opération.

Définition 5.4.9. *Soient A, B deux automates. On définit $\sqsubseteq$ la relation réflexive et transitive comme : $A \sqsubseteq B$ si il existe une fonction ϱ qui renomme les états de A dans les états de B et telle que toutes les transitions renommées de Δ_A sont contenues dans Δ_B :*

$$A \sqsubseteq B \iff \exists \varrho : \mathcal{Q}_A \to \mathcal{Q}_B,\ \varrho(\Delta_A) \subseteq \Delta_B \ \wedge\ \varrho(Q_{F_A}) \subseteq Q_{F_B} \tag{5.1}$$

On étend l'application du renommage ϱ pour les ensembles et les configurations de la manière suivante :

- $\varrho(\{q_i\}_1^n)$ est utilisée pour $\{\varrho(q_i)\}_1^n$
- $\varrho(c) = \begin{cases} f(\varrho(c_1), \ldots, \varrho(c_n)) & \text{si } c = f(c_1, \ldots, c_n) \\ c & \text{si } c \in \mathcal{F}_0 \\ \varrho(q) & \text{si } c = q \in Q \end{cases}$
- $\varrho(c \to q)$ est utilisée pour $\varrho(c) \to \varrho(q)$
- $\varrho(\Delta)$ signifie $\{\varrho(c \to q) \mid c \to q \in \Delta\}$.

Le lemme suivant montre que la *normalisation* et la *fusion d'états* préserve la relation $\sqsubseteq$.

Lemme 5.4.10. *Soit A un automate d'arbres, et Δ_A son ensemble de transitions. On considère A' tel que :*

1. *si* $\Delta_{A'} = \Delta_A \cup \texttt{Norm}(r\sigma \to q)$ *alors* $A \sqsubseteq A'$
2. *si* $A' = Merge(A, q_1, q_2)$ *alors* $A \sqsubseteq A'$

où $Merge(A, q_1, q_2)$ l'automate obtenu par la fusion des états q_1 et q_2 de A.

Démonstration. 1. Le premier point est très facile à montrer puisqu'on a évidemment $\Delta_{A'} \supseteq \Delta_A$ quelque soient $r\sigma$ et q choisis. Il suffit de prendre $\varrho = id$, on peut immédiatement conclure $A \sqsubseteq A'$.

2. On considère Δ_A l'ensemble des transitions de A. Soient q_1 et q_2 deux états que l'on veut fusionner. On peut alors appliquer à Δ_A un fonction de renommage ϱ qui donne le même résultat que la fusion d'états pour $q_1 = q_2$:

$$\varrho(q) = \begin{cases} \text{si } (q = q_2) \\ \quad q_1 \\ \text{sinon} \\ \quad q \end{cases}$$

Ainsi la fusion d'états construit $\Delta_{A'} = \varrho(\Delta_A)$ et par la définition 5.4.9 on en déduit $\Delta_A \sqsubseteq \Delta_{A'}$.

□

Théorème 5.4.11. *Pour $A^0_{\mathcal{R},E}$ un automate d'arbres donné, $\mathcal{R}$ un ensemble de règles de réécriture et E un ensemble d'équations, après k étapes de complétion on obtient $A^k_{\mathcal{R},E}$ tel que $A^0 \sqsubseteq A^k_{\mathcal{R},E}$.*

Démonstration. Par induction sur k :

- Comme $\sqsubseteq$ est réflexive, on a évidemment $A^0_{\mathcal{R},E} \sqsubseteq A^0_{\mathcal{R},E}$.
- Soit $A^k_{\mathcal{R},E}$ l'automate d'arbres obtenu après k étapes de complétion tel que $A^0_{\mathcal{R},E} \sqsubseteq A^k_{\mathcal{R},E}$. Par définition de la complétion $A^{k+1}_{\mathcal{R},E}$ est construit à partir de $A^k_{\mathcal{R},E}$ en appliquant successivement la normalisation puis la fusion. Alors on a $A^k_{\mathcal{R},E} \sqsubseteq A^{k+1}_{\mathcal{R},E}$. Par transitivité de $\sqsubseteq$, à partir de$A^0 \sqsubseteq A^k_{\mathcal{R}}$ et $A^k_{\mathcal{R}} \sqsubseteq A^{k+1}_{\mathcal{R}}$ on en déduit immédiatement que $A^0 \sqsubseteq A^{k+1}_{\mathcal{R}}$. En effet, la relation $\sqsubseteq$ est transitive, pour cela il suffit de considérer la composition des fonctions de renommage, et le fait que l'inclusion d'ensemble est aussi une relation transitive.

□

Maintenant, on peut définir la propriété de complétude vis-à-vis de cette relation d'ordre :

Théorème 5.4.12. *(Complétude) Pour deux automates d'arbres donnés A et B, si on a $A \sqsubseteq B$ alors il existe $\prod$ un arbre de preuve pour $\emptyset \vdash_{A,B} \{\#(q_f) \mid q_f \in \mathcal{Q}_{F_A}\} \Subset \{\#(q'_f) \mid q'_f \in \mathcal{Q}_{F_B}\}$.*

Démonstration. Par définition de $A \sqsubseteq B$, on sait qu'il existe une fonction de renommage ϱ.

Tout d'abord, on montre par induction sur l'arbre de preuve que pour tout Γ et q, $\Gamma \vdash_{A,B} q \Subset \varrho(q)$:
L'hypothèse d'induction est alors $\forall \Gamma,\ q,\ q_i$,

$$\frac{\prod_i}{\Gamma \cup \{q \Subset \varrho(q)\} \vdash_{A,B} q_i \Subset \varrho(q_i)}$$

On veut construire un arbre de preuve pour $\Gamma \vdash_{A,B} q \Subset \varrho(q)$.
Deux cas sont envisageables :

– si $q \Subset \varrho(q) \in \Gamma$ alors on peut conclure immédiatement :

$$\text{(Axiom)}\ \frac{}{\Gamma' \cup \{q \Subset \varrho(q)\} \vdash_{A,B} q \Subset \varrho(q)}$$

– Sinon,il faut appliquer la règle Induction pour obtenir un arbre de la forme :

$$\text{(Split-l)}\ \frac{\dfrac{\prod_{c_1}}{\Gamma \cup \{q \Subset \varrho(q)\} \vdash_{A,B} c_1 \Subset \{c'_k | c'_k \to \varrho(q)\}_1^m} \quad \ldots\ldots \quad \dfrac{\prod_{c_n}}{\Gamma \cup \{q \Subset \varrho(q)\} \vdash_{A,B} c_n \Subset \{c'_k | c'_k \to \varrho(q)\}_1^m}}{\text{(Induction)}\ \dfrac{\Gamma \cup \{q \Subset \varrho(q)\} \vdash_{A,B} \{c_i | c_i \to q\}_1^n \Subset \{c'_k | c'_k \to \varrho(q)\}_1^m}{\Gamma \vdash_{A,B} q \Subset \varrho(q)}}$$

A partir de l'hypothèse $\varrho(\Delta_A) \subseteq \Delta_B$ pour chaque transition $c \to q$ de Δ_A, on a $\varrho(c \to q) \in \Delta_B$.

$$\text{Ainsi, pour tout } (c \to q) \in \Delta_A, \text{ on a } \varrho(c) \in \{c'_k | c'_k \to \varrho(q)\}_1^m$$

Pour chaque $c_i = f_i(q_{i_1}, \ldots, q_{i_n})$ on peut construire l'arbre correspondant $\prod_{c_i}$ dont chaque branche est conclue par $\prod_{i_j}$ une instance de l'hypothèse d'induction pour q_{i_j} l'état correspondant :

$$\text{(Config)}\ \frac{\dfrac{\prod_{i_1}}{\Gamma \cup \{q \Subset \varrho(q)\} \vdash_{A,B} q_{i_1} \Subset \varrho(q_{i_1})} \quad \ldots\ldots \quad \dfrac{\prod_{i_n}}{\Gamma \cup \{q \Subset \varrho(q)\} \vdash_{A,B} q_{i_n} \Subset \varrho(q_{i_n})}}{\text{(Weak-r)}\ \dfrac{\Gamma \cup \{q \Subset \varrho(q)\} \vdash_{A,B} c_i \Subset \varrho(c_i)}{\Gamma \cup \{q \Subset \varrho(q)\} \vdash_{A,B} c_i \Subset \{c'_k | c'_k \to \varrho(q)\}_1^m}}$$

D'autre part, on sait que pour tout Γ et $q \in Q_A$ il existe un arbre de preuve $\prod_q$ pour tout but $\Gamma \vdash_{A,B} q \Subset \varrho(q)$.
En particulier, cela est vrai pour $\Gamma = \emptyset$ et pour tout état q de Q_{F_A}. Comme on a $A \sqsubseteq B \implies \varrho(Q_{F_A}) \subseteq Q_{F_B}$, on peut construire un arbre de preuve tel que :

$$\text{(Split-l)}\ \frac{\text{(Config)}\ \dfrac{\text{(Weak-r)}\ \dfrac{\dfrac{\prod_{q_{f_1}}}{\emptyset \vdash_{A,B} q_{f_1} \Subset \varrho(q_{f_1})}}{\emptyset \vdash_{A,B} \#(q_{f_1}) \Subset \#(\varrho(q_{f_1}))}}{\emptyset \vdash_{A,B} q_{f_1} \Subset Q_{F_B}} \quad \ldots\ldots \quad \dfrac{\text{(Weak-r)}\ \dfrac{\text{(Config)}\ \dfrac{\dfrac{\prod_{q_{f_n}}}{\emptyset \vdash_{A,B} q_{f_1} \Subset \varrho(q_{f_1})}}{\emptyset \vdash_{A,B} \#(q_{f_n}) \Subset \#(\varrho(q_{f_n}))}}{\emptyset \vdash_{A,B} \#(q_{f_n}) \Subset \{\#(q) \mid Q_{F_B}\}}}{\emptyset \vdash_{A,B} \{\#(q) \mid q \in Q_{F_A}\} \Subset \{\#(q) \mid q \in Q_{F_B}\}}$$

□

Ainsi, on peut assurer que pour un automate $A^k_{\mathcal{R},E}$ obtenu après k étapes de complétion à partir $A^0_{\mathcal{R},E}$, il existe une preuve $\prod$ pour le but $\emptyset \vdash_{A^0, A^k_{\mathcal{R}}} \{\#(q) \mid q \in \mathcal{Q}_{F_0} \Subset \{\#(q') \mid q' \in \mathcal{Q}_{F_k}\}$. Il peut être vu comme une conséquence des deux théorèmes précédents.

A propos de l'incomplétude : Pour obtenir une procédure de décision pour tout automate d'arbres (*i.e.* des automates d'arbres obtenus autrement que par la complétion), la règle "Weak-r" doit être modifiée. C'est à cause de cette règle que le système est incomplet en oubliant de prendre en compte d'autres manière de construire l'union.

Exemple 5.4.13. *Soient deux automates d'arbres A et B tels que :*

$$A = \left\{ \begin{array}{rcl} a & \to & q_1 \\ b & \to & q_2 \\ c & \to & q_2 \\ f(q_1, q_2) & \to & \boldsymbol{q} \end{array} \right\} \text{ and } B = \left\{ \begin{array}{rcl} a & \to & q'_1 \\ b & \to & q'_2 \\ c & \to & q'_3 \\ f(q'_1, q'_2) & \to & \boldsymbol{q'} \\ f(q'_1, q'_3) & \to & \boldsymbol{q'} \end{array} \right\}$$

Cet exemple illustre l'incomplétude du système. Les deux automates d'arbres reconnaissent clairement le même langage $\mathcal{L}(A, q) = \mathcal{L}(B, q')$. Si on peut construire une preuve pour le but $\emptyset \vdash_{B,A} q' \Subset q$, ce n'est en revanche pas le cas pour le but $\emptyset \vdash_{A,B} q \Subset q'$. En effet, la règle "Weak-r" ne peut montrer le but $\Gamma \vdash_{A,B} \{f(q_1, q_2)\} \Subset \{f(q'_1, q'_2), f(q'_1, q'_3)\}$ qu'à partir du but $\Gamma \vdash_{A,B} f(q_1, q_2) \Subset f(q'_1, q'_2)$ ou $\Gamma \vdash_{A,B} f(q_1, q_2) \Subset f(q'_1, q'_3)$ qui sont tous les deux faux.

5.4.14 Complexité

Comme expliqué précédemment dans la section 5.1, l'algorithme standard pour vérifier l'inclusion $\mathcal{L}(A) \subseteq \mathcal{L}(B)$ est basé sur le calcul de $\overline{B}$ l'automate complément. Cependant, pour les automates d'arbres non-déterministes, la taille [18] de $\overline{B}$ peut être exponentiellement plus grande que la taille de B. L'algorithme proposé ici n'a pas cet inconvénient et utilise seulement une taille mémoire qui est polynomiale par rapport à la taille des automates considérés.

On pose $|Q|$ le nombre maximum d'états des deux automates d'arbres A et B. Les arbres de preuves construits en utilisant les règles de déduction données sont au plus de hauteur $|Q|^2$. On doit cela aux règles 'Induction' et 'Axiom' qui assurent que chaque problème d'inclusion $q \Subset q'$ sera analysé seulement une fois par branche. Comme on a $q \in Q_A$ et $q' \in Q_B$ et que l'on sait que les cardinal de Q_A et Q_B est borné par $|Q|$, la longueur de la branche est au plus bornée par $|Q|^2$. De plus, l'algorithme `inclusion` explorant chaque branche de l'arbre de preuve une par une, la mémoire utilisée est donc bornée par $|Q|^2$, et donc polynomiale.

Cependant, la complexité d'une exécution de l'algorithme tel qu'il est implémenté est exponentielle. En effet, même si chaque $q \Subset q'$ n'est considéré qu'une seule fois par branche, le nombre de branches à considérer est exponentiel par rapport à Q et un même couple $q \Subset q'$ peut être rencontré dans plusieurs branches et donc le but peut-être reprouvé plusieurs fois. Une optimisation très simple de cet algorithme consiste à tabuler une partie des résultats pour certains des couples $q \Subset q'$. Ainsi, on en utilisant le lemme 5.4.6 pour chacun des couples $q \Subset q'$ pour lesquels la propriété a été prouvée on peut l'ajouter à la liste des hypothèses des buts suivants. Par exemple, en utilisant cette optimisation pour tous

les couples, chaque couple n'est alors plus considéré qu'une *seule fois*, dans tout l'arbre de preuve. Si les opérations d'accès à la table était réalisé en temps constant, cela ramènerait la complexité polynomiale en temps bornée par $|Q|^2$. Néanmoins, on n'a jamais besoin de tabuler la totalité des états car cela augmente par ailleurs la consommation de mémoire). En fait, on peut voir cette liste de couples, comme une stratégie pour définir un ordre sur les sous-branches de la preuve que l'on souhaite réutiliser plusieurs fois. L'approche est correcte car si une branche ne peut être prouvée pour une certaine paire d'états, elle ne sera pas ajoutée à l'environnement. On peut donc définir soit à la main ou par l'intermédiaire d'un oracle la liste d'états pour laquelle on souhaite construire et conserver les preuves.

5.5 Formalisation de la clôture par réécriture

Dans cette section, on s'intéresse à définir formellement le prédicat `IsClosed`, la fonction `closure` et on montre que cette fonction est correcte par rapport à la propriété de clôture `IsClosed`. On rappelle que pour vérifier qu'un automate d'arbres $A = \langle Q_F,\ \Delta \rangle$ est clos pour un système de règles de réécriture $\mathcal{R}$, il est suffisant de montrer que pour tout terme $t \in \mathcal{L}(A)$, si le terme t' est atteignable à partir de t par $\rightarrow_{\mathcal{R}}^*$ alors t' est aussi reconnu par l'automate. En se basant sur les spécifications de la réécriture en Coq et des automates d'arbres présentées dans les sections 5.2 et 5.3, on peut définir plus formellement le prédicat `IsClosed` et le théorème `closure_sound` à montrer :

```
Definition IsClosed (R : list rule) (A : t_aut) : Prop :=
   ∀ q t t', IsRec A.delta q t →  Reachable R t t' →  IsRec A.delta q t'.

Theorem closure_sound:
      ∀ R A', closure R A' = true →  IsClosed R A'.
```

L'algorithme utilisé pour vérifier la clôture de A par $\mathcal{R}$ calcule pour chaque règle $l \rightarrow r \in \mathcal{R}$ l'ensemble de toutes les substitutions σ telles que $l\sigma \rightarrow_\Delta^* q$ puis vérifie pour chaque σ que $r\sigma \rightarrow_\Delta^* q$. Ensuite, la preuve de correction consiste à prouver que si la fonction `closure` répond vrai, alors le langage $\mathcal{L}(A)$ est clos par $\rightarrow_{\mathcal{R}}$.

On donne des indications pour définir la fonction `closure`. Tout d'abord, pour toute règle $l \rightarrow r$ de $\mathcal{R}$, il faut trouver toutes les substitutions $\sigma : \mathcal{X} \mapsto Q$ pour chaque état $q \in Q$ telles que $l\sigma \rightarrow_\Delta^* q$, ce qui correspond au *problème de filtrage*. Dans un second temps, il faut vérifier que pour tout couple état-substitution q, σ trouvé, la paire critique est bien résolue c'est à dire $r\sigma \rightarrow_\Delta^* q$. Enfin, dans le théorème de correction, on doit montrer que toutes les substitutions $\sigma : \mathcal{X} \mapsto Q$ couvrent l'ensemble des substitutions sur les termes, *i.e.* les substitution de la forme $\sigma' : \mathcal{X} \mapsto \mathcal{T}(\mathcal{F})$, et ainsi que tous les termes atteignables sont couverts.

On rappelle que le problème de filtrage $l \trianglelefteq q$ consiste à trouver toutes les substitutions $\sigma : \mathcal{X} \mapsto Q$ pour chaque état $q \in Q$ telles que $l\sigma \rightarrow_\Delta^* q$. L'algorithme résolvant ce problème est initialement défini dans [29]. Ici, on présente une version plus algorithmique correspondant à la version implémentée en Coq :

$$\text{(Var)}\ \frac{}{x \trianglelefteq q,\ S \vdash_A \{\sigma \cup \{x \mapsto q\} \mid \sigma \in S\}}(x \in \mathcal{X})$$

$$\text{(Unfold)}\ \frac{t_1 \trianglelefteq q_1,\ S_0 \vdash_A S_1 \quad \ldots \quad t_n \trianglelefteq q_n,\ S_{n-1} \vdash_A S_n}{f(t_1,\ldots,t_n) \trianglelefteq f(q_1,\ldots q_n),\ S_0 \vdash_A S_n}$$

$$\text{(Delta)}\ \frac{f(t_1,\ldots,t_n) \trianglelefteq f(q_1^1,\ldots,q_n^1)\ S_0 \vdash_A S_1 \quad \ldots \quad f(t_1,\ldots,t_n) \trianglelefteq f(q_1^m,\ldots,q_n^m),\ S_0 \vdash_A S_m}{f(t_1,\ldots,t_n) \trianglelefteq q,\ S_0 \vdash_A \bigcup_1^m S_i}(\text{avec } \{f(q_1^i,\ldots,q_n^i) \to q \in$$

On énonce le problème de filtrage pour la règle $l \to r$ et l'état q de l'automate A, par $l \trianglelefteq q,\ \{id\} \vdash_A S$ où S contient l'ensemble des substitutions telles que $l\sigma \to_\Delta^* q$. On remarque que l'ensemble initial est le singleton contenant id la substitution identité.

Théorème 5.5.1. *Pour toute substitution σ telle que $l\sigma \to_\Delta^* q$, alors cette substitution est déduite par l'algorithme de filtrage.*

Démonstration. La preuve est similaire à la preuve de complétude 3.9.4 de l'algorithme de filtrage donnée au chapitre 3. □

L'implémentation de la fonction `matching_rec` est Coq très proche des règles de cet algorithme. L'ensemble des substitutions est définie par une `list` de substitutions, et les substitutions sont définies comme des fonctions. La signature Coq de la fonction de filtrage est alors :

```
Definition substitutions := list substitution.

Definition matching_rec :
      Delta.t → substitutions → state → term → substitutions :=

   (* ... Body ... *)
.
```

Le but $l \trianglelefteq q,\ \{id\} \vdash_A S$ se pose par l'intermédiaire de la définition `matching` dont l'évaluation renvoie S. Lorsque le filtrage échoue, c'est à dire si il n'existe aucune substitution telle que $l\sigma \to_\Delta^* q$, la fonction `matching` retourne simplement la liste vide `nil`, ce qui correspond simplement à l'ensemble vide. Pour cet algorithme, la terminaison est syntaxique : la fonction `matching_rec` est définie par récurrence structurelle sur le terme l passé en argument à la fonction. On énonce la propriété de complétude du filtrage de la manière suivante :

```
(* Le singleton {id} est represente par id_subst::nil *)
Definition matching D q l := matching_rec D (id_subst::nil) l q.

(* L'expression (Fv l) calcule l'ensemble des variables du terme l *)
Theorem matching_spec:
   ∀ Δ l σ q,
      Qsubst σ → TFX l → linear l → IsRed Δ q (l @ σ) →

            ∃ σ', σ' ∈(matching Δ q l) ∧ (∀ x, x ∈(Fv l) → σ' x = σ x).
```

Le prédicat `Qsubst: substitution → Prop` s'assure que la substitution σ ne substitue les variables que par des états, ce qui se résume à $\sigma : \mathcal{X} \to Q$. Le prédicat `linear l` s'assure que le terme `l` est le membre d'une règle linéaire à gauche. Le prédicat `IsRed` est une simple

extension du prédicat `IsRec` introduit à la section 5.3 pour étendre la reconnaissabilité des termes clos $\mathcal{T}(\mathcal{F})$ à l'ensemble des configurations $\mathcal{T}(\mathcal{F} \cup Q)$. On peut noter aussi une différence entre l'énoncé du théorème 5.5.1 et son équivalent `matching_spec`. En effet, dans ce dernier cas, on ne montre pas directement que la substitution σ appartient à l'ensemble `(matching Δ q l)` mais qu'il existe une substitution σ' telle que son image est égale à celle de σ pour chaque variable `x` de `l` le terme filtré. Cette différence s'explique par l'usage implicite de l'extensionnalité dans la preuve papier. Cette propriété n'est pas vraie en Coq et plus généralement en informatique, puisqu'il est toujours possible de donner deux implémentations différentes pour une même fonction. Par exemple, les deux substitutions suivantes sont équivalentes du point de vue de leur résultat mais pas égales du point de vue syntaxique :

```
Definition σ x :=
  if (id_eq x 0) then
     Some 1
  else if (id_eq x 1) then
     Some 2
  else None.
```

```
Definition σ' x :=
  if (id_eq x 1) then
     Some 2
  else if (id_eq x 0) then
     Some 1
  else None.
```

En fait, la propriété énoncée telle quelle dans le théorème est suffisante, il n'est pas nécessaire d'avoir l'égalité entre σ et σ'. Cela étant, le schéma de la preuve Coq reste similaire à la preuve donnée ci-dessus.

La deuxième partie de la fonction `closure` consiste à vérifier que pour chaque substitution σ telle $\sigma \in S$ où S est l'ensemble de substitutions calculé par le filtrage, on a $r\sigma \rightarrow_{\Delta}^{*} q$, où r est le membre droit de la règle de réécriture. Cette vérification est effectuée par la fonction `all_reduce_dec`. Comme dans le cas de l'inclusion, on utilise une spécification avec une signature riche. Donnée par le théorème suivant, cette spécification nous indique que `all_reduce_dec Δ q r S` permet de conclure soit que toutes les configurations `rσ` sont réduites en l'état `q`, soit qu'il existe une substitution telle que la réduction ne soit pas possible.

```
Definition all_reduce_dec Δ q (r: term) (S: substitutions): boolopt substitution.

Theorem all_reduce_dec_spec:
   ∀ Δ q r S, all_reduce_dec Δ q r S = ok →
      ∀ σ, σ ∈ S → IsRed Δ q (r @ s).
```

Démonstration. La preuve ici est très simple. Elle se fait par induction sur `S` la liste des substitutions. Si la liste est vide la propriété est vérifiée. Sinon on décompose la liste en $\sigma :: l$. On se base alors sur un théorème qui énonce la décidabilité du prédicat `IsRed`. Ainsi `IsRed Δ q t` est équivalent à la relation de reconnaissabilité $t \rightarrow_{\delta}^{*} q$ pour $t \in \mathcal{T}(\mathcal{F} \cup Q)$. Or la reconnaissabilité est bien sûr décidable, ce qui s'énonce en Coq par le théorème suivant

```
Lemma IsRed_dec:
   ∀ Δ q t, {IsRed Δ q t} + {¬ IsRed Δ q t}.
```

On regarde alors le résultat de `IsRed_dec Δ q (r @ σ)` :

- Si la réponse est négative, alors il n'est pas nécessaire d'aller plus loin, la propriété est violée. En fait, la fonction `all_reduce_dec` ne renvoie pas simplement `ok` ou `ko`, mais renvoie un contre-exemple. Donc la fonction va renvoyer `ko σ`.
- Lorsque la réponse est positive, on a alors une preuve `IsRed Δ q (r @ σ)`. En utilisant l'hypothèse d'induction sur le reste de la liste l, on regarde le résultat. En cas d'échec, on se contente de faire suivre le contre-exemple rencontré dans la liste, puisque si il existe $\sigma' \in l$ qui viole le prédicat `IsRed`, on a bien $\sigma' \in \sigma :: l$ qui est un contre-exemple pour `all_reduce_dec Δ q r (σ :: l)`. Sinon, on sait que toutes les substitutions de l respectent la propriété. Comme c'est aussi le cas pour σ, on en déduit que pour toutes les substitutions `s` de la liste $\sigma :: l$, on a bien `IsRed Δ q (r @ s)`.

□

Il est important de remarquer que la signature de la fonction `all_reduce_dec` est basée sur le type de données `boolopt` qui correspond à construire un type de données isomorphe au booléen paramétré par un type. En cas d'échec, on aimerait récupérer la substitution la substitution pour laquelle la clôture n'est pas vérifiée, celle-ci constituant simplement un contre-exemple. Cela permet de donner un caractère plus informatif à la procédure de vérification, en précisant la nature de l'échec : la règle de réécriture $l \to r$, l'état q, et la substitution σ. Pour cela, on a donc introduit le type suivant :

```
Inductive boolopt (A: Type): Type :=
  | ok : sumdec A
  | ko : A →  sumdec A.
```

Avec une expertise un peu plus avancée, on peut montrer que le type `boolopt` est équivalent au type `option`. Si d'un point de vue logique les types sont effectivement équivalents, il en est tout autrement d'un point de vue syntaxique. En effet, il est possible de les utiliser tous les deux avec la construction **`if`** … **`then`** … **`else`** … qui se réduit pour tout type isomorphe à `bool` mais qui suppose toujours que la branche gauche correspond au cas "positif", ce qui n'est pas le cas dans l'utilisation du type `option` lorsqu'on l'utilise pour renvoyer un contre-exemple : dans ce cas, on suppose que le constructeur `Some: A → option A` correspond à la branche négative puisqu'il est sensé contenir un contre-exemple. Or la définition du type option introduit dans l'ordre le constructeur `Some` suivi de `None` ce qui correspond implicitement à l'ordre branche "positive" puis branche "négative". C'est pour cette raison que l'on introduit le type `boolopt` pour être cohérent par rapport à cette notion de branchement. Privilégier le type `boolopt` par rapport au type `option` évite d'avoir à renommer systématiquement par la négative toutes les fonctions dans le cas où l'on veut renvoyer un contre-exemple, ce qui peut-être une source d'incompréhension et d'erreur.

En combinant les fonctions `matching` et `all_red`, on obtient l'algorithme pour vérifier toutes les paires critiques trouvées à l'état q avec la règle $l \to r$:

```
Definition closure_at_state Δ q l r :=
   catch
          all_red Δ q r (matching Δ q l)
```

```
(fun σ ⇒ Closure_ko q l r σ).

Theorem closure_at_state_spec :
  ∀ Δ q l r, closure_at_state Δ q l r = ok →
    (∀ σ, IsRed Δ q (l @ σ) → IsRed Δ q (r @ σ)).
```

Pour une règle donnée $l \to r$ et un état donné q, cette fonction répond `ok` si pour toute substitution $\sigma : \mathcal{X} \mapsto Q$ telle que $l\sigma \to_\Delta^* q$ alors on sait que $r\sigma \to_\Delta^* q$. Sinon, on peut voir l'emploi de la fonction `catch` qui en cas d'échec de la vérification par la fonction `all_dec` capture le contre-exemple (une substitution) pour construire et retourner l'ensemble des éléments qui forment le contexte à l'origine l'échec : la valeur de retour `ko(Closure_ko q l r σ)` nous indique que la clôture a échoué pour $r\sigma \not\to_\Delta^* q$ alors que l'on a bien $l\sigma \to_\Delta^* q$, la paire critique semble ne pas être résolue. La preuve du théorème `closure_at_state_spec` est une conséquence immédiate du théorème `matching_spec` et du théorème `all_reduce_dec_spec`.

Des configurations aux termes. Pour l'instant la vérification est faite seulement pour les substitutions calculées par le filtrage qui sont de la forme $\sigma : \mathcal{X} \mapsto Q$ (`Qsubst σ`). Symboliquement, la réécriture de la configuration $l\sigma$ en $r\sigma$ à l'état q consiste en la réécriture de n'importe quel terme clos de la forme $l\sigma' \to_\Delta^* q$ en un terme clos $r\sigma' \to_\Delta^* q$. On va donc transférer cette propriété de clôture aux substitutions $\sigma' : \mathcal{X} \mapsto \mathcal{T}(\mathcal{F})$. On pourra alors en déduire que si la vérification par `closure_at_state Δ q l r` est un succès à l'état q pour la règle $l \to r$ alors pour tout terme clos $u \in \mathcal{L}(\Delta, q)$ si u se réécrit en v à la racine par $l \to r$ alors $v \in \mathcal{L}(\Delta, q)$.

En fait, on doit être capable de passer des substitutions de $\mathcal{X} \to Q$ aux substitutions de $\mathcal{X} \to \mathcal{T}(\mathcal{F})$, mais on a aussi besoin de montrer que l'inverse est possible. Pour cela on a besoin des deux théorèmes suivants :

```
Theorem IsRed_subst: ∀ Δ c σ q, TFX c →

  (∀ x_i, x_i ∈Fv c → ∃q_i, s x = Some (State q_i)) →
    IsRed Δ q (c @ σ) →

  ∀ σ',
    (∀ x_i q_i, x_i ∈Fv c → σ x_i = Some (State q_i) → IsRec Δ q_i ((Var x_i) @ σ')) →
    IsRec Δ q (c @ σ').
```

Ce lemme permet de composer avec les prédicats de reconnaissabilité : on considère tous les termes de la forme $c\sigma' \in \mathcal{T}(\mathcal{F})$ avec $\sigma' : \mathcal{X} \mapsto \mathcal{T}(\mathcal{F})$ qui se réduisent $c\sigma' \to_\Delta^* c\sigma$ avec $\sigma : \mathcal{X} \mapsto Q$. On peut alors en déduire que si $c\sigma \to_\Delta^* q$ alors on a $c\sigma' \to_\Delta^* q$. La "petite" difficulté dans ce théorème est la formalisation $c\sigma' \to_\Delta^* c\sigma$ qui ne s'exprime pas immédiatement avec les prédicats `IsRec` et `IsRed`. On est obligé d'exprimer la propriété plus localement, c'est à dire d'exprimer comment est reconnu le sous-terme $\sigma'(x_i)$ associé à chaque variable x_i de c. Ce qui revient à décomposer la relation $c\sigma' \to_\Delta^* c\sigma$ en une suite de réductions de la forme $\sigma'(x_i) \to_\Delta^* \sigma(x_i)$.

```
Theorem IsRec_subst: ∀ c s q D, IsRec D q (c @ s) →
```

```
   ∃ s',
        (∀ x, ¬ x ∈ Fv c →  s' x = None)
     ∧
        (∀ x, x ∈ Fv c →  ∃ q', s' x = Some (State q') ∧
IsRec D q' ((Var x) @ s))
     ∧
        IsRed D q (c @ s').
```

Dualement, le théorème montre que tout terme clos de la forme $c\sigma \in \mathcal{T}(\mathcal{F})$ réduit par l'automate en l'état q par est forcément réécrit en une configuration intermédiaire de la forme $c\sigma'$ avec $\sigma' : \mathcal{X} \mapsto Q$. Ce qui correspond à :

$$\forall \sigma,\ \exists \sigma' \text{ t.q. } c\sigma \to_{\Delta}^{*} c\sigma' \text{ et } c\sigma' \to_{\Delta}^{*} q$$

Le théorème `IsRec_subst` utilise la même astuce pour formaliser la réduction de $c\sigma$ en $c\sigma'$. Pour ces deux théorèmes la preuve se fait simplement par induction sur le terme $c \in \mathcal{T}(\mathcal{F}, \mathcal{X})$.

En utilisant ces deux théorèmes, on peut reformuler le théorème `closure_at_state_spec` en transférant la propriété sur les termes clos, c'est à dire la relation de réécriture à la racine(`Trew u v`) entre deux termes clos :

```
Lemma closure_at_state_spec :
   ∀ Δ lr,
      ∀ l r Hsub Hl Hr, lr = LRule l r Hsub Hl Hr →
         closure_at_state Δ q l r = ok →
            ∀ u v, IsRec D q u →  TRew lr u v →  IsRec D q v.
```

Lorsque le membre droit est une variable... Dans le cas où le membre droit de la règle est une variable c'est à dire que l'on a une règle de la forme $f(x) \to x$. Supposons qu'il existe une substitution $\sigma : \mathcal{X} \to Q$ telle l'on ait la relation $f(q') \to_{\Delta}^{*} q$ à l'état $q \neq q'$. Comme on considère des automates d'arbres sans ε-transition, on ne peut pas avoir $q \to_{\Delta}^{*} q'$, la réflexivité de la relation ne pouvant s'appliquer à q et q'. Comme le prédicat `IsRed Δ q (Var x @ σ)` est faux, la procédure `closure_at_state Δ q l r` va échouer et rendre la substitution σ comme contre-exemple. Dans ce type de cas, la complétion (sans ε-transition) duplique chaque transition de la forme $f(q_1, \ldots, q_n) \to q'$ en une $f(q_1, \ldots, q_n) \to q$, ce qui est évidemment suffisant pour avoir la relation l'inclusion $\mathcal{L}(\Delta, q') \subseteq \mathcal{L}(\Delta, q)$. On ajoute donc la fonction `closure_at_q'` qui vérifie ce critère syntaxique :

```
Definition closure_at_q' :
        Delta.t →  state →  term →  ident →  boolopt closure_ko :=
   (*   ... Body ... *)
.

Lemma closure_at_q'_spec : ∀ Δ q l x,
      linear l →  In x (Fv l) →
         closure_at_q' D q l x = ok →
             ∀ s, IsRec D q (l @ s) →  IsRec D q (Var x @ s).
```

Il ne reste plus qu'à vérifier la propriété de clôture pour toutes les règles de réécriture à tous les états de l'automate. C'est le rôle jouée par la fonction `closure`, qui itère sur l'ensemble des couples règle-état de $\mathcal{R}$ × `(Delta.dom Δ)`.

```
(* L'ensemble [R: list lrule] est lineaire gauche *)

Lemma closure_spec_0 :
   ∀ Δ R, closure Δ R = ok →
      ∀ q lr, lr ∈ R →
         ∀ u v, IsRec D q u →  TRew lr u v →  IsRec D q v.
```

Enfin il est facile d'en déduire que la relation est suffisante pour en déduire la clôture de la relation de réécriture par contexte avec le prédicat `LRew` (la formulation inductive de la réécriture pour toute position p) puis par réflexivité et transitivité `ReachableL`. Ce qui permet de conclure à la reconnaissabilité de tout terme atteignable :

```
Theorem closure_spec:
   ∀ Δ R, closure Δ R = ok →
         ∀ q u v, IsRec Δ q u → Reachable R u v → IsRec Δ q v.
```

5.5.2 Extraction et benchmarks

A partir de la spécification formelle en Coq, il est possible d'extraire une implémentation du `checker` en OCaml. Le code extrait est interfacé avec un analyseur syntaxique qui accepte les fichiers générés par Timbuk. Si cette opération se déroule globalement bien, il subsiste quelques subtilités dues au mécanisme d'extraction dont il faut tenir compte pour que le `checker` OCaml soit conforme à sa spécification. En effet, pour produire des termes OCaml à partir des termes Coq, l'extraction nettoie les termes Coq de tout ce qui n'a pas trait au calcul, en particulier tout ce qui est du type **`Prop`** soit toutes les propriétés. Par exemple, si l'on reprend le cas de la définition d'une règle de réécriture linéaire à gauche, on a :

```
Inductive l_rule: Set :=
| Lrule (l r: term)(Hsub: subseteq (Fv r) (Fv l)) ... (Hl: linear l): l_rule.
```

L'extraction produit le type OCaml suivant :

```
type l_rule = Lrule of term * term
```

L'extraction supprime dans le type OCaml, la propriété `H` qui permet de s'assurer que la règle est bien formée et `Hl` qui s'assure que le membre gauche est bien linéaire. L'absence de ces propriétés dans le type OCaml rend le type de données plus permissif. Cela peut poser des problèmes pour les fonctions extraites puisque leur comportement peut être non spécifié dans le cas où les propriétés `H` et `Hl` ne sont pas assurées. Or, le type OCaml nous permet de construire des règles de réécriture qui en fait ne sont ni bien formées ni linéaires à gauche : $f(x,x) \to g(y)$. Si l'on ne tient pas compte de cette remarque, on peut par construire un `checker` OCaml qui n'est plus conforme à la spécification Coq. La solution la plus simple pour contourner le problème consiste à écrire une fonction qui vérifie que la règle `l_rule` est bien formée et linéaire à gauche en OCaml. Cependant, on peut toujours

se demander si cette fonction est correcte vis-à-vis de la spécification de propriétés `H` et `Hl` qui est donnée en Coq...Pour cette raison, on préférera définir une fonction qui construira une règle conforme à la spécification à partir d'un simple couple de termes `(l, r)`.

```
Definition build_l_rule: term → term → option l_rule :=
        (* ... Body ... *)
```

On peut alors légitimement se poser la même question pour l'extraction du type `term`. Mais dans ce cas, la définition Coq ne contient pas de propriété supplémentaire, ce qui implique que la totalité de la définition est extraite vers OCaml. Ainsi, on en déduit un critère raisonnablement simple pour interfacer correctement en OCaml, du code extrait à du code externe : les types de données mis en commun, doivent être le résultat d'une extraction totale.

Le tableau 5.1 résume quelques tests d'efficacité. Pour chaque test, le tableau donne la taille des automates d'arbres (A^0 initial et $A^*_{\mathcal{R},E}$ pour le complété) en nombre de transitions et nombre d'états. Pour chaque système de règles de réécriture $\mathcal{R}$, on donne le nombre de règles. La colonne 'CS' indique le nombre d'étapes de complétion nécessaires pour compléter A^0 en $A^*_{\mathcal{R},E}$ et la colonne 'CT' donne le temps de calcul pour effectuer ces étapes. Enfin, on donne le temps (colonne 'CKT') et la mémoire (colonne 'CKM') consommés par le `checker` pour vérifier que l'automate $A^*_{\mathcal{R},E}$ est bien un point fixe. Le point important à relever ici est que le temps de complétion est souvent très long (parfois plus d'une journée), mais que globalement le temps nécessaire à la vérification est très rapide puisque se limitant à une poignée de secondes. Ces résultats plus que satisfaisants soulignent notablement l'intérêt d'une vérification systématique des automates complétés. Les quatre tests sont issus de programmes Java compilés par Copster [5] vers un système de réécriture en utilisant la technique détaillée dans [7]. Pour tous les exemples excepté `List2.java`, l'automate complété a été produit par Timbuk. Concernant l'exemple `List2.java`, le calcul a été complété en utilisant une implémentation en Tom [50] réalisée par Yohan Boichut et Emilie Balland [4]. Cette version offre des performances accrues en temps d'exécution par rapport à Timbuk. Celle-ci repose sur une représentation interne qui n'est plus basée sur des automates d'arbres, mais qui transforme à posteriori le résultat en un automate clos par réécriture tout en respectant les propriétés de l'algorithme initial. Ce qui implique que ces résultats lorsqu'ils sont corrects sont acceptés par le `checker`. Les exemples `List1.java` et `List2.java` correspondent à deux programmes Java manipulant des listes chaînées d'entiers. Dans un cas ils sont sensés être tous positifs, alors que dans l'autre ils sont négatifs. Les entiers sont lus sur l'entrée standard définie comme une séquence infinie d'entiers. L'ensemble d'équations utilisé permet d'abstraire les entiers en deux classes d'équivalence les entiers positifs ou nuls, et les entiers strictement négatifs. On abstrait aussi la représentation mémoire des listes, en capturant dans la même classe d'équivalence toutes les listes quelque soit leur longueur. L'exemple `Ex_poly.java` est l'exemple utilisé dans [7] pour illustrer comment définir une abstraction pour réaliser une analyse k-CFA sur des programmes Java grâce à la complétion d'automates d'arbres. Enfin l'exemple `Bad_Fixp.java` est le même problème que `Ex_poly.java` dont l'automate complété $A^*_{\mathcal{R},E}$ a été volontairement corrompu : l'automate non clos est alors rejeté par le

checker.

Name	$A^0_{\mathcal{R},E}$	$A^*_{\mathcal{R},E}$	$\mathcal{R}$	CS	CT	CKT	CKM
List1.java	118/82	422/219	228	180	$\approx$ 3 j.	0,9s	2,3 Mo
List2.java	1/1	954/364	308	473	1h30	2,2s	3,1 Mo
Ex_poly.java	88/45	951/352	264	161	$\approx$ 1 j.	2,5s	3,3 Mo
Bad_Fixp	88/45	949/352	264	161	$\approx$ 1 j.	1,6s	3,2 Mo

FIGURE 5.1: Vérification de quelques résultats

5.6 Conclusion

Ce chapitre résume l'approche adoptée pour définir un checker certifié en Coq pour la complétion d'automates d'arbres. Le point important ici est le transfert de la certification non pas sur une implémentation spécifique de l'algorithme mais plutôt sur le résultat. De ce fait, le checker reste valide même si l'implémentation de la complétion change, ou est améliorée. En fait, il existe à ce jour d'autres implémentations que celle proposée par Timbuk pour la complétion d'automates d'arbres : on peut citer parmi les versions récentes Tomed Timbuk [4] implémenté en Tom [50] un langage de manipulation d'arbres, ainsi qu'une implémentation en SICStus Prolog reposant sur une spécification des systèmes de réécriture par des clauses de Horn [28]. Le checker est utilisé par les auteurs de ces différents outils afin de vérifier la correction des résultats obtenus. Le deuxième aspect fondamental est qu'un checker en OCaml est extrait directement de la preuve de correction de la spécification Coq grâce au mécanisme d'extraction de Coq. A la vue des performances, on peut conclure qu'il est raisonnable d'exécuter le checker sur chaque résultat pour en vérifier correction, puisque le coût de ressources pour la vérification reste négligeable devant le coût nécessaire à la construction du point-fixe. Le troisième point important est l'attention portée lors de la formalisation pour obtenir un outil efficace. En particulier, le recours à un algorithme d'inclusion particulier permet d'éviter le facteur exponentiel de l'algorithme d'inclusion standard et de justifier en grande partie des performances mentionnées ci-dessus. Enfin, l'approche se base principalement sur des arbres dès lors que des structures de données nécessitent des accès aléatoires. Bien que leur utilisation puisse rendre les preuves plus complexes, on peut établir une projection des arbres dans les listes. L'astuce permet ainsi de transférer les propriétés des arbres vers leur projection dans les listes ce qui simplifie les preuves, puisqu'il est généralement plus facile de raisonner sur les listes que sur les arbres. Enfin, la principale évolution de la complétion est l'utilisation des ε-transitions pour résoudre les paires critiques. Cette approche qui est utilisée comme base dans les techniques d'analyses des chapitres précédents peut être couverte en nettoyant l'automate point-fixe de ses ε-transitions. D'autre part, le théorème 5.4.8 sur la complétude de l'algorithme d'inclusion reste valide pour les automates avec des ε-transitions. La seule partie qu'il est nécessaire d'adapter, est la vérification de la clôture par le système de

réécriture, ce qui peut être réalisé très facilement. Ensuite, on peut se poser la question d'étendre la clôture pour les systèmes de règles de réécriture non linéaires à gauche qui sont requis pour formaliser certains protocoles cryptographiques. Le problème est un peu plus délicat, et pose déjà un challenge du point de vue de la complétion elle-même. La plupart du temps la prise en compte d'un système de règles non linéaires à gauche effondre les performances. Il existe alors plusieurs astuces qui peuvent être mises en oeuvre, mais demandent pour chacune une méthode de vérification particulière.

Dans les pistes à explorer, on peut imaginer exécuter le `checker` directement en Coq, et grâce à la réflexivité en déduire des preuves de non-atteignabilité pour systèmes de transitions en Coq. La preuve est alors d'une part constituée du point-fixe calculé au préalable par un outil externe et d'autre part par la vérification par le `checker` directement en Coq. Enfin, à la vue des performances du `checker`, il est envisageable d'imaginer l'adaptation des analyses basées sur la complétion d'automates d'arbres vers un modèle à la Proof-Carrying Code. Dans un tel modèle, le `checker` est embarqué dans un environnement où l'on ne fait pas confiance aux programmes étrangers au système. L'installation d'un programme nécessite alors une preuve de sa sûreté (programme fiable, non-malicieux, préservant la politique de confidentialité...), et cette preuve doit être vérifiable facilement. Dans ce cas, l'automate produit par la complétion peut constituer cette preuve, dont la correction est facile à vérifier.

Bibliographie générale

[1] P. A. Abdulla, A. Legay, A. Rezine, and J. d'Orso. Simulation-based iteration of tree transducers. In *TACAS*, volume 3440 of *LNCS*, pages 30–40. Springer, 2005.

[2] Agda. Chalmers University of Technology. `http://unit.aist.go.jp/cvs/Agda/`.

[3] F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

[4] E. Balland, Y. Boichut, T. Genet, and P.-E. Moreau. Towards an Efficient Implementation of Tree Automata Completion. In *AMAST'08*, volume 5140 of *LNCS*. Springer, 2008.

[5] N. Barré, F. Besson, T. Genet, L. Hubert, and L. Le Roux. Copster homepage, 2009. `http://www.irisa.fr/celtique/genet/copster`.

[6] Y. Boichut, R. Courbis, P.-C. Heam, and O. Kouchnarenko. Finer is better : Abstraction refinement for rewriting approximations. In *RTA*, LNCS 5117, pages 48–62. Springer, 2008.

[7] Y. Boichut, T. Genet, T. Jensen, and L. Leroux. Rewriting Approximations for Fast Prototyping of Static Analyzers. In *RTA*, volume 4533 of *LNCS*, pages 48–62. Springer, 2007.

[8] B. Boigelot, A. Legay, and P. Wolper. Iterating transducers in the large (extended abstract). In *CAV*, LNCS, pages 223–235. Springer, 2003.

[9] A. Bouajjani, P. Habermehl, A. Rogalewicz, and T. Vojnar. Abstract Regular Tree Model Checking. In *Infinity'05*, volume 149(1), pages 37–48, 2006.

[10] A. Bouajjani, P. Habermehl, A. Rogalewicz, and T. Vojnar. Abstract Regular Tree Model Checking of Complex Dynamic Data Structures. In *SAS'06*, volume 4134, pages 52–70. Springer, 2006.

[11] A. Bouajjani, P. Habermehl, and T. Vojnar. Abstract regular model checking. In R. Alur and D. Peled, editors, *CAV*, volume 3114 of *Lecture Notes in Computer Science*, pages 372–386. Springer, 2004.

[12] A. Bouajjani, B. Jonsson, M. Nilsson, and T. Touili. Regular model checking. In *CAV*, volume 1855 of *LNCS*, pages 403–418. Springer-Verlag, 2000.

[13] B. Boyer and T. Genet. Verifying Temporal Regular properties of Abstractions of Term Rewriting Systems. In *Proc. of RULE'09*. EPTCS, 2009.

[14] B. Boyer, T. Genet, and T. Jensen. Certifying a Tree Automata Completion Checker. In *IJCAR'08*, volume 5195 of *LNCS*. Springer, 2008.

[15] W. S. Brainerd. Tree generating regular systems. 14 :217–231, 1969.

[16] J. Büchi. On a decision method in restricted second order arithmetic. In *International Congress on Logic, Methodoly and philosophy of Science*, pages 1–11. Stanford University Press, 1960.

[17] E. M. Clarke. Counterexample-Guided Abstraction Refinement. In *TIME*, page 7. IEEE Computer Society, 2003.

[18] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, C. Löding, S. Tison, and M. Tommasi. Tree automata techniques and applications. 2008.

[19] The coq proof assistant. Inria. `http://coq.inria.fr/`.

[20] The Coq Standard Library. Inria. `http://www.lix.polytechnique.fr/coq/stdlib/`.

[21] J. Coquidé, M. Dauchet, R. Gilleron, and S. Vágvölgyi. Bottom-up tree pushdown automata and rewrite systems. volume 488, pages 287–298, 1991.

[22] P. Cousot and R. Cousot. Abstract interpretation : A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252, 1977.

[23] M. Dauchet and S. Tison. The theory of ground rewrite systems is decidable. pages 242–248, June 1990.

[24] G. Dowek. Le langage mathématique et les langages de programmation, juin 1997.

[25] O. G. Edmund M. Clarke and D. A. Peled. *Model Checking*. MIT Press, 2000.

[26] G. Feuillade, T. Genet, and V. Viet Triem Tong. Reachability Analysis over Term Rewriting Systems. *Journal of Automated Reasonning*, 33 (3-4) :341–383, 2004.

[27] Frama-C homepage. Inria Cea. `http://frama-c.com/`.

[28] J. Gallagher and M. Rosendahl. Approximating term rewriting systems : a horn clause specification and its implementation. In *LPAR'08*, volume 5330. Springer, 2008.

[29] T. Genet. Decidable approximations of sets of descendants and sets of normal forms (extended version). Technical Report RR-3325, 1997.

[30] T. Genet. Decidable approximations of sets of descendants and sets of normal forms. volume 1379, pages 151–165, 1998.

[31] T. Genet. Reachability analysis of rewriting for software verification. Université de Rennes 1, 2009. Habilitation.

[32] T. Genet and F. Klay. Rewriting for Cryptographic Protocol Verification. volume 1831, 2000.

[33] T. Genet and R. Rusu. Equational tree automata completion. *JSC*, 45 :574–597, 2010.

[34] T. Genet and V. Viet Triem Tong. Timbuk – a Tree Automata Library. IRISA / Université de Rennes 1, 2001. `http://www.irisa.fr/celtique/genet/timbuk/`.

[35] R. Gerth, D. Peled, M. Y. Vardi, and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In P. Dembinski and M. Sredniawa, editors, *PSTV*, volume 38 of *IFIP Conference Proceedings*, pages 3–18. Chapman & Hall, 1995.

[36] R. Gilleron and S. Tison. Regular tree languages and rewrite systems. 24 :157–175, 1995.

[37] R. Goscinny and A. Uderzo. Le Bouclier arverne, 1968.

[38] D. Huet, Gérard Lankford. On the uniform halting problem for term rewriting systems. Technical report, IRIA, Mars 1978.

[39] The proof assistant isabelle. http ://www.cl.cam.ac.uk/research/hvg/Isabelle/.

[40] F. Jacquemard. Decidable approximations of term rewriting systems. pages 362–376, 1996.

[41] The KeY project. `http://www.key-project.org`.

[42] P. Lammich. Tree Automata in Isabelle. `http://afp.sourceforge.net/entries/Tree-Automata.shtml`.

[43] X. Leroy, D. Doligez, J. Garrigue, D. Rémy, and J. Vouillon. The Objective Caml system release 3.00 – Documentation and user's manual, 2000. `http://caml.inria.fr/ocaml/htmlman/`.

[44] J. Meseguer, M. Palomino, and N. Martí-Oliet. Equational abstractions. *TCS*, 403 :239–264, 2008.

[45] Rewriting and approximations for java applications verification. http ://www.irisa.fr/celtique/genet/RAVAJ/.

[46] P. Réty. Regular Sets of Descendants for Constructor-based Rewrite Systems. volume 1705, 1999.

[47] X. Rival and J. Goubault-Larrecq. Experiments with finite tree automata in coq. In *Proc. of TPHOL'01*, LNCS. Springer, 2001.

[48] K. Salomaa. Deterministic Tree Pushdown Automata and Monadic Tree Rewriting Systems. *J. Comput. Syst. Sci.*, 37 :367–394, 1988.

[49] T. Takai. A Verification Technique Using Term Rewriting Systems and Abstract Interpretation. In *RTA*, volume 3091, pages 119–133. Springer, 2004.

[50] Tom Homepage. `http://tom.loria.fr`.

[51] P. Wolper and B. Boigelot. Verifying systems with infinite but regular state spaces. In *CAV*, volume 1427 of *LNCS*, pages 88–97. Springer-Verlag, 1998.

Résumé

Cette thèse s'intéresse à la vérification de programmes modélisés sous forme de systèmes de règles de réécriture. La vérification de propriétés est basée sur une analyse statique semi-automatique qui construit une sur-approximation, représentée par un automate d'arbres, de l'ensemble des termes atteignables. L'analyse est paramétrée par une abstraction qui doit être suffisamment précise pour que la propriété attendue puisse être vérifiée. Or, il est difficile de construire une telle abstraction à priori. On propose un mécanisme original de raffinement automatique par élagage de l'automate d'arbres lorsque la sur-approximation calculée, trop imprécise, est susceptible de contenir de fausses alarmes. Non seulement l'analyse s'applique à la vérification de propriétés de sûreté par non-atteignabilité, mais on montre qu'elle peut être adaptée afin de vérifier des propriétés temporelles, notamment sur le graphe des appels de méthodes d'un programme Java. Enfin, les outils réalisant cette analyse reposent sur des implémentations optimisées, clairement éloignées de la spécification originale. Pour accroître la confiance en ces outils, on fournit un vérificateur chargé de la validation de leurs résultats à posteriori. La spécification et la correction de ce validateur sont formulées et démontrées dans l'assistant de preuves Coq.

Abstract

This thesis addresses the verification of programs, symbolized as term rewriting systems. Program properties are verified using a semiautomatic static analysis that returns a tree automaton recognizing an over-approximation of reachable terms. This analysis is parameterized by an abstraction that has to be precise enough to check the expected property. However, it is generally hard to give such an abstraction to the analysis. Using tree automaton pruning, we propose an original mechanism of automatic refinement, which allows us to avoid false alarms that are contained in the over-approximation. The technique is initially designed to check safety properties by unreachability. We show how to extend it to check temporal properties, especially for properties about the graph of method calls for a Java program. Finally, to increase their performance, the tools performing this analysis are very optimized and their implementation is quite far from of the original specification. To trust the results of these tools, we provide a checker that is in charge of validating the results. The specification and the correction of the checker are designed and proved in the proof assistant Coq.

VU :

Le Directeur de Thèse
(Nom et Prénom)

VU :

Le Responsable de l'École Doctorale

VU pour autorisation de soutenance

Rennes, le

Le Président de l'Université de Rennes 1

Guy CATHELINEAU

VU après soutenance pour autorisation de publication :

Le Président de Jury,
(Nom et Prénom)