Analyse d'atteignabilité pour les programmes fonctionnels avec stratégie d'évaluation en profondeur

Yann Salmon

ANNÉE 2015

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
École doctorale Matisse

présentée par

Yann Salmon

Préparée à l'U.M.R. 6074 I.R.I.S.A.
Institut de recherche en informatique et systèmes aléatoires
U.F.R. d'informatique et électronique

Analyse d'atteignabilité pour les programmes fonctionnels avec stratégie d'évaluation en profondeur

Thèse soutenue à Rennes
le 7 décembre 2015
devant le jury composé de

Géraud Sénizergues
Professeur à l'Université de Bordeaux
Rapporteur

Pierre Réty
Maitre de conférences à l'Université d'Orléans
Rapporteur

Thomas Jensen
Directeur de recherches à l'I.N.R.I.A.
Examinateur

Hélène Kirchner
Directrice de recherches à l'I.N.R.I.A.
Examinatrice

Sylvain Salvati
Chargé de recherches à l'I.N.R.I.A.
Examinateur

Thomas Genet
Maitre de conférences à l'Université de Rennes 1
Directeur de thèse

Thèse de doctorat

Analyse d'atteignabilité pour les programmes fonctionnels avec stratégie d'évaluation en profondeur

Yann Salmon

1er mars 2016

Université de Rennes 1

Table des matières

Table des figures

1 Introduction

1.1 Contexte

L'informatique est née de l'étude et de la formalisation du calcul. Dès l'antiquité, des nécessités concrètes ont requis l'étude de longueurs, de surfaces, de volumes, de grandeurs numériques. Les mathématiques émergèrent de la recherche systématique de propriétés sur ces objets. Les démonstrations rigoureuses ont nécessité la définition formelle de concepts comme le rectangle, le cercle, plus tard l'équation, la fonction numérique, etc.

Ces manipulations requerraient des calculs et des raisonnements, lesquels furent donc des outils centraux des mathématiques et des autres sciences, sans être véritablement des objets d'étude systématique et générale à part entière. Au dix-neuvième siècle, la recherche des fondements des mathématiques, qui se manifeste entre autres par le développement de la théorie des ensembles de Cantor, la volonté de réduire les mathématiques à la logique de Frege et certains des problèmes de Hilbert, conduisent à des interrogations plus fines sur la nature du raisonnement et du calcul. Naissent alors les controverses entre finitistes de Brouwer et les hilbertiens. D'outils, certes importants, le raisonnement et le calcul deviennent des objets d'étude qui doivent à leur tour être formalisés.

Les différentes approches pour formaliser la notion intuitive de calcul aboutissent dans les années 1930 lorsque Gödel, avec Herbrand, Church et Turing proposent chacun un modèle — fonctions récursives, lambda-calcul, machine de Turing — de cette notion et qu'est démontrée l'équivalence entre ces trois modèles. La thèse de Church postule que ce triple modèle est la bonne formalisation de la notion de calcul, que ce qui est calculable au sens intuitif est ce qui est calculable au sens de ces trois modèles. Car ces études montrent, théorème d'incomplétude de Gödel et indécidabilité du problème de l'arrêt des machines de Turing en tête, que la volonté de Hilbert de réduire le raisonnement mathématique à des étapes de calcul est irréalisable.

Si les incohérences de la théorie naïve des ensembles de Cantor ont pu être contournées par les théories de Zermelo et Fraenkel, l'échec du projet hilbertien, après celui de Frege, clôt une période qui se voulait être la finalisation, la formalisation totale et complète des mathématiques. Mais il ouvre la voie à une nouvelle discipline, car si tout n'est pas calculable, il est pertinent d'étudier ce qui l'est. De plus, les études menées sur les limites des modèles du calcul ont conduit Turing à

décrire une machine universelle, un programme qui peut recevoir comme donnée le texte d'un autre programme et l'exécuter.

À partir du dix-huitième siècle se développent également l'industrialisation et l'usage de machines pour la production. Sont conçues et construites des machines comme le métier à tisser de Jacquard (1801), dont la nouveauté est que le mouvement complexe des différentes pièces mécaniques servant à la réalisation du motif du tissu n'est plus provoqué manuellement par un artisan mais automatiquement selon la position des trous d'une carte perforée. L'esprit humain n'intervient donc plus de façon critique au moment de la production du motif mais au moment de l'écriture d'un objet matériel destiné à provoquer le production. En cela, le métier Jacqard se distingue de machines antérieures construites ou imaginées, comme la Pascaline, qui servaient à réaliser un calcul numérique. C'est le lien avec la production qui rend intéressant l'écriture d'un programme sur carte perforée : s'il est utile de pouvoir lancer plusieurs fois la production d'un même tissu, exécuter plusieurs fois un même calcul arithmétique ne présente pas d'intérêt.

Il est toutefois un domaine où programmer un calcul arithmétique pour une exécution répétée est utile : la construction de tables de valeurs. Suivant Jacquard, Babbage construit une machine analytique programmable par cartes perforées. Babbage ne la destine qu'aux calculs de tables pour lequel il l'a conçue, mais Ada Lovelace, qui écrivait les programmes, comprend que cette machine est capable d'effectuer des manipulations symboliques générales. Elle est considérée comme l'ancêtre des ordinateurs actuels. Les progrès techniques permettront, dans les années 1940, le remplacement de la vapeur par l'électricité et de certains engrenages par des tubes à vide puis des transistors. Le principe reste le même : la machine est conçue en vue d'une application déterminée, souvent le calcul de tables de valeurs, et on entre les programmes sous une forme — un langage-machine — dont la lecture par la machine provoque physiquement le comportement attendu.

Chaque programme, pensé par celui qui veut faire exécuter un calcul par la machine, doit être traduit en une suite d'instructions élémentaires du langage de la machine. Cette opération est fastidieuse. Grace Hopper comprend qu'elle relève du calcul, qu'elle peut donc être confiée à la machine à calculer et qu'il est pertinent de le faire : elle conçoit un programme dont l'entrée est la description d'un programme dans un langage agréable et dont la sortie est un programme réalisant la même tâche, mais exprimé dans le langage de la machine. Il s'agit du premier compilateur.

Cela facilite l'écriture de programmes et, associé aux progrès techniques qui réduisent le cout et l'encombrement des machines, donne le départ de l'expansion de l'informatique que nous connaissons aujourd'hui.

Les programmes informatiques sont aujourd'hui présents dans tous les domaines. Leur présence n'est plus seulement un ajout aux dispositifs préexistants, ils constituent souvent l'armature et le pivot des systèmes. Ils traitent des données importantes, et sont parfois le seul moyen de traitement de ces données. Le tissu pro-

duit par le métier Jacquard pouvait être, et était, contrôlé indépendamment de tout système informatique. Actuellement, les objets produits par l'industrie à l'aide de machines pilotées par des programmes sont également contrôlés. Mais ce contrôle s'appuie sur des mesures effectuées par des appareils eux-mêmes pilotés par un dispositif informatisé ou qui font l'objet d'un traitement informatique.

Les conséquences sont nombreuses. Une erreur de programmation a causé la destruction en plein vol d'une fusée Ariane. La plupart des signes monétaires n'ont d'existence que dans les différents systèmes d'information des banques. Ce n'est pas en bombardant — comme ç'avait encore été le cas à Osirak en 1981 — les usines de centrifugation que des tiers ont ralenti les activités d'enrichissement d'uranium de l'Iran mais en exploitant, à l'aide du ver Stuxnet, un défaut du programme de contrôle de ces machines.

La question de l'adéquation du comportement effectif des programmes informatiques avec leur comportement attendu se pose ainsi avec une acuité toujours plus grande, et plus généralement celle de leur maitrise. Il s'agit également d'un enjeu de souveraineté nationale, le cyberespace étant désormais considéré comme un théâtre d'opérations qui intersecte les théâtres d'opérations traditionnels — terre, mer, air.

Pour autant, faut-il chercher à établir l'inviolabilité de tous les programmes ? Yanis Varoufakis a été ministre des finances de la Grèce de janvier à juillet 2015. Il a indiqué qu'en raison du fait que les systèmes informatiques de son ministère étaient sous le contrôle des créanciers du pays, il avait dû trouver le moyen de s'y introduire subrepticement, afin, le cas échéant, de mettre en œuvre des procédures élaborées par le gouvernement investi de la légitimité populaire. Cela n'aurait pas été possible si ces logiciels avaient été parfaitement sécurisés.

Cette question est en fait celle de la maitrise et du contrôle des systèmes d'information par les personnes qui sont chargées des traitements qu'ils mettent en œuvre, et l'analyse des programmes contribue à cette maitrise parce qu'elle peut aussi contribuer à la compréhension des systèmes.

1.2 Contributions

Nous proposons une nouvelle présentation de la complétion d'automate d'arbres de [Gen09] et surtout une adaptation de cette méthode à la stratégie d'évaluation en profondeur [GS12]. Cette adaptation permet une meilleure précision dans l'analyse des programmes qui s'exécutent avec cette stratégie, phénomène constaté expérimentalement et établi formellement dans [GS15].

Nous présentons également une contribution pratique qui consiste en une implémentation de notre méthode dans l'outil Timbuk [GV01]. Cette implémentation,

encore prototypique, permet d'observer des résultats de notre méthode et de la comparer à celle développée dans [CLM15].

Nous montrons enfin comment cette adaptation de la complétion d'automate à la stratégie en profondeur s'inscrit dans un réseau de travaux qui peut déboucher sur un analyseur de programmes fonctionnels par réécriture et automates d'arbres.

1.3 Plan

Après une introduction générale du problème d'atteignabilité (chapitre 2) et du formalisme retenu pour aborder son étude (chapitre 3), nous passons en revue quelques méthodes utilisées pour le résoudre et donnons une présentation de la complétion d'automate de [Gen09] qui sert de base à notre contribution (chapitre 4).

Notre principale contribution théorique est présentée au chapitre 5. Nous commençons par rappeler l'intérêt d'une adaptation des méthodes d'analyse aux stratégies d'évaluation et par expliquer quelles méthodes ont été mises en œuvre à cette fin par d'autres auteurs (section 5.1). Après avoir rappelé les résultats classiques préliminaires (section 5.2.1), nous présentons notre adaption de la complétion d'automate (section 5.2.2) puis montrons qu'elle jouit de théorèmes de correction et précision semblables à ceux connus pour la méthode classique (section 5.2.3).

Le chapitre 6 traite des aspects pratiques. Nous rappelons tout d'abord les grandes lignes du fonctionnement de l'outil Timbuk [GV01] (section 6.1), avant d'exprimer la façon dont notre contribution est actuellement implémentée (section 6.2). Ceci nous donne l'occasion d'une comparaison avec la méthode de [CLM15] (section 6.2.4).

Dans le chapitre 7, nous citons des travaux qui permettraient de franchir les étapes restantes pour aboutir à une analyse de programmes OCaml [Ler+14] (sections 7.1 à 7.3). Nous indiquons aussi (section 7.4) comment la complétion d'automate peut s'inscrire dans le cadre théorique général de l'interprétation abstraite [CC77], dont nous rappelons les grandes lignes en sections 7.4.1 et 7.4.2.

2 Vérification et atteignabilité

2.1 Les programmes

Un programme est la description d'un calcul à effectuer par une machine. Il peut être écrit directement dans le langage de la machine ou dans un langage, dit langage de programmation, destiné à être traduit automatiquement dans le langage de la machine au moyen d'un autre programme appelé compilateur. Le langage de programmation doit pouvoir être traité par les opérations calculatoires du compilateur et doit donc proposer une syntaxe formalisée. Pour que le programmeur sache quel programme écrire en vue du comportement qu'il souhaite, le langage de programmation doit aussi proposer une sémantique précise correspondant à sa syntaxe.

Il existe plusieurs types de styles syntaxiques. On peut décrire les programmes par des graphes dont les sommets sont des instructions élémentaires et les arcs représentent les passages possibles d'une instruction à l'autre. Cette représentation, quoique non totalement linguistique, est une forme de langage de programmation. Elle est par ailleurs toujours utilisée dans l'analyse des programmes sous le nom de graphe de flot de contrôle. Les langages de programmation impératifs sont très répandus. Ils reposent sur les concepts d'affectation d'une valeur à une variable, de séquence, de test et de boucle. Pour notre part, nous utiliserons des langages de programmation dits fonctionnels. Ces langages connaissent également le test, mais, dans l'acception restrictive que nous faisons nôtre ici, ni la séquence, ni la boucle, ni l'affectation. En revanche, il est permis de définir des fonctions récursives. Le langage dans lequel nous exprimerons nos exemples est un fragment fonctionnel du langage OCaml [Ler+14].

Il existe plusieurs types de sémantiques [Win93], qui décrivent avec plus ou moins de précision le déroulement attendu de l'exécution des programmes. En première approximation, la sémantique dénotationelle [Sch86] d'un programme est la fonction mathématique dont ce programme réalise le calcul.

Exemple 1.
Le programme OCaml `minimum` de la figure 2.1 dénote une fonction mathématique dont on peut établir que c'est la fonction qui associe à une liste non vide le plus petit de ses éléments.

`[h]` désigne une liste à un seul élément, noté `h` : notre fonction renvoie en ce cas la valeur de cet unique élément. Pour une liste `h::t` n'ayant pas un seul élément, on

```
1 let min x y =
2    if x < y then x else y
3
4  let rec minimum l = match l with
5    | [h] -> h
6    | h::t -> min h (minimum t)
```

Figure 2.1 – Un programme OCaml

note `h` sa tête et `t` sa queue ; on calcule le minimum de la queue à l'aide du même programme, puis on synthétise. 1◁

2.2 Correction et vérification

2.2.1 Spécification

Un programme est écrit pour répondre à un problème, exécuter une tâche. Pour pouvoir espérer démontrer formellement que le programme a un comportement conforme à ce qui est attendu, il faut d'abord formaliser cette attente. Tel est l'objet de la spécification.

La spécification décrit les paramètres admissibles et le résultat souhaité, en fonction de ces paramètres. Le programme correspondant est censé décrire, en fonction de ces mêmes paramètres, un calcul permettant d'obtenir ce résultat.

Exemple 2.
Une spécification pour l'exemple précédent indique que les valeurs admissibles pour le paramètre `l` sont des listes non vides d'éléments ordonnés, et que le résultat renvoyé est le plus petit de ces éléments. On peut restreindre la spécification à des listes non vides d'entiers, par exemple. 2◁

Étant donné des paramètres effectifs, l'exécution du programme avec ces paramètres est correcte lorsqu'elle fournit le résultat attendu. Le programme est correct lorsque son exécution est correcte pour tous les paramètres admis par la spécification.

Parler du résultat issu de l'exécution de l'appel suppose que les étapes de calcul qui s'enchainent dans cet appel s'achèvent à un moment : si le calcul se poursuit indéfiniment, il n'y a pas de résultat. Il est donc également crucial de s'assurer de la terminaison d'un appel. On dit d'un programme qu'il termine sans plus de précision lorsque tous les appels autorisés par la spécification terminent.

Exemple 3.
L'appel `minimum [4;8;7]` termine et renvoie `4`.

L'appel `minimum []` ne termine pas à proprement parler (une erreur se produit) et on ne peut donc parler du résultat qu'il renvoie. Cependant, comme la liste vide n'est pas une valeur admissible pour le paramètre `l` d'après la spécification donnée de la fonction `minimum`, la non-terminaison de cet appel n'est pas un problème.

On peut démontrer que pour toute liste non vide `l`, l'appel `minimum l` se termine (sans erreur) et renvoie bien le minimum des éléments de `l`. Le programme `minimum` est donc correct. 3 ◁

Vérifier qu'un programme termine est une tâche en général difficile et l'établissement de la preuve de terminaison est elle-même sujette à erreur du fait de sa longueur et de sa complexité. Il est donc tentant d'automatiser le plus possible la recherche de cette preuve.

Tel est l'objet des vérificateurs de programme : analyser des programmes afin de prouver automatiquement des propriétés à leur sujet.

2.2.2 Problème de l'arrêt

La tâche de vérifier la terminaison des programmes ne peut être entièrement automatisée. Elle peut l'être si on la restreint à des catégories restreintes de programmes, mais est insoluble dans sa forme générale.

Théorème 4 (Indécidabilité du problème de l'arrêt [Tur36]).
Soit la question : étant donné un programme P et x un paramètre concret pour ce programme, l'exécution de P avec le paramètre x se termine-t-elle après un nombre fini d'étapes ?

Il n'existe pas d'algorithme permettant de répondre à cette question. 4 ■

Il en découle plus généralement la non-existence d'un algorithme général pour déterminer si un programme donné vérifie une propriété donnée [Ric53].

C'est pourquoi les outils de preuve de programme sont utilisés comme suit : si l'outil a réussi à prouver la propriété cherchée, c'est qu'elle est vraie ; si l'outil n'a pas réussi à prouver la propriété, elle peut être fausse comme elle peut être vraie. Ce point de vue est le contraire de celui adopté pour les tests : si le test détecte un problème, c'est qu'il y en a un ; si le test ne détecte rien, il peut ne pas y avoir de problème comme il peut y en avoir un.

2.3 Atteignabilité et non-atteignabilité

Nous souhaitons étudier et développer des méthodes qui assurent le bon comportement des programmes, c'est-à-dire des méthodes de preuve. Une telle méthode

vise à établir, étant donné un programme — un calcul à effectuer — et une donnée intiale, que les états successifs au cours du calcul resteront confinés au domaine des états admissibles. Ou encore, qu'il n'est pas possible d'atteindre un mauvais état.

Il est donc pertinent d'exprimer l'ensemble des états de calcul qui peuvent être atteints par un programme à partir d'un paramètre.

Exemple 5.
Reprenons le programme de la figure 2.1. Il est impossible d'effectuer `minimum []` : cette expression est donc, pour ce programme, un « mauvais état », qui ne doit jamais être atteint.

Ici, étant donné un état initial, par exemple `minimum [4;8;7]`, on observe que la définition de `minimum` conduit à se trouver dans des états comportant les termes `minimum [8;7]` puis `minimum [7]`. Il résulte de la présence du premier cas dans le **`match`** que `minimum [7]` fournit immédiatement un résultat ; on ne cherche pas à évaluer `minimum []`. 5◁

Mais un paramètre ne suffit pas pour obtenir une preuve : on doit considérer comme point de départ l'ensemble des paramètres admissibles et déterminer les états atteignables.

Exemple 6.
Pour le programme de la figure 2.1, prouver qu'on n'effectue jamais `minimum []` nécessite de considérer l'ensemble de toutes les listes non vides (d'entiers, si on a posé cette restriction dans la spécification). 6◁

Pour cela, nous devons nous donner un formalisme pour exprimer les états de calculs, les ensembles de tels états et programmes. Nous utiliserons la réécriture.

Exemple 7.
L'évolution du calcul de `minimum [4;8;7]` peut se formaliser par

$$\begin{aligned}
\mathit{minimum}([4;8;7]) &\to \mathit{min}(4, \mathit{minimum}([8;7])) \\
&\to \mathit{min}(4, \mathit{min}(8, \mathit{minimum}([7]))) \\
&\to \mathit{min}(4, \mathit{min}(8,7)) \\
&\vdots && \text{étapes omises} \\
&\to \mathit{min}(4,7) \\
&\vdots && \text{étapes omises} \\
&\to 4.
\end{aligned}$$

La flèche indique que l'expression (ou terme) à sa gauche peut être réécrite en l'expression à sa droite. 7◁

3 Atteignabilité en réécriture

3.1 Les systèmes de réécriture de termes

Nous exposons maintenant le formalisme qui est utilisé dans la suite de ce document. Les notions exposées ici sont classiques, on en trouvera des présentations dans [BN98 ; Ter03].

Ce formalisme met en jeu des termes et des règles de réécriture agissant sur ces termes. Un terme représente donc un état de calcul à effectuer, tandis que les règles de réécriture décrivent les transformations qui peuvent être apportées à cet état pour faire progresser le calcul. Un terme auquel aucune règle de réécriture n'est applicable est appelé une forme normale ; cette notion recoupe celle de valeur. Ainsi, ce sont les règles de réécriture qui donnent leur sens aux termes.

3.1.1 Termes

Les termes sont écrits à l'aide de symboles de fonctions, y compris les constantes, et de variables. Cet alphabet de base est appelé signature. Les variables trouveront leur rôle particulier dans la définition des substitutions.

Définition 8 (Signature).
Une signature est la donnée, pour chaque entier naturel k jusqu'à un maximum, d'un ensemble fini de symboles, dit ensemble des symboles d'arité k, ainsi que d'un ensemble au plus dénombrable dont les éléments sont appelés variables. Tous ces ensembles sont deux à deux disjoints. L'ensemble des variables est usuellement noté $\mathcal{X}$. L'ensemble des symboles de fonction d'arité k est usuellement noté Σ_k, et l'ensemble des symboles de fonctions, toutes arités confondues, est noté Σ.

Une telle signature est notée $(\Sigma, \mathcal{X})$. On note Σ au lieu de $(\Sigma, \varnothing)$.

Les symboles de fonctions d'arité 0 sont appelés constantes. 8 ◀

Remarque 9. Dans une signature, les ensembles Σ_k sont vides à partir d'un certain rang, de sorte que Σ, qui est leur réunion, est fini.

Dans toute la suite, $(\Sigma, \mathcal{X})$ est une signature.

La notion de terme est assez rigide et impose le respect de l'arité des symboles mis en jeu.

Définition 10 (Terme).
On appelle terme sur la signature $(\Sigma, \mathcal{X})$
— toute variable $x \in \mathcal{X}$,
— toute constante $c \in \Sigma_0$ et
— toute construction de la forme $f(t_1, \ldots, t_k)$ où $f \in \Sigma_k$ et $t_1, \ldots, t_k$ sont des termes sur $(\Sigma, \mathcal{X})$.
L'ensemble des termes sur la signature $(\Sigma, \mathcal{X})$ est noté $\mathrm{T}(\Sigma, \mathcal{X})$. 10 ◀

Avant de donner quelques exemples, nous exposons quelques notions classiques sur les termes.

Définition 11 (Terme clos).
Un terme est dit clos lorsqu'il ne contient pas de variable. L'ensemble des termes clos sur $(\Sigma, \mathcal{X})$ n'est autre que l'ensemble des termes sur $(\Sigma, \varnothing)$, noté $\mathrm{T}(\Sigma)$. 11 ◀

Définition 12 (Linéarité).
Un terme est linéaire lorsque chaque variable y apparait au plus une fois. 12 ◀

Exemple 13.
Considérons la signature qui comprend un symbole f d'arité deux, un symbole g d'arité un et un symbole c d'arité zéro. Le terme $f(g(x), y)$ est linéaire : chaque variable x et y apparait une seule fois. Le terme $f(g(x), x)$ n'est pas linéaire : la variable x apparait deux fois. Le terme $f(g(c), c)$ est linéaire : la répétition de constantes ou de sous-termes clos est autorisée. 13 ◁

Définition 14 (Substitution).
Soit $(\Sigma, \mathcal{X})$ et $(\Sigma', \mathcal{X}')$ deux signatures telles que $\Sigma' \subseteq \Sigma$. Une substitution de $(\Sigma, \mathcal{X})$ vers $(\Sigma', \mathcal{X}')$ est donnée par une application $\mu : \mathcal{X} \longrightarrow \mathrm{T}(\Sigma', \mathcal{X}')$ dont l'action, notée à droite, s'étend en une fonction de $\mathrm{T}(\Sigma, \mathcal{X})$ vers $\mathrm{T}(\Sigma, \mathcal{X}')$ comme suit :
— si t est une constante, $t\mu = t$,
— si t est une variable, $t\mu = \mu(t)$ et
— si $t = f(t_1, \ldots, t_k)$, alors $t\mu = f(t_1\mu, \ldots, t_k\mu)$. 14 ◀

Remarque 15. Fréquemment, on s'abstient de définir μ sur les variables de $\mathcal{X}$ qui n'apparaissent pas effectivement dans les termes auxquels elle est susceptible d'être appliquée.

Pour parler avec précision des sous-termes d'un terme, il convient de définir la notion de position.

Définition 16 (Position).
Une position est un mot sur l'alphabet $\mathbb{N}_*$. Le mot vide est noté Λ.

Soit t un terme. L'ensemble des positions de t, noté $\mathrm{Pos}(t)$, est défini par induction :
— si t est une variable ou une constante, $\mathrm{Pos}(t) = \{\Lambda\}$ et

— si $t = f(t_1, \ldots, t_k)$, $\mathrm{Pos}(t) = \{\Lambda\} \cup \bigcup_{i=1}^{k} \{i\}.\mathrm{Pos}(t_i)$. 16 ◀

Remarque 17. On dit de positions qu'elles sont parallèles si aucune n'est un préfixe d'une autre, au sens des mots sur l'alphabet $\mathbb{N}_*$.

Définition 18 (Sous-terme).
Soit t un terme et $\xi \in \mathrm{Pos}(t)$. Le sous-terme de t à la position ξ, noté $t_{|\xi}$, est défini par induction :
— si $\xi = \Lambda$, alors on définit $t_{|\xi} = t$ et
— si $\xi = i.\xi'$, alors nécessairement t est de la forme $f(t_1, \ldots, t_k)$ avec $k \geq i$ et on peut définir $t_{|\xi} = t_{i|\xi'}$. 18 ◀

Exemple 19.
Soit le terme $t = f(g(x), f(c, y))$. On a $t_{|2.1} = c$ et $t_{|1} = g(x)$. 19 ◁

Définition 20 (Contexte).
Soit $(\Sigma, \mathcal{X})$ une signature et $\Box$ un symbole ne faisant partie ni de Σ ni de $\mathcal{X}$. On appelle contexte sur $(\Sigma, \mathcal{X})$ tout terme de $\mathrm{T}(\Sigma, \mathcal{X} \cup \{\Box\})$ dans lequel $\Box$ apparait exactement une fois.

Soit C un contexte et t un terme. On note $C[t]$ le terme obtenu en remplaçant $\Box$ par t dans C, c'est-à-dire $C\mu$ pour la substitution $\mu : \Box \mapsto t$. 20 ◀

Définition 21 (Remplacement).
Soit t un terme, ξ une position de t et s un terme. Soit C le contexte tel que $t = C\left[t_{|\xi}\right]$. On note $t[s]_\xi$ le terme $C[s]$.

Soit $\{\xi_1, \ldots, \xi_n\}$ est un ensemble de positions parallèles de t et $\{s_1, \ldots, s_n\}$ un ensemble de termes. On note $t[s_1, \ldots, s_n]_{\xi_1, \ldots, \xi_n}$ au lieu de $t[s_1]_{\xi_1}[\ldots]_{\ldots}[s_n]_{\xi_n}$.

Lorsque l'ensemble de positions et l'ensemble de termes sont indexés par un même troisième ensemble I, on note $t[s_i, i \in I]_{\xi_i, i \in I}$ au lieu de $t[\{s_i \mid i \in I\}]_{\{\xi_i \mid i \in I\}}$. 21 ◀

Exemple 22.
Soit le terme $t = f(g(x), f(c, y))$ et $s = g(y)$. On a $t[s]_{2.1} = f(g(x), f(g(y), y))$ et $t[t]_1 = f(f(g(x), f(c, y)), f(c, y))$. 22 ◁

3.1.2 Systèmes de réécriture de termes

Dans la section précédente, les termes sont considérés isolément ; les règles de réécriture permettent d'exprimer les calculs qui peuvent être effectués sur ces termes.

Définition 23 (Système de réécriture de termes (TRS)).
Une règle de réécriture est un couple de termes $(\ell, r) \in \mathrm{T}(\Sigma, \mathcal{X}) \times \mathrm{T}(\Sigma, \mathcal{X})$ tel que ℓ n'est pas une variable et toute variable de r apparait aussi dans ℓ. Une telle règle est notée $\ell \to r$.

Un système de réécriture de termes est un ensemble de règles de réécriture.

Un système de réécriture R induit une relation sur les termes, dite relation de réécriture, encore notée $\to_R$ ou $\to$: pour tous termes s et t, $s \to_R t$ si et seulement s'il existe une règle $\ell \to r$ dans R, un contexte C et une substitution μ tels que $s = C[\ell\mu]$ et $t = C[r\mu]$.

Étant donné un ensemble de termes L, on note $R(L) = \{t \mid \exists s \in L, s \to_R t\}$. 23 ◀

Remarque 24. Nous n'appliquerons la réécriture qu'entre termes clos. En particulier, les substituions μ seront de $(\Sigma, \mathcal{X})$ vers $\mathrm{T}(\Sigma)$ et les contexte C seront sur $\mathrm{T}(\Sigma)$.

Définition 25 (Forme normale).
Un terme t est une forme normale pour le système R si $R(\{t\}) = \varnothing$.

On note $\textsc{Irr}(R)$ l'ensemble des formes normales de R. 25 ◀

Exemple 26.
Considérons la même signature que dans l'exemple 13 et soit R le système de réécriture constitué des règles

$$\begin{aligned} f(g(x), g(y)) &\to f(x, y) \\ f(c, g(y)) &\to g(y) \\ f(g(x), c) &\to g(x) \\ f(c, c) &\to c. \end{aligned}$$

Les formes normales sont la constante c et tous les termes de la forme $g^n(c)$ pour $n \in \mathbb{N}_*$:

$$\textsc{Irr}(R) = \{g^n(c) \mid n \in \mathbb{N}\}.$$

Au travers de cet exemple, on constate que les formes normales des systèmes de réécriture correspondent aux valeurs de la sémantique des programmes : la réécriture d'une expression complexe $f(g^m(c), g^n(c))$ jusqu'à une forme normale correspond au calcul de la valeur de la fonction f en l'argument $g^m(c), g^n(c)$. Le terme $g^n(c)$ peut être vu comme une représentation de l'entier naturel n. L'appel $f(g^m(c), g^n(c))$ calcule la valeur absolue de $m - n$. 26 ◁

Les questions d'existence et d'unicité de la valeur trouvent leur traduction dans les notions de terminaison et de confluence des systèmes de réécriture. Avant de les exposer, posons quelques notations.

Définition 27.
On note $\to_R^*$, ou $\to^*$ s'il n'y a pas d'ambigüité, la clôture réflexive et transitive de $\to_R$.

Pour L un ensemble de termes, on note $R^*(L) = \left\{ t \;\middle|\; \exists s \in L, s \to_R^* t \right\}$ et $R^!(L) = R^*(L) \cap \text{IRR}(R)$.

Les éléments de $R^!(\{t\})$, quand il y en a, sont appelés des formes normales de t (pour R). 27 ◀

Remarque 28. Pour tout ensemble de termes L, on a $R^*(L) \supseteq L$.

Définition 29 (Terminaison).
Un système de réécriture R est dit terminant s'il n'admet pas de chaine de réductions infinie. 29 ◀

Lemme 30.
Pour un système terminant R, pour tout terme t, $R^!(\{t\}) \neq \emptyset$, c'est-à-dire que tout terme admet au moins une forme normale. 30 ■

Définition 31 (Confluence).
Un système de réécriture R est dit confluent si pour tous termes s, t_1, t_2 tels que $t_1, t_2 \in R^*(\{s\})$, $R^*(\{t_1\}) \cap R^*(\{t_2\}) \neq \emptyset$. 31 ◀

Lemme 32.
Pour un système confluent R, tout terme t admet au plus une forme normale. 32 ■

Précisons le lien entre les formes normales et les valeurs. Il est classique de distinguer, parmi les symboles de fonction de la signature, ceux dont les règles du système de réécriture considéré donnent une définition.

Définition 33 (Symbole défini, symbole constructeur, terme constructeur).
Soit $(\Sigma, \mathcal{X})$ une signature et R un système de réécriture sur cette signature. Soit f un symbole de fonction. On dit que f est un symbole défini (par R) s'il existe une règle $\ell \to r$ de R telle que f est le symbole de tête de ℓ. Sinon, on dit que f est un symbole constructeur.

On note $\mathcal{C}$ l'ensemble des symboles constructeurs. On appelle terme constructeur un terme construit sur la signature $(\mathcal{C}, \mathcal{X})$. 33 ◀

Exemple 34.
Dans l'exemple 26, f est un symbole défini tandis que c et g sont des symboles constructeurs. 34 ◁

Lemme 35.
Tout terme constructeur est une forme normale, c'est-à-dire $\text{IRR}(R) \supseteq \text{T}(\mathcal{C})$. 35 ■

Les formes normales attendues sont les termes constructeurs. Cependant, il se peut qu'une définition donnée par R ne soit pas complète. Ainsi, dans l'exemple 26, si l'on omet la règle $f(c, g(y)) \to g(y)$, alors le terme $f(c, g(c))$ est une forme normale car aucune règle ne peut lui être appliquée, et ce bien qu'il s'agisse d'un terme constructeur.

Définition 36 (Complétude suffisante).
On dit d'un système de réécriture qu'il est suffisamment complet si l'ensemble de ses formes normales coïncide avec l'ensemble de ses termes constructeurs, c'est-à-dire $\textsc{Irr}(R) = \mathrm{T}(\mathcal{C})$. 36 ◀

Enfin, signalons que les systèmes de réécriture permettent d'exprimer tous les algorithmes.

Théorème 37.
Le formalisme des systèmes de réécriture de termes est Turing-complet. 37 ■

3.2 Correction et atteignabilité

Le théorème 37 exprime que tous les programmes peuvent être représentés par des systèmes de réécriture de termes. Ce formalisme fournit donc un cadre possible pour l'étude de leur correction.

Un terme représente l'état courant de l'exécution d'un programme, c'est-à-dire le calcul qui reste à faire. On peut donc déterminer des termes qui correspondent à des calculs qui ne devraient pas se présenter, comme une division par zéro. Ces mauvais termes ne doivent pas apparaitre dans l'ensemble des étapes de l'exécution d'un programme correct.

Le problème de correction se traduit donc en un problème de (non-)atteignabilité. Étant donné un système de réécriture R qui représente un programme, un ensemble de termes L_0 qui représente les états initiaux considérés pour le démarrage de l'exécution du programme et un ensemble de termes L_b qui représente les états correspondant à des erreurs, a-t-on $R^*(L_0) \cap L_b = \varnothing$?

Dans le chapitre suivant, nous expliquons les écueils auxquels chacun doit faire face pour résoudre ce problème ainsi que les solutions qui ont été proposées.

3.3 Linéarité

Nous serons amenés à faire des hypothèses de linéarité à gauche sur les systèmes de réécriture que nous traiterons. Quelles sont les restrictions apportées par ces hypothèses ?

Définition 38 (Linéarité à gauche, à droite).
Soit R un système de réécriture. On dit que R est linéaire à gauche si pour toute règle $\ell \to r$ de R, le terme ℓ est linéaire. On dit que R est linéaire à droite si pour toute règle $\ell \to r$ de R, le terme r est linéaire. On dit (parfois) qu'un système R est linéaire s'il est linéaire à gauche et linéaire à droite. 38 ◀

La linéarité à gauche n'est pas une contrainte gênante pour qui veut utiliser des TRS pour représenter des programmes fonctionnels (elle l'est cependant si l'on souhaite par exemple décrire et vérifier des protocoles cryptographiques, comme dans [GK00]). En effet, le membre gauche d'une règle de réécriture correspond, dans cette optique, à la donnée des paramètres formels de la fonction. Il est alors naturellement linéaire.

En revanche, une contrainte de linéarité à droite limite l'expressivité : il est impossible de parler plus d'une fois d'un même paramètre formel dans le corps de la fonction.

Exemple 39.
La fonction suivante

```
1 let f x y z = if x = y then x else z
```

pourrait se traduire, dans un cadre approprié, par le système de réécriture suivant

$$
\begin{aligned}
f(x,y,z) &\to if(eq(x,y),x,z) \\
if(true,x,y) &\to x \\
if(false,x,y) &\to y \\
eq(x,y)\ldots & \qquad \text{défini de façon à se réduire à } \textit{true} \text{ ou } \textit{false}
\end{aligned}
$$

qui est linéaire à gauche mais pas à droite, à cause de la première règle. 39 ◁

4 Résolution du problème d'atteignabilité

Le théorème 37 énonce que les systèmes de réécriture sont un formalisme Turing-complet, il n'est donc pas étonnant que les problèmes d'indécidabilité de la section 2.2.2 y trouvent leur traduction.

Chacun des problèmes suivants est indécidable :

1. étant donné un TRS R, dire s'il est terminant ;
2. étant donné un TRS R, dire s'il est confluent ;
3. étant donné un TRS R et deux termes s et t, dire si $s \to_R^* t$.

Ainsi, la donnée d'un ensemble de termes sous la forme $R^*(L)$ n'est pas une description effective, faute de pouvoir décider quels sont les termes de cet ensemble. Il est illusoire d'espérer décider si un tel ensemble contient des « mauvais termes », comme proposé en section 3.2 !

4.1 Régularité

Pour répondre à ce problème, nous allons maintenant présenter le formalisme des automates d'arbres [Com+08]. La notion d'automate d'arbres est analogue à celle d'automate de mots, et il en découle une notion de régularité analogue à celle qu'on trouve dans le cas des mots sur un alphabet. Ce formalisme présente l'avantage d'être conforme à la structure arborescente des termes.

Nous allons disposer de théorèmes sur les langages réguliers qui nous permettront de décider la vacuité de leurs intersections et donc de tenter de donner une réponse au problème d'atteignabilité.

Définition 40 (Automate d'arbre).

Soit Σ une signature. Un automate d'arbres $\mathcal{A}$ sur Σ est donné par

— un ensemble fini d'états Q,

— un ensemble de delta-transitions de la forme $f(q_1, \ldots, q_k) \rightarrowtail q'$ où k est un entier naturel, $f \in \Sigma_k$ et $q_1, \ldots, q_k, q' \in Q$,

— un ensemble d'epsilon-transitions de la forme $q \rightarrowtail q'$, où $q, q' \in Q$ et

— une partie Q_F de Q dont les éléments sont appelés états terminaux.

Une configuration de $\mathcal{A}$ est un terme de $\mathrm{T}(\Sigma, Q)$. [1]

1. En particulier, les termes clos sur Σ sont des configurations de tout automate sur Σ.

Étant donné deux configurations c et c' d'un automate $\mathcal{A}$, on dit que $\mathcal{A}$ reconnait c en c' (en une étape) s'il existe un contexte sur (Σ, Q) et une transition $\tau \rightarrowtail \tau'$ de $\mathcal{A}$ tels que $c = C[\tau]$ et $c' = C[\tau']$. Cette relation est encore notée $\rightarrowtail$ ou $\rightarrowtail_{\mathcal{A}}$. On note par une étoile sa clôture réflexive et transitive. 40 ◀

Définition 41 (Langage).
On appelle langage sur Σ tout ensemble de termes clos sur Σ. 41 ◀

Définition 42 (Langage reconnu, langage régulier).
Soit $\mathcal{A}$ un automate sur Σ. Pour tout ensemble X d'états de $\mathcal{A}$, on note $\mathscr{L}(\mathcal{A}, X) = \left\{t \in \mathrm{T}(\Sigma) \;\middle|\; \exists q \in X, t \rightarrowtail_A^* q\right\}$. On dit que $\mathscr{L}(\mathcal{A}, X)$ est le langage reconnu par $\mathcal{A}$ en X. Si $X = \{q\}$, on note $\mathscr{L}(\mathcal{A}, q)$ au lieu de $\mathscr{L}(\mathcal{A}, \{q\})$.

Si l'ensemble Q_F des états terminaux de $\mathcal{A}$ a été précisé, on note $\mathscr{L}(\mathcal{A}) = \mathscr{L}(\mathcal{A}, Q_F)$ et on appelle ce langage le langage reconnu par $\mathcal{A}$.

On dit d'un langage qu'il est régulier s'il est reconnu par un automate d'arbres. 42 ◀

Remarque 43. La notion d'état final sert donc seulement à parler du langage reconnu par l'automate avec une notation plus légère : $\mathscr{L}(\mathcal{A})$ au lieu de $\mathscr{L}(\mathcal{A}, Q_F)$ pour Q_F déterminé.

Cependant, dans ce document, nous nous intéresserons aux langages reconnus par les automates en chacun de leurs états et pas seulement en leurs états finaux. En conséquence, la notion d'état final sera peu utilisée, et il pourra arriver qu'on ne précise pas, dans un automate, quels sont les états finaux.

Exemple 44.
Considérons une signature Σ comprenant deux symboles de constantes c et d, un symbole de fonction unaire g et un symbole de fonction binaire f. Construisons un automate d'arbres qui reconnait le langage

$$L = \{f(g^m(c), g^n(d)) \mid m \in \mathbb{N}, n \in \mathbb{N}_*\}.$$

Sont à construire un ensemble d'états Q, une partie Q_f de Q pour les états terminaux et une relation de transition Δ.

Il nous faut un état pour reconnaitre chacune des deux constantes (il ne peut s'agir du même, sinon on ne pourrait pas distinguer $f(g(c), g(c))$ et $f(g(c), g(d))$). Soit donc les états et les transitions $c \to q_c$, $d \to q_d$. Une fois qu'on a lu un c, composer formellement par g ne change rien vis-à-vis de L : prenons donc $g(q_c) \to q_c$. En revanche, les arbres que nous voulons accepter doivent avoir au moins un $g(\dots)$ dans le deuxième argument de f : il faut distinguer la présence d'un g, soit donc l'état $q_{g(d)}$ et les transitions $g(q_d) \to q_{g(d)}$ et $g(q_{g(d)}) \to q_{g(d)}$. Enfin, soit $f(q_c, q_{g(d)}) \to q_f$. On prend donc $Q = \{q_c, q_d, q_{g(d)}, q_f\}$, $Q_f = \{q_f\}$ et Δ comme indiqué ci-dessus. Les transitions initiales sont $c \to q_c$ et $d \to q_d$.

Une exécution de cet automate sur l'arbre $f(g(c), g(d))$ est représentée en figure 4.1.
44 ◁

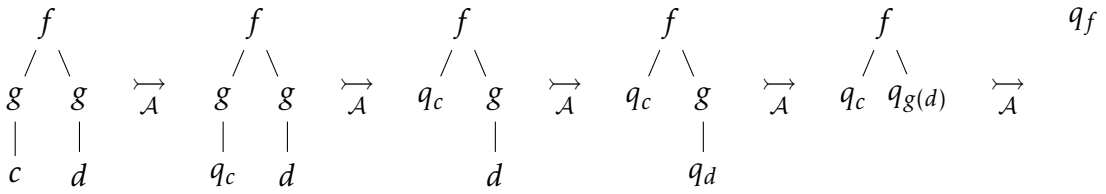

FIGURE 4.1 – Une exécution de l'automate de l'exemple 44

Exemple 45.

Considérons la signature Σ qui comprend les symboles d'arité zéro *nil* et 0, les symboles d'arité un *succ*, *pred* et *filter* et le symbole d'arité deux *cons*. Nous souhaitons que 0 représente in fine l'entier zéro, *succ* l'opération successeur sur les entiers, *pred* l'opération prédécesseur, et que *nil* représente la liste vide et *cons* le constructeur de listes d'entiers. Enfin, *filter* est censé représenter la fonction qui élimine tous les zéros d'une liste d'entiers. Cela nous conduit à définir un système de réécriture R composé des règles suivantes.

$$\begin{aligned} \mathit{filter}(\mathit{nil}) &\to \mathit{nil},\\ \mathit{filter}(\mathit{cons}(\mathit{succ}(X), Y)) &\to \mathit{cons}(\mathit{succ}(X), \mathit{filter}(Y)),\\ \mathit{filter}(\mathit{cons}(\mathit{pred}(X), Y)) &\to \mathit{cons}(\mathit{pred}(X), \mathit{filter}(Y)),\\ \mathit{filter}(\mathit{cons}(0, Y)) &\to \mathit{filter}(Y),\\ \mathit{pred}(\mathit{succ}(X)) &\to X,\\ \mathit{succ}(\mathit{pred}(X)) &\to X \end{aligned}$$

Nous souhaitons étudier les descendants par R du terme $\mathit{filter}(\mathit{cons}(\mathit{pred}(\mathit{succ}(0)), \mathit{nil}))$, qui représente l'appel du filtrage des zéros sur une liste à un élément qui est (sémantiquement) l'entier zéro mais écrit comme le prédécesseur de 1. Considérons pour cela l'automate $\mathcal{A}_0$ sur la signature Σ comprenant les états q_n, q_0, q_s, q_p, q_c et q_f et les transitions

$$\begin{aligned} \mathit{nil} &\rightarrowtail q_n\\ 0 &\rightarrowtail q_0\\ \mathit{succ}(q_0) &\rightarrowtail q_s\\ \mathit{pred}(q_s) &\rightarrowtail q_p\\ \mathit{cons}(q_p, q_n) &\rightarrowtail q_c\\ \mathit{filter}(q_c) &\rightarrowtail q_f \end{aligned}$$

On a

$$\mathscr{L}(\mathcal{A}_0, q_f) = \{\mathit{filter}(\mathit{cons}(\mathit{pred}(\mathit{succ}(0)), \mathit{nil}))\}$$

et

$$R(\mathscr{L}(\mathcal{A}_0, q_f)) = \{\mathit{filter}(\mathit{cons}(0, \mathit{nil})), \quad \mathit{cons}(\mathit{pred}(\mathit{succ}(0)), \mathit{filter}(\mathit{nil}))\}.$$

On voit au travers de cet exemple que la réécriture générale produit des résultats qui ne sont pas conformes à l'intention qui était la nôtre en concevant le système R. 45 ◁

Il est facile de concevoir un automate dont un certain état q ne reconnait aucun terme : il suffit par exemple qu'aucune transition n'aboutisse à q. Mais un tel état est alors inutile et il vaut mieux le supprimer. La notion d'accessibilité formalise cela.

Définition 46 (Accessibilité dans les automates).
Soit $\mathcal{A}$ un automate et q un état de $\mathcal{A}$. On dit que l'état q est accessible s'il reconnait au moins un terme, c'est-à-dire si $\mathscr{L}(\mathcal{A}, q) \neq \emptyset$.

On dit d'un automate $\mathcal{A}$ qu'il est accessible si tous ses états le sont. 46 ◀

Lemme 47.
Étant donné un automate, on peut construire un automate accessible qui reconnait le même langage. 47 ■

Sans rappeler tous les résultats classiques sur les langages réguliers d'arbres [Com+08], disons qu'ils sont également analogues à ceux qu'on connait sur les langages réguliers de mots. En particulier :

Lemme 48.
L'intersection de deux langages réguliers est un langage régulier, et on sait effectivement construire un automate la reconnaissant.

On sait décider si le langage reconnu par un automate est vide. 48 ■

Remarque 49. Par récurrence, on peut établir que l'intersection d'un nombre fini de langages réguliers est un langage régulier (il en va de même pour l'union). Cependant, la classe des langages réguliers n'est pas stable par intersection ni par union quelconque. Ainsi, pour n entier naturel, le langage $L_n = \{g(f^n(c), f^n(c))\}$ est régulier (car fini), mais le langage $\{g(f^n(c), f^n(c)) \mid n \in \mathbb{N}\}$, qui est la réunion de tous les langages L_n, n'est pas régulier.

Remarque 50. Il arrivera que des transitions portent des couleurs, $\mathfrak{R}$ ou $\mathfrak{E}$, dont la signification sera introduite au fur et à mesure. Il arrivera aussi qu'étant donné un automate $\mathcal{A}$ dont certaines transitions portent la couleur $\mathfrak{R}$, nous voulions considérer l'automate auquel on a retiré les transitions portant cette couleur. Cet automate est noté $\mathcal{A}^{\not\mathfrak{R}}$.

4.2 Préservation de la régularité

Tout langage n'est pas régulier. Aussi, se placer dans le cadre des langages réguliers peut apparaitre restrictif. Cela ne l'est pas vraiment en pratique pour ce

qui concerne la définition du langage initial L_0. La restriction apparait dans la définition de l'ensemble des termes d'erreur L_b : nous ne pourrons vérifier que des propriétés régulières. Ainsi, il n'est pas possible de construire un langage régulier L_b qui représenterait l'ensemble des nombres premiers, ou encore un langage du type $\left\{g(f^k(c), h^\ell(d)) \;\middle|\; k \neq \ell\right\}$ qui pourrait être utilisé pour assurer qu'un programme renvoie deux objets (par exemple des listes) de même taille. Cependant, le cadre régulier couvre de nombreux cas.

La difficulté principale à laquelle nous sommes confrontés réside plutôt dans la régularité de $R^*(L_0)$. En effet, nous savons décider la vacuité de l'intersection de deux langages, à condition qu'ils soient tous deux réguliers. La régularité de L_b étant supposée, de nombreux auteurs se sont intéressés à la transmission de la régularité de L_0 à $R^*(L_0)$. Cette propriété, la préservation de la régularité par un système de réécriture, est indécidable [GT95] et a toutes les chances d'être fausse dès que le système comprend une règle dont le membre droit n'est pas linéaire.

Exemple 51.
Considérons la signature composée du symbole d'arité zéro c, des symboles d'arité un s et f et du symbole d'arité deux g. Soit R le système constitué de la seule règle $f(x) \to g(x,x)$ et soit $L_0 = \left\{f(s^k(c)) \;\middle|\; k \in \mathbb{N}\right\}$. Le langage L_0 est régulier, mais

$$R^*(L_0) = \left\{g(s^k(c), s^k(c)) \;\middle|\; k \in \mathbb{N}\right\}$$

ne l'est pas. Ainsi, R ne préserve pas la régularité. 51 ◁

De nombreuses classes ont été étudiées au cours des trois dernières décennies par de nombreux auteurs [Tak04 ; Coq+91 ; Rét99] ; Genet en donne une hiérarchie dans [Gen09]. Ces classes sont définies par des restrictions sur la forme des règles de réécriture ; toutes imposent que les membres droits des règles soient linéaires. Il s'agit d'une restriction forte pour la représentation de programmes fonctionnels, comme indiqué en section 3.3.

4.3 Surapproximation : complétion d'automate

La démarche dans laquelle s'inscrit ce travail consiste non à tenter de traiter le cas où $R^*(L_0)$ est régulier, mais à construire un langage régulier K tel que $K \supseteq R^*(L_0)$, c'est-à-dire une surapproximation régulière de ce dernier ensemble.

Une telle surapproximation n'existe pas toujours [BH08].

Pour construire une telle surapproximation, il est pertinent de compléter l'automate reconnaissant le langage initial L_0. C'est l'objet des travaux menés depuis 20 ans par Thomas Genet et ses équipes successives [Gen97 ; Gen09 ; GR10 ; Boy10]. Les méthodes de complétion d'automate d'arbres connaissent plusieurs variantes et

plusieurs formalisations. L'objectif de ce document n'est pas de les recenser, mais de présenter la variante récente servant de base à la nôtre dans un formalisme adapté, jetant les bases d'une possible unification des différentes présentations en usage.

Partant d'un automate donné $\mathcal{A}$, une étape de la procédure de complétion de $\mathcal{A}$ selon une règle $\ell \to r$ d'un système de réécriture R consiste à énumérer les configurations de $\mathcal{A}$ de la forme $\ell\sigma$ (où l'on a remplacé les variables de ℓ par des états de $\mathcal{A}$), à vérifier, pour chacune d'elle, si cette configuration est reconnue par $\mathcal{A}$ en un état q ($\ell\sigma \rightarrowtail^*_{\mathcal{A}} q$) et, dans l'affirmative, à regarder le comportement de $\mathcal{A}$ sur la configuration $r\sigma$. Si l'automate $\mathcal{A}$ ne reconnait pas $r\sigma$ en le même état q, la procédure de complétion rajoute les transitions nécessaires pour ce faire. Cette étape est itérée jusqu'à ce que l'ajout de nouvelles transitions ne soit plus nécessaire.

La terminaison de ce procédé n'est pas garantie en général, mais on montre que s'il termine, alors il fournit nécessairement un automate reconnaissant une surapproximation K de l'ensemble $R^*(L_0)$.

Avant d'exposer formellement la complétion d'automate d'arbres, nous devons aborder quelques notions supplémentaires.

4.3.1 Préliminaires

4.3.1.1 Normalisation d'une transition d'automate

Ajouter une transition à un automate est une chose aisée. Mais tout n'est pas transition d'automate ! Il résulte de la définition 40 que le membre gauche d'une transition (ce qui est reconnu en une étape) doit être soit un état (cas des epsilon-transitions), soit une constante ou une configuration de la forme $f(q_1, \ldots, q_k)$ où les $q_1, \ldots, q_k$ sont des états (cas des delta-transitions). Ainsi, aucune transition ne peut avoir pour membre gauche une configuration aussi compliquée que $f(q_1, a)$ où a est une constante, ni $f(g(q), h(q_1, q_2))$.

Il y a donc lieu de procéder par étapes, c'est ce qu'on appelle la normalisation. Cette procédure consiste à décomposer progressivement la configuration à traiter et à introduire de nouveaux états et transitions. Ainsi, pour normaliser $f(q_1, a)$ vers q', on commencera par constater que q_1 est déjà un état, puis on sélectionnera un état q_a et on s'assurera que $a \rightarrowtail q_a$, avant d'ajouter une transition $f(q_1, q_a) \rightarrowtail q'$.

La description de cette procédure soulève deux questions : comment sélectionne-t-on les états intermédiaires qui reconnaissent les sous-configurations ; dans quel ordre parcourt-on la configuration à normaliser ?

Il existe deux réponses pertinentes à la première question. On peut choisir de systématiquement rajouter un nouvel état. On peut aussi rechercher dans l'automate étudié si la sous-configuration en question n'est pas déjà reconnue en un certain état et sélectionner un de ceux-là, et n'ajouter un nouvel état qu'en cas d'échec de

cette tentative. La première méthode présente l'avantage d'être plus simple à exposer. La seconde, celui de produire des automates moins gros, moins redondants. Ces choix ont d'autres conséquences que nous explorerons plus loin.

La question de l'ordre de parcours des sous-configurations est beaucoup moins cruciale. Elle doit être déterminée dans l'implémentation des algorithmes, mais la théorie ne repose pas sur un choix particulier en cette matière.

4.3.1.2 Réécriture modulo équations

Le processus de complétion d'automate ajoute de nouveaux états, à cause de la normalisation. Il s'auto-alimente donc, ce qui nuit à la terminaison. Plusieurs méthodes ont été développées pour contrer ce phénomène. Dans ce document, nous ne traiterons que la fusion équationnelle introduite dans [GR10] et utilisée par toutes les versions ultérieures de la complétion. Nous introduisons ici la notion d'équation entre termes et celle de réécriture modulo de telles équations. Ces notions seront utiles à la compréhension de l'algorithme de fusion équationnelle décrit en 4.3.2.2.

Les équations entre termes n'ont pas été inventées pour la complétion équationnelle [BN98]. Les théories équationnelles sont à la fois un objet d'étude et un outil de la réécriture de termes.

On rappelle qu'une relation d'équivalence est une relation binaire réflexive, symétrique et transitive.

Définition 52 (Congruence).
Soit $(\Sigma, \mathcal{X})$ une signature. Une relation de congruence $\equiv$ sur $\mathrm{T}(\Sigma, \mathcal{X})$ est une relation d'équivalence qui est compatible avec le processus de construction des termes : pour tout entier naturel k, pour tout $f \in \Sigma_k$, pour tous $t_1, \ldots, t_k, s_1, \ldots, s_k \in \mathrm{T}(\Sigma, \mathcal{X})$, si pour tout $i \in [\![1\,;k]\!]$, $t_i \equiv s_i$, alors $f(t_1, \ldots, t_k) \equiv f(s_1, \ldots, s_k)$. 52 ◀

Définition 53 (Théorie équationnelle).
Soit $(\Sigma, \mathcal{X})$ une signature. Une équation sur cette signature est un couple de termes (g, d), noté $g \equiv d$.

Une théorie équationnelle est un ensemble E de telles équations. Deux termes s et t sont équivalents modulo E, ce qu'on note $s \equiv_E t$, s'il existe un contexte C, une équation $(g, d) \in E$ et une substitution θ tels que $s = C[g\theta]$ et $t = C[d\theta]$. 53 ◀

Lemme 54.
La relation $\equiv_E$ induite par une théorie équationnelle E est une relation de congruence sur $\mathrm{T}(\Sigma, \mathcal{X})$. 54 ■

Remarque 55. Le quotient $\mathrm{T}(\Sigma)/_{\equiv_E}$ est structuré par la signature Σ : étant donné un symbole de fonction f d'arité k et des classes $\varkappa_1, \ldots, \varkappa_k$, on définit la classe $\varkappa = f(\varkappa_1, \ldots, \varkappa_k)$ comme étant la classe commune des termes de la forme $t = f(t_1, \ldots, t_k)$ où chaque t_i est un représentant de la classe $\varkappa_i$. L'indépendance du choix des représentants est assuré par les propriétés de congruence.

Il est possible d'introduire la réécriture comme moyen d'étudier, si possible de décider, une théorie équationnelle donnée (c'est ce que fait [BN98]).

Exemple 56.
Soit la théorie équationnelle donnée par les axiomes de l'arithmétique, dont on donne un extrait :

$$\begin{aligned} x + 0 &= 0 \\ x + S(y) &= S(x + y) \\ x \times 0 &= 0 \\ x \times S(y) &= x \times y + x \end{aligned}$$

Un moyen de décider que des égalités comme $S(S(S(0))) + S(S(0)) = S(S(S(S(0)))) + S(0)$ sont une conséquence de ces axiomes est d'orienter chacun d'eux de la gauche vers la droite (car on a pris soin ici de les présenter convenablement !) et de calculer que le terme de gauche et le terme de droite se réécrivent en $S(S(S(S(S(0)))))$. [56] ◁

Lorsque nous avons dit, dans l'introduction de section 3.1, que « ce sont les règles de réécriture qui donnent leur sens aux termes », nous avons quelque peu triché pour donner une présentation des systèmes de réécriture qui se lie bien aux sections qui précédaient. En réalité, c'est bien souvent une théorie équationnelle qui donne le sens des choses, et la construction d'un système de réécriture à partir d'icelle est déjà un moyen de l'étudier.

Cette approche est fondatrice pour la logique moderne. Dans [Fre92], Gottlob Frege souhaite étudier ce qui constitue le sens et la dénotation des termes, et pour ce faire commence par analyser des énoncés d'identité. Ce document n'a pas pour objet de commenter cette analyse, mais on voit au travers de cette démarche qu'on touche du doigt le sens lorsqu'on maitrise les égalités.

Il s'agit là du problème du mot : étant donné une théorie équationnelle et deux termes, ces termes sont-ils en relation ? Ce problème est indécidable et l'orientation des équations par des règles de réécriture participe de la recherche d'une procédure de décision dans les cas favorables.

Théorème 57.
Le problème du mot d'une théorie équationnelle E est décidable si E est fini et que le système de réécriture obtenu en traitant chaque équation $\ell \equiv r$ comme une règle $\ell \to r$ est terminant et confluent. [57] ■

Mais trouver une orientation pertinente pour une équation n'est pas toujours possible. C'est le cas notamment des équations qui décrivent la commutativité d'une opération. La réécriture modulo ces équations non orientables est une réponse à cette difficulté. Elle traite non plus les termes, mais les classes d'équivalence de termes.

Définition 58.
Soit E une théorie équationnelle et t un terme. On note $[t]_E$ la classe d'équivalence de t par la relation d'équivalence $\equiv_E$: $[t]_E = \{u \in \mathrm{T}(\Sigma, \mathcal{X}) \mid t \equiv_E u\}$. 58 ◀

Définition 59 (Réécriture modulo équations).
Soit R un système de réécriture et E une théorie équationnelle portant sur la même signature $(\Sigma, \mathcal{X})$. Soit $s, t \in \mathrm{T}(\Sigma, \mathcal{X})$. On dit que s se réécrit en t par R modulo E en une étape, et on note $s \to_{R/E} t$ s'il existe $s', t' \in \mathrm{T}(\Sigma, \mathcal{X})$ tels que $s \equiv_E s'$, $t \equiv_E t'$ et $s' \to_R t'$.

Alternativement, on dit que la classe $[s]_E$ se réécrit en la classe $[t]_E$ en une étape, ce qu'on note $[s]_E \to_{R/E} [t]_E$ ou encore $s \to_{R/E} t$, s'il existe $s' \in [s]_E$ et $t' \in [t]_E$ tels que $s' \to_R t'$. 59 ◀

Remarque 60. Les deux définitions alternatives précédentes sont équivalentes.

Exemple 61.
Considérons les axiomes de la théorie des groupes :

$$\begin{aligned} x \star e &= x \\ e \star x &= x \\ x \star x^{\star -1} &= e \\ x^{\star -1} \star x &= e \\ (x \star y) \star z &= x \star (y \star z) \end{aligned}$$

Les premiers axiomes s'orientent naturellement. Le dernier permet en fait de considérer des mots $x \star y \star z$ et d'appliquer les autres axiomes où l'on veut ; en procédant ainsi, on effectue implicitement des calculs modulo associativité. De même, l'axiome de commutativité des groupes abéliens,

$$x \star y = y \star x$$

ne peut pas être orienté dans un sens où l'autre. On préfèrera travailler avec un système R contenant les règles

$$\begin{aligned} x \star e &\to x \\ e \star x &\to x \\ x \star x^{\star -1} &\to e \\ x^{\star -1} \star x &\to e \end{aligned}$$

appliqué à des termes modulo E, avec les deux équations restantes. 61 ◁

Lemme 62.
Soit R un système de réécriture et E une théorie équationnelle portant sur la même signature $(\Sigma, \mathcal{X})$. Pour tous $s, t \in \mathrm{T}(\Sigma, \mathcal{X})$, $[s]_E \to_{R/E} [t]_E$ si et seulement si $R([s]_E) \cap [t]_E \neq \varnothing$. 62 ■

Démonstration.
Il suffit de voir les classes d'équivalences pour ce qu'elles sont : des ensembles de termes, c'est-à-dire des langages. Dans le sens direct, le t' de la définition 59 fournit un élément commun à $R([s]_E)$ et $[t]_E$. Dans le sens réciproque, soit $t' \in R([s]_E) \cap [t]_E$; par définition de l'opérateur $R(\cdot)$, il existe $s' \in [s]_E$ tel que $s' \to_R t'$, ce qui permet de conclure. □

Définition 63.
On définit comme pour la réécriture classique la clôture réflexive et transitive $\to^*_{R/E}$. On définit aussi, pour tout langage L, $(R/E)(L) = \left\{t \in \mathrm{T}(\Sigma,\mathcal{X}) \;\middle|\; \exists s \in L, s \to_{R/E} t\right\}$ et $(R/E)^*(L) = \left\{t \in \mathrm{T}(\Sigma,\mathcal{X}) \;\middle|\; \exists s \in L, s \to^*_{R/E} t\right\}$ 63 ◀

Remarque 64. Dans la définition de $\to^*_{R/E}$, et donc aussi de $(R/E)^*(\cdot)$, la théorie équationnelle entre en jeu entre chaque étape de réécriture, et non seulement au début à la fin : il ne s'agit pas d'un $(R^*)/E$ mais bien d'un $(R/E)^*$.

Définition 65.
Pour tout langage L, on note encore $[L]_E$ la réunion des classes d'équivalence des éléments de L : $[L]_E = \bigcup_{t \in L} [t]_E$. 65 ◀

Remarque 66. Pour tout langage L, on a $(R/E)(L) \supseteq [R(L)]_E$. On a $(R/E)^*(L) \supseteq L$ mais pas nécessairement $(R/E)^*(L) \supseteq [L]_E$. Si $\varkappa$ est une classe d'équivalence ou une réunion de classes d'équivalences, alors $[\varkappa]_E = \varkappa$.

Lemme 67.
Si $\varkappa$ est une classe d'équivalence pour E, alors $(R/E)(\varkappa) = [R(\varkappa)]_E$. 67 ■

Démonstration.
Soit $u \in \mathrm{T}(\Sigma,\mathcal{X})$ tel que $\varkappa = [u]_E$. Pour tout $t \in \mathrm{T}(\Sigma,\mathcal{X})$,

$$\begin{aligned}
t \in (R/E)(\varkappa) &\Leftrightarrow \exists s \in \varkappa, \exists s' \in [s]_E, \exists t' \in [t]_E, s' \to_R t' \\
&\Leftrightarrow \exists s \equiv_E u, \exists s' \equiv_E s, \exists t' \equiv_E t, s' \to_R t' \\
&\Leftrightarrow \exists t' \equiv_E t, \exists s' \equiv_E u, s' \to_R t' \\
&\Leftrightarrow \exists t' \equiv_E t, \exists s' \in \varkappa, s' \to_R t' \\
&\Leftrightarrow \exists t' \equiv_E t, t' \in R(\varkappa) \\
&\Leftrightarrow t \in [R(\varkappa)]_E .
\end{aligned}$$

□

Le champ des théories équationnelles est labouré par de nombreux chercheurs et donne de nombreux fruits. Pour notre part, nous nous servirons d'elles non pour décrire ou appréhender la réalité, mais pour l'abstraire et l'approximer. Ainsi, plutôt que des équations dans le genre des exemples 56 et 61, nous utiliserons dans certains cas l'équation $s(s(X)) = s(X)$. Cette équation n'est « pas vraie », car elle confond dans une même classe tous les entiers naturels non nuls. Mais c'est précisément ce que nous voudrons, afin d'effectuer des approximations.

En effet, la présente section est introduite par le problème que pose la création de nouveaux états pour le processus de complétion d'automate d'arbres. La fusion équationnelle, dont nous donnerons la définition formelle ultérieurement, est une réponse à ce problème qui repose sur un jeu d'équations d'approximation E, fixé d'avance et qui est utilisé pour fusionner entre eux des états de l'automate en cours de complétion. Cette opération de fusion a un avantage : elle favorise la terminaison de la complétion. Elle a un cout : une perte de précision, un grossissement de la surapproximation K que l'on peut obtenir à la fin du processus. Le cadre de la réécriture modulo équations permettra, sous certaines hypothèses, de quantifier cette perte de précision et d'identifier des jeux d'équations E pour lesquels cette perte de précision n'a en réalité pas lieu.

Les « bonnes équations », qui font converger la complétion tout en préservant une précision suffisante, dépendent du système de réécriture R, du langage initial L_0 et de l'automate $\mathcal{A}_0$ qui le reconnait ainsi que l'objectif (le langage L_b dont on veut montrer que $R^*(L_0)$ est disjoint). Trouver ces équations n'est pas toujours possible, puisqu'il n'existe pas toujours de surapproximation régulière convenable [BH08]. C'est, d'une façon générale, un exercice ad hoc et difficile. Valérie Murat d'une part, et Benoit Boyer d'autre part, proposent des méthodes pour générer automatiquement des équations idoines [Mur14] et pour raffiner automatiquement les surapproximations effectuées en exploitant les contre-exemples trouvés [Boy10]. Nous en reparlerons au chapitre 6.

4.3.1.3 Équations et automates : E-cohérence

Nous aurons donc à appliquer des équations à des automates. En particulier, nous manipulerons des epsilon-transitions traduisant des équations. Ces epsilon-transitions porteront la couleur $\mathfrak{E}$. En raison de la symétrie de la relation de congruence induite par E, ces epsilon-transitions seront croisées : lorsque nous voudrons traduire le fait qu'un état q_1 et un état q_2 reconnaissent des termes E-équivalents, nous ajouterons une transition $q_1 \overset{\mathfrak{E}}{\rightarrowtail} q_2$ et une transition $q_2 \overset{\mathfrak{E}}{\rightarrowtail} q_1$. Nous adopterons pour une telle paire d'epsilon-transitions croisées la notation $q_1 \overset{\mathfrak{E}}{\rightleftarrows} q_2$. Généralisons cette notation.

Définition 68.
Soit $\mathcal{A}$ un automate et c_1, c_2 deux configurations de $\mathcal{A}$. On dit que $c_1 \overset{\mathfrak{E}}{\rightleftarrows}_{\mathcal{A}} c_2$ lorsque $c_1 \overset{\mathfrak{E}}{\rightarrowtail}_{\mathcal{A}} c_2$ et $c_2 \overset{\mathfrak{E}}{\rightarrowtail}_{\mathcal{A}} c_1$.

On note $\overset{\mathfrak{E},*}{\rightleftarrows}_{\mathcal{A}}$ la clôture réflexive et transitive de $\overset{\mathfrak{E}}{\rightleftarrows}_{\mathcal{A}}$. 68 ◀

Remarque 69. $c_1 \overset{\mathfrak{E},*}{\rightleftarrows}_{\mathcal{A}} c_2$ est plus fort que $c_1 \overset{\mathfrak{E},*}{\rightarrowtail}_{\mathcal{A}} c_2$ et $c_2 \overset{\mathfrak{E},*}{\rightarrowtail}_{\mathcal{A}} c_1$: pour que $c_1 \overset{\mathfrak{E},*}{\rightleftarrows}_{\mathcal{A}} c_2$, il faut que chacune des étapes d'un chemin $c_1 \overset{\mathfrak{E},*}{\rightarrowtail}_{\mathcal{A}} c_2$ soit réversible. En pratique

cependant, les transitions de couleur $\mathfrak{E}$ seront toujours ajoutées par paires croisées, et cette distinction ne trouvera pas à se manifester.

De la même façon que nous avons défini la réécriture modulo E, nous souhaiterons utiliser une automate $\mathcal{A}$ sur les classes de termes. Pour cela, il faut que l'automate considéré soit E-cohérent. Un automate E-cohérent est un automate qui ne mélange pas plusieurs classes d'équivalences de $\equiv_E$ dans un même état, de sorte qu'on peut considérer qu'il reconnait des classes d'équivalence.

Définition 70 (E-cohérence).
Soit E un ensemble d'équations et $\mathcal{A}$ un automate. On dit que l'automate $\mathcal{A}$ est E-cohérent si pour tout état q de $\mathcal{A}$ et pour tous $s, t \in \mathscr{L}(\mathcal{A}, q)$, $s \equiv_E t$.

Si $\mathcal{A}$ est un automate E-cohérent, pour tout état q de $\mathcal{A}$ tel que $\mathscr{L}(\mathcal{A}, q) \neq \emptyset$, on note $[q]_E^{\mathcal{A}}$ la classe commune par $\equiv_E$ des termes de $\mathscr{L}(\mathcal{A}, q)$.

Cette notation est étendue aux configurations c de $\mathcal{A}$ par induction :
— si c est un état tel que $\mathscr{L}(\mathcal{A}, c) \neq \emptyset$, $[c]_E^{\mathcal{A}}$ est défini comme précédemment,
— si c est un état tel que $\mathscr{L}(\mathcal{A}, c) = \emptyset$, on note $[c]_E^{\mathcal{A}} = \bot$,
— si c est une constante, on note $[c]_E^{\mathcal{A}} = [c]_E$,
— si $c = f(c_1, \ldots, c_k)$ avec un indice i tel que $[c_i]_E^{\mathcal{A}} = \bot$, on note $[c]_E^{\mathcal{A}} = \bot$,
— si $c = f(c_1, \ldots, c_k)$ avec pour tout indice i, $[c_i]_E^{\mathcal{A}} \neq \bot$, alors pour tout indice i, on choisit un terme t_i dans la classe $[c_i]_E^{\mathcal{A}}$, on construit le terme $t = f(t_1, \ldots, t_k)$ et on note $[c]_E^{\mathcal{A}} = [t]_E$. 70 ◀

Exemple 71.
Si E est vide ou si E ne contient que des équations qui sont naturellement vraies, comme $x = x$ ou $f(x, c) = f(x, c)$, alors la relation $\equiv_E$ est l'égalité syntaxique entre termes. Un automate E-cohérent pour cet ensemble d'équations doit alors reconnaitre en chaque état au plus un terme : cela n'est possible que si le langage à reconnaitre est fini.

Plus généralement, pour qu'un langage L puisse être reconnu par un automate E-cohérent, il faut que l'ensemble quotient $\mathrm{T}(\Sigma)/_{\equiv_E}$ soit fini. 71 ◁

Il n'est donc pas toujours possible de disposer d'un automate E-cohérent pour reconnaitre un langage L donné, mais on peut en ce cas rajouter des équations.

Lemme 72.
Si $E \subseteq E'$, alors tout automate E-cohérent est également E'-cohérent. 72 ■

Démonstration.
Les classes de $\equiv_{E'}$ sont des réunions de classes de $\equiv_E$, l'obligation de ne reconnaitre en chaque état que des termes d'une même classe devient donc plus permissive quand E grossit. □

4.3.2 Procédure de complétion d'automate d'arbre

Soit R un système de réécriture ; nous le supposerons bientôt linéaire à gauche. Nous allons décrire d'abord la procédure de complétion dite exacte, qui ne fait pas intervenir d'équations et n'opère aucune fusion d'états. Nous décrirons ultérieurement la fusion équationnelle, qui, associée à la completion exacte, constitue la complétion équationnelle.

4.3.2.1 Completion exacte

Nous allons décrire la notion de paire critique, qui est paradoxalement dénotée par un triplet, avant d'exposer ce qui sera l'invariant de correction de la procédure puis une étape de complétion exacte. Nous allons devoir faire jouer sur le même terrain des éléments d'origines différentes : les termes sur lesquels porte R, qui sont dans $\mathrm{T}(\Sigma, \mathcal{X})$, et les configurations de l'automate $\mathcal{A}$, qui sont dans $\mathrm{T}(\Sigma, Q)$. Nous allons pour cela utiliser des substitutions de $(\Sigma, \mathcal{X})$ vers $(\varnothing, Q)$.

Définition 73 (Paire critique).
Soit $\ell \to r$ une règle de R, q un état de $\mathcal{A}$ et σ une substitution de $\mathrm{T}(\Sigma, \mathcal{X})$ vers $(\varnothing, Q)$. On dit que le triplet $(\ell \to r, \sigma, q)$ est une paire critique si

1. $\ell\sigma \rightarrowtail^*_{\mathcal{A}} q$ et
2. $r\sigma \not\rightarrowtail^*_{\mathcal{A}} q$. 73 ◀

L'existence d'une paire critique signe donc un défaut de compatibilité entre le système de réécriture R et l'automate $\mathcal{A}$; la complétion de ces paires critiques vise à résorber ce défaut. L'invariant suivant, qui sera vérifié lorsque le système R est linéaire à gauche, exprime le fait que la notion de paire critique capture totalement le cas où le langage $\mathscr{L}(\mathcal{A}, q)$ d'un état q n'est pas clos par R, c'est-à-dire que $R(\mathscr{L}(\mathcal{A}, q)) \not\subseteq \mathscr{L}(\mathcal{A}, q)$.

Invariant 74.
Pour tout état q de $\mathcal{A}$, pour tout $u \in \mathscr{L}(\mathcal{A}, q)$, pour tout $v \in R(\{u\})$, ou bien $v \in \mathscr{L}(\mathcal{A}, q)$, ou bien il existe une paire critique $(\ell \to r, \sigma, q_0)$ dans $\mathcal{A}$ et un contexte C sur $\mathrm{T}(\Sigma)$ tels que $u \rightarrowtail^*_{\mathcal{A}} C[\ell\sigma] \rightarrowtail^*_{\mathcal{A}} C[q_0] \rightarrowtail^*_{\mathcal{A}} q$ et $v \rightarrowtail^*_{\mathcal{A}} C[r\sigma]$. 74 ◀

La complétion d'une paire critique (figure 4.2) consiste à ajouter des transitions de $r\sigma$ vers q. Lorsque $r\sigma$ n'est pas réduit à un état (ce qui se produit lorsque r n'est pas une variable), il est possible que nous devions normaliser cette configuration, comme expliqué en section 4.3.1.1. Ce processus, que nous allons décrire dans quelques lignes, introduit éventuellement des nouveaux états. Il réutilise ou introduit de nouvelles delta-transitions.

On aboutit donc toujours à une situation où $r\sigma \rightarrowtail^* q'$, avec $q' = r\sigma$ lorsque r est une variable, et q' un état éventuellement nouveau dans le cas contraire. Dans tous

les cas, $q' \neq q$ sinon nous n'aurions pas de paire critique. Il reste donc à ajouter une epsilon-transition de q' vers q. Afin d'en garder la trace et de les distinguer de celles qui seront ajoutées par application des équations, ces epsilon-transitions porteront la couleur $\mathfrak{K}$.

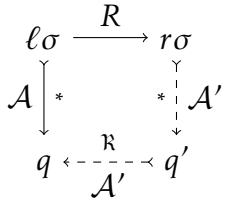

Figure 4.2 – Complétion d'une paire critique

Revenons maintenant à la normalisation et rappelons plus précisément ses objectifs avant d'en donner une définition formelle. La normalisation de c dans $\mathcal{A}$, notée $\mathrm{Norm}_{\mathcal{A}}(c)$, est constituée d'un ensemble de transitions nouvelles N_Δ, d'un ensemble d'états nouveaux N_Q et d'un état q' tels que $q' \in N_Q$ et dans l'automate $\mathcal{A}'$ enrichi par ces nouveaux éléments, $c \rightarrowtail^*_{\mathcal{A}'} q'$.

Définition 75 (Normalisation).
Soit $\mathcal{A} = (\Sigma, Q, \Delta, F)$ un automate et c une configuration de cet automate. On définit $\mathrm{Norm}_{\mathcal{A}}(c)$ par induction.

Si c est un état, alors $\mathrm{Norm}_{\mathcal{A}}(c) = (\varnothing, \varnothing, c)$.

Si c est une configuration élémentaire, c'est-à-dire de la forme $f(q_1, \ldots, q_k)$ avec k un entier naturel, $f \in \Sigma_k$ et $q_1, \ldots, q_k \in Q$, alors :
— s'il existe $q' \in Q$ tel que $f(q_1, \ldots, q_k) \rightarrowtail_{\mathcal{A}} q'$, alors $\mathrm{Norm}_{\mathcal{A}}(c) = (\varnothing, \varnothing, q')$;
— sinon, prenant $q' \notin Q$ arbitraire, $\mathrm{Norm}_{\mathcal{A}}(c) = (\{q'\}, \{f(q_1, \ldots, q_k) \rightarrowtail_{\mathcal{A}} q'\}, q')$.

Si $c = f(c_1, \ldots, c_k)$ avec k un entier naturel, $f \in \Sigma_k$ et $c_1, \ldots, c_k$ des configurations, alors on note $(Q_i)_{i \in [\![1;k+1]\!]}$, $(\Delta_i)_{i \in [\![1;k+1]\!]}$, $(\mathcal{A}_i)_{i \in [\![1;k+1]\!]}$, $(Q'_i)_{i \in [\![1;k+1]\!]}$, $(\Delta'_i)_{i \in [\![1;k+1]\!]}$ et $(q'_i)_{i \in [\![1;k+1]\!]}$ les suites ainsi construites :
— $Q_1 = Q$ et $\Delta_1 = \Delta$,
— pour tout $i \in [\![1\,;k]\!]$, $Q_{i+1} = Q_i \cup Q'_i$ et $\Delta_{i+1} = \Delta_i \cup \Delta'_i$,
— pour tout $i \in [\![1\,;k+1]\!]$, $\mathcal{A}_i = (\Sigma, Q_i, \Delta_i, F)$,
— pour tout $i \in [\![1\,;k]\!]$, $(Q'_i, \Delta'_i, q'_i) = \mathrm{Norm}_{\mathcal{A}_i}(c_i)$,
— $(Q'_{k+1}, \Delta'_{k+1}, q'_{k+1}) = \mathrm{Norm}_{\mathcal{A}_{k+1}}\left(f(q'_1, \ldots, q'_k)\right)$.
On définit $\mathrm{Norm}_{\mathcal{A}}(c) = \left(\bigcup_{i=1}^{k+1} Q'_i, \bigcup_{i=1}^{k+1} \Delta'_i, q'_{k+1}\right)$. 75 ◀

Remarque 76. Dans la définition précédente, la notation $\mathcal{A}$ dans $\mathrm{Norm}_{\mathcal{A}}(c)$ est générique. Dans le processus de complétion d'un automate $\mathcal{A}$, les normalisations sont faites dans l'automate $\mathcal{A}^{\mathfrak{K}}$.

Définition 77 (Complétion d'une paire critique).
Soit $pc = (\ell \to r, \sigma, q)$ une paire critique pour l'automate $\mathcal{A}$ et le système de réécriture R. Notant $(Q', \Delta', q') = \mathrm{Norm}_{\mathcal{A}^R}(r\sigma)$, la complétion de cette paire critique est $\mathrm{Comp}_{\mathcal{A}}(pc) = \left(Q', \Delta', \left\{q' \overset{R}{\rightarrowtail} q\right\}\right)$. 77 ◀

Définition 78 (Étape de complétion).
Soit $\mathcal{A}_n = (\Sigma, Q_n, \Delta_n, F)$ un automate. Soit R un système de réécriture. Une étape de complétion de $\mathcal{A}_n$ par R consiste en la construction d'un automate $\mathcal{A}_{n+1} = (\Sigma, Q_{n+1}, \Delta_{n+1}, F)$ par les opérations suivantes.

Soit PC l'ensemble des paires critiques de $\mathcal{A}_n$. Pour tout $pc \in PC$, notons $(Q'_{pc}, \Delta'_{pc}, \Delta^R_{pc}) = \mathrm{Comp}_{\mathcal{A}_n}(pc)$. On définit $Q_{n+1} = Q \cup \bigcup_{pc \in PC} Q'_{pc}$ et $\Delta_{n+1} = \Delta \cup \bigcup_{pc \in PC} \Delta'_{pc} \cup \Delta^R_{pc}$.

Exemple 79.
Reprenons notre exemple 45. On rappelle le système de réécriture R

$$\begin{aligned}
\mathit{filter}(\mathit{nil}) &\to \mathit{nil},\\
\mathit{filter}(\mathit{cons}(\mathit{succ}(X), Y)) &\to \mathit{cons}(\mathit{succ}(X), \mathit{filter}(Y)),\\
\mathit{filter}(\mathit{cons}(\mathit{pred}(X), Y)) &\to \mathit{cons}(\mathit{pred}(X), \mathit{filter}(Y)),\\
\mathit{filter}(\mathit{cons}(0, Y)) &\to \mathit{filter}(Y),\\
\mathit{pred}(\mathit{succ}(X)) &\to X,\\
\mathit{succ}(\mathit{pred}(X)) &\to X
\end{aligned}$$

et les transitions de l'automate $\mathcal{A}_0$

$$\begin{aligned}
\mathit{nil} &\rightarrowtail q_n\\
0 &\rightarrowtail q_0\\
\mathit{succ}(q_0) &\rightarrowtail q_s\\
\mathit{pred}(q_s) &\rightarrowtail q_p\\
\mathit{cons}(q_p, q_n) &\rightarrowtail q_c\\
\mathit{filter}(q_c) &\rightarrowtail q_f.
\end{aligned}$$

On a

$$\mathscr{L}(\mathcal{A}_0, q_f) = \{\mathit{filter}(\mathit{cons}(\mathit{pred}(\mathit{succ}(0)), \mathit{nil}))\}$$

et

$$R(\mathscr{L}(\mathcal{A}_0, q_f)) = \{\mathit{filter}(\mathit{cons}(0, \mathit{nil})), \quad \mathit{cons}(\mathit{pred}(\mathit{succ}(0)), \mathit{filter}(\mathit{nil}))\}.$$

Il y a une paire critique PC_1 dans $\mathcal{A}_0$ avec la règle

$$\mathit{filter}(\mathit{cons}(\mathit{pred}(X), Y)) \to \mathit{cons}(\mathit{pred}(X), \mathit{filter}(Y)),$$

la substitution

$$\sigma_1 = \{X \mapsto q_s, Y \mapsto q_n\}$$

et l'état q_f. Elle est résolue par l'ajout de transitions pour reconnaitre la configuration correspondant au membre droit $cons(pred(q_s),filter(q_n))$ en l'état q_f. La normalisation de cette configuration va réutiliser la transition $pred(q_s) \rightarrowtail q_p$. Elle doit ensuite créer un nouvel état q_{N1} tel que $filter(q_n) \rightarrowtail q_{N1}$, puis un nouvel état q_{N2} tel que $cons(q_p, q_{N1}) \rightarrowtail q_{N2}$. On ajoute ensuite $q_{N2} \overset{\mathfrak{R}}{\rightarrowtail} q_f$. De ces opérations résulte $\mathrm{Comp}_{\mathcal{A}_0}(PC_1)$.

Il y a une autre paire critique PC_2 dans $\mathcal{A}_0$ avec la règle

$$pred(succ(X)) \to X,$$

la substitution

$$\sigma_2 = \{X \mapsto q_0\}$$

et l'état q_p. On la résout en ajoutant la transition $q_0 \overset{\mathfrak{R}}{\rightarrowtail} q_p$, ce qui correspond à $\mathrm{Comp}_{\mathcal{A}_0}(PC_2)$.

Il n'y a pas d'autre paire critique dans $\mathcal{A}_0$, de sorte que la première étape de complétion exacte est terminée et $\mathcal{A}_1$ est construit.

La normalisation effectuée a donné naissance à une nouvelle paire critique dans $\mathcal{A}_1$ avec $f(n) \to n$, la substitution vide et l'état q_{N1}. 79 ◁

L'énumération des paires critiques d'un automate est un problème algorithmique non trivial. Nous passons sous silence cette difficulté qui n'a pas de conséquence sur les théorèmes de la présente partie et en dirons davantage dans le chapitre 6 relatif à l'outil Timbuk.

En notant $\mathcal{A}_0$ un automate reconnaissant le langage L_0 dont on souhaite étudier les descendants par R, on peut donc construire par ce procédé une suite d'automates $(\mathcal{A}_n)$. Nous allons montrer que si R est linéaire à gauche, alors chaque automate de cette suite vérifie l'invariant 74. Dans le cas où le processus de complétion s'arrête par absence de paire critique dans un certain automate $\mathcal{A}_N$, nous en déduirons que l'automate $\mathcal{A}_N$ reconnait une surapproximation de $R^*(L_0)$.

La linéarité est cruciale pour pouvoir intercaler un $\ell\sigma$ dans un chemin de reconnaissance $\ell\mu \rightarrowtail^* q$.

Théorème 80 (Théorème d'interpolation).
Soit $\ell \in \mathrm{T}(\Sigma, \mathcal{X})$ un terme linéaire. Soit μ une substitution de $(\Sigma, \mathcal{X})$ vers Σ. Soit $\mathcal{A} = (\Sigma, Q, \Delta, F)$ un automate et q un état de $\mathcal{A}$ tel que $\ell\mu \rightarrowtail^*_{\mathcal{A}} q$. Il existe une substitution σ de $(\Sigma, \mathcal{X})$ vers $(\varnothing, Q)$ telle que $\ell\mu \rightarrowtail^*_{\mathcal{A}} \ell\sigma \rightarrowtail^*_{\mathcal{A}} q$. 80 ■

Démonstration.
Soit $X \subseteq \mathcal{X}$ l'ensemble des variables qui apparaissent dans ℓ. Comme ℓ est linéaire, pour tout $x \in X$, il existe une *unique* position ξ_x telle que $\ell_{|\xi_x} = x$.

On peut toujours réorganiser un chemin de reconnaissance d'un automate en échangeant des applications de transitions dans des sous-arbres parallèles. Ainsi, il existe des états q_{ξ_x}, $x \in X$ tels que $\ell\mu \rightarrowtail^*_{\mathcal{A}} (\ell\mu)\left[q_{\xi_x}, x \in X\right]_{\xi_x, x\in X} \rightarrowtail^*_{\mathcal{A}} q$. Par définition des positions ξ_x, $x \in X$, on a $(\ell\mu)\left[q_{\xi_x}, x \in X\right]_{\xi_x, x\in X} = \ell\left[q_{\xi_x}, x \in X\right]_{\xi_x, x\in X}$. De plus, pour tout $x \in X$, on a $x\mu \rightarrowtail^*_{\mathcal{A}} q_{\xi_x}$.

L'unicité de la position ξ_x pour chaque $x \in X$ associe à chaque variable un unique état q_{ξ_x}, ce qui permet de définir une substitution σ de $(\Sigma, \mathcal{X})$ vers $(\varnothing, Q)$ qui à tout $x \in X$ associe q_{ξ_x}. Par construction, on a $\ell\left[q_{\xi_x}, x \in X\right]_{\xi_x, x\in X} = \ell\sigma$.

Par conséquent, $\ell\mu \rightarrowtail^*_{\mathcal{A}} \ell\sigma \rightarrowtail^*_{\mathcal{A}} q$. □

Exemple 81 (Nécessité de la linéarité pour l'interpolation).
Soit R le système de réécriture constitué de la seule règle $f(x,x) \to g(x)$. Soit a une constante. Soit $\mathcal{A}$ un automate disposant des transitions $a \rightarrowtail q_1$, $a \rightarrowtail q_2$ et $f(q_1, q_2) \rightarrowtail q_f$.

On a $f(a,a) \to_R g(a)$ par la substitution $\mu : x \mapsto a$, mais il n'existe pas de substitution σ de $(\Sigma, \mathcal{X})$ vers $(\varnothing, Q)$ telle que $f(x,x)\sigma \rightarrowtail^*_{\mathcal{A}} q_f$. En effet, pour σ, il faut fixer à x la valeur q_1 ou q_2, ce qui donne à chaque fois une configuration où l'un de ces deux états apparait deux fois. Or seul $f(q_1, q_2)$ est reconnu en q_f.

Le théorème d'interpolation, dont l'hypothèse de linéarité n'est pas vérifiée, voit sa conclusion mise en défaut, et il en résulte que l'invariant 74 n'est pas vérifié par $\mathcal{A}$ et R. 81 ◁

Remarque 82. L'interpolation reste possible avec des termes non-linéaires si l'automate étudié est déterministe. Mais déterminiser $\mathcal{A}$ à chaque étape de complétion serait beaucoup trop couteux, et de fait $\mathcal{A}$ ne sera pas déterministe dès la deuxième étape de complétion.

Imposer au système de réécriture R d'être linéaire à gauche est en revanche très raisonnable : la définition d'une fonction de plusieurs variables se fait avec des variables deux-à-deux distinctes.

Lemme 83.
Soit R un système de réécriture linéaire à gauche. Soit $\mathcal{A}_0$ un automate. Soit $\mathcal{A}$ un automate obtenu à partir de $\mathcal{A}_0$ par application d'un nombre quelconque d'étapes de complétion par R. $\mathcal{A}$ vérifie l'invariant 74. 83 ■

Démonstration.
Soit q un état de $\mathcal{A}$. Soit $u \in \mathscr{L}(\mathcal{A}, q)$ et $v \in R(\{u\})$. On a $u \to_R v$, donc, par définition de la réécriture, il existe une règle $\ell \to r$ de R, un contexte C sur $\mathrm{T}(\Sigma)$ et une substitution μ de $(\Sigma, \mathcal{X})$ vers Σ tels que $u = C[\ell\mu]$ et $v = C[r\mu]$.

On a $u = C[\ell\mu] \rightarrowtail^*_{\mathcal{A}} q$, donc il existe un état q_0 tel que $\ell\mu \rightarrowtail^*_{\mathcal{A}} q_0$ et $C[q_0] \rightarrowtail^*_{\mathcal{A}} q$. Comme ℓ est linéaire, par théorème d'interpolation, il existe une substitution σ de

$(\Sigma, \mathcal{X})$ vers $(\varnothing, Q)$ telle que $\ell\mu \rightarrowtail^*_{\mathcal{A}} \ell\sigma \rightarrowtail^*_{\mathcal{A}} q_0$. Par ailleurs, toute variable de r est aussi une variable de ℓ, donc $r\mu \rightarrowtail^*_{\mathcal{A}} r\sigma$, puis $v \rightarrowtail^*_{\mathcal{A}} C[r\sigma]$.

Si $r\sigma \rightarrowtail^*_{\mathcal{A}} q_0$, on obtient $v \rightarrowtail^*_{\mathcal{A}} q$, soit la première branche de l'alternative de l'invariant 74.

Sinon $(\ell \to r, \sigma, q_0)$ est une paire critique, dont on a montré précédemment qu'elle vérifie les conditions de la deuxième branche. □

Corolaire 84.
Si l'automate $\mathcal{A}$ du lemme précédent est sans paire critique, alors $\mathscr{L}(\mathcal{A})$ est clos par R, c'est-à-dire que $R(\mathscr{L}(\mathcal{A})) \subseteq \mathscr{L}(\mathcal{A})$. 84 ■

Démonstration.
Soit $v \in R(\mathscr{L}(\mathcal{A}))$. Il existe $u \in \mathscr{L}(\mathcal{A})$ tel que $v \in R(\{u\})$. Il existe un état final q tel que $u \in \mathscr{L}(\mathcal{A}, q)$. Par l'invariant 74 et en l'absence de paire critique dans $\mathcal{A}$, $v \in \mathscr{L}(\mathcal{A}, q)$, donc $v \in \mathscr{L}(\mathcal{A})$. □

Théorème 85 (Correction de la complétion exacte).
Soit R un système de réécriture linéaire à gauche. Soit $\mathcal{A}_0$ un automate dont les transitions sont dépourvues de couleurs. On suppose que la complétion exacte de $\mathcal{A}_0$ par R termine, c'est-à-dire qu'elle produit un automate point-fixe $\mathcal{A}_N$. Alors $\mathscr{L}(\mathcal{A}_N) \supseteq R^*(\mathscr{L}(\mathcal{A}_0))$. 85 ■

Démonstration.
Le corollaire 84 est applicable à $\mathcal{A}_N$: $\mathscr{L}(\mathcal{A}_N)$ est clos par R. De plus, la complétion exacte ajoute des transitions sans jamais en retirer, donc fait grossir les langages reconnus : $\mathscr{L}(\mathcal{A}_N) \supseteq \mathscr{L}(\mathcal{A}_0)$. Or $R^*(\mathscr{L}(\mathcal{A}_0))$ est le plus petit langage clos par R et contenant $\mathscr{L}(\mathcal{A}_0)$, d'où le résultat. □

4.3.2.2 Fusion équationnelle

Nous disposons maintenant d'un processus de complétion. Il produit un résultat correct pourvu qu'il aboutisse à un point fixe. Mais la complétion peut à son tour alimenter indéfiniment le processus en nouvelles paires critiques. Cela résulte de l'adjonction de nouveaux états rendue nécessaire par la normalisation, comme nous l'avons constaté dans l'exemple 79.

Si le système R est superficiel à droite, c'est-à-dire que les variables des membres droits des règles ont une profondeur maximale de 1, on n'a pas besoin de normalisation (sauf éventuellement un nombre fini de fois). Il en résulte que la complétion termine toujours sur de tels systèmes, et que les systèmes superficiels à droite et linéaires préservent la régularité, résultat démontré dans un autre cadre par [Sal88] (cité par [GGJ08]).

Cependant, nous ne pouvons nous contenter de traiter des systèmes aussi contraints : on ne peut pas définir des fonctions récursives très intéressantes avec des systèmes

superficiels puisqu'il est impossible de composer des fonctions ou de reconstruire des données dans le membre droit.

Pour favoriser l'obtention d'un point fixe dans le cadre puissant des systèmes linéaires à gauche, nous allons « fusionner » des états. Cette fusion sera paramétrée par un ensemble d'équations E. Il aura pour conséquence, en général, une perte de précision sur le résultat. Nous allons décrire la fusion équationnelle et donner une quantification de la précision de la surapproximation.

Pour fusionner des états, nous allons appliquer des équations. De même qu'une paire critique dénote une incompatibilité entre l'automate et le système de réécriture considérés, une situation d'application d'une équation trahit une complexité excessive de l'automate au regard d'une équation.

Définition 86 (Situation d'application).
Soit $\mathcal{A}$ un automate, $s = t$ une équation et θ une substitution de $(\Sigma, \mathcal{X})$ vers $(\varnothing, Q)$. Soit q_s, q_t deux états de $\mathcal{A}$. On dit que $(s = t, \theta, q_s, q_t)$ est une situation d'application (de l'équation $s = t$) si

— $s\theta \rightarrowtail^*_{\mathcal{A}/\mathcal{X}} q_s$,
— $t\theta \rightarrowtail^*_{\mathcal{A}/\mathcal{X}} q_t$ et
— il n'est pas le cas que $q_s \overset{E,*}{\rightleftarrows}_{\mathcal{A}} q_t$. 86 ◀

Définition 87 (Application d'une équation).
Soit $\mathcal{A}$ un automate et $(s = t, \theta, q_s, q_t)$ une situation d'application d'une équation. L'application de cette équation est la paire de transitions $\{q_s \overset{E}{\rightarrowtail} q_t, q_t \overset{E}{\rightarrowtail} q_s\}$. 87 ◀

Une étape de fusion équationnelle consiste à énumérer toutes les situations d'applications et à ajouter les transitions correspondantes. Contrairement à la définition d'une étape de complétion, nous choisissons de considérer immédiatement les nouvelles situations d'application qui pourraient résulter de ces ajouts : comme nous n'ajoutons aucun état lors de la fusion équationnelle, il est certain qu'après un certain nombre d'itérations, il n'y aura plus aucune situation d'application.

Définition 88 (Étape de fusion équationnelle).
Soit $\mathcal{A} = (\Sigma, Q, \Delta, F)$ un automate et E un ensemble d'équations. Une étape de fusion équationnelle de $\mathcal{A}$ par E consiste à construire un automate $\mathcal{A}' = (\Sigma, Q, \Delta', F)$ avec Δ' défini comme suit.

Soit $(\Delta_n)_{n\in\mathbb{N}}$, $(\mathcal{A}_n)_{n\in\mathbb{N}}$ et $(S_n)_{n\in\mathbb{N}}$ les suites ainsi définies :
— $\Delta_0 = \Delta$,
— pour tout n entier naturel, $\mathcal{A}_n = (\Sigma, Q, \Delta_n, F)$,
— pour tout n entier naturel, S_n est l'ensemble des situations d'applications d'équations de E dans $\mathcal{A}_n$,
— pour tout n entier naturel, Δ_{n+1} est la réunion de Δ_n et de toutes les transitions ajoutées par application des situations de S_n.

La suite $(\Delta_n)_{n\in\mathbb{N}}$ est stationnaire, c'est-à-dire constante à partir d'un rang N. On définit $\Delta' = \Delta_N$. 88 ◀

Définition 89 (Étape de complétion équationnelle).
Une étape de complétion équationnelle est une étape de complétion exacte suivie d'une étape de fusion équationnelle. 89 ◀

Invariant 90.
L'automate $\mathcal{A}^{\not K}$ est accessible et E-cohérent. De plus, pour tout état q, $\mathscr{L}(\mathcal{A}, q) \subseteq (R/E)^*\left([q]_E^{\mathcal{A}^{\not K}}\right)$. 90 ◀

Lemme 91.
Un automate produit par complétion équationnelle vérifie l'invariant 74. 91 ■

Démonstration.
La démonstration du lemme 83 ne faisant pas intervenir le procédé de construction de l'automate, elle reste encore valable. □

Lemme 92.
La fusion équationnelle préserve l'invariant 90. 92 ■

Démonstration.
Soit $\mathcal{A}$ un automate vérifiant l'invariant et $(s = t, \theta, q_s, q_t)$ une situation d'application. Soit $\mathcal{B}$ l'automate obtenu en ajoutant à $\mathcal{A}$ les transitions $q_s \overset{E}{\longmapsto} q_t$ et $q_t \overset{E}{\longmapsto} q_s$. L'accessibilité de $\mathcal{B}^{\not K}$ découle de celle de $\mathcal{A}^{\not K}$: on n'a pas ajouté de nouvel état. Nous allons montrer que pour tout état q, les termes de $\mathscr{L}(\mathcal{B}^{\not K}, q)$ ont une classe commune par E, qui sera bien sûr la même que pour $\mathscr{L}(\mathcal{A}^{\not K}, q)$, soit $[q]_E^{\mathcal{A}^{\not K}}$.

Remarquons qu'il résulte de l'existence de la situation d'application que $[q_s]_E^{\mathcal{A}^{\not K}} = [q_t]_E^{\mathcal{A}^{\not K}}$. En effet, prenant $s\mu$ et $t\mu$ deux termes reconnus respectivement par $\mathcal{A}^{\not K}$ en $s\theta$ et $t\theta$ (il en existe car cet automate est accessible), on a $s\mu \equiv_E t\mu$ par définition de $\equiv_E$. Nous notons $\varkappa_0$ cette classe commune.

Soit q un état et soit $u \in \mathscr{L}(\mathcal{B}^{\not K}, q)$. Si $u \in \mathscr{L}(\mathcal{A}^{\not K}, q)$, alors u est dans la classe $[q]_E^{\mathcal{A}^{\not K}}$ par définition. Sinon, les chemins de reconnaissance de u utilisent au moins une des deux transitions ajoutées. Pour simplifier le discours, nous supposerons disposer d'un chemin utilisant une seule fois la seule transition $q_s \overset{E}{\longmapsto} q_t$; la généralisation se fait par des considérations de symétrie et par récurrence sur le nombre minimal de transitions nouvelles utilisées.

Il existe ainsi un contexte C sur $\mathrm{T}(\Sigma)$ et un terme u_0 tels que $u = C[u_0]$, $u_0 \overset{*}{\longmapsto}_{\mathcal{A}^{\not K}} q_s$ et $C[q_t] \overset{*}{\longmapsto}_{\mathcal{A}^{\not K}} q$. On déduit de la E-cohérence de $\mathcal{A}^{\not K}$ que, d'une part, u_0 est dans la classe $[q_s]_E^{\mathcal{A}^{\not K}} = \varkappa_0$ et, d'autre part, que la classe de la configuration $C[q_t]$, notée $\varkappa$,

est $[q]_E^{\mathcal{A}^{\not\perp}}$. Ainsi, par congruence, u est dans la classe $\varkappa$. Finalement, u est dans la classe $[q]_E^{\mathcal{A}^{\not\perp}}$.

Ainsi est établie la E-cohérence de $\mathcal{B}^{\not\perp}$; de plus, pour tout état q, $[q]_E^{\mathcal{B}^{\not\perp}} = [q]_E^{\mathcal{A}^{\not\perp}}$. Nous allons montrer la seconde propriété de l'invariant par récurrence sur le nombre b de transitions nouvelles utilisées. À chaque étape, nous ne traiterons que le cas de la transition $q_s \overset{\mathrm{E}}{\rightarrowtail} q_t$, la généralisation se faisant par symétrie.

Soit donc un état q et un terme $u \in \mathscr{L}(\mathcal{B}, q)$, la reconnaissance se faisant sans utiliser les nouvelles transitions. Alors $u \in \mathscr{L}(\mathcal{A}, q)$, donc, par l'invariant sur $\mathcal{A}$, $u \in (R/E)^*\left([q]_E^{\mathcal{A}^{\not\perp}}\right)$.

Considérons le cas d'un état q et un terme $u \in \mathscr{L}(\mathcal{B}, q)$ avec b utilisations des nouvelles transitions : il existe un terme u_0 et un contexte C tels que $u = C[u_0]$, $u_0 \rightarrowtail_{\mathcal{B}}^* q_s$ et $C[q_t] \rightarrowtail_{\mathcal{B}}^* q$, avec à chaque fois strictement moins de b nouvelles transitions. Nous notons $\varkappa$ la classe de la configuration $C[q_s]$, de sorte que $\varkappa = C[\varkappa_0]$. Par hypothèse de récurrence, $u_0 \in (R/E)^*(\varkappa_0)$ et, notant v_0 un terme quelconque de $\mathscr{L}(\mathcal{A}^{\not\perp}, q_t)$ et $v = C[v_0]$, $v \in (R/E)^*\left([q]_E^{\mathcal{A}^{\not\perp}}\right)$. Ainsi, $\varkappa$ étant la classe commune de $C[q_s]$, $C[q_t]$ et v, on a $[q]_E^{\mathcal{A}^{\not\perp}} \rightarrow_{R/E}^* \varkappa \rightarrow_{R/E}^* [u]_E$, c'est-à-dire $u \in (R/E)^*\left([q]_E^{\mathcal{A}^{\not\perp}}\right)$.

Par récurrence, pour tout état q, $\mathscr{L}(\mathcal{B}, q) \subseteq (R/E)^*\left([q]_E^{\mathcal{A}^{\not\perp}}\right)$. Finalement, $\mathcal{B}$ vérifie l'invariant 90. □

Lemme 93.
Soit $(\ell \to r, \sigma, q)$ une paire critique dans un automate $\mathcal{A}$ vérifiant l'invariant 90. On a $[r\sigma]_E^{\mathcal{A}^{\not\perp}} \subseteq [\mathscr{L}(\mathcal{A}, r\sigma)]_E \subseteq (R/E)^*\left([q]_E^{\mathcal{A}^{\not\perp}}\right)$. 93 ■

Démonstration.
Par définition, $[r\sigma]_E^{\mathcal{A}^{\not\perp}} = \left[\mathscr{L}(\mathcal{A}^{\not\perp}, r\sigma)\right]_E$, ce qui établit la première inclusion.

Pour montrer la seconde inclusion, il suffit en fait d'établir $\mathscr{L}(\mathcal{A}, r\sigma) \subseteq (R/E)^*\left([q]_E^{\mathcal{A}^{\not\perp}}\right)$. Soit donc $u \in \mathscr{L}(\mathcal{A}, r\sigma)$: il existe une substitution μ de $(\Sigma, \mathcal{X})$ vers Σ telle que $u = r\mu$ et pour toute variable x de r, $x\mu \rightarrowtail_{\mathcal{A}}^* x\sigma$. Pour toute variable y apparaissant dans ℓ mais non dans r, on étend μ en choisissant un terme arbitraire $y\mu \in \mathscr{L}(\mathcal{A}, y\sigma)$. Ainsi, par définition de la réécriture, $\ell\mu \rightarrow_R r\mu$, c'est-à-dire que $u \in R(\{\ell\mu\})$. Or, par l'invariant 90, $\mathscr{L}(\mathcal{A}, q) \subseteq (R/E)^*\left([q]_E^{\mathcal{A}^{\not\perp}}\right)$, donc $u \in (R/E)^*\left([q]_E^{\mathcal{A}^{\not\perp}}\right)$. □

Lemme 94.
Soit R un système de réécriture linéaire à gauche et E un ensemble d'équations. Soit $\mathcal{A}_0$ un automate accessible et E-cohérent dont les transitions sont dépourvues de couleurs. Soit $\mathcal{A}$ un automate produit à partir de $\mathcal{A}_0$ par complétion équationnelle par R et E. Alors $\mathcal{A}$ vérifie l'invariant 90. 94 ■

Démonstration.
Nous allons procéder par induction. Comme $\mathcal{A}_0$ ne possède pas de $\mathfrak{K}$-transition, l'invariant se réduit aux hypothèses d'accessibilité et de E-cohérence.

Supposons qu'après un certain nombre d'étapes, l'automate courant $\mathcal{A}$ vérifie l'invariant et montrons que cette propriété est conservée par l'étape suivante de complétion équationnelle.

La normalisation préserve la E-cohérence de $\mathcal{A}^{\not\mathfrak{K}}$. En effet, les seules nouvelles transitions ajoutées le sont vers des nouveaux états, qui reconnaissent des termes uniquement par la configuration qui a motivé leur création.

Considérons maintenant l'adjonction d'une transition $q' \xrightarrow{\mathfrak{K}} q$ complétant une paire critique $(\ell \to r, \sigma, q)$. Notons $\mathcal{B}$ l'automate ainsi obtenu et montrons qu'il vérifie l'invariant. Les propriétés portant sur $\mathcal{B}^{\not\mathfrak{K}}$ ne sont pas en cause car cet automate est égal à $\mathcal{B}^{\not\mathfrak{K}}$. Il reste à prouver que $\mathscr{L}(\mathcal{B}, q) \subseteq (R/E)^*\left([q]_E^{\mathcal{A}^{\not\mathfrak{K}}}\right)$. Comme $\mathscr{L}(\mathcal{B}, q) = \mathscr{L}(\mathcal{A}, q) \cup \mathscr{L}(\mathcal{A}, q')$, il suffit de vérifier que $\mathscr{L}(\mathcal{A}, q') \subseteq (R/E)^*\left([q]_E^{\mathcal{A}^{\not\mathfrak{K}}}\right)$. On a, par l'invariant sur $\mathcal{A}$, $\mathscr{L}(\mathcal{A}, q') \subseteq (R/E)^*\left([q']_E^{\mathcal{A}^{\not\mathfrak{K}}}\right)$. Mais par construction de q' par normalisation, $[q']_E^{\mathcal{A}^{\not\mathfrak{K}}} = [r\sigma]_E^{\mathcal{A}^{\not\mathfrak{K}}}$. Et par le lemme 93, cet ensemble est inclus dans $(R/E)^*\left([q]_E^{\mathcal{A}^{\not\mathfrak{K}}}\right)$. □

Théorème 95 (Précision).
Soit E un ensemble d'équations. Soit $\mathcal{A}_0$ un automate accessible dont on a fixé l'ensemble des états finaux. Supposons que $\mathcal{A}_0$ est E-cohérent. Soit R un TRS linéaire à gauche. Soit $\mathcal{A}$ obtenu à partir de $\mathcal{A}_0$ après un certain nombre d'étapes de complétion équationnelle. On a

$$\mathscr{L}(\mathcal{A}) \subseteq (R_{\text{in}}/E)^*(\mathscr{L}(\mathcal{A}_0)).$$ 95 ■

Démonstration.
Comme $\mathcal{A}_0$ ne contient pas de $\mathfrak{K}$-transition, $\mathcal{A}_0^{\not\mathfrak{K}} = \mathcal{A}_0$. Ainsi, pour $\mathcal{A}_0$, la troisième partie de l'invariant 90 découle des deux premières, lesquelles sont vérifiées par hypothèse. Le lemme précédent assure donc que l'invariant 90 est encore vérifié par $\mathcal{A}$. En considérant la troisième partie de cet invariant et en l'appliquant pour les états finaux de $\mathcal{A}$ (qui sont ceux de $\mathcal{A}_0$), on obtient le résultat. □

5 Gérer les stratégies

Nous présentons ici les éléments théoriques de notre principale contribution, ses éléments pratiques se trouvant dans le chapitre 6.

5.1 Introduction

5.1.1 Stratégies dans les langages de programmation

La sémantique de OCaml [Ler+14] impose l'ordre dans lequel doivent être évaluées les sous-expressions d'une expression complexe.

```
let plusplus i =
  let res = !i in
  incr i;
  res

let print_ij i j =
  print_string ("1er arg = " ^ (string_of_int i) ^ "\n");
  print_string ("2e  arg = " ^ (string_of_int j) ^ "\n")

let test _ =
  let i = ref 0 in
  print_ij (plusplus i) (plusplus i)
```

Figure 5.1 – Illustration de l'évaluation de droite à gauche de OCaml

Ainsi, avec le programme de la figure 5.1, un appel à la fonction `test` provoquera l'affichage des deux lignes

```
1er arg = 1
2e  arg = 0
```

car l'argument le plus à droite de l'appel à `print_ij` est évalué d'abord, ce qui par effet de bord provoque une première incrémentation de la référence `i`, puis l'argument de gauche est évalué, avec cette même référence qui a subi une première incrémentation. D'autres langages, comme C++, n'imposent pas d'ordre d'évaluation

```
#include <iostream>
using namespace std;

void print_ij(int i, int j) {
  cout << "1er arg = " << i << endl;
  cout << "2e  arg = " << j << endl;
}

void test(void) {
  int i = 0;
  print_ij(i++, i++);
}
```

Figure 5.2 – Correspondant en C++ du programme 5.1

des arguments d'une fonction. C'est pourquoi dans le programme de la figure 5.2, un appel à `test` peut afficher, selon les choix faits par le compilateur, ou bien

```
1er arg = 1
2e  arg = 0
```

ou bien

```
1er arg = 0
2e  arg = 1
```

selon que les arguments ont été évalués de droite à gauche ou de gauche à droite. La seule garantie apportée par la spécification de C++ est qu'au début de l'exécution du corps de la fonction `print_ij`, tous ses paramètres effectifs ont été évalués et les valeurs liées aux paramètres formels.

Il en résulte qu'une analyse de programmes C++ indépendante des compilateurs doit considérer comme possibles ces deux situations, tandis qu'une analyse de programmes OCaml peut tirer parti de la stratégie d'évaluation imposée pour donner des résultats plus précis.

Dans les deux exemples ci-dessus, les expressions constituant les paramètres effectifs de la fonction `print_ij` ont été évalués, puis le corps de `test_ij` a été exécuté avec les valeurs ainsi calculées. Cette stratégie, dite appel par valeur ou évaluation en profondeur d'abord, est systématique pour C++, OCaml et de nombreux langages. Elle n'est cependant pas universelle. Par exemple, le langage Haskell [Jon+14] utilise une stratégie d'appel par nom.

Considérons les programmes suivants, rédigés en OCaml en figure 5.3 et en Haskell en figure 5.4. La fonction principale, `suite`, vise à calculer, de façon baroque,

le x^e terme de la suite définie par

$$u_0 = 0,$$
$$u_x = u_{x-1} + x.$$

La fonction `sumList` permet d'énumérer tous les termes de cette liste et la fonction `nth` permet d'extraire le terme voulu.

```
let rec sumList x y =
  (x+y) :: sumList (x+y) (y+1)

let rec nth i (h::t) =
   if i = 0 then h else nth (i-1) t

let suite x =
   nth x (sumList 0 0)
```

FIGURE 5.3 – Tentative de calcul de u en OCaml

```
sumList x y =
  (x+y) : sumList (x+y) (y+1)

nth i (h:t) =
  if i = 0 then h else nth (i-1) t

suite x =
  nth x (sumList 0 0)
```

FIGURE 5.4 – Calcul de u en Haskell

Bien entendu, la version OCaml ne se comporte pas comme on le voudrait : l'appel `suite 1` ne termine pas (il est interrompu en raison du débordement de la pile de récursion), tout simplement parce que l'appel `sumList 0 0` ne termine pas (ni pour aucune autre valeur des paramètres, du reste) et que OCaml doit d'abord calculer la valeur de cet appel pour ensuite exécuter la fonction `nth`. En revanche, la version Haskell est correcte : l'appel `suite 2` ne provoque le calcul des termes de la liste que dans la mesure où on en a besoin. L'appel `sumList 0 0` ne termine pas non plus en Haskell, mais le comportement d'un interpréteur Haskell est très différent de celui d'un toplevel OCaml : OCaml ne renvoie aucune valeur, faute d'avoir pu trouver la fin de la liste, tandis que Haskell produit effectivement et indéfiniment des termes de la liste.

Ces deux programmes se traduisent naturellement en le système de réécriture de la figure 5.5. En réécriture générale (sans stratégie), ce système a des réductions

infinies à partir de $suite(s(s(0)))$, mais ce terme admet aussi une forme normale. Il constitue donc une surapproximation du comportement du programme OCaml et du programme Haskell, et toute analyse de ce système qui ne ferait pas usage de stratégies de réécritures mimant l'un de ces langages serait condamnée à ne pouvoir établir que des propriétés également vérifiées par l'autre.

Étant donné un système de réécriture, il est possible [Plo75] de construire un système de réécriture qui, sans stratégie, a le comportement du premier avec une stratégie donnée. On peut ainsi transformer le système 5.5 pour obtenir un système qui a, sans stratégie, le comportement du programme OCaml 5.3 ou du programme Haskell 5.4.

Cependant, ces transformations tendent à alourdir le système. Ainsi, l'analyse du bytecode Java par complétion d'automate dans [Boi+07] a requis une telle transformation, et la lourdeur des termes représentant l'état de la machine virtuelle Java a encore été augmentée.

Dans [CLM15] est proposée une méthode générale pour traduire des stratégies de réécriture en des systèmes de réécriture générale. Comme les stratégies que l'on considère sont décidables, elles peuvent bien sûr être représentées par des systèmes de réécritures. Les travaux des auteurs de [CLM15] ont pour avantage de fournir une traduction explicite et systématique pour une grande classe de stratégies.

Dans un premier temps, les auteurs s'appuient sur un langage de description de stratégies qui présente l'intérêt de considérer l'application (une fois) d'une règle de réécriture (générale) comme une forme élémentaire de stratégie. Ce langage est doté d'un symbole spécial pour dénoter l'impossibilité d'appliquer une stratégie donnée à un terme donné. Ainsi, la stratégie qui correspond à la règle classique générale $\ell \to r$ échoue pour tous les termes qui ne sont pas de la forme $\ell\sigma$ avec σ une substitution ; elle produit alors le symbole spécial dénotant l'échec. Le langage de stratégie dispose également d'opérateurs qui permettent de combiner des stratégies en spécifiant lesquelles doivent être essayées en premier.

Ainsi, le comportement usuel d'un système de réécriture R se décrit comme : essayer d'appliquer la première règle du système ; si cela échoue, essayer la deuxième, etc. ; si toutes échouent, ne rien faire. Le comportement de R^* est obtenu en exprimant la répétition de la stratégie précédente.

La stratégie en profondeur pour R se décrit récursivement : appliquer d'abord la stratégie sur tous les sous-termes, puis essayer d'appliquer toutes les règles de R sur le terme complet en résultant, puis recommencer.

Tout ceci permet de compiler le langage des stratégies vers les systèmes de réécriture classiques, au prix d'une augmentation significative du nombre des règles.

Dans ce chapitre, nous proposons de ne pas transformer le système de réécriture mais au contraire d'adapter la procédure de complétion elle-même. Ces résultats ont été publiés dans [GS15].

$$
\begin{aligned}
sumList(x,y) &\to cons(plus(x,y), sumList(plus(x,y), s(y)))\\
nth(0, cons(h,t)) &\to h\\
nth(s(i), cons(h,t)) &\to nth(i,t)\\
suite(x) &\to nth(x, sumList(0,0))\\
plus(0,x) &\to x\\
plus(s(x),y) &\to s(plus(x,y))
\end{aligned}
$$

FIGURE 5.5 – Un TRS pour calculer u

La relation de réécriture $\to_R$ induite par un système R peut être restreinte pour suivre une stratégie S : on aura simplement $u \to_R^S v$ si et seulement si $u \to_R v$ et ce pas de réécriture vérifie le critère de la stratégie S. Nous donnons maintenant la définition de la stratégie de réécriture en surface (« outermost ») et de la stratégie en profondeur (« innermost »), qui correspondent respectivement à l'appel par nom et à l'appel par valeur.

Définition 96 (Réécriture en surface).
Soit R un système de réécriture et deux termes s et t. On dit que R permet de réécrire s en t selon la stratégie en surface, et on note $s \to_{R_{\text{out}}} t$, si et seulement s'il existe une règle $\ell \to r$ dans R, un contexte C et une substitution μ tels que $s = C[\ell\mu]$ et $t = C[r\mu]$ et pour toute position p strictement au dessus du trou de C, le terme $s_{|p}$ n'est l'instance d'aucune règle de R. 96 ◀

Haskell utilise une stratégie « outermost ».

Définition 97 (Réécriture en profondeur).
Soit R un système de réécriture R et deux termes s et t. On dit que R permet de réécrire s en t selon la stratégie en profondeur, et on note $s \to_{R_{\text{in}}} t$, si et seulement s'il existe une règle $\ell \to r$ dans R, un contexte C et une substitution μ tels que $s = C[\ell\mu]$ et $t = C[r\mu]$ et tout sous-terme strict de $\ell\mu$ est une forme normale pour la réécriture générale par R. 97 ◀

Il peut arriver que plusieurs sous-termes de s vérifient la condition ci-dessus. Dans ce cas, chacun d'eux peut être réduit par la stratégie en profondeur. OCaml utilise une stratégie « rightmost-innermost » : parmi les sous-termes qui peuvent être réduits par la stratégie en profondeur, c'est toujours celui situé le plus à droite qui l'est. Il s'agit donc d'une stratégie encore plus contrainte.

On trouvera plus d'informations sur la réécriture avec stratégies et sur les propriétés des systèmes de réécriture utilisés avec des stratégies dans [Ter03 ; Toy05]. Nous allons nous concentrer sur la stratégie en profondeur : elle qui correspond en

partie à l'évaluation des programmes OCaml et c'est pour cette stratégie que nous proposons une extension de la complétion d'automate d'arbres.

Définition 98.
Pour un langage L et un système de réécriture R, on note $R_{\text{in}}(L) = \left\{t \mid \exists s \in L, s \to_{R_{\text{in}}} t\right\}$ et $R^*_{\text{in}}(L) = \left\{t \mid \exists s \in L, s \to^*_{R_{\text{in}}} t\right\}$. 98 ◀

5.1.2 Stratégie en profondeur

Comme dans le cas de la réécriture générale, l'étude de la préservation de la régularité des langages par la réécriture avec stratégie en profondeur est un axe d'étude qui intéresse les chercheurs. Réty et Vuotto semblent avoir été les premiers à étudier la préservation par réécriture sous stratégie, dans [RV02]. Ils montrent que celle-ci n'est ni une condition nécessaire ni une condition suffisante de la préservation par réécriture générale : un système qui préserve la régularité par réécriture générale peut ne pas la préserver lorsqu'on impose une stratégie ; un système qui ne préserve pas la régularité par réécriture générale peut cependant être préservant pour une stratégie. Ils transposent ensuite les résultats de [Rét99] aux cas de la stratégie en surface et de la stratégie en profondeur, au prix de restrictions supplémentaires. Il reste que ces études de préservation de la régularité se font sous l'hypothèse restrictive de la linéarité à droite.

Exemple 99.
Considérons à nouveau l'exemple 79. La réécriture de $\mathit{filter}(\mathit{cons}(\mathit{pred}(\mathit{succ}(0)), \mathit{nil}))$ en $\mathit{cons}(\mathit{pred}(\mathit{succ}(0)), \mathit{filter}(\mathit{nil}))$ n'est pas conforme à la stratégie en profondeur car $\mathit{pred}(\mathit{succ}(0))$ n'est pas une forme normale. Nous voulons nous abstenir de compléter la paire critique PC_1 de l'exemple 79. 99 ◁

5.2 Complétion d'automate avec stratégie en profondeur

Comment adapter la complétion d'automate à la stratégie en profondeur ? Strictement, une adaptation n'est pas nécessaire sur le plan de la correction. En effet, pour tout système R et tout langage L, $R^*_{\text{in}}(L) \subseteq R^*(L)$, de sorte que tout surapproximation correcte de $R^*(L)$ est aussi une surapproximation correcte de $R^*_{\text{in}}(L)$. Cependant, procéder ainsi est frustrant, car on surapproxime un ensemble, $R^*(L)$, qui contient strictement (la plupart du temps) celui qui nous intéresse, $R^*_{\text{in}}(L)$.

5.2.1 Préliminaires

5.2.1.1 Formes normales et régularité

La complétion équationnelle fonctionne dans le cadre des systèmes de réécriture linéaires à gauche. Cela nous permet de proposer une adaptation de la complétion à l'étude de la réécriture avec stratégie en profondeur. Il nous faut en effet pour cela distinguer les formes normales des formes réductibles, car c'est là le critère discriminant de cette stratégie. Or on dispose justement du théorème suivant [GB85 ; GT95 ; CJ03].

Théorème 100.
Soit R un système de réécriture linéaire à gauche. L'ensemble de ses formes normales, $\textsc{Irr}(R)$, est régulier. Plus précisément, il existe un automate déterministe et complet $\mathcal{IRR}(R)$ disposant d'un état distingué non final p_{red} et tel que $\mathcal{L}(\mathcal{IRR}(R)) = \textsc{Irr}(R)$. 100 ■

Remarque 101. L'état p_{red} est le seul état non final de $\mathcal{IRR}(R)$.

Exemple 102.
Considérons le système R de l'exemple 45, dont on rappelle les membres gauches :

$$\begin{aligned} \mathit{filter}(\mathit{nil}) &\to \dots, \\ \mathit{filter}(\mathit{cons}(\mathit{succ}(X), Y)) &\to \dots, \\ \mathit{filter}(\mathit{cons}(\mathit{pred}(X), Y)) &\to \dots, \\ \mathit{filter}(\mathit{cons}(0, Y)) &\to \dots, \\ \mathit{pred}(\mathit{succ}(X)) &\to \dots, \\ \mathit{succ}(\mathit{pred}(X)) &\to \dots \end{aligned}$$

On observe que les formes réductibles les plus simples ont pour symbole racine *filter*, *pred* ou *succ*. Pour les règles faisant intervenir *filter*, on constate que le symbole situé directement sous *filter* est soit *nil* soit *cons*, et que ces deux symboles n'interviennent dans aucun autre membre gauche. Outre l'état p_{red}, l'automate $\mathcal{IRR}(R)$ doit donc comporter un état p_{list} pour reconnaitre les listes (termes dont le symbole racine est *nil* ou *cons*). Il doit aussi comporter un état p_p pour les termes de la forme $\mathit{pred}(\dots)$, un état p_s pour ceux de la forme $\mathit{succ}(\dots)$, un état p_0 pour 0. Enfin, il doit y avoir un état p_{var} pour reconnaitre tous les termes qui ne sont pas des sous-termes des membres gauches des règles de R, mais peuvent participer d'un terme réductible en tant qu'instance d'une variable apparaissant dans les membres gauches.

Finalement, on a un ensemble de 3 états $P_{int} = \{p_0, p_p, p_s\}$ destiné à reconnaitre les termes qui sont censés désigner des entiers. Contrairement au cas des listes, on doit en avoir 3 et non un seul car par exemple il faut distinguer $\mathit{succ}(0)$ et $\mathit{pred}(0)$

puisque $succ(succ(0))$ est une forme normale tandis que $succ(pred(0))$ n'en est pas une. On aura donc les transitions suivantes

$$
\begin{aligned}
0 &\rightarrowtail p_0 && \\
succ(p_p) &\rightarrowtail p_{\text{red}} && \\
succ(p) &\rightarrowtail p_s && \text{pour tout } p \text{ sauf } p_p \text{ et } p_{\text{red}} \\
pred(p_s) &\rightarrowtail p_{\text{red}} && \\
pred(p) &\rightarrowtail p_p && \text{pour tout } p \text{ sauf } p_s \text{ et } p_{\text{red}} \\
nil &\rightarrowtail p_{list} && \\
cons(p,p') &\rightarrowtail p_{list} && \text{pour tout } p \in P_{int} \text{ et tout } p' \neq p_{\text{red}} \\
filter(p_{list}) &\rightarrowtail p_{\text{red}} &&
\end{aligned}
$$

À ce « noyau », il faut ajouter les transitions qui traduisent le fait qu'un terme est réductible dès qu'un de ses sous-termes est réductible :

$$
\begin{aligned}
f(p_{\text{red}}) &\rightarrowtail p_{\text{red}} && \text{pour chaque } f \text{ symbole de fonction unaire} \\
cons(p_{\text{red}},p) &\rightarrowtail p_{\text{red}} && \text{pour tout } p \\
cons(p,p_{\text{red}}) &\rightarrowtail p_{\text{red}} && \text{pour tout } p
\end{aligned}
$$

et enfin faire reconnaitre toutes les autres constructions par p_{var} :

$$
\begin{aligned}
f(p) &\rightarrowtail p_{var} && \text{pour tous } f \text{ et } p \text{ tels qu'il n'y a pas déjà de transition } f(p) \rightarrowtail \dots \\
cons(p,p') &\rightarrowtail p_{var} && \text{pour tous } p \text{ et } p' \text{ tels qu'il n'y a pas déjà de transition } cons(p,p') \rightarrowtail \dots
\end{aligned}
$$

Remarquons que si la dénomination des états de $\mathcal{IRR}(R)$ ainsi que le processus par lequel il est décrit font penser à des types (listes, entiers), il n'y a aucun typage dans le système de réécriture. Le terme $t = succ(cons(nil, 0))$ est licite. Il est reconnu par $\mathcal{IRR}(R)$ en l'état p_{var} (c'est une forme normale), de sorte que $prec(t)$ est reconnu en p_{red}. 102 ◁

Cet automate permet en fait de définir une relation de congruence $\equiv_{\mathcal{IRR}(R)}$ qui discrimine les termes selon leur statut de forme normale ou réductible. L'ensemble des termes réductibles constitue la classe d'équivalence associée à p_{red}.

Définition 103.
Soit R un système de réécriture linéaire à gauche et $\mathcal{IRR}(R)$ un automate répondant aux critère du théorème 100. On définit la relation d'équivalence $\equiv_{\mathcal{IRR}(R)}$ sur $\mathrm{T}(\Sigma)$ par : pour tout $u, v \in \mathrm{T}(\Sigma)$, notant p_u et p_v les uniques états de $\mathcal{IRR}(R)$ tels que $u \in \mathcal{L}(\mathcal{IRR}(R), p_u)$ et $v \in \mathcal{L}(\mathcal{IRR}(R), p_v)$, on a $u \equiv_{\mathcal{IRR}(R)} v$ si et seulement si $p_u = p_v$. 103 ◀

Lemme 104.
La relation $\equiv_{\mathcal{IRR}(R)}$ est une relation de congruence. 104 ■

Démonstration.
C'est évidemment une relation d'équivalence. Soit k un entier naturel, $f \in \Sigma_k$ et $t_1, \ldots, t_k, s_1, \ldots, s_k \in \mathrm{T}(\Sigma)$ tels que pour tout $i \in [\![1\,;k]\!]$, $t_i \equiv_{\mathcal{IRR}(R)} s_i$. Soit $t = f(t_1, \ldots, t_k)$ et $s = f(s_1, \ldots, s_k)$. Montrons que $t \equiv_{\mathcal{IRR}(R)} s$. Il suffit de reprendre la définition : pour tout $i \in [\![1\,;k]\!]$, il y a un état p_i de $\mathcal{IRR}(R)$ tel que $t_i, s_i \in \mathscr{L}(\mathcal{IRR}(R), p_i)$. Soit p l'état de $\mathcal{IRR}(R)$ destination de l'unique transition $f(p_1, \ldots, p_k) \rightarrowtail p$ de l'automate. On a bien $t \equiv_{\mathcal{IRR}(R)} s$. □

De la même façon que la notion de E-cohérence nous a permis de quantifier la précision de la surapproximation produite par la complétion équationnelle usuelle, l'utilisation d'un automate $\mathcal{IRR}(R)$-cohérent, c'est-à-dire tel que chaque état reconnait des termes qui sont de la même classe pour $\equiv_{\mathcal{IRR}(R)}$, sera fructueuse pour la définition d'une procédure de complétion équationnelle compatible avec la stratégie en profondeur.

5.2.1.2 Automates produits

Le gestion de deux relations de congruence nous conduit à utiliser des automates dont les états sont des couples. Nous introduisons d'abord la notion classique [Com+08] d'automate produit.

Définition 105 (Automate produit).
Soit $\mathcal{A} = (\Sigma, Q, Q_F, \Delta)$ et $\mathcal{B} = (\Sigma, P, P_F, \Gamma)$ deux automates. On construit l'automate produit $\mathcal{A} \times \mathcal{B}$ à partir de l'ensemble d'états $Q \times P$. On note $\langle q, p \rangle$ les états de $\mathcal{A} \times \mathcal{B}$. Cette notation est étendue aux configurations par induction : pour un symbole $f \in \Sigma$ et des configurations $\gamma_1 = f(q_1, \ldots, q_k)$ de $\mathcal{A}$ et $\gamma_2 = f(p_1, \ldots, p_k)$ de $\mathcal{B}$, on note $\langle \gamma_1, \gamma_2 \rangle$ la configuration $f(\langle q_1, p_1 \rangle, \ldots, \langle q_k, p_k \rangle)$ de $\mathcal{A} \times \mathcal{B}$.

L'ensemble des delta-transitions de $\mathcal{A} \times \mathcal{B}$ est alors celui des $\langle \gamma_1, \gamma_2 \rangle \rightarrowtail \langle q, p \rangle$ avec $\gamma_1 \rightarrowtail_{\mathcal{A}} q$ et $\gamma_2 \rightarrowtail_{\mathcal{B}} p$. L'ensemble des epsilon-transition est celui des $\langle q, p \rangle \rightarrowtail \langle q', p \rangle$ avec $q \rightarrowtail_{\mathcal{A}} q'$ et des $\langle q, p \rangle \rightarrowtail \langle q, p' \rangle$ avec $p \rightarrowtail_{\mathcal{B}} p'$.

L'ensemble d'états finaux est $Q_F \times P_F$. 105 ◀

Remarque 106. La définition classique présentée ci-dessus de l'ensemble des états finaux par $Q_F \times P_F$ conduit à ce que $\mathscr{L}(\mathcal{A} \times \mathcal{B}) = \mathscr{L}(\mathcal{A}) \cap \mathscr{L}(\mathcal{B})$. Cela ne correspond pas à l'utilisation que nous ferons des produits. Aussi, il nous arrivera, après avoir considéré un automate produit, de modifier « à la main » l'ensemble des états finaux pour l'adapter à notre besoin.

Plus généralement, nous serons amenés à considérer des automates dont l'ensemble des états est un produit cartésien.

Nous aurons aussi besoin, inversement, de réaliser des projections sur la partie gauche ou droite d'un automate de couples.

Définition 107 (Projections).
Soit $\mathcal{A} = (\Sigma, Q, Q_F, \Delta)$ un automate de couples, soit $\tau \rightarrowtail \rho$ une de ses transitions et $\langle q, p \rangle$ un de ses états. On note $\Pi_1(\langle q, p \rangle) = q$ et on étend $\Pi_1(\cdot)$ aux configurations par induction : $\Pi_1(f(\gamma_1, \ldots, \gamma_k)) = f(\Pi_1(\gamma_1), \ldots, \Pi_1(\gamma_k))$. On définit $\Pi_1(\tau \rightarrowtail \rho) = \Pi_1(\tau) \rightarrowtail \Pi_1(\rho)$ ainsi que $\Pi_1(\mathcal{A}) = (\Sigma, \Pi_1(Q), \Pi_1(Q_F), \Pi_1(\Delta))$. $\Pi_2(\cdot)$ est défini pour tous ces objets de la même façon pour les composantes droites. 107 ◀

Remarque 108. Utiliser $\Pi_1(\mathcal{A})$ revient à oublier la précision apportée par les composantes droites des états. Par conséquent, $\mathscr{L}(\Pi_1(\mathcal{A}), q) \supseteq \mathscr{L}(\mathcal{A}, \langle q, p \rangle)$.

5.2.1.3 Compatibilité

Définition 109 (Compatibilité avec $\mathcal{IRR}(R)$).
On dit qu'un automate de couple $\mathcal{A}$ est compatible avec $\mathcal{IRR}(R)$ si, pour toute configuration c et pour tout état $\langle q, p \rangle$ de $\mathcal{A}$, $\Pi_2(c)$ est une configuration et p est un état de $\mathcal{IRR}(R)$, et si $c \overset{*}{\underset{\mathcal{A}}{\rightarrowtail}} \langle q, p \rangle$, alors $\Pi_2(c) \overset{*}{\underset{\mathcal{IRR}(R)}{\rightarrowtail}} p$. 109 ◀

5.2.2 Procédure de complétion innermost

Nous donnons ici une description théorique de la complétion. Les algorithmes utilisés en pratique seront donné dans le chapitre 6.

Étant donné le système de réécriture R, nous construisons l'automate déterministe complet $\mathcal{IRR}(R)$. Partant d'un automate $\mathcal{A}_\circ$ sans epsilon-transition reconnaissant le langage initial L_0 (comme dans la complétion classique), nous construisons l'automate produit $\mathcal{A}_0 = \mathcal{A}_\circ \times \mathcal{IRR}(R)$. Ce calcul de produit n'est fait qu'une seule fois au début, nous allons le mettre à jour lors des étapes de complétion. Grâce à lui, nous pouvons discriminer les termes en fonction de leur situation par rapport à la réécriture en profondeur. Il nous est alors possible de définir une notion de paire critique innermost de façon effective puis une procédure de complétion exacte. Enfin, nous pourrons également dire quand il est possible d'appliquer une équation de surapproximation.

5.2.2.1 Complétion exacte

Une paire critique innermost est une paire critique au sens de la complétion classique qui vérifie en plus une propriété relative aux sous-termes.

Définition 110 (Paire critique innermost).
La paire $(\ell \to r, \sigma, \langle q, p \rangle)$, avec $\ell \to r \in R$, $\sigma : \mathcal{X} \to Q_{\mathcal{A}}$ et $\langle q, p \rangle \in Q_{\mathcal{A}}$, est critique si

1. $\ell\sigma \overset{*}{\underset{\mathcal{A}}{\rightarrowtail}} \langle q, p \rangle$,

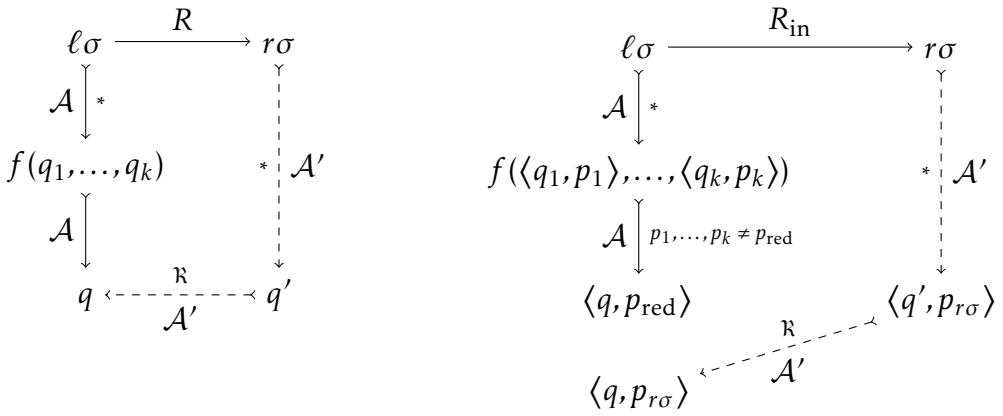

FIGURE 5.6 – Comparaison de la complétion standard et de la complétion adaptée

2. il n'y a pas de p' tel que $r\sigma \xrightarrow[\mathcal{A}]{*} \langle q,p'\rangle$ et
3. pour toute sous-configuration γ à la profondeur 1 de $\ell\sigma$, l'état $\left\langle q_\gamma,p_\gamma\right\rangle$ tel que $\gamma \xrightarrow[\mathcal{A}]{*} \left\langle q_\gamma,p_\gamma\right\rangle$ dans le chemin de reconnaissance de la condition 1 est tel que $p_\gamma \neq p_{\text{red}}$. 110 ◀

Remarque 111. Puisqu'une paire critique dénote une possibilité de réécriture, le p de la définition 110 est nécessairement p_{red}.

Exemple 112.

Reprenons encore une fois la situation des exemples 45–102 et considérons la règle $filter(cons(pred(X),Y)) \to cons(pred(X),filter(Y))$, la substitution $\sigma_1 = \{X \mapsto \langle q_s,p_s\rangle, Y \mapsto \langle q_n,p_{nil}\rangle\}$ et l'état $\left\langle q_f,p_{\text{red}}\right\rangle$. Cela correspond à la paire critique PC_1 de l'exemple 79. Cependant, et conformément au souhait que nous formulions dans l'exemple 99, cette situation ne constitue pas une paire critique au sens de la nouvelle définition. En effet, le chemin de reconnaissance est

$$\begin{aligned} filter(cons(pred(\langle q_s,p_s\rangle),\langle q_n,p_{nil}\rangle)) &\rightarrowtail filter(cons(\left\langle q_p,p_{\text{red}}\right\rangle,\langle q_n,p_{nil}\rangle)) \\ &\rightarrowtail filter(\langle q_c,p_{\text{red}}\rangle) \\ &\rightarrowtail \left\langle q_f,p_{\text{red}}\right\rangle \end{aligned}$$

et l'on constate donc la présence d'un p_{red} à la profondeur 1.

Il y a cependant une paire critique innermost dans $\mathcal{A}_0$ avec la règle $pred(succ(X)) \to X$, la substitution $\sigma_2 = \{X \mapsto \langle q_0,p_0\rangle\}$ et l'état $\left\langle q_p,p_{\text{red}}\right\rangle$. 112 ◁

Le processus de normalisation sera analogue à celui utilisé dans le cas classique, avec une modification technique : pour faciliter la formulation de la preuve de

précision et limiter le nombre de cas à traiter, nous réutilisons pas les états déjà existants, mais introduisons à chaque fois un nouvel état. Ce n'est pas gênant car il est possible de fusionner après coup ces états supplémentaires.

Définition 113 (Normalisation avec nouveaux états).
Soit $\mathcal{A} = (\Sigma, Q, \Delta, F)$ un automate et c une configuration de cet automate. On définit $\mathrm{NormInn}_{\mathcal{A}}(c)$ par induction comme l'est $\mathrm{Norm}_{\mathcal{A}}(c)$ dans la définition 75, avec la modification suivante pour le cas de base.

Si c est une configuration élémentaire, c'est-à-dire de la forme $f(\langle q_1, p_1\rangle, \ldots, \langle q_k, p_k\rangle)$ avec k un entier naturel, $f \in \Sigma_k$ et $\langle q_1, p_1\rangle, \ldots, \langle q_k, p_k\rangle \in Q$, alors :

— peu importe qu'il existe $\langle q', p'\rangle \in Q$ tel que $f(\langle q_1, p_1\rangle, \ldots, \langle q_k, p_k\rangle) \rightarrowtail_{\mathcal{A}} \langle q', p'\rangle$ ou qu'il n'existe pas de tel état : on choisit $q'' \notin \Pi_1(Q)$ arbitraire, on note p'' l'unique état de $\mathcal{IRR}(R)$ tel que $f(p_1, \ldots, p_k) \rightarrowtail_{\mathcal{IRR}(R)} p''$ et on définit

$$\mathrm{NormInn}_{\mathcal{A}}(c) = (\{\langle q'', p''\rangle\}, \{f(\langle q_1, p_1\rangle, \ldots, \langle q_k, p_k\rangle) \rightarrowtail \langle q'', p''\rangle\}, \langle q'', p''\rangle).$$

113 ◀

Pour compléter une paire critique (figure 5.6), nous devons, comme dans le cas classique, « fermer le carré ». Cependant, ici, afin de conserver la cohérence des composantes $\langle \cdot, p\rangle$, nous ne pouvons pas le refermer sur lui-même car cela conduirait à ajouter une epsilon-transition du type $\langle q', p_{r\sigma}\rangle \rightarrowtail \langle q, p_{\mathrm{red}}\rangle$. Or, il est clair que si nous voulons que nos composantes droites aient un sens, une epsilon-transition ne peut se faire qu'à composante droite identique. Dans le cas contraire, l'automate n'est plus compatible avec $\mathcal{IRR}(R)$. Il faut donc ajouter une transition $\langle q', p_{r\sigma}\rangle \rightarrowtail \langle q, p_{r\sigma}\rangle$, avec $p_{r\sigma} \neq p_{\mathrm{red}}$ en général.

Cela pose quelques difficultés : à ce stade, nous avons rendu l'automate compatible avec la réécriture innermost $\ell\sigma \to r\sigma$, mais seulement lorsque celle-ci se fait à la racine, mais l'automate présente toujours une différence de comportement pour les configurations de la forme $C[\ell\sigma]$ et $C[r\sigma]$ avec C un contexte non trivial. En effet, le fait que le carré n'ait pas été refermé exactement sur lui-même signifie que la partie en $\langle \cdot, p_{r\sigma}\rangle$, après avoir progressé jusqu'à $C[\langle q, p_{r\sigma}\rangle]$, ne peut pas utiliser les transitions où apparait $\langle q, p_{\mathrm{red}}\rangle$ pour reconnaitre la partie propre du contexte C.

Nous allons donc devoir dupliquer ces transitions : pour toute transition où apparait $\langle q, p_{\mathrm{red}}\rangle$, nous devons ajouter (si elle n'est pas déjà présente par ailleurs), une transition analogue dans laquelle $\langle q, p_{\mathrm{red}}\rangle$ est remplacé par $\langle q, p_{r\sigma}\rangle$. Bien sûr, il ne suffit pas de le faire une fois, il faut tour à tour considérer les états qu'on peut atteindre à partir de $\langle q, p_{\mathrm{red}}\rangle$.

Heureusement, ce processus ne peut pas conduire à la formation de nouvelles paires critiques. En effet, on n'y effectue aucune normalisation et donc on n'introduit aucune nouvelle composante gauche $\langle q, \cdot\rangle$ (pour la même raison, on est certain que ces opérations de duplication terminent). De plus, on n'aboutit pas, à de nouvelles situations $\langle q'', p_{\mathrm{red}}\rangle$ à droite qui n'apparaissaient pas déjà à gauche (en fait,

en partant de $\langle q, p_{\text{red}}\rangle$, tous les états dont on construit un analogue ont déjà une composante droite égale à p_{red}).

Formalisons ce qui vient d'être expliqué.

Définition 114 (Transitions supplémentaires).
Soit $pc = (\ell \to r, \sigma, \langle q, p_{\text{red}}\rangle)$ une paire critique innermost pour l'automate $\mathcal{A}$ et le système de réécriture R. Notons $(Q', \Delta', \langle q', p_{r\sigma}\rangle) = \text{NormInn}_{\mathcal{A}^{\not\mathcal{R}}}(r\sigma)$.

Soit $\gamma \rightarrowtail \langle q'', p_{\text{red}}\rangle$ une transition de $\mathcal{A}$ avec γ une configuration élémentaire où intervient $\langle q, p_{\text{red}}\rangle$, c'est-à-dire que γ est de la forme $f(\dots, \langle q, p_{\text{red}}\rangle, \dots)$. Soit $\gamma_{[p_{\text{red}}/p_{r\sigma}]}$ la configuration obtenue en remplaçant dans γ l'état $\langle q, p_{\text{red}}\rangle$ par $\langle q, p_{r\sigma}\rangle$. Notons p'' l'unique état de $\mathcal{IRR}(R)$ tel que $\Pi_2\left(\gamma_{[p_{\text{red}}/p_{r\sigma}]}\right) \rightarrowtail_{\mathcal{IRR}(R)} p''$. La transition supplémentaire associée à $\gamma \rightarrowtail \langle q'', p_{\text{red}}\rangle$ est $\gamma_{[p_{\text{red}}/p_{r\sigma}]} \rightarrowtail \langle q'', p''\rangle$.

On appelle $\text{TSuppInn}_{\mathcal{A}}(pc)$ l'ensemble de transitions obtenu en rassemblant toutes les transitions supplémentaires associées aux transitions de $\mathcal{A}$ qui font intervenir l'état $\langle q, p_{\text{red}}\rangle$ puis, successivement, à celles faisant intervenir les états $\langle q'', p''\rangle$ où aboutissent des transitions supplémentaires précédemment considérées. 114 ◀

Définition 115 (Complétion d'une paire critique).
Soit $pc = (\ell \to r, \sigma, \langle q, p\rangle)$ une paire critique innermost pour l'automate $\mathcal{A}$ et le système de réécriture R. Notant $(Q', \Delta', \langle q', p_{r\sigma}\rangle) = \text{NormInn}_{\mathcal{A}^{\not\mathcal{R}}}(r\sigma)$, la complétion de cette paire critique est $\text{CompInn}_{\mathcal{A}}(pc) = \left(Q', \Delta' \cup \text{TSuppInn}_{\mathcal{A}}(pc), \left\{q' \overset{\mathcal{R}}{\rightarrowtail} q\right\}\right)$. 115 ◀

La notion d'étape de complétion est identique à celle du cas classique.

5.2.2.2 Fusion équationnelle

Nous avons déjà observé que la compatibilité avec $\mathcal{IRR}(R)$ interdit la présence d'une epsilon-transition entre deux états dont la composante $\langle \cdot, p\rangle$ est différente. Par conséquent, il n'est pas possible d'appliquer une équation entre deux tels états, ce qui conduit à une définition plus restrictive de la notion de situation d'application.

Définition 116 (Situation d'application).
Soit $\mathcal{A}$ un automate, $s = t$ une équation et θ une substitution de $(\Sigma, \mathcal{X})$ vers $(\varnothing, Q)$. Soit $\langle q_s, p\rangle, \langle q_t, p\rangle$ deux états de $\mathcal{A}$. On dit que $(s = t, \theta, \langle q_s, p\rangle, \langle q_t, p\rangle)$ est une situation d'application (de l'équation $s = t$) si
— $s\theta \rightarrowtail^*_{\mathcal{A}} \langle q_s, p\rangle$,
— $t\theta \rightarrowtail^*_{\mathcal{A}} \langle q_t, p\rangle$ et
— il n'est pas le cas que $\langle q_s, p\rangle \overset{\mathcal{E},*}{\rightleftarrows}_{\mathcal{A}} \langle q_t, p\rangle$. 116 ◀

Définition 117 (Application d'une équation).
Soit $\mathcal{A}$ un automate et $(s = t, \theta, \langle q_s, p\rangle, \langle q_t, p\rangle)$ une situation d'application d'une équation. L'application de cette équation est la paire de transitions $\{\langle q_s, p\rangle \xrightarrow{E} \langle q_t, p\rangle, \langle q_t, p\rangle \xrightarrow{E} \langle q_s, p\rangle\}$. 117 ◀

La fusion équationnelle est définie comme dans le cas classique : énumération des situations d'application, puis fusion, jusqu'à l'obtention d'un point fixe, laquelle est garantie par le fait qu'on n'ajoute aucun nouvel état.

Comme dans le cas classique, une étape de complétion équationnelle consiste en une étape de complétion exacte suivie d'une étape de fusion équationnelle.

5.2.3 Preuves de correction et précision

Nous allons adapter les preuves données en section 4.3.2.

5.2.3.1 Préliminaires

Avant de prouver que les automates construits vérifient des invariants semblables au cas classique, énonçons un résultat de composition de contexte. Ce résultat est évident dans le cas classique mais mérite une mention ici.

Lemme 118.
Soit $\mathcal{A}$ un automate cohérent avec $\mathcal{IRR}(R)$, $pc = (\ell \to r, \sigma, \langle q, p\rangle)$ une paire critique de $\mathcal{A}$, $p_{r\sigma}$ l'état de $\mathcal{IRR}(R)$ tel que $r\sigma \xrightarrow[\mathcal{IRR}(R)]{*} p_{r\sigma}$ et $\mathcal{B}$ l'automate résultant de la complétion de cette paire critique. Soit C un contexte sur $\mathrm{T}(\Sigma)$ et $\langle q_1, p_1\rangle$ un état de $\mathcal{A}$ tel que $C[\langle q, p\rangle] \xrightarrow[\mathcal{A}]{*} \langle q_1, p_1\rangle$. Alors il existe un état p_2 de $\mathcal{IRR}(R)$ tel que $C[\langle q, p_{r\sigma}\rangle] \xrightarrow[\mathcal{B}]{*} \langle q_1, p_2\rangle$. 118 ■

Démonstration.
Notons que $p = p_1 = p_{\mathrm{red}}$.

Nous devons montrer que toutes les transition du chemin $C[\langle q, p_{\mathrm{red}}\rangle] \xrightarrow[\mathcal{A}]{*} \langle q_1, p_{\mathrm{red}}\rangle$ admettent un correspondant en partant de $C[\langle q, p_{r\sigma}\rangle]$. Observons tout d'abord que les transitions qui servent à reconnaitre des sous-termes situés en des positions de C parallèles au trou peuvent être conservées. Il reste à montrer que pour chaque transition mettant en jeu une position au dessus du lieu de la complétion, il existe une transition correspondante. Ces transitions sont précisément celles ajoutées par $\mathrm{TSuppInn}_{\mathcal{A}}(pc)$. □

Remarque 119. Les transitions de $\mathrm{TSuppInn}_{\mathcal{A}}(pc)$ sont nécessaires à ce lemme. En effet, considérons $R = \{g(f(b)) \to g(f(a)), f(a) \to c\}$, $\mathcal{A}_\circ = \{b \rightarrowtail q_b, f(q_b) \rightarrowtail q_{fb}, g(q_{fb}) \rightarrowtail q_{gfb}\}$. L'automate des formes normales peut être décrit ainsi $\mathcal{IRR}(R) = \{a \rightarrowtail p_a, b \rightarrowtail p_b, c \rightarrowtail p_c, f(p_a) \rightarrowtail p_{\mathrm{red}}, g(p_a) \rightarrowtail p_c, f(p_b) \rightarrowtail p_{fb}, g(p_{fb}) \rightarrowtail p_{\mathrm{red}}, f(p_c) \rightarrowtail p_c, g(p_c) \rightarrowtail p_c\}$. Il y a une paire critique $PC_1 = (g(f(b)) \to g(f(a)), \varnothing, \left\langle q_{gfb}, p_{\mathrm{red}} \right\rangle)$ dans $\mathcal{A}_\circ \times \mathcal{IRR}(R)$, qui est complétée par l'ajout des transitions $a \rightarrowtail \langle q_{N1}, p_a \rangle$, $f(\langle q_{N1}, p_a \rangle) \rightarrowtail \langle q_{N2}, p_{\mathrm{red}} \rangle$, $g(\langle q_{N2}, p_{\mathrm{red}} \rangle) \rightarrowtail \langle q_{N3}, p_{\mathrm{red}} \rangle$ et $\langle q_{N3}, p_{\mathrm{red}} \rangle \rightarrowtail \left\langle q_{gfb}, p_{\mathrm{red}} \right\rangle$, ce qui produit l'automate $\mathcal{A}_1$. Ici, $\mathrm{TSuppInn}_{\mathcal{A}}(PC_1)$ est vide.

Il y a une paire critique $PC_2 = (f(a) \to c, \varnothing, \langle q_{N2}, p_{\mathrm{red}} \rangle)$ dans $\mathcal{A}_1$, qu'on complète par les transitions $c \rightarrowtail \langle q_{N4}, p_c \rangle$ et $\langle q_{N2}, p_c \rangle$. Notons $\mathcal{A}_2^\natural$ l'automate ainsi construit. Voyons ce qui se produit en l'absence des transitions de $\mathrm{TSuppInn}_{\mathcal{A}_1}(PC_2)$.

Observons que $g(c) \in R_{\mathrm{in}}(g(f(a)))$ et $g(f(a)) \in \mathscr{L}(\mathcal{A}_2^\natural, \left\langle q_{gfb}, p_{\mathrm{red}} \right\rangle)$ à cause de la complétion de PC_1. Mais nous avons seulement $g(c) \rightarrowtail_{\mathcal{A}_2^\natural} g(\langle q_{N4}, p_c \rangle) \rightarrowtail_{\mathcal{A}_2^\natural} g(\langle q_{N2}, p_c \rangle)$, et cette dernière configuration n'est pas reconnue. Ainsi, $g(c) \notin \mathscr{L}(\mathcal{A}_2^\natural, \left\langle q_{gfb}, p' \right\rangle)$ pour tout p'.

Détaillons comment obtenir $\mathrm{TSuppInn}_{\mathcal{A}_1}(PC_2)$. Comme il existe une transition $g(\langle q_{N2}, p_{\mathrm{red}} \rangle) \rightarrowtail \langle q_{N3}, p_{\mathrm{red}} \rangle$, nous ajoutons une transition $g(\langle q_{N2}, p_c \rangle) \rightarrowtail \langle q_{N3}, p_c \rangle$. Puis, comme $q_{N3} \notin \mathcal{A}_\circ$ et que $\langle q_{N3}, p_{\mathrm{red}} \rangle \rightarrowtail \left\langle q_{gfb}, p_{\mathrm{red}} \right\rangle$, nous ajoutons $\langle q_{N3}, p_c \rangle \rightarrowtail \left\langle q_{gfb}, p_c \right\rangle$. Il n'y a pas d'autre transition à ajouter : nous avons construit $\mathcal{A}_2$, et on a bien $g(c) \in \mathscr{L}(\mathcal{A}_2, \left\langle q_{gfb}, p_c \right\rangle)$.

5.2.3.2 Correction

À l'invariant 74 correspond un invariant qui tient compte de la détection de du caractère « en profondeur » des paires critiques.

Invariant 120.
D'une part, l'automate $\mathcal{A}$ est compatible avec $\mathcal{IRR}(R)$.

D'autre part, pour tout état $\langle q, p_{\mathrm{red}} \rangle$ de $\mathcal{A}$, pour tout $u \in \mathscr{L}(\mathcal{A}, \langle q, p_{\mathrm{red}} \rangle)$, pour tout $v \in R_{\mathrm{in}}(\{u\})$, ou bien il existe p tel que $v \in \mathscr{L}(\mathcal{A}, \langle q, p \rangle)$, ou bien il existe une paire critique innermost $(\ell \to r, \sigma, \langle q_0, p_{\mathrm{red}} \rangle)$ dans $\mathcal{A}$ et un contexte C sur $\mathrm{T}(\Sigma)$ tels que $u \rightarrowtail_{\mathcal{A}}^* C[\ell\sigma] \rightarrowtail_{\mathcal{A}}^* C[\langle q_0, p_{\mathrm{red}} \rangle] \rightarrowtail_{\mathcal{A}}^* \langle q, p_{\mathrm{red}} \rangle$ et $v \rightarrowtail_{\mathcal{A}}^* C[r\sigma]$. 120 ◀

Théorème 121.
Un automate produit par complétion exacte vérifie cet invariant. 121 ■

Démonstration.
L'automate initial $\mathcal{A}_0 = \mathcal{A}_\circ \times \mathcal{IRR}(R)$ est compatible avec $\mathcal{IRR}(R)$. De plus, toutes les étapes de la complétion exacte n'introduisent des transitions $\gamma \rightarrowtail \langle q, p \rangle$ qu'avec

un p choisi explicitement pour que $\Pi_2(\gamma) \rightarrowtail_{\mathcal{IRR}(R)} p$. Ainsi, la compatibilité de $\mathcal{A}$ avec $\mathcal{IRR}(R)$ est assurée à toute étape de complétion exacte.

La deuxième partie de l'invariant se démontre en reprenant la preuve du lemme 83 mutatis mutandis. Soit $\langle q, p_{\text{red}}\rangle$ un état de $\mathcal{A}$, $u \in \mathscr{L}(\mathcal{A}, \langle q, p_{\text{red}}\rangle)$ et $v \in R_{\text{in}}(u)$. Par définition de la réécriture en profondeur, il y a une règle $\ell \to r$ de R, une substitution μ de $(\Sigma, \mathcal{X})$ vers Σ et un contexte C tels que $u = C[\ell\mu]$, $v = C[r\mu]$ et tout sous-terme strict de u est en forme normale. Soit $u_0 = \ell\mu$ et $v_0 = r\mu$. Il existe $\langle q_0, p_{\text{red}}\rangle$ tel que $u_0 \in \mathscr{L}(\mathcal{A}, \langle q_0, p_{\text{red}}\rangle)$ et $C[\langle q_0, p_{\text{red}}\rangle] \overset{*}{\underset{\mathcal{A}}{\rightarrowtail}} \langle q, p_{\text{red}}\rangle$.

Par linéarité de ℓ, il existe $\sigma : \mathcal{X} \to Q_{\mathcal{A}}$ tel que $\ell\mu \overset{*}{\underset{\mathcal{A}}{\rightarrowtail}} \ell\sigma \overset{*}{\underset{\mathcal{A}}{\rightarrowtail}} \langle q_0, p_{\text{red}}\rangle$ et $r\mu \overset{*}{\underset{\mathcal{A}}{\rightarrowtail}} r\sigma$. Il en résulte que $u \overset{*}{\underset{\mathcal{A}}{\rightarrowtail}} C[\ell\sigma] \overset{*}{\underset{\mathcal{A}}{\rightarrowtail}} C[\langle q_0, p_{\text{red}}\rangle] \overset{*}{\underset{\mathcal{A}}{\rightarrowtail}} \langle q, p_{\text{red}}\rangle$ and $v \overset{*}{\underset{\mathcal{A}}{\rightarrowtail}} C[r\sigma]$.

Supposons qu'il n'y a pas de p_0 tel que $v_0 \in \mathscr{L}(\mathcal{A}, \langle q_0, p_0\rangle)$ et montrons que $(\ell \to r, \sigma, \langle q_0, p_{\text{red}}\rangle)$ est une paire critique dans $\mathcal{A}$. D'abord, par hypothèse, il n'y a pas de p tel que $r\sigma \overset{*}{\underset{\mathcal{A}}{\rightarrowtail}} \langle q_0, p\rangle$. Les conditions 1 et 2 de la définition 110 sont donc remplies. Supposons que $\ell = f(\gamma_1, \ldots, \gamma_k)$ et montrons que la condition 3 de la définition 110 est également vérifiée. [1] Pour tout $i \in [\![1\,;k]\!]$, soit $\langle q_i, p_i\rangle$ l'état de $\mathcal{A}$ tel que $\gamma_i\mu \overset{*}{\underset{\mathcal{A}}{\rightarrowtail}} \gamma_i\sigma \overset{*}{\underset{\mathcal{A}}{\rightarrowtail}} \langle q_i, p_i\rangle$ dans le chemin de reconnaissance de $\ell\sigma$. Puisque $\mathcal{A}$ est cohérent avec $\mathcal{IRR}(R)$, pour tout $i \in [\![1\,;k]\!]$, on a $\gamma_i\mu \overset{*}{\underset{\mathcal{IRR}(R)}{\rightarrowtail}} p_i$. Les sous-termes stricts de $\ell\mu$ étant aussi des sous-termes stricts de u, ils sont en forme normale et donc $p_i \neq p_{\text{red}}$. La condition 3 est ainsi remplie. □

Il résulte de ce théorème que, à l'instar du cas classique, la complétion d'automate innermost fournit, lorsqu'elle aboutit à un point-fixe, un automate qui reconnait une surapproximation de $R^*_{\text{in}}(L_0)$.

Comme dans le cas classique, ce résultat de correction reste valable pour la complétion équationnelle car sa démonstration ne repose nullement sur l'absence de transitions issues de l'application d'équations.

5.2.3.3 Précision

Adapter les résultats du cas classique est encore possible mais plus délicat.

Nous devons faire l'hypothèse que l'automate initial ne mélange pas plusieurs classes d'équivalences au sein d'un même état. Cependant, la notion de classe d'un état est techniquement plus compliquée que dans le cas classique, car il faut gérer les composantes $\langle \cdot, p\rangle$.

1. Si ℓ est une constante, alors la condition 3 est vraie.

Définition 122 (Séparation des classes de E).
L'automate $\mathcal{A}$ sépare les classes de E si pour tout $q \in \Pi_1(Q_\mathcal{A})$, il existe un terme s tel que pour tout $p \in \Pi_2(Q_\mathcal{A})$, $\mathscr{L}(\mathcal{A}, \langle q, p\rangle) \subseteq [s]_E$. 122 ◀

Définition 123 (Classe d'équivalence d'une configuration).
Soit $\mathcal{A}$ un automate qui sépare les classes de E. Soit q la composante gauche d'un état ($q \in \Pi_1(\mathcal{A})$). S'il existe $p \in \Pi_2(\mathcal{A})$ tel que $\mathscr{L}(\mathcal{A}, \langle q, p\rangle) \neq \varnothing$, alors la classe commune de tous les termes reconnus par $\mathcal{A}$ en des états de la forme $\langle q, \cdot\rangle$ est notée $[q]_E^\mathcal{A}$. S'il n'y a pas de tel p, on note $[q]_E^\mathcal{A} = \bot$.

Nous étendons cette notation aux configurations de $\mathcal{A}$ comme suit. On note $[\langle q, p\rangle]_E^\mathcal{A} = [q]_E^\mathcal{A}$. Soit $c = f(c_1, \ldots, c_k)$ une configuration de $\mathcal{A}$. S'il existe un $i \in [\![1\,;k]\!]$ tel que $[c_i]_E^\mathcal{A} = \bot$, alors soit $[c]_E^\mathcal{A} = \bot$. Sinon, pour tout $i \in [\![1\,;k]\!]$, soit $t_i \in \mathscr{L}(\Pi_1(\mathcal{A}), \Pi_1(c_i))$. On note $[c]_E^\mathcal{A} = [f(t_1, \ldots, t_n)]_E$. 123 ◀

Définition 124 (Séparation totale).
Soit $\mathcal{A}$ un automate séparant les classes de E. On dit que cette séparation est totale si pour toute configuration c de $\mathcal{A}$, $[c]_E^\mathcal{A} \neq \bot$. 124 ◀

Les remarques techniques suivantes rendront la preuve de précision plus lisible.

Définition 125 (Équations de réflexivité).
Pour une signature $(\Sigma, \mathcal{X})$, on appelle ensemble de ses équation de réflexivité l'ensemble

$$E_r = \{f(x_1, \ldots, x_k) = f(x_1, \ldots, x_k) \mid k \in \mathbb{N} \wedge f \in \Sigma_k \wedge x_1, \ldots, x_k \text{ sont distincts}\}.$$

125 ◀

Remarque 126. Pour faciliter la formulation de la preuve et limiter le nombre de cas à traiter, nous utilisons une procédure de normalisation qui ne réutilise pas les états déjà existants, mais introduit à chaque fois un nouvel état. Pour contrebalancer cette création systématique, nous ajoutons à E l'ensemble E_r des équations de réflexivité : la fusion équationnelle va fusionner les états superflus ainsi créés. D'autre part, les classes d'équivalences de $\equiv_E$ sont les mêmes que celles de $\equiv_{E \cup E_r}$. Enfin, la fusion équationnelle augmente la taille des langages reconnus, donc ce détour est valide.

L'invariant de précision est appelé R_{in}/E-cohérence.

Définition 127 (R_{in}/E-cohérence).
Un automate $\mathcal{A}$ est R_{in}/E-cohérent si

1. $\mathcal{A}^{\not\kappa}$ sépare totalement les classes de E,
2. $\mathcal{A}$ est accessible et

3. pour tout état $\langle q, p\rangle$ de $\mathcal{A}$, $\mathscr{L}(\mathcal{A}, \langle q, p\rangle) \subseteq (R_{\text{in}}/E)^*\left([q]_E^{\mathcal{A}^{\not\leftarrow}}\right)$. 127 ◀

Lemme 128.
Soit $\mathcal{A}$ un automate qui sépare totalement les classes de E. Soit $(s = t, \theta, \langle q_1, p\rangle, \langle q_2, p\rangle)$ une situation d'application d'une équation de E in $\mathcal{A}$. On a $[q_1]_E^{\mathcal{A}} = [q_2]_E^{\mathcal{A}}$. 128 ■

Démonstration.
Il suffit de montrer que $[s\theta]_E^{\mathcal{A}} = [t\theta]_E^{\mathcal{A}}$. La séparation des classes de E par $\mathcal{A}$ étant totale, pour tout x du domaine de θ, il existe un terme $x\mu \in \mathscr{L}(\Pi_1(\mathcal{A}), x\theta)$. Ceci permet de construire une instance $s\mu \equiv_E t\mu$ de l'équation considérée. Mais $[s\theta]_E^{\mathcal{A}} = [s\mu]_E$ et $[t\theta]_E^{\mathcal{A}} = [t\mu]_E$. □

Lemme 129.
Soit $\mathcal{A}$ un automate R_{in}/E-cohérent et cohérent avec $\mathcal{IRR}(R)$, soit $(\ell \to r, \sigma, \langle q, p_{\text{red}}\rangle)$ une paire critique qu'on complète par l'ajout d'une transition $\langle q', p_{r\sigma}\rangle \overset{\mathfrak{R}}{\rightarrowtail} \langle q, p_{r\sigma}\rangle$. Nous supposons que les étapes de normalisation ont déjà été faites dans $\mathcal{A}$. On a $\mathscr{L}(\mathcal{A}, r\sigma) \subseteq (R_{\text{in}}/E)^*\left([q]_E^{\mathcal{A}^{\not\leftarrow}}\right)$. 129 ■

Démonstration.
Soit $t \in T(\Sigma)$ tel que $t \overset{*}{\underset{\mathcal{A}}{\rightarrowtail}} r\sigma \overset{*}{\underset{\mathcal{A}^{\not\leftarrow}}{\rightarrowtail}} \langle q', p_{r\sigma}\rangle$. Pour toute variable x de r, soit $x\mu$ le sous-terme de t à la position où x apparait dans r, de sorte que $t = r\mu$. Pour toute variable y apparaissant dans ℓ mais non dans r, soit $y\mu$ un terme quelconque de $\mathscr{L}(\mathcal{A}, y\sigma)$. Comme $\mathcal{A}$ est cohérent avec $\mathcal{IRR}(R)$ et que la paire critique remplit la condition 3 de la définition 110, tout sous-terme strict de $\ell\mu$ est en forme normale. Ainsi, $t = r\mu \in R_{\text{in}}(\ell\mu)$. De plus, $\ell\mu \in \mathscr{L}(\mathcal{A}, \langle q, p_{\text{red}}\rangle)$ et $\mathcal{A}$ est R_{in}/E-cohérent, donc $t \in (R_{\text{in}}/E)^*\left([q]_E^{\mathcal{A}^{\not\leftarrow}}\right)$. □

Théorème 130.
La fusion équationnelle préserve la R_{in}/E-cohérence. 130 ■

Démonstration.
Soit $(s = t, \theta, \langle q_1, p_0\rangle, \langle q_2, p_0\rangle)$ une situation d'application d'une équation de E dans $\mathcal{A}$ et soit $\mathcal{B}$ l'automate résultant de la fusion de $\langle q_1, p_0\rangle$ et $\langle q_2, p_0\rangle$. Soit $\langle q, p\rangle$ un état.

Montrons que $\mathscr{L}(\mathcal{B}^{\not\leftarrow}, \langle q, p\rangle) \subseteq [q]_E^{\mathcal{A}^{\not\leftarrow}}$. Considérons $u = C[u_0] \overset{*}{\underset{\mathcal{A}^{\not\leftarrow}}{\rightarrowtail}} C[\langle q_1, p_0\rangle] \overset{\mathfrak{E}}{\rightarrowtail}$ $C[\langle q_2, p_0\rangle] \overset{*}{\underset{\mathcal{A}^{\not\leftarrow}}{\rightarrowtail}} \langle q, p\rangle$. On a $u_0 \in [q_1]_E^{\mathcal{A}^{\not\leftarrow}}$, donc, par le lemme 128, $u_0 \in [q_2]_E^{\mathcal{A}^{\not\leftarrow}}$. Par conséquent, $u \in C[[q_2]_E^{\mathcal{A}^{\not\leftarrow}}]$, c'est-à-dire $[q]_E^{\mathcal{A}^{\not\leftarrow}}$. Les autres cas sont soit triviaux, soit réductibles à celui-ci.

Montrons ensuite que $\mathscr{L}(\mathcal{B}, \langle q, p\rangle) \subseteq (R_{\text{in}}/E)^*\left([q]_E^{\mathcal{A}'}\right)$. Considérons $u = C[u_0] \xrightarrow[\mathcal{A}]{}^* C[\langle q_1, p_0\rangle] \xrightarrow{\mathfrak{E}} C[\langle q_2, p_0\rangle] \xrightarrow[\mathcal{A}]{}^* \langle q, p\rangle$. On a $u_0 \in (R_{\text{in}}/E)^*\left([q_1]_E^{\mathcal{A}'}\right)$, *ie.* $u_0 \in (R_{\text{in}}/E)^*\left([q_2]_E^{\mathcal{A}'}\right)$. Ainsi $u \in (R_{\text{in}}/E)^*\left(C[[q_2]_E^{\mathcal{A}'}]\right)$.

Si $C[\langle q_2, p_0\rangle] \xrightarrow[\mathcal{A}']{}^* \langle q, p\rangle$, alors $C[[q_2]_E^{\mathcal{A}'}] = [q]_E^{\mathcal{A}'}$, puis $u \in (R_{\text{in}}/E)^*\left([q]_E^{\mathcal{A}'}\right)$.

Supposons maintenant que $C[\langle q_2, p_0\rangle] \xrightarrow[\mathcal{A}]{}^* \langle q, p\rangle$ avec une seule $\mathfrak{K}$-transition, c'est-à-dire que $C[\langle q_2, p_0\rangle] \xrightarrow[\mathcal{A}']{}^* D[r\sigma] \xrightarrow[\mathcal{A}']{}^* D[\langle q_3', p_3\rangle] \xrightarrow{\mathfrak{K}} D[\langle q_3, p_3\rangle] \xrightarrow[\mathcal{A}']{}^* \langle q, p\rangle$. Il y a une paire critique correspondante, notons-là $(\ell \to r, \sigma, \langle q_3, p_{\text{red}}\rangle)$. Par lemme 129, $\mathscr{L}(\mathcal{A}, r\sigma) \subseteq (R_{\text{in}}/E)^*\left([q_3]_E^{\mathcal{A}'}\right)$. D'autre part, $D[[q_3]_E^{\mathcal{A}'}] = [q]_E^{\mathcal{A}'}$, donc $\mathscr{L}(\mathcal{A}, D[r\sigma]) \subseteq (R_{\text{in}}/E)^*\left([q]_E^{\mathcal{A}'}\right)$. Puisque $(R_{\text{in}}/E)^*$ est un opérateur sur les classes d'équivalences par E, tout terme équivalent à un terme de $\mathscr{L}(\mathcal{A}, D[r\sigma])$ est également un descendant de $[q]_E^{\mathcal{A}'}$ par R_{in}/E. Mais puisque $C[[q_2]_E^{\mathcal{A}'}] = D[[r\sigma]_E^{\mathcal{A}'}]$, u est un descendant d'un tel terme, donc $u \in (R_{\text{in}}/E)^*\left([q]_E^{\mathcal{A}'}\right)$.

Finalement, dans le paragraphe ci-dessus, on a seulement exploité $D[[q_3]_E^{\mathcal{A}'}] \subseteq (R_{\text{in}}/E)^*\left([q]_E^{\mathcal{A}'}\right)$ (on avait $D[[q_3]_E^{\mathcal{A}'}] = [q]_E^{\mathcal{A}'}$) : on peut donc utiliser ce raisonnement dans une induction sur le nombre de $\mathfrak{K}$-transitions présentes dans le chemin de $C[\langle q_2, p_0\rangle] \xrightarrow[\mathcal{A}]{}^* \langle q, p\rangle$, et conclure. □

L'utilisation systématique de nouveaux états pour la normalisation simplifie l'étude d'une étape de complétion car le lemme suivant est alors trivial.

Lemme 131.
La normalisation préserve la R_{in}/E-cohérence, ainsi que l'ajout des transitions de $\text{TSuppInn}_{\mathcal{A}}(pc)$. 131 ■

Lemme 132.
La complétion de paire critique préserve la R_{in}/E-cohérence. 132 ■

Démonstration.
Soit $(\ell \to r, \sigma, \langle q, p_{\text{red}}\rangle)$ une paire critique, qu'on complète par normalisation et l'ajout de la transition $\langle q', p'\rangle \xrightarrow{\mathfrak{K}} \langle q, p'\rangle$. Soit $\mathcal{B}$ l'automate ainsi constuit. Il suffit de prouver que $\mathscr{L}(\mathcal{B}, \langle q', p'\rangle) \subseteq (R_{\text{in}}/E)^*\left([q]_E^{\mathcal{A}'}\right)$. Grâce à l'utilisation systématique de nouveaux états pour la normalisation, on a $\mathscr{L}(\mathcal{B}, \langle q', p'\rangle) = \mathscr{L}(\mathcal{A}, r\sigma)$. On conclut grâce au lemme 129. □

Nous pouvons donc énoncer le théorème de précision analogue au cas classique.

Théorème 133.
La complétion équationnelle innermost préserve la R_{in}/E-cohérence. 133 ■

Théorème 134 (Précision).
Soit E un ensemble d'équations. Soit $\mathcal{A}_0 = \mathcal{A}_\circ \times \mathcal{IRR}(R)$, où $\mathcal{A}_\circ$ est un automate pour lequel on a fixé des états finaux déterminés. On ne conserve dans $\mathcal{A}_0$ que les états accessibles et, par exception à la règle générale de construction des automates produits, on convient qu'un état $\langle q, p\rangle$ de $\mathcal{A}_0$ est final dès lors que q est final dans $\mathcal{A}_\circ$. Supposons que $\mathcal{A}_0$ sépare les classes de E. Soit R un TRS linéaire à gauche. Soit $\mathcal{A}$ obtenu à partir de $\mathcal{A}_0$ après un certain nombre d'étapes de complétion équationnelle innermost. On a

$$\mathcal{L}(\mathcal{A}) \subseteq (R_{\text{in}}/E)^*(\mathcal{L}(\mathcal{A}_\circ \times \mathcal{IRR}(R))). \qquad 134\ ■$$

Démonstration.
Comme $\mathcal{A}_0$ ne contient pas de $\mathfrak{K}$-transition, $\mathcal{A}_0^{\not\mathfrak{K}} = \mathcal{A}_0$. Ainsi, pour $\mathcal{A}_0$, la troisième condition (R_{in}/E)-cohérence découle des deux premières. De plus, $\mathcal{A}_0$ sépare les classes de E et est accessible, donc il sépare totalement les classes de E. Ainsi, $\mathcal{A}_0$ est (R_{in}/E)-cohérent. Le théorème 133 montre que cette propriété est transmise à $\mathcal{A}$, ce qui établit le résultat. □

6 Timbuk

Dans ce chapitre, nous présentons les méthodes concrètes qui ont été employées pour mettre en œuvre dans le logiciel Timbuk les méthodes décrites au chapitre précédent. Cela nécessite l'exposition du fonctionnement général du logiciel Timbuk.

Timbuk [GV01] est créé par Thomas Genet et Valérie Viet Triem Tong en 2000 comme bibliothèque de manipulation d'automates d'arbres et comme implémentation de la complétion d'automate décrite dans [Gen97] (une méthode, antérieure à celle rappelée en section 4.3, qui n'utilise pas d'équations pour exprimer les approximations).

Seul ce deuxième aspect, la mise en œuvre des méthodes de complétion d'automate, est conservé dans la version actuelle de Timbuk, Timbuk3, dont une variante sert de base à l'implémentation expérimentale de notre méthode tenant compte de la stratégie en profondeur.

6.1 Timbuk3

Le code de Timbuk3 [Gen+14] n'est pas basé sur celui des versions antérieures ; il a été entièrement réécrit par Thomas Genet et Yohan Boichut. Le code de Timbuk2 faisait un usage très abondant de foncteurs OCaml, ce qui en fait une sorte de cathédrale : un ensemble magnifique achevé, mais peu propice à des extensions expérimentales.

L'approche retenue pour Timbuk3 est plus pragmatique, plus concrète et plus impérative.

Outre les différents éléments d'entrée-sortie (interface utilisateur, parseur pour les descriptions d'automates), l'essentiel du fonctionnement de Timbuk3 tourne autour de son module de manipulation d'automate. En effet, les systèmes de réécriture d'une part, les ensembles d'équations d'autre part, sont également représentés par des automates.

Nous présentons ces formats brièvement et renvoyons à [Gen09 ; FGT04] pour plus de détails.

6.1.1 Représentation des automates

La description d'un automate par la liste de ses transitions, présentée dans les exposés théoriques, n'est utilisée que marginalement, comme format intermédiaire pour les entrées-sorties et quelques manipulations. Cette représentation ne permettrait en effet pas de mener de façon efficace les opérations de la complétion.

La représentation retenue est double ; chaque transition $f(q_1,\ldots,q_n) \rightarrowtail q'$ est représentée deux fois. Dans une première table de hachage (transtable), cette transition est représentée par une clé q' à laquelle est associé le couple $(f,(q_1,\ldots,q_n))$. Dans une seconde table (satable), cette transition est représentée, pour chaque $i \in [\![1\,;n]\!]$, par une clé q_i à laquelle est associé $(f,(q_1,\ldots,q_n),q')$. Est également créée et maintenue une table (ftable) qui étant donné un symbole $f \in \Sigma$, permet de connaitre l'ensemble des états q qui interviennent dans les transitions de symbole f.

Ces représentations ne permettent pas de façon efficace de déterminer en quel état est reconnu un terme, mais cela est sans importance car l'algorithme de complétion procède par filtrage.

6.1.2 Conversion des TRS en automates

D'une façon générale, l'exécution d'un TRS requiert la capacité, pour chaque terme t, de détecter s'il est de la forme $C[\ell\sigma]$ pour certains C, σ, $\ell \rightarrow r$ et de générer le terme $C[r\sigma]$ correspondant. Cependant, dans l'optique de détecter et compléter des paires critiques, il n'est pas utile de tenir compte d'un éventuel contexte C, si bien qu'il suffit, pour les substitutions potentielles σ, de vérifier si $t = \ell\sigma$ pour un certain membre gauche ℓ.

On peut utiliser un automate pour cela, dont chaque état représente un sous-terme des membres gauches des règles et dont les états terminaux correspondent aux membres gauches complets. À cet automate est adjoint une table de correspondance entre les états terminaux représentant chacun un ℓ et le terme r correspondant dans la règle $\ell \rightarrow r$. Dans une signature $(\Sigma,\mathcal{X})$, l'ensemble des variables est infini. Cependant, la description d'un TRS donné ne peut, par la force des choses, faire intervenir qu'un nombre fini de variables différentes : la gestion des variables ne pose donc pas de difficulté ; elle peuvent être traitées à l'instar des constantes, en prenant soin de maintenir une table de correspondance des symboles d'arité zéro qui dénotent des variables.

L'automate des membres gauches sommairement décrit ci-dessus peut être obtenu à partir d'un automate constitué d'aucun état par application de l'algorithme de normalisation aux membres gauches du TRS, vus comme des configurations de cet automate.

Exemple 135.
Considérons le TRS de la figure 5.5, savoir :

$$\begin{aligned} sumList(X,Y) &\to cons(plus(X,Y), sumList(plus(X,Y), s(Y))) \\ nth(0, cons(H,T)) &\to H \\ nth(s(I), cons(H,T)) &\to nth(I,T) \\ suite(X) &\to nth(X, sumList(0,0)) \\ plus(0,X) &\to X \\ plus(s(X),Y) &\to s(plus(X,Y)). \end{aligned}$$

Pour construire un automate correspondant à ce TRS, on part d'un automate vide et on normalise chacun des membres gauches vers un état ad hoc.

La normalisation de $sumList(X,Y)$ provoque l'introduction de trois états q_0, q_1 et q_2 tels que

$$\begin{aligned} X &\rightarrowtail q_0 \\ Y &\rightarrowtail q_1 \\ sumList(q_0,q_1) &\rightarrowtail q_2. \end{aligned}$$

Outre ces transitions, on met en place une table de correspondance pour les états qui sont issus de variables

q_0	X
q_1	Y

ainsi qu'une table de correspondance pour les membres droits des règles

q_2	$cons(plus(X,Y), sumList(plus(X,Y), s(Y)))$

La normalisation de la deuxième règle conduit à l'ajout des états q_3, q_4, q_5, q_6, q_7 et des transitions

$$\begin{aligned} 0 &\rightarrowtail q_3 \\ H &\rightarrowtail q_4 \\ T &\rightarrowtail q_5 \\ cons(q_4,q_5) &\rightarrowtail q_6 \\ nth(q_3,q_6) &\rightarrowtail q_7 \end{aligned}$$

ainsi que la mise à jour des deux tables de correspondance, celle des variables

q_0	X
q_1	Y
q_4	H
q_5	T

et celle des membres droits des règles

q_2	$cons(plus(X,Y), sumList(plus(X,Y), s(Y)))$
q_7	H

La normalisation de la troisième règle peut utiliser les transitions déjà introduites pour reconnaitre le $cons(H,T)$. On n'introduit que les états q_8, q_9, q_{10} et les transitions

$$\begin{aligned} I &\rightarrowtail q_8 \\ s(q_8) &\rightarrowtail q_9 \\ nth(q_8, q_6) &\rightarrowtail q_{10}. \end{aligned}$$

Les tables deviennent

q_0	X
q_1	Y
q_4	H
q_5	T
q_8	I

et

q_2	$cons(plus(X,Y), sumList(plus(X,Y), s(Y)))$
q_7	H
q_{10}	$nth(I,T)$

On procède de même pour les règles suivantes. L'algorithme de filtrage peut alors détecter, pour tout terme t, si t est reconnu en l'un des états q_2, q_7, q_{10}, ... La table des membres droits permet à la fois de connaitre la règle applicable et le membre droit associé. Le procédé de construction de l'automate assure de plus qu'un terme t qui est reconnu en q_{10} par exemple est de la forme

$$t = nth(s(t_1), cons(t_2, t_3))$$

avec t_1 reconnu en q_8, t_2 reconnu en q_4 et t_3 reconnu en q_5. La table des variables associe à chacun de ces états la variable idoine, ce qui permet de construire la substitution μ sous-jacente à l'instance de la règle en question. 135 ◁

À partir de Timbuk3 ont été écrites plusieurs variantes, chacune mettant en œuvre un raffinement ou une extension de la procédure classique de complétion équationnelle.

TimbukLTA Créé par Valérie Murat, cette version ajoute la gestion de treillis pour la manipulation d'espaces de valeurs tels que les entiers machine [Mur14].

TimbukCEGAR Cette version ajoute la possibilité d'étiqueter les transitions par des formules ainsi qu'un module de gestion de ces formules qui permet la mise en œuvre du raffinement automatique des surapproximations à l'aide de contre-exemples [Boy10].

6.2 TimbukSTRAT

Bien que le raffinement guidé par les contre-exemples ne soit pas possible à ce stade, le code de TimbukSTRAT est basé sur celui de TimbukCEGAR, dont on a désactivé le module de gestion des formules.

6.2.1 Mise en œuvre de la complétion équationnelle

Dans le chapitre 5, on a exposé la méthode de complétion innermost comme une version modifiée de la complétion classique qui travaille sur un automate produit $\mathcal{A} \times \mathcal{I}_{\mathcal{RR}}(R)$. Cette présentation est utile car elle permet de décalquer les théorèmes de correction et précision ainsi que leurs preuves.

Cependant, mettre en œuvre cette présentation de la complétion innermost dans Timbuk aurait conduit à adapter l'algorithme de complétion pour traiter les couples $\langle q, p \rangle$, et par ricochet l'ensemble du code.

Thomas Genet et moi-même avons donc décidé de revenir à ce qui est l'essence de l'adaptation : il s'agit d'inhiber la complétion de certaines paires critiques classiques lorsque les états mis en jeu ne reconnaissent aucune forme normale.

Un nouveau module a donc été écrit pour gérer le produit $\mathcal{A} \times \mathcal{I}_{\mathcal{RR}}(R)$, mais la complétion n'est pas faite sur cet automate, qui sert seulement à inhiber dans le module principal, marginalement modifié à cet effet, la complétion des paires critiques ne correspondant pas à la stratégie en profondeur. Plus précisément, lors de l'énumération des substitutions à prendre en compte pour un membre gauche, celles ne donnant pas naissance à une paire critique innermost sont rejetées.

6.2.2 Construction de l'automate $\mathcal{I}_{\mathcal{RR}}(R)$

Pour mener à bien ces opérations, il est nécessaire de disposer d'un automate reconnaissant les formes normales de R. On peut utiliser Timbuk2 pour construire un automate des formes normales : cet outil permet de générer un automate des termes réductibles, mais cet automate n'est ni déterministe ni complet. Il faut donc le déterminiser, calculer son complémentaire et le compléter. C'est la procédure qui a été suivie pour générer les exemples de [GS15]. Elle est cependant insatisfaisante parce que les algorithmes mis en œuvre pour la déterminisation et la complétion sont génériques et ne peuvent tirer parti des propriétés de l'automate à constuire.

Nous avons donc entrepris d'écrire un module permettant la génération d'un automate $\mathcal{IRR}(R)$ conforme à l'énoncé du théorème 100. Il procède en trois grandes étapes : tout d'abord, calculer un automate qui discrimine les termes et sous-termes des membres gauche de règles de R ; ensuite, étendre puis compléter cet automate en tenant compte de ses particularités.

La première étape est une adaptation de l'algorithme utilisé pour construire un automate représentant R en section 6.1.2.

On tient compte du fait que, cette fois, on cherche seulement à savoir si un terme est réductible par le système, mais sans accorder d'importance à la règle précisément applicable : la table de correspondance entre les états reconnaissant les membres gauches et les membres droits n'a pas lieu d'être. On n'a donc pas besoin que les différents membres gauches soient reconnus dans des états différents. Ainsi, si l'on reprend le système de l'exemple 135, il n'y a pas lieu de distinguer les états q_2, q_7 et q_{10} : dans l'automate $\mathcal{IRR}(R)$, on n'a qu'un seul état p_{red} pour tous les membres gauches.

De même, il n'est pas nécessaire de dinstinguer les différentes variables : comme on ne s'intéresse qu'aux membres gauches, il importe peu qu'une variable s'appelle X ou Y, étant entendu que les membres gauches sont linéaires. Par conséquent, il n'y a pas non plus de table de correspondance états–variables et les états q_0, q_1, q_4, q_5, q_8 de l'exemple 135 n'ont pas lieu d'être distincts : on n'aura qu'un seul état p_{var} pour toutes les variables.

On applique donc aux membres gauches un algorithme de normalisation modifié : il y a deux états distingués p_{red} et p_{var}, et un terme qui est une variable est systématiquement normalisé vers p_{var} tandis que tout terme qui se trouve être un membre gauche de règle est normalisé vers p_{red}. Une vérification est effectuée à cette fin avant toute introduction d'un nouvel état.

Exemple 136.

Appliquons cet algorithme modifié au système de l'exemple 135. La normalisation de la première règle introduit

$$\begin{aligned} X &\rightarrowtail p_{\text{var}} \\ Y &\rightarrowtail p_{\text{var}} \\ \mathit{sumList}(p_{\text{var}}, p_{\text{var}}) &\rightarrowtail p_{\text{red}}. \end{aligned}$$

La deuxième ajoute

$$\begin{aligned} 0 &\rightarrowtail p_2 \qquad\qquad p_{\text{red}} \text{ est en fait } p_0 \text{ et } p_{\text{var}} \text{ est } p_1 \\ H &\rightarrowtail p_{\text{var}} \\ T &\rightarrowtail p_{\text{var}} \\ \mathit{cons}(p_{\text{var}}, p_{\text{var}}) &\rightarrowtail p_3 \\ \mathit{nth}(p_2, p_3) &\rightarrowtail p_{\text{red}}. \end{aligned}$$

Et la troisième :

$$I \rightarrowtail p_{\text{var}}$$
$$s(p_{\text{var}}) \rightarrowtail p_4$$
$$nth(p_4, p_3) \rightarrowtail p_{\text{red}}.$$

La règle $suite(X) \rightarrow \dots$ donne

$$suite(p_{\text{var}}) \rightarrowtail p_{\text{red}}.$$

Les règles sur *plus* introduisent chacune une transition :

$$plus(p_2, p_{\text{var}}) \rightarrowtail p_{\text{red}}$$
$$plus(p_4, p_{\text{var}}) \rightarrowtail p_{\text{red}}.$$

En effet, bien que dans la règle avec *nth*, les entiers non nuls aient été exprimés avec $s(I)$ et dans la règle du *plus* avec $s(X)$, on bénéficie de l'unicité de l'état reconnaissant les variables.

On pourrait constater — c'est un algorithme classique de simplification d'automate — que dans cet exemple, les états p_2 et p_4 jouent des rôles identiques : ils peuvent donc être fusionnés. 136 ◁

L'automate ainsi construit est déterministe mais pas complet, et il ne reconnait pas non plus toutes les formes normales, seulement celles qui sont des sous-termes stricts des règles.

Exemple 137.
Dans l'exemple précédent, le terme $suite(0)$ n'est pas reconnu par l'automate. Il devrait être reconnu en p_{red} car il s'agit d'un terme réductible, mais pour cela, il manque soit une transition $0 \rightarrowtail p_{\text{var}}$ (qui permettrait d'appliquer ensuite $suite(p_{\text{var}}) \rightarrowtail p_{\text{red}}$), soit une transition $suite(p_2) \rightarrowtail p_{\text{red}}$. La première solution n'est pas compatible avec l'exigence d'un automate déterministe car on a déjà $0 \rightarrowtail p_2$. C'est donc la deuxième possibilité qui sera appliquée. 137 ◁

Il faut étendre l'automate. L'extension consiste, pour chaque transition où p_{var} apparait à gauche $f(\dots, p_{\text{var}}, \dots) \rightarrowtail p'$, à ajouter, pour chaque état p qui n'est ni p_{red}, ni p_{var}, une transition $f(\dots, p, \dots) \rightarrowtail p'$. Cette opération traduit le fait que chaque fois qu'une variable apparait dans une règle, n'importe quel terme est admissible à cette position, y compris les sous-termes des règles.

Exemple 138.
Comme indiqué, il faut ajouter une transition $suite(p_2) \rightarrowtail p_{\text{red}}$. Mais aussi $suite(p_3) \rightarrowtail p_{\text{red}}$ et $suite(p_4) \rightarrowtail p_{\text{red}}$. De même, à partir de la transition $plus(p_2, p_{\text{var}}) \rightarrowtail p_{\text{red}}$, on doit ajouter les 3 transitions $plus(p_2, p_i) \rightarrowtail p_{\text{red}}$ pour $i \in [\![2\,;4]\!]$. Enfin, pour les transitions où p_{var} apparait 2 fois, il faut énumérer les p_i pour chacune des positions (ce qui fait 3^2 transitions à chaque fois). 138 ◁

Il reste encore à rendre l'automate complet, c'est-à-dire qu'il faut énumérer toutes les configurations élémentaires qui n'apparaissent pas dans les transitions déjà construites et les faire reconnaitre en un état. Contrairement à l'algorithme classique, on n'introduit pas un nouvel état puits, mais on va utiliser p_{red} et p_{var} à bon escient.

Exemple 139.
Dans l'exemple filé, on ne peut pour l'heure construire aucune liste parce que l'on n'a pas de constructeur de liste vide. En fait, la signature comprend également une constante *nil*. Elle doit être considérée pour la complétion de l'automate. Dans le cas présent, *nil* est une forme normale, donc il ne doit pas être reconnu en p_{red}. En revanche, il peut intervenir dans des instances de membres gauches des règles comme instance d'une variable. On introduit donc $nil \rightarrowtail p_{\text{var}}$.

De même, le terme $plus(suite(0), 0)$, pour étrange qu'il soit, doit être reconnu en un état, plus précisément p_{red} car c'est un terme réductible (puisque son sous-terme $suite(0)$ l'est). Avec les transitions existantes, $plus(suite(0), 0)$ est reconnu en la configuration $plus(p_{\text{red}}, p_2)$. On ajoute donc $plus(p_{\text{red}}, p_2) \rightarrowtail p_{\text{red}}$. 139 ◁

Pour toute configuration $f(p_1, \ldots, p_n)$ non encore reconnue, il faut ajouter une transition $f(p_1, \ldots, p_n) \rightarrowtail p_{\text{red}}$ s'il existe $i \in [\![1\,;n]\!]$ tel que $p_i = p_{\text{red}}$ et une transition $f(p_1, \ldots, p_n) \rightarrowtail p_{\text{var}}$ dans le cas contraire. Le premier cas traduit le fait que si un terme t est réductible, alors tout terme u dont t est un sous-terme est encore réductible. Le second cas traduit le fait que si une configuration ne permet pas naturellement de progresser dans la reconnaissance d'un membre gauche d'une règle, elle doit être rangée parmi les termes quelconques qui sont admissibles chaque fois qu'une variable est présente.

L'automate ainsi construit est bien complet — on a tout fait pour — et il demeure déterministe parce que les étapes d'extension et de complétion n'ajoutent de transitions que pour des configurations qui ne sont pas déjà reconnues.

La première étape de construction n'est pas plus couteuse que l'étape classique de traduction du système de réécriture en un automate des membres gauches. En revanche, les étapes d'extension et de complétion sont couteuses en temps et en mémoire — le problème de calcul de $\mathcal{IRR}(R)$ est EXPTIME-complet [Com13]. On peut cependant envisager de ne pas les exécuter lors de la génération de $\mathcal{IRR}(R)$, mais au contraire d'introduire ces transitions à la volée lorsqu'elles sont utiles à la construction du produit $\mathcal{A}_0 \times \mathcal{IRR}(R)$ — et seulement dans ce cas.

6.2.3 Limites

TimbukSTRAT a été développé pour aider au calcul d'exemples de ce que permet la complétion innermost. L'implémentation actuelle n'est cependant pas complètement satisfaisante au regard de la définition de la méthode. En effet, le lien

ténu entre l'algorithme de complétion existant et le nouveau module de gestion de $\mathcal{A} \times \mathcal{IRR}(R)$, s'il a permis d'obtenir rapidement des résultats, a pour effet que le produit $\mathcal{A} \times \mathcal{IRR}(R)$ n'est pas utilisé à plein : l'algorithme principal ne traite toujours que des composantes $\langle q, \cdot \rangle$ et la fonction du nouveau module qui doit déterminer si une substitution doit être conservée ne peut donc directement exploiter un couple $\langle q, p \rangle$.

Cette implémentation a suffi à donner des résultats concrets encourageants, mais en général elle est suceptible de donner des surapproximations plus grossières que ce qui est présenté au chapitre 5.

On peut parfaire davantage cette implémentation. Mais comme on l'a vu, Timbuk présente actuellement de nombreuses variantes qui ne sont pas compatibles entre elles, et il serait sans doute profitable de remédier d'abord à ce problème.

6.2.4 Comparaison avec la méthode de [CLM15]

Notre complétion d'automate permet de surapproximer le comportement d'un système de réécriture sous la stratégie en profondeur. La méthode de [CLM15] consiste à ne pas modifier les procédures habituelles d'analyse de TRS mais à transformer le système R initial en un système R' qui encode le comportement de R soumis à la stratégie en profondeur. Nous comparons les ordres de grandeur des structures mises en jeu d'une part lors de l'exécution de notre méthode — calcul de $\mathcal{IRR}(R)$ puis application de la complétion adaptée à la stratégie — et d'autre part lorsqu'on applique la transformation de [CLM15] puis qu'on applique au système résultant la complétion classique.

Prenons pour R le système de l'exemple 135 et considérons le langage intial

$$L_0 = \left\{ suite(s^k(0)) \;\middle|\; k \in \mathbb{N} \right\}.$$

Exécuter R à partir d'un terme de L_0 revient à tenter de calculer le k^{e} terme de la suite u de l'exemple des figures 5.3, 5.4 et 5.5 de la section 5.1.1 (page 47). Si on applique la stratégie en profondeur, celle de OCaml présentée en figure 5.3, ce calcul ne termine pas, ce qui se traduit par le fait que le langage $R^*(L_0)$ ne possède aucune valeur, aucune forme normale et notamment aucun entier $s^k(0)$, mais seulement l'ensemble infini des étapes de calcul.

Notre méthode d'analyse consiste à calculer l'automate des formes normales. Cet automate comprend 4 états et 74 transitions. Ensuite, nous appliquons la procédure de complétion équationnelle à partir d'un automate à 2 états et 3 transitions qui reconnait L_0. La complétion équationnelle converge en 4 étapes et produit un automate point-fixe à 7 états et 13 transitions (figure 6.1). On constate sur cet automate l'absence de terme de la forme $s^k(0)$ dans $R^*_{\text{in}}(L_0)$: les seuls termes reconnus en l'état final q_0 sont de la forme $suite(\dots)$ (terme de départ), soit de la forme $nth(\dots)$ (via q_3). Nous avons ainsi prouvé la non-terminaison du système de la figure 5.5

lorsqu'il est utilisé avec la stratégie en profondeur, et donc la non-terminaison du programme OCaml de la figure 5.3, ce qu'il n'était pas possible de faire directement avec la complétion classique.

$$
\begin{aligned}
s(q_1) &\rightarrowtail q_1 & \qquad 0 &\rightarrowtail q_1\\
suite(q_1) &\rightarrowtail q_0 & \qquad plus(q_1, q_1) &\rightarrowtail q_4\\
sumList(q_4, q_1) &\rightarrowtail q_5 & \qquad nth(q_1, q_2) &\rightarrowtail q_3\\
sumList(q_1, q_1) &\rightarrowtail q_2 & \qquad cons(q_4, q_5) &\rightarrowtail q_6\\
q_3 &\rightarrowtail q_0 & \qquad q_6 &\rightarrowtail q_5\\
q_6 &\rightarrowtail q_2 & \qquad q_4 &\rightarrowtail q_1\\
q_1 &\rightarrowtail q_4
\end{aligned}
$$

État final : q_0

Figure 6.1 – Les transitions de l'automate point-fixe obtenu par notre méthode

La convergence de la complétion a été rendue possible par l'application des équations de surapproximation de la figure 6.2.

$$
\begin{aligned}
s(s(X)) &= s(X)\\
plus(0, X) &= X\\
plus(X, Y) &= plus(X, Y)
\end{aligned}
$$

Figure 6.2 – Les équations utilisées pour obtenir la convergence de la complétion

Avec l'outil de [CLM15], on commence par produire, à partir de R (qui comprend 6 règles), un système R' à 620 règles portant sur une signature étendue à 63 symboles de fonction (dont l'arité la plus élevée est 3) et 706 variables. L'application de la méthode de complétion classique — puisque la stratégie est déjà encodée dans le système — à ce système R' et un automate reconnaissant L_0 (3 états, 4 transitions) converge, grâce aux équations de la figure 6.2, en 85 étapes et produit un automate à 113 états et 216 transitions. Il n'est évidemment pas pertinent de reproduire ici ces transitions, mais les outils de Timbuk permettent de vérifier que cet automate point-fixe ne reconnait pas non plus de terme de la forme $s^k(0)$.

Ces deux méthodes fournissent ici des résultats aussi probants pour ce qui est de la preuve de non-terminaison, et ce dans des temps de calculs de l'ordre de la seconde dans les deux cas. La méthode de [CLM15] a l'avantage de la généralité (on peut étudier différentes stratégies). La nôtre au contraire a celui de la spécialisation, qui permet ici de fournir un automate point-fixe bien plus lisible. Cela est utile

pour des applications qui vont au delà de la simple preuve, notamment l'assistance au développement de programme. On peut en effet vouloir utiliser la complétion d'automate pour donner une surapproximation lisible des résultats possibles d'une fonction OCaml. Des travaux sont entrepris en ce sens [GKV15], et il est clair qu'un système de réécriture ou un automate ayant subi la transformation de [CLM15] perd en lisibilité.

7 Vers l'analyse de programmes fonctionnels

Dans ce chapitre, nous indiquons dans quelle mesure notre complétion d'automate innermost peut permettre d'envisager des analyses de programmes fonctionnels par automate d'arbres.

On s'intéresse plus spécialement aux programmes fonctionnel écrits en OCaml [Ler+14], qui est un langage de programmation fonctionnel d'ordre supérieur. Il comporte également des éléments de programmation impérative et objet, mais nous ne nous intéresserons pas à ces fragments. OCaml est un langage largement utilisé dans le domaine de l'analyse de programmes [Coq15 ; Cou+15] mais également pour écrire des applications variées [Pie15 ; FJ15]. La capacité de l'analyser revêt donc un intérêt pratique.

Pour analyser le fragment fonctionnel de OCaml par complétion d'automate, on doit l'adapter de plusieurs façons. Le respect de la stratégie d'appel par valeur de OCaml par notre complétion avec stratégie en profondeur permet une meilleure précision des analyses. Nous dirons (section 7.1) quelques mots à propos de la gestion de la priorité qui existe entre certaines règles.

Mais d'autres défis restent à relever et nous mentionnerons les travaux qui y contribuent ou qui permettent d'envisager des angles d'approche : gérer les termes d'ordre supérieur (section 7.2) et manipuler efficacement des types comme les entiers machines (section 7.3). Pour rendre plus perceptible la difficulté liée à l'ordre supérieur, nous exposons d'abord (section 7.2.1) une méthode d'analyse développée à Oxford par l'équipe du Professeur Ong qui utilise des formalismes plus complexes que les langages réguliers mais permet d'exprimer plus directement certaines des fonctionnalités de OCaml : les PMRS. Enfin, nous montrons que des liens peuvent être tissés entre les méthodes de complétion d'automate et le cadre général de l'interprétation abstraite (section 7.4).

7.1 Priorités sur les règles

Pour traiter convenablement un programme OCaml, il faut respecter les règles de priorité sur les `match` : dans une construction `match`, les différents cas sont envisagés dans l'ordre du texte du programme, et seul le premier cas valide est exécuté.

Par exemple, dans le programme d'insertion dans une liste triée présenté figure 7.1, le deuxième cas n'est testé que si le premier n'est pas activé, et le troisième cas ne filtre, de fait, que la liste vide. Une traduction brutale de chacun de ces trois cas en trois règles de réécriture engendre une imprécision car ces trois règles seront sur un pied d'égalité dans le système de réécriture et le système admettra donc des descendants qui n'ont pas de réalité dans le cadre de la sémantique de OCaml. Cependant, on peut éliminer ce problème en effectuant une traduction plus précise, ou même un pré-traitement sur le code OCaml de façon à ce que les clauses des cas soient disjointes, ce qui dans notre exemple est très simple (figure 7.2).

```
1 let rec insere l x = match l with
2   | h::t when h >= x -> x::h::t
3   | h::t -> h::(insere t x)
4   | _ -> [x]
```

Figure 7.1 – Un programme d'insertion (pour un tri par insertion)

```
1 let rec insere l x = match l with
2   | h::t -> if h >= x then x::h::t else h::(insere t x)
3   | [] -> [x]
```

Figure 7.2 – Programme à clauses disjointes

Dans le cas général cependant, cette transformation produit une inflation exponentielle du nombre de motifs. Suivant [Kra08], qui propose un procédé de transformation systématique, considérons l'exemple de la figure 7.3. Dans cet exemple, le dernier motif désigne de fait tous les triplets qui ne possèdent pas de A. Or cela ne peut être exprimé par un seul motif OCaml disjoint des précédents ni par une seule règle de réécriture : on doit énumérer toutes les combinaisons de B et de C. Pour traiter un tel exemple avec un type à n constructeurs et des k-uplets, il faut remplacer le dernier motif par k^{n-1} règles.

Le langage de stratégies de [CLM15], qui permet d'exprimer « essayer S_1, et si cela échoue essayer S_2 » grâce à l'introduction d'un symbole dénotant l'échec, fournit également un cadre intéressant pour traiter cette question. La génération de règles pour indiquer l'échec nécessite également d'énumérer les B et les C, mais en une seule position à la fois, comme le montre la figure 7.4 dans une présentation simplifiée qui n'explicite pas le symbole d'échec. On n'ajoute ainsi que $k(n-1)+1$ règles.

```
type abc = A | B | C

let possedeA uplet = match uplet with
  | (A, _, _) -> true
  | (_, A, _) -> true
  | (_, _, A) -> true
  | _ -> false
```

FIGURE 7.3 – Un cas problématique pour la gestion des priorités

$$possedeA(x,y,z) \rightarrow possedeA_0(x,y,z) \qquad \text{on commence par tester le motif 0}$$

$$possedeA_0(A,y,z) \rightarrow true$$
$$possedeA_0(B,y,z) \rightarrow possedeA_1(B,y,z) \qquad \text{le motif 0 ne s'applique pas, on essaye le 1}$$
$$possedeA_0(C,y,z) \rightarrow possedeA_1(C,y,z)$$

$$possedeA_1(x,A,z) \rightarrow true \qquad \text{arrivé ici, on sait que } x \text{ n'est pas A}$$
$$possedeA_1(x,B,z) \rightarrow possedeA_2(x,B,z)$$
$$possedeA_1(x,C,z) \rightarrow possedeA_2(x,C,z)$$

$$possedeA_2(x,y,A) \rightarrow true$$
$$possedeA_2(x,y,B) \rightarrow possedeA_3(x,y,B)$$
$$possedeA_2(x,y,C) \rightarrow possedeA_3(x,y,C)$$

$$possedeA_3(x,y,z) \rightarrow false \qquad \text{par construction, } x,y,z \text{ ne peuvent être A}$$

FIGURE 7.4 – Traduction de l'exemple figure 7.3 selon la méthode de [CLM15]

7.2 Gestion de l'ordre supérieur

7.2.1 Une analyse de flot de contrôle d'ordre supérieur par PMRS

7.2.1.1 Présentation

Les PMRS [OR11] ont été développés comme des variantes de HORS. Un HORS est une grammaire d'ordre supérieur (schéma de récursion) : il décrit l'ensemble des termes qui peuvent être produits à partir d'un germe.

Pour illustrer les PMRS et leur utilisation, nous reprenons un exemple de [OR11]. Dans cet article, Ong et Ramsay décrivent une méthode qui prend comme entrée un programme exprimé comme PMRS ainsi qu'une propriété qu'on souhaite véri-

fier. Supposons un programme de filtrage de liste selon un prédicat, qu'on pourrait écrire comme suit en OCaml.

```
1 let rec filter P = function
2   | [] -> []
3   | X::L -> if P X then X::(filter P L) else filter P L
```

Figure 7.5 – Un programme de filtrage

Plus loin, on supposera que l'on souhaite vérifier qu'une liste dont on a filtré zéro ne contient pas de zéro.

Une règle ordinaire (pure) de PMRS, comme d'un HORS, est de la forme $Fx_1 \dots x_n \to r$. Dans le membre gauche, F est un symbole de fonction et $x_1, \dots, x_n$ sont des symboles de variables. Le membre droit, ici noté r, est une expression construite à partir de tels symboles. Les PMRS introduisent la possibilité de règle à motif, de la forme $Fx_1 \dots x_{n-1} p \to \dots$. Le motif p permet d'avoir des règles différentes selon les données. Ainsi, l'exemple choisi ci-dessus peut se traduire par les deux règles à motif $\mathrm{filter}\, P\, \mathrm{nil} \to \mathrm{nil}$ et $\mathrm{filter}\, P (\mathrm{cons}\, X L) \to \mathrm{if} (P X) (\mathrm{cons}\, X (\mathrm{filter}\, P L)) (\mathrm{filter}\, P L)$. Cette faculté d'introduire des termes constructeurs dans le motif permet de décomposer et parcourir récursivement les données.

Ce formalisme, comme celui des TRS, est Turing-complet et on est donc confronté à des problèmes d'indécidabilité. Il y a donc lieu de recourir à une surapproximation, ici les PMRS faibles.

7.2.1.2 PMRS faible puis raffinement par contre-exemple

Les PMRS faibles le sont dans le sens que pour une règle à motif p, les variables de p ne peuvent apparaitre dans le membre droit de la règle. Ceci interdit de décomposer les données de façon récursive : $\mathrm{filter}\, P (\mathrm{cons}\, X L) \to \mathrm{if} (P X) (\mathrm{cons}\, X (\mathrm{filter}\, P L)) (\mathrm{filter}\, P L)$ est invalide car L et X se trouvent dans le motif ainsi que dans le membre droit (la présence de P à droite n'est en revanche pas problématique parce que dans le membre gauche, cette variable n'apparait qu'en dehors du motif). Combiné au fait que l'ensemble des symboles est fini, cette restriction assure que l'espace des données exploré par un PMRS faible est fini. Contrairement aux PMRS généraux, on dispose donc de de résultats de décidabilité sur le model-checking des PMRS faibles.

Pour construire un PMRS faible approximant un PMRS donné, pour chaque règle à motif, on remplace chaque variable L n'ayant plus le droit de cité dans le membre droit par un nouveau symbole de fonction $\mathbf{L}$. Pour que ce PMRS faible surapproxime le comportement du PMRS de départ, il faut donner pour ce nouveau symbole des règles lui permettant de produire un sur-ensemble de tous les termes auxquels L

pouvait faire référence lors de l'application de la règle d'origine. Par exemple, **L** produirait par exemple toutes les listes possibles.

Ceci est un facteur d'imprécision pour les variables des motifs, même si Ong et Ramsay prennent soin de mener une analyse des liaisons entre variables et termes pour guider cette première surapproximation. Nous ne détaillons pas cette étape et procédons à des surapproximations grossières. Ainsi, la règle

$$\text{filter}\, P(\text{cons}\, X L) \to \text{if}(P X)(\text{cons}\, X(\text{filter}\, P L))(\text{filter}\, P L) \tag{7.1}$$

devient

$$\text{filter}\, P(\text{cons}\, X L) \to \text{if}(P \mathbf{X})(\text{cons}\, \mathbf{X}(\text{filter}\, P \mathbf{L}))(\text{filter}\, P \mathbf{L}), \tag{7.2}$$

avec **L** qui génère toutes les listes et **X** qui génère tous les entiers. Pour simplifier, intéressons-nous seulement à l'imprécision causée par **X** : chacune de ses deux occurrences dans le membre droit peut produire zéro comme des entiers non nuls, et ce indépendamment, ce qui fait que l'opération de filtrage de zéro par le if devient inopérante.

Le raffinement consiste à détecter par la faute de quelle règle faible des contre-exemples sont apparus dans le PMRS faible, et à tenter d'en déduire des contre-exemples dans le PMRS d'origine. Si de tels contre-exemples existent, la propriété étudiée n'est pas vérifiée par le programme. Dans le cas contraire, il est procédé à un raffinement des règles du PMRS faible qui ont engendré ces faux contre-exemples. En effet, si les PMRS faibles interdisent la présence d'une variable de motif dans le membre droit, cette interdiction n'a pas cours s'agissant des constantes. Ainsi la règle (7.1) du PMRS initial peut être décomposée selon que X est 0 ou un entier non nul s Y, et remplacée par ces deux règles. La première de ces deux règles,

$$\text{filter}\, P(\text{cons}\, 0 L) \to \text{if}(P 0)(\text{cons}\, 0(\text{filter}\, P L))(\text{filter}\, P L), \tag{7.3}$$

admet une traduction faible en

$$\text{filter}\, P(\text{cons}\, 0 L) \to \text{if}(P 0)(\text{cons}\, 0(\text{filter}\, P \mathbf{L}))(\text{filter}\, P \mathbf{L}). \tag{7.4}$$

On peut donc remplacer, dans le PMRS faible, la règle imprécise (7.2) par les deux règles (7.4) ainsi que

$$\text{filter}\, P(\text{cons}(\text{s}\, Y) L) \to \text{if}(P(\text{s}\, \mathbf{Y}))(\text{cons}(\text{s}\, \mathbf{Y})(\text{filter}\, P \mathbf{L}))(\text{filter}\, P \mathbf{L}), \tag{7.5}$$

ce qui assure la distinction entre zéro et les autres entiers.

Les PMRS faibles permettent donc de décomposer quand même les données, mais jusqu'à une profondeur bornée.

Le formalisme de Ong et Ramsay, plus complexe que celui des automates d'arbres, permet d'exprimer et de vérifier des propriétés non régulières. Il permet d'exprimer directement les programmes d'ordre supérieur.

7.2.2 Gestion de l'ordre supérieur avec les systèmes de réécriture

La capacité d'exprimer des règles d'ordre supérieur est un point important pour parvenir à une analyse de programmes fonctionnels par automates d'arbres.

Le formalisme des PMRS permet d'exprimer des constructions d'ordre supérieur, ce qui n'est pas le cas des systèmes de réécriture que nous utilisons actuellement. Des adaptations doivent être menées. Par exemple, la transposition brutale de la règle de PMRS

$$\text{filter}\, P(\text{cons}\, XL) \to \text{if}(PX)(\text{cons}\, X(\text{filter}\, PL))(\text{filter}\, PL)$$

donne une pseudo-règle de réécriture

$$\mathit{filter}(P, \mathit{cons}(X, L)) \to \mathit{if}(P(X), \mathit{cons}(X, \mathit{filter}(P, L)), \mathit{filter}(P, L)) \tag{7.6}$$

qui n'est pas valide parce que le symbole P est d'arité zéro dans le membre gauche et d'arité un dans le membre droit. De plus, P est une variable, et on ne dispose pas de variable d'arité différente de zéro.

Pour éliminer ce problème de variables d'arité non nulle, la méthode la plus respectueuse du formalisme déjà connu des systèmes de réécriture est d'introduire un symbole explicite d'application. Introduite par [JM79], elle consiste à ne conserver qu'un seul symbole d'opérateur défini, @, dénotant de façon explicite l'application d'une fonction à son argument. Les fonctions sont alors représentées par des symboles d'arité zéro, et en particulier les variables qui représentent des fonctions. Ainsi, l'expression

$$\mathit{if}(P(X))$$

peut être transformée en

$$@(\mathit{if}, @(P, X)), \tag{7.7}$$

ce qui fait bien intervenir seulement des variables d'arité zéro. Mais cette transformation alourdit le TRS et elle pose des problèmes pour le respect des stratégies : dans le terme (7.7), l'argument réel de la fonction *if* n'est pas un sous-terme sous ce symbole.

Il est également possible d'étendre la définition des TRS pour admettre des variables d'arité non nulle. C'est ce que fait [JRR] et c'est ce que nous faisons pour la suite de cette partie. Le terme $P(X)$ de la pseudo-règle (7.6) devient donc licite.

Il reste que l'arité du symbole P varie entre le membre gauche et le membre droit de cette règle. Cela traduit le fait qu'à gauche, on parle de la fonction elle-même, tandis qu'à droite on l'applique. Une solution est de remplacer le symbole P, qui désigne une fonction à un argument, par une application abstraite $P(\diamond)$ et de l'identifier avec $P = \diamond \mapsto P(\diamond)$. En lambda-calcul, cette représentation $\lambda\diamond.P\diamond$ est appelée forme longue de P.

Il s'agit donc d'adjoindre à la signature un ensemble de symboles spéciaux $\diamond_1, \diamond_2, \ldots$ qui permet d'écrire la règle (7.6) comme suit

$$\mathit{filter}(P(\diamond_1), \mathit{cons}(X, L)) \to \mathit{if}(P(X), \mathit{cons}(X, \mathit{filter}(P(\diamond_1), L)), \mathit{filter}(P(\diamond_1), L)). \quad (7.8)$$

La présence de variables d'arité non nulle impose de repenser le concept de substitution et d'autoriser plus généralement l'écriture de termes d'arité non nulle et d'applications partielles.

Une difficulté théorique première est que l'unification et le pattern-matching d'ordre supérieur ne sont pas décidables [Jou05]. Mais les programmes fonctionnels présentent des propriétés particulières qui permettent de les traduire dans des formalismes d'ordre supérieur restreint pour lesquels on dispose d'algorithmes qui en permettent l'exécution effective. Différents formalismes pour la réécriture d'ordre supérieur ont été développés par plusieurs auteurs ; Jouannaud en donne une revue et propose un cadre général.

Une première restriction naturelle est l'utilisation de motifs analogue à celle des PMRS : les symboles qui apparaissent sous un symbole défini dans un membre gauche de règle doivent être des symboles constructeurs ou des variables.

Une autre restriction qui peut être apportée sans douleur aux systèmes qu'on envisage de traiter est le respect d'un système de types. Le langage OCaml, comme tous les langages de programmation fonctionnelle, dispose en effet d'un tel système et l'impose. En réécriture, l'introduction d'un tel système conduit à considérer des sortes et des signatures assorties. Ces systèmes, qui peuvent aussi être considérés au premier ordre, ont déjà été utilisés par Genet pour établir des résultats sur la précision de la complétion équationnelle [Gen14].

Ces différentes approches permettent de résoudre certains problèmes énoncés en ordre supérieur. Considérons le problème de filtrage de liste de la figure 7.5 appliqué avec un prédicat de parité des entiers naturel, qu'on définit récursivement (figure 7.6).

En filtrant une liste d'entiers avec le prédicat de parité, on ne doit pas obtenir de liste comportant un entier impair. Timbuk3 permet de prouver ce résultat en complétant un automate qui reconnait les termes de la forme $@(@(\mathit{filter}, \mathit{even}), q_{lnat})$ avec q_{lnat} un état qui reconnait les listes de naturels : les listes reconnues par l'automate complété ne comprennent que des entiers pairs.

La méthode employée dans [OR11] ne permet pas d'obtenir ce résultat : le raffinement du PMRS faible par les contre-exemples successifs ne termine pas, car dans tous les cas, le motif raffiné sera sans variable et de profondeur finie, ce qui ne permet pas de décrire tous les entiers pairs ou tous les entiers impairs. En effet, on part, comme dans l'exemple du filtrage des entiers non nuls, de la règle faible (7.2)

$$\mathrm{filter}\, P(\mathrm{cons}\, X L) \to \mathrm{if}(P\mathbf{X})(\mathrm{cons}\, \mathbf{X}(\mathrm{filter}\, P\mathbf{L}))(\mathrm{filter}\, P\mathbf{L}),$$

$$
\begin{aligned}
if(true, X, Y) &\to X \\
if(false, X, Y) &\to Y \\
@(even, 0) &\to true \\
@(even, s(0)) &\to false \\
@(even, s(s(X))) &\to @(even, X) \\
@(@(filter, F), nil) &\to nil \\
@(@(filter, F), cons(X, Y)) &\to if(@(F, X), cons(X, @(@(filter, F), Y)), @(@(filter, F), Y))
\end{aligned}
$$

Figure 7.6 – TRS pour filtrer les entiers pairs, avec ordre supérieur encodé

mais cette fois, la détection de contre-exemple conduit à distinguer entre un et les autres entiers, et donc à remplacer cette règle par

$$\text{filter}\, P(\text{cons}\, 0 L) \to \text{if}(P0)(\text{cons}\, 0(\text{filter}\, P\mathbf{L}))(\text{filter}\, P\mathbf{L}), \tag{7.9}$$

$$\text{filter}\, P(\text{cons}(\text{s}\, 0)L) \to \text{if}(P(\text{s}\, 0))(\text{cons}(\text{s}\, 0)(\text{filter}\, P\mathbf{L}))(\text{filter}\, P\mathbf{L}) \text{ et} \tag{7.10}$$

$$\text{filter}\, P(\text{cons}(\text{s}(\text{s}\, X))L) \to \text{if}(P(\text{s}(\text{s}\, \mathbf{X})))(\text{cons}(\text{s}(\text{s}\, \mathbf{X}))(\text{filter}\, P\mathbf{L}))(\text{filter}\, P\mathbf{L}). \tag{7.11}$$

Mais la règle (7.11) provoque encore un contre-exemple (pour l'entier trois), ce qui conduit à la raffiner pour distinguer les entiers deux, trois, et plus grands que trois. On aura ainsi un nouveau contre-exemple pour cinq, etc.

Il reste cependant à étudier comment ces différentes approches du codage de l'ordre supérieur en réécriture peuvent être mises en œuvre systématiquement d'une façon compatible avec les méthodes de complétion actuellement développées. En particulier, il serait souhaitable d'établir à l'ordre supérieur — éventuellement avec encodage — les résultats de terminaison de la complétion d'automate démontrés dans le cadre des programmes fonctionnels dans [Gen15].

7.3 Types built-ins

OCaml utilise des types de base riches et efficaces que les systèmes de réécriture ne permettent pas de restituer. Dans sa thèse de doctorat, Valérie Murat [Mur14] a proposé une extension des systèmes de réécriture, des automates d'arbres et de la complétion d'automate d'arbres pour permettre l'utilisation directe de constantes et d'opérations interprétées. Elle propose également une version de Timbuk implémentant ces contributions.

En réécriture, on représente usuellement les entiers naturel à la Peano : on se donne une constante 0 et un symbole de fonction unaire s. Procéder ainsi donne

naissance à des systèmes de réécriture très lourds et à des automates très volumineux. En effet, pour représenter le seul langage très simple

$$\left\{s^k(0) \;\middle|\; k \in [\![327\,;594]\!]\right\},$$

il faut un automate à 595 états, chacun d'eux permettant la reconnaissance d'un entier.

L'extension de [Mur14] consiste à autoriser dans le système de réécriture la présence de termes interprétés, par exemple les entiers machine et les opérateurs sur ces entiers. Ces opérateurs ne sont pas régis par des règles de réécriture mais par une sémantique dénotationelle usuelle : la transformation du terme interprété $25+9$ en le terme 34 se fait en une étape, à la manière des machines à oracle.

Le problème de la représentation effective de ces valeurs se pose dans l'automate : certains états doivent reconnaitre non pas des termes classiques, mais des valeurs interprétées. Pour cela, Valérie Murat exploite la relation d'ordre sur ces valeurs, supposées fournir une structure de treillis.

Définition 140 (Treillis).
Soit $(T,\sqsubseteq)$ un ensemble ordonné. On dit que $(T,\sqsubseteq)$ est un treillis si pour tous $p,q \in T$, l'ensemble $\{p,q\}$ admet une borne supérieure et une borne inférieure pour $\sqsubseteq$. Autrement dit, s'il existe $m,M \in T$ tels que

$$\begin{array}{lll} m \sqsubseteq p, & m \sqsubseteq q, & \forall x \in T, (x \sqsubseteq p \text{ et } x \sqsubseteq q) \Rightarrow x \sqsubseteq m, \\ M \sqsupseteq p, & M \sqsupseteq q, & \forall x \in T, (x \sqsupseteq p \text{ et } x \sqsupseteq q) \Rightarrow x \sqsupseteq M. \end{array}$$ 140 ◀

Dans un treillis, la borne supérieure (respectivement inférieure) d'une paire est unique, ce qui nous permet d'introduire l'opérateur borne supérieure (respectivement inférieure).

Définition 141 (Opérateur borne supérieure, inférieure).
Soit $(T,\sqsubseteq)$ un treillis. On note $\sqcup$ l'opérateur infixe qui à deux éléments $p,q \in T$ associe $p \sqcup q$, la borne supérieure de $\{p,q\}$. De même, on note $\sqcap$ l'opérateur borne inférieure. 141 ◀

Ainsi, les états reconnaissent des ensembles de valeurs interprétées, et les transitions de l'automate utilisent les opérateurs de borne supérieure. Valérie Murat définit une notion de paire critique pour ces automates d'arbres à treillis (dont la puissance expressive excède celle des automates classiques), puis une procédure de complétion.

Murat implémente ses méthodes dans une version de Timbuk, TimbukLTA.

7.4 Lien avec l'interprétation abstraite

L'interprétation abstraite est une méthode d'élaboration d'analyses statiques inventée par Patrick Cousot dans [CC77]. Exprimer la complétion d'automate en termes d'interprétation abstraite permettrait de donner des nouvelles pistes pour son développement et rendrait possible la combinaison d'analyses par complétion d'automate avec des analyses exprimées par d'autres types d'interprétation abstraite, comme le souligne [CC95].

7.4.1 Présentation informelle de l'interprétation abstraite

Considérons le programme représenté en figure 7.7. Le langage dans lequel il est écrit possède une sémantique standard conforme au sens commun — nous choisissons un langage impératif pour cette présentation car cela nous parait plus simple. Nous supposons que le langage ne traite que des entiers. Ainsi, la division par 2 d'un entier impair est interdite. Ici, nous devons donc nous assurer qu'au moment de réaliser la division de la ligne 10, la variable `a` contient une valeur paire, et cela dans toute exécution du programme.

```
1  var a,b,c,d,e;
2  input c;
3  input d;
4  b = 0;
5  a = 2 * d;
6  while (b < c) do {
7    a = a + 2;
8    b = b + 1;
9  }
10 e = a / 2;
11 output e;
```

Figure 7.7 – Un programme pour présenter l'interprétation abstraite

Bien entendu, il est aisé de le vérifier à la main en faisant une preuve. Mais voyons comment aborder ce problème par le biais de l'interprétation abstraite.

Dans le cas présent, l'état de la mémoire dans laquelle s'exécute le programme est entièrement déterminé par la donnée de la valeur de chacune des variables qui apparaissent dans le texte du programme. Pour des raisons de simplicité d'exposition, nous considérons que toute variable est initialisée à zéro dès sa déclaration. En prenant les variables dans l'ordre alphabétique, l'espace des environnements est donc identifiable au produit cartésien $\mathbb{Z}^5$:

$$Env = \mathbb{Z}^5.$$

Nous voulons nous assurer qu'au moment d'exécuter l'instruction de la ligne 10, la première variable contient une valeur paire, c'est-à-dire que l'environnement courant est dans $2\mathbb{Z} \times \mathbb{Z}^4$. Nous ne pouvons pas faire cette analyse statique avec la sémantique standard, car nous ne pouvons parcourir toutes les valeurs possibles pour `c` et `d`.

L'interprétation abstraite consiste à définir une sémantique plus grossière de ce langage et à exécuter le programme considéré en utilisant cette sémantique grossière, appelée sémantique abstraite. Ici, nous allons considérer un espace de valeurs abstrait constitué des 3 symboles P, I et $\top$ (qui dénoteront les valeurs paires, impaires, et celles dont la parité est inconnue). Il est fréquent de signaler par un dièse ce qui relève du domaine abstrait. L'espace des environnements abstraits est donc

$$Env^\sharp = \{\mathsf{P}, \mathsf{I}, \top\}^5.$$

Dans cette sémantique abstraite, l'instruction `input c;` affecte simplement la valeur $\top$ à la variable `c`. Les constantes `0` et `2` représentent la valeur P, la constante `1` représente la valeur I. Nous devons maintenant donner la sémantique abstraite des opérateurs + et $*$, ce que nous faisons en figure 7.8. Notons que ces tables ont été dressées à la main, en utilisant des connaissances d'arithmétique élémentaire.

$[\![+]\!]^\sharp$	P	I	$\top$
P	P	I	$\top$
I	I	P	$\top$
$\top$	$\top$	$\top$	$\top$

$[\![*]\!]^\sharp$	P	I	$\top$
P	P	P	P
I	P	I	$\top$
$\top$	P	$\top$	$\top$

Figure 7.8 – Sémantique abstraite des opérateurs

La sémantique abstraite des opérateurs nous permet de déterminer que juste après la ligne 5, la variable `a` contient la valeur abstraite P.

Le point délicat est bien sûr la boucle : nous ne savons pas si elle va être empruntée, ni combien de fois. Nous devons donc envisager plusieurs cas possibles.

Supposons que la boucle est empruntée une première fois : à la ligne 7, l'expression `a + 2` vaut P, et donc l'affectation donne à `a` la valeur P (qu'elle avait déjà) ; à la ligne 8, l'expression `b + 1` vaut I (car `b` vaut P depuis la ligne 4), et l'affectation courante donne donc à `b` la valeur I.

Considérons maintenant un passage quelconque dans la boucle. Par propagation, nous pouvons affirmer que la valeur de `a` est toujours P (elle n'est pas modifiée par le passage dans la boucle). En revanche, selon que nous sommes, par exemple, dans le premier ou dans le deuxième passage de la boucle, nous ne savons pas si `b` contient la valeur P ou la valeur I. Nous n'avons pas d'autre choix que de considérer cette valeur comme inconnue : $\top$.

Mais qu'importe : au début de la ligne 10, nous ne savons pas si la boucle a été exécutée ni combien de fois, donc nous n'avons pour `b` pas de valeur plus précise

que $\top$. En revanche, nous avons vu que quelque soit la situation, `a` contient la valeur P. C'est précisément ce que nous voulions vérifier.

Notons que si la ligne 5 du programme avait été ainsi rédigée : `a = d + d;`, notre analyse aurait échoué, car elle n'aurait pas pu déterminer que cette instruction affecte nécessairement une valeur paire à `a`. En effet, à la ligne 5, `d` contient la valeur abstraite $\top$, si bien que la sémantique abstraite de + donne le résultat $\top$.

7.4.2 Cadre formel confortable

7.4.2.1 Univers et connexions de Galois

Les valeurs abstraites envisagées pour une analyse sont appelées propriétés abstraites, leur ensemble forme l'univers abstrait. De même que dans la section précédente, nous considérions $\top$ comme moins précis que P, nous avons besoin d'une relation d'ordre sur l'univers abstrait pour dénoter la précision relative des propriétés abstraites. Étant donné deux propriétés p et q, nous souhaitons que l'univers abstrait possède une propriété qui puisse tenir le rôle de « p ou q ». Cette propriété doit être moins précise que p (au-dessus de p, pour notre relation d'ordre), moins précise que q (au-dessus de q), mais plus précise que toute autre propriété qui est à la fois au-dessus de p et de q.

Finalement, il est souhaitable que toute paire de propriétés $\{p,q\}$ possède une borne supérieure pour la relation d'ordre. De même, il faut qu'elle possède une borne inférieure, qui joue le rôle de « p et q ». Ces considérations nous conduisent à imposer que l'univers abstrait soit un treillis, notions dont nous avons donné la définition page 140. On dispose dans ce cadre d'un opérateur borne supérieure et d'un opérateur borne inférieure.

Dans l'univers concret, on identifie chaque propriété (concrète) à l'ensemble des environnements qui vérifient cette propriété. Dans l'exemple de la section précédente, $2^{\mathbb{Z}\times\mathbb{Z}^4}$ est une propriété concrète. Plus généralement, les propriétés concrètes sont des parties de Env. Ainsi, l'univers concret est l'ensemble des parties de Env, $\mathcal{P}(Env)$.

Remarque 142. Ainsi défini, un univers concret est un treillis : il suffit de le munir de l'inclusion. L'opérateur borne supérieure est l'union $\cup$, l'opérateur borne inférieure est l'intersection $\cap$.

Si l'on veut pouvoir considérer la plus petite propriété qui subsume un ensemble de propriétés, on a besoin d'un treillis complet.

Définition 143 (Treillis complet).
Soit $(T,\sqsubseteq)$ un treillis. On dit qu'il est complet si toute partie non vide X de T admet une borne inférieure et une borne supérieure, notées respectivement $\bigsqcap X$ et $\bigsqcup X$. 143 ◀

Remarque 144. Un treillis complet non vide T admet nécessairement un plus petit élément, noté

$$\bot = \bigsqcap T,$$

et un plus grand élément, noté

$$\top = \bigsqcup T.$$

Deux treillis complets étant pris comme univers, l'un concret $\mathcal{U}$, l'autre abstrait $\mathcal{U}^\sharp$, il faut mettre en place une connexion entre eux, afin de pouvoir concrétiser les propriétés abstraites et abstraire les propriétés concrètes. On se donne à cette fin une fonction de concrétisation $\gamma : \mathcal{U}^\sharp \to \mathcal{U}$ et une fonction d'abstraction $\alpha : \mathcal{U} \to \mathcal{U}^\sharp$.

L'univers concret $\mathcal{U}$ est l'ensemble des parties $\mathcal{P}(Env)$ d'un certain ensemble d'environnements possibles, muni de la relation d'ordre de l'inclusion. C'est pourquoi on utilise des opérateurs « ronds » $(\cup, \subseteq)$ pour les objets concrets, et des opérateurs « carrés » $(\sqcup, \sqsubseteq)$ pour les objets abstraits.

La connexion entre $\mathcal{U}$ et $\mathcal{U}^\sharp$ ne peut pas être quelconque : elle doit respecter les relations d'ordre qui représentent la précision relative des différentes propriétés.

Dans l'univers concret, une petite propriété P (au sens de l'inclusion) est plus fine qu'une propriété Q qui la contient. Il est raisonnable de demander que, dans ce cas, le représentant de P dans l'univers abstrait soit plus petit que celui de Q. Autrement dit, il est souhaitable que α soit croissante.

De même, une propriété abstraite fine doit se concrétiser en une propriété concrète fine. Ainsi, γ aussi doit être croissante.

On souhaite que l'approximation effectuée soit correcte. Cela requiert notamment que, pour toute propriété concrète P, $\gamma \circ \alpha(P)$ ne doit pas contredire P. Ainsi, si un objet vérifie la propriété concrète P, il doit aussi vérifier $\gamma \circ \alpha(P)$. On doit donc avoir $\gamma \circ \alpha(P) \supseteq P$. Cette propriété, $\forall P \in \mathcal{U}$, $\gamma \circ \alpha(P) \supseteq P$, est appelée extensivité de $\gamma \circ \alpha$. On la note $\gamma \circ \alpha \dot{\supseteq} \mathrm{Id}_{\mathcal{U}}$.

On souhaite que α fournisse la meilleure approximation possible de chaque propriété concrète. Soit y une propriété abstraite et $P = \gamma(y)$ sa concrétisation. Par construction, y est une approximation possible de P, donc on doit avoir $\alpha(P) \sqsubseteq y$, *ie.*$\alpha \circ \gamma(y) \sqsubseteq y$. Cette propriété est appelée rétractivité de $\alpha \circ \gamma$.

Ces contraintes sont rassemblées dans la notion de connexion de Galois :

Définition 145 (Connexion de Galois).
On dit d'un couple (α, γ) formé d'une fonction d'abstraction et d'une fonction de concrétisation entre un univers concret $\mathcal{U}$ et un univers abstrait $\mathcal{U}^\sharp$ qu'il est une connexion de Galois si

(i) α est croissante,

(ii) γ est croissante,

(iii) $\gamma \circ \alpha \dot{\supseteq} \mathrm{Id}_{\mathcal{U}}$ (extensivité) et

(iv) $\alpha \circ \gamma \dot{\sqsubseteq} \mathrm{Id}_{\mathcal{U}^\sharp}$ (rétractivité). 145 ◀

Remarque 146. La notion de connexion de Galois est exprimée de manière équivalente comme suit. Soit $(T_1, \preccurlyeq_1)$ et $(T_2, \preccurlyeq_2)$ deux treillis complets. Soit $\alpha : T_1 \longrightarrow T_2$ et $\gamma : T_2 \longrightarrow T_1$. On dit que (α, γ) est une connexion de Galois entre T_1 et T_2 si pour tout $x_1 \in T_1$ et pour tout $x_2 \in T_2$, on a $\alpha(x_1) \preccurlyeq_2 x_2 \Leftrightarrow x_1 \preccurlyeq_1 \gamma(x_2)$.

La notion de connexion de Galois est assez contraignante : la définition de l'une des deux fonctions en jeu impose l'autre, comme l'énonce le lemme suivant.

Lemme 147.
Si (α, γ) forme une connexion de Galois entre $\mathcal{U}$ et $\mathcal{U}^\sharp$, alors pour tout $P \in \mathcal{U}$ (respectivement pour tout $p \in \mathcal{U}^\sharp$), on a

$$\alpha(P) = \bigsqcap \left\{ x \in \mathcal{U}^\sharp \;\middle|\; P \subseteq \gamma(x) \right\}$$

et

$$\gamma(p) = \bigcup \{ X \in \mathcal{U} \mid \alpha(X) \sqsubseteq p \}. \qquad 147 \blacksquare$$

7.4.2.2 Opérateurs et leurs abstractions

Après avoir abstrait les objets manipulés par les programmes, il faut abstraire les manipulations. Chaque étape d'exécution d'un programme transforme l'environnement courant. On s'intéresse donc aux fonctions de $\mathcal{U}$ dans $\mathcal{U}$, qu'on appelle des opérateurs. Pour chaque opérateur concret agissant sur les environnements, il faut construire un opérateur abstrait agissant sur les abstractions de ces environnements.

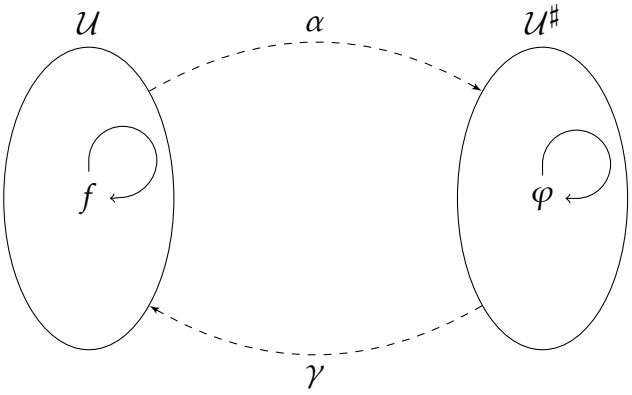

Figure 7.9 – Domaines, opérateurs concrets et abstraits

Soit $f : \mathcal{U} \longrightarrow \mathcal{U}$ un opérateur concret. On souhaite construire un opérateur abstrait $\varphi : \mathcal{U}^\sharp \longrightarrow \mathcal{U}^\sharp$ tel que pour toute approximation correcte x de P, $\varphi(x)$ soit une approximation correcte de $f(P)$.

Définition 148 (Approximation correcte).
Soit f un opérateur concret et φ un opérateur abstrait. On dit que φ est une approximation correcte de f si pour tout $P \in \mathcal{U}$ et pour tout $x \in \mathcal{U}^\sharp$ tel que $\alpha(P) \sqsubseteq x$, on a $\alpha(f(P)) \sqsubseteq \varphi(x)$ (ou, de manière équivalente : si $P \subseteq \gamma(x)$ alors $f(P) \subseteq \gamma(\varphi(x))$). 148 ◀

On dispose d'un théorème de caractérisation des approximations correctes :

Théorème 149.
Soit f un opérateur concret croissant et φ un opérateur abstrait croissant. On a équivalence entre

(i) φ est une approximation correcte de f,

(ii) $\alpha \circ f \dot{\sqsubseteq} \varphi \circ \alpha$,

(iii) $\alpha \circ f \circ \gamma \dot{\sqsubseteq} \varphi$ et

(iv) $f \circ \gamma \dot{\subseteq} \gamma \circ \varphi$. 149 ■

Corolaire 150.
Soit f un opérateur concret croissant. $\alpha \circ f \circ \gamma$ est une approximation correcte de f. De plus, c'est la meilleure approximation croissante de f.

En pratique, les opérateurs concrets que nous manipulerons seront croissants. Il est donc intéressant de définir la notion d'approximation optimale en se fondant sur le corolaire précédent.

Définition 151 (Approximation optimale).
Soit f un opérateur concret croissant. On dit que $\alpha \circ f \circ \gamma$ est l'approximation optimale de f, et on la note $f^{\mathcal{U}^\sharp}$. 151 ◀

7.4.3 Perspectives

Considérons le treillis complet $\mathcal{P}(\mathrm{T}(\Sigma))$ muni de l'inclusion ensembliste. Pour un système de réécriture R et un langage initial L_0, le langage des termes atteignables $R^*(L_0)$ peut être vu comme le plus petit point fixe de l'opérateur

$$F_R : \begin{array}{ccc} \mathcal{P}(\mathrm{T}(\Sigma)) & \longrightarrow & \mathcal{P}(\mathrm{T}(\Sigma)) \\ X & \longmapsto & L_0 \cup X \cup \{t \mid \exists s \in X, s \to_R t\} \end{array}.$$

Remarque 152. Pour tout langage L qui contient L_0, on a $F_R(L) = L \cup R(L)$.

Ce point fixe existe et vaut $\bigcup_{n\in\mathbb{N}} F_R^n(\varnothing)$, mais on ne peut en général le calculer par itérations successives en calculant les $F_R^n(\varnothing)$ car ce calcul ne termine pas ; la suite des itérés, croissante, n'étant pas stationnaire.

Cela n'a rien d'étonnant et c'est pourquoi on recourt à des abstractions. Des domaines abstraits à base d'automates ou de grammaires régulières sont décrites dans [CC95] et [GP02]. Étant donné une signature Σ, les automates sur cette signature sont naturellement ordonnés par la relation $\sqsubseteq$ qu'on définit ainsi

$$\mathcal{A}_1 \sqsubseteq \mathcal{A}_2 \Longleftrightarrow \mathscr{L}(\mathcal{A}_1) \subseteq \mathscr{L}(\mathcal{A}_2).$$

Cependant, il ne s'agit pas d'une relation d'ordre car il se peut que $\mathscr{L}(\mathcal{A}_1) = \mathscr{L}(\mathcal{A}_2)$ (et donc que $\mathcal{A}_1 \sqsubseteq \mathcal{A}_2$ et $\mathcal{A}_2 \sqsubseteq \mathcal{A}_1$) alors que $\mathcal{A}_1 \neq \mathcal{A}_2$, pour des raisons triviales de noms des états ou à cause d'une structure plus profondément différente entre ces deux automates. On peut lever ce problème de façon classique en quotientant l'ensemble des automates par la relation d'équivalence sous-jacente, ou plus explicitement en se restreignant à des automates minimaux (et en ignorant les problèmes de noms d'états). On dispose alors d'une structure de treillis : l'automate borne supérieure de $\mathcal{A}_1$ et $\mathcal{A}_2$ est celui reconnaissant l'union des deux langages, etc.

Il reste qu'un tel treillis n'est pas complet (l'ensemble des langages réguliers est stable par intersection finie mais pas par intersection ni union quelconque, comme on l'a observé en remarque 49 page 28) et de plus, en général, un calcul itératif ne terminera pas davantage que dans le domaine concret des termes clos.

Cela n'interdit pas de considérer ce domaine abstrait et la fonction de concrétisation

$$\gamma : \begin{array}{ccc} \text{automates sur } \Sigma & \longrightarrow & \mathcal{P}(\mathrm{T}(\Sigma)) \\ \mathcal{A} & \longmapsto & \mathscr{L}(\mathcal{A}) \end{array}$$

ainsi que l'opérateur abstrait

$$F_R^\sharp : \begin{array}{ccc} \text{automates sur } \Sigma & \longrightarrow & \text{automates sur } \Sigma \\ \mathcal{A} & \longmapsto & \mathcal{A}' \end{array},$$

où nous notons $\mathcal{A}'$ l'automate obtenu en appliquant une étape de complétion à $\mathcal{A}$.

Cet opérateur $F_R^\sharp$ est correct au sens de Cousot si l'on restreint le domaine abstrait aux seuls automates qui reconnaissent au moins L_0.

Théorème 153.
L'opérateur $F_R^\sharp$ restreint au domaine abstrait des seuls automates qui reconnaissent au moins L_0 est correct au sens de Cousot et Cousot dans [CC95, Section 6.2], c'est-à-dire que

$$F_R \circ \gamma \dot{\subseteq} \gamma \circ F_R^\sharp.$$ 153 ■

Démonstration.
Par définition, la relation

$$F_R \circ \gamma \dot{\subseteq} \gamma \circ F_R^\sharp$$

signifie : pour tout automate $\mathcal{A}$ du domaine abstrait,

$$F_R(\gamma(\mathcal{A})) \subseteq \gamma(F_R^\sharp(\mathcal{A})),$$

relation équivalente, par définition de γ, à

$$F_R(\mathscr{L}(\mathcal{A})) \subseteq \mathscr{L}(F_R^\sharp(\mathcal{A})).$$

Or, une étape de complétion ajoute des transitions à l'automate, donc on a toujours $\mathscr{L}(F_R^\sharp(\mathcal{A})) \supseteq \mathscr{L}(\mathcal{A})$, et comme toutes les paires critiques de $\mathcal{A}$ sont complétées, on a aussi $\mathscr{L}(F_R^\sharp(\mathcal{A})) \supseteq R(\mathscr{L}(\mathcal{A}))$. Comme la restriction du domaine apporte $\mathscr{L}(\mathcal{A}) \supseteq L_0$, et donc aussi $\mathscr{L}(F_R^\sharp(\mathcal{A})) \supseteq L_0$, on a bien $\mathscr{L}(F_R^\sharp(\mathcal{A})) \supseteq L_0 \cup \mathscr{L}(\mathcal{A}) \cup R(\mathscr{L}(\mathcal{A}))$, ce dernier ensemble n'étant autre que $F_R(\mathscr{L}(\mathcal{A}))$. □

$F_R^\sharp$ apparait donc comme une abstraction correcte de F_R dans le domaine restreint et la complétion d'automate comme le calcul d'un point fixe de cet opérateur. Le théorème 85 s'interprète alors comme le fait que le langage reconnu par ce point-fixe est une abstraction correcte de $R^*(L_0)$, ce qui est conforme à la théorie de l'interprétation abstraite.

Ce résultat est suffisant pour se raccrocher à la théorie de l'interprétation abstraite, ce qui est intéressant car elle permet d'exprimer des combinaisons d'analyses variées ; on pourrait ainsi recueillir les fruits de plusieurs méthodes d'analyse combinées.

Il est par ailleurs agréable de constater que dans le cas où la complétion termine, on peut exprimer une connexion de Galois sur un domaine restreint.

On peut se restreindre à un domaine dépendant de R comme envisagé dans [CC95, Section 6.3] (mais pas à un domaine fini, car cela voudrait dire que la complétion d'automate termine toujours). En effet, R étant fixé, les automates minimaux sur la signature Σ ne sont pas tous susceptibles d'être produits par le processus de complétion. Les seules transitions ajoutées, outre les epsilon-transitions, sont celles produites par la normalisation des configurations. Ces configurations sont nécessairement des sous-termes ou portions de sous-termes des membres droits des règles de R.

Pour les systèmes R tels que la complétion termine, ce domaine est fini et donc complet. Dans ce cas, soit donc $\mathcal{U}_R^\sharp$ l'ensemble (quotienté par les relations d'équivalences techniques) des automates sur la signature Σ dont le langage contient L_0 et dont les configurations sont, outre celles nécessaires à la reconnaissance de L_0, issues des sous-termes des membres droits de R. On conserve la définition antérieure

de γ et on définit

$$\alpha : \begin{array}{rcl} \mathcal{P}(\mathrm{T}(\Sigma)) & \longrightarrow & \mathcal{U}_R^\sharp \\ L & \longmapsto & \bigsqcap \left\{ \mathcal{A} \in \mathcal{U}_R^\sharp \;\middle|\; L \subseteq \mathscr{L}(\mathcal{A}) \right\} \end{array}.$$

On dispose alors d'une connexion de Galois.

8 Conclusion

Nous nous sommes intéressés au problème de la vérification des programmes. Plus précisément, nous nous sommes concentrés sur les possibilités d'améliorer la précision d'analyses de programmes fonctionnels en tenant compte de la stratégie d'évaluation.

Pour la stratégie en profondeur, nous proposons une adaptation de la méthode d'analyse par complétion d'automate d'arbres. Cette adaptation fournit des résultats intéressants, même si elle impose de calculer l'automate $\mathcal{IRR}(R)$ des formes normales. Des améliorations peuvent être envisagées pour ne pas calculer explicitement tous les états de cet automate. On peut également réfléchir à des adaptations pour d'autres stratégies.

Une autre méthode pour tenir compte des stratégies est de transformer le programme initial pour encoder ce comportement. Cette méthode, développée de façon systématique dans [CLM15] est complémentaire à la nôtre et les principes sous-tendant son élaboration peuvent nourrir les améliorations de notre méthode envisagées ci-dessus. Certains des défis qui restent à relever sont communs à notre méthode et à celle de [CLM15]. En particulier, le nombre élevé d'états de $\mathcal{IRR}(R)$ est analogue à l'explosion du nombre de règles dans les systèmes de [CLM15] due à l'explicitation des anti-termes (constructions qui dénotent tous les termes de $\mathrm{T}(\Sigma)$ qui ne sont pas d'une certaine forme), explosion dont la limitation est également signalée par les auteurs comme perspective d'amélioration.

Nous avons également présenté comment notre adaptation s'inscrit dans le paysage des variantes de la complétion d'automate. Il reste à intégrer ces différentes versions éparses. Établir des liens entre la complétion d'automate et l'interprétation abstraite permet également d'envisager des bénéfices pour l'établissement de résultats de correction et précision grâce aux théorèmes et plus généralement au cadre formel de cette théorie.

Bibliographie

[BH08] Yohan Boichut et Pierre-Cyrille Héam. « A theoretical limit for the safety verification techniques with resular fix-point compuations ». In : *Information Processing Letters* 108 (2008), p. 1–2.

[BN98] Franz Baader et Tobias Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

[Boi+07] Yohan Boichut et al. « Rewriting Approximations for Fast Prototyping of Static Analyzers ». In : *RTA*. T. 4533. LNCS. Springer, 2007, p. 48–62.

[Boy10] Benoît Boyer. « Réécriture d'automates certifiée pour la vérification de modèles ». Thèse de doctorat. Université de Rennes 1, 2010.

[CC77] Patrick Cousot et Radhia Cousot. « Abstract interpretation : a unified lattice model for static analysis of programs by construction or approximation of fixpoints ». In : *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. POPL '77. Los Angeles, California : ACM, 1977, p. 238–252. URL : http://doi.acm.org/10.1145/512950.512973.

[CC95] Patrick Cousot et Radhia Cousot. *Formal Language, Grammar and Set-Constraint-Based Program Analysis by Abstract Interpretation*. 1995.

[CJ03] Hubert Comon et Florent Jacquemard. « Ground reducibility is EXPTIME-complete ». In : *Inf. Comput.* 187.1 (2003), p. 123–153.

[CLM15] Horatiu Cirstea, Serguei Lenglet et Pierre-Etienne Moreau. « A faithful encoding of programmable strategies into term rewriting systems ». In : *26th International Conference on Rewriting Techniques and Applications (RTA 2015)*. Sous la dir. de Maribel Fernández. T. 36. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany : Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2015, p. 74–88. ISBN : 9783939897859. DOI : http://dx.doi.org/10.4230/LIPIcs.RTA.2015.74. URL : http://drops.dagstuhl.de/opus/volltexte/2015/5190.

[Com+08] Hubert Comon et al. *Tree Automata Techniques and Applications*. 2008.

[Com13] Hubert Comon. *Complexité du calcul de l'automate des formes normales*. Communication privée. 2013.

[Coq+91] Jean-Luc Coquidé et al. « Bottom-up tree pushdown automata and rewrite systems ». In : *Rewriting Techniques and Applications*. Sous la dir. de Ronald Book. T. 488. Lecture Notes in Computer Science. 10.1007/3-540-53904-2_104. Springer Berlin / Heidelberg, 1991, p. 287–298. URL : http://dx.doi.org/10.1007/3-540-53904-2_104.

[Coq15] Équipe de développement de Coq. *Assistant de preuve et langage Coq*. 2015. URL : https://coq.inria.fr/.

[Cou+15] Patrick Cousot et al. *Analyseur statique de logiciels temps-réel embarqués (ASTRÉE)*. 2015. URL : http://www.astree.ens.fr/.

[FGT04] Guillaume Feuillade, Thomas Genet et Valérie Viet Triem Tong. « Reachability Analysis over Term Rewriting Systems ». In : *J. Autom. Reasoning* 33.3-4 (2004), p. 341–383. DOI : 10.1007/s10817-004-6246-0. URL : http://dx.doi.org/10.1007/s10817-004-6246-0.

[FJ15] Matteo Frigo et Steven G. Johnson. *The Fastest Fourier Transform in the West*. 2015. URL : http://www.fftw.org/.

[Fre92] Gottlob Frege. « Über Sinn und Bedeutung ». In : *Zeitschrift für Philosophie und philosophische Kritik* 100 (1892), p. 25–50. Traduit dans [Fre94].

[Fre94] Gottlob Frege. *Écrits logiques et philosophiques*. Trad. par Claude Imbert. Seuil, 1994.

[GB85] Jean H. Gallier et Ronald V. Book. « Reductions in Tree Replacement Systems ». In : *Theor. Comput. Sci.* 37 (1985), p. 123–150.

[Gen+14] Thomas Genet et al. *Timbuk 3 – A Tree Automata Completion Tool*. IRISA / Université de Rennes 1. http://www.irisa.fr/celtique/genet/timbuk/. 2008–2014.

[Gen09] Thomas Genet. « Reachability analysis of rewriting for Software verification ». Thèse d'habilitation à diriger des recherches. Université de Rennes 1, 2009.

[Gen14] Thomas Genet. *A Note on the Precision of the Tree Automata Completion*. Research Report. IRISA, déc. 2014, p. 13. URL : https://hal.inria.fr/hal-01091393.

[Gen15] Thomas Genet. « Termination criteria for tree automata completion ». In : *Journal of Logical and Algebraic Methods in Programming* (2015), ISSN : 2352-2208. DOI : http://dx.doi.org/10.1016/j.jlamp.2015.05.003. URL : http://www.sciencedirect.com/science/article/pii/S2352220815000504.

[Gen97] Thomas Genet. « Decidable Approximations of Sets of Descendants and Sets of Normal Forms ». In : *Proc. 9th RTA Conf., Tsukuba (Japan), volume 1379 of LNCS*. Springer-Verlag, 1997, p. 151–165.

[GGJ08] Adria Gascon, Guillem Godoy et Florent Jacquemard. « Closure of Tree Automata Languages under Innermost Rewriting ». Anglais. In : *8th International Workshop on Reduction Strategies in Rewriting and Programming (WRS)*. T. 237. ENTCS. Hagenberg, Autriche : Elsevier, 2008, p. 23–38. doi : 10.1016/j.entcs.2009.03.033. url : http://hal.inria.fr/inria-00578966.

[GK00] Thomas Genet et Francis Klay. *Rewriting for Cryptographic Protocol Verification (extended version)*. Technical Report 3921. INRIA, 2000.

[GKV15] Thomas Genet, Barbara Kordy et Amaury Vansyngel. « Vers un outil de vérification formelle légere pour OCaml ». In : *AFADL*. Bordeaux, France, 2015, p. 6. url : https://hal.inria.fr/hal-01194538.

[GP02] John P. Gallagher et Germán Puebla. « Abstract Interpretation over Non-Deterministic Finite Tree Automata for Set-Based Analysis of Logic Programs ». In : *In Fourth International Symposium on Practical Aspects of Declarative Languages, number 2257 in LNCS*. Springer-Verlag, 2002, p. 243–261.

[GR10] Thomas Genet et Vlad Rusu. « Equational Tree Automata Completion ». In : *Journal of Symbolic Computation* 45 (2010), p. 574–597.

[GS12] Thomas Genet et Yann Salmon. « Proving Reachability Properties on Term Rewriting Systems with Strategies ». In : *International Workshop on Strategies*. Manchester, Royaume-Uni, 2012.

[GS15] Thomas Genet et Yann Salmon. « Reachability Analysis of Innermost Rewriting ». In : *26th International Conference on Rewriting Techniques and Applications (RTA 2015)*. Sous la dir. de Maribel Fernández. T. 36. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany : Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2015, p. 177–193. isbn : 9783939897859. doi : http://dx.doi.org/10.4230/LIPIcs.RTA.2015.177. url : http://drops.dagstuhl.de/opus/volltexte/2015/5196.

[GT95] Rémy Gilleron et Sophie Tison. « Regular Tree Languages and Rewrite Systems ». In : *Fundamenta informaticae* 24 (1995), p. 157–175.

[GV01] Thomas Genet et Valérie Viet Triem Tong. *Timbuk – A Tree Automata Library*. IRISA / Université de Rennes 1. http://www.irisa.fr/celtique/genet/timbuk/. 2001.

[JM79] Neil D. Jones et Steven S. Muchnick. « Flow Analysis and Optimization of LISP-like structures ». In : *POPL*. 1979, p. 244–256.

[Jon+14] Simon Peyton Jones et al. *The Haskell Programming Language*. http://www.haskell.org. 1990–2014.

[Jou05] Jean-Pierre Jouannaud. « Higher-Order Rewriting : Framework, Confluence and Termination ». English. In : *Processes, Terms and Cycles : Steps on the Road to Infinity*. Sous la dir. d'Aart Middeldorp et al. T. 3838. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2005, p. 224–250. isbn : 9783540309116. doi : 10.1007/11601548_14. url : http://dx.doi.org/10.1007/11601548_14.

[JRR] Jean-Pierre Jouannaud, Femke van Rammsdonk et Albert Rubio. *Higher-order Rewriting with Types and Arities*. url : http://www.lix.polytechnique.fr/~jouannaud/articles/horta.pdf.

[Kra08] Alexander Krauss. « Pattern Minimization Problems over Recursive Data Types ». In : *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*. ICFP '08. Victoria, BC, Canada : ACM, 2008, p. 267–274. isbn : 9781595939197. doi : 10.1145/1411204.1411242. url : http://doi.acm.org/10.1145/1411204.1411242.

[Ler+14] Xavier Leroy et al. *The Objective Caml system*. INRIA. http://caml.inria.fr/ocaml/ 1996–2014.

[Mur14] Valérie Murat. « Tree automata extensions for verification of infinite states systems ». Theses. Université Rennes 1, juin 2014. url : https://tel.archives-ouvertes.fr/tel-01065696.

[OR11] C.-H. Luke Ong et Steven J. Ramsay. « Verifying Higher-Order Functional Programs with Pattern Matching Algebraic Data Types ». In : *Proceedings of the 38th ACM SIGACT-SIGPLAN Symposium on the Principles of Programming Languages (POPL '11)*. 2011.

[Pie15] Benjamin C. Pierce. *Unison*. 2015. url : http://www.cis.upenn.edu/~bcpierce/unison/.

[Plo75] Gordon D. Plotkin. « Call-by-Name, Call-by-Value and the lambda-Calculus ». In : *Theor. Comput. Sci.* 1.2 (1975), p. 125–159.

[Rét99] Pierre Réty. « Regular Sets of Descendants for Constructor-Based Rewrite Systems ». In : *Logic for Programming and Automated Reasoning*. Sous la dir. d'Harald Ganzinger, David McAllester et Andrei Voronkov. T. 1705. Lecture Notes in Computer Science. 10.1007/3-540-48242-3_10. Springer Berlin / Heidelberg, 1999, p. 148–160. url : http://dx.doi.org/10.1007/3-540-48242-3_10.

[Ric53] Henry Gordon Rice. « Classes of Recursively Enumerable Sets and Their Decision Problems ». In : *Transactions of the American Mathematical Society* 74.2 (1953), p. 358–366. url : http://www.jstor.org/stable/1990888.

[RV02] Pierre Réty et Julie Vuotto. « Regular Sets of Descendants by some Rewrite Strategies ». In : *Proc. 13th RTA Conf., Copenhagen (Denmark)*. T. 2378. LNCS. Springer-Verlag, 2002.

[Sal88] Kai Salomaa. « Deterministic Tree Pushdown Automata and Monadic Tree Rewriting Systems ». In : *J. Comput. Syst. Sci.* 37.3 (1988), p. 367–394. doi : 10.1016/0022-0000(88)90014-1. url : http://dx.doi.org/10.1016/0022-0000(88)90014-1.

[Sch86] David A. Schmidt. *Denotational semantics : a methodology for language development*. Boston, London : Allyn et Bacon, 1986. isbn : 0205089747.

[Tak04] Toshinori Takai. *A Verification Technique Using Term Rewriting Systems and Abstract Interpretation*. Rapport. Japan Science et Technology Corporation, 2004.

[Ter03] Terese. *Term Rewriting Systems*. Cambridge University Press, 2003.

[Toy05] Yoshihito Toyama. « Reduction Strategies for Left-Linear Term Rewriting Systems ». English. In : *Processes, Terms and Cycles : Steps on the Road to Infinity*. Sous la dir. d'Aart Middeldorp et al. T. 3838. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2005, p. 198–223. isbn : 9783540309116. doi : 10.1007/11601548_13. url : http://dx.doi.org/10.1007/11601548_13.

[Tur36] Alan Mathison Turing. « On Computable Numbers, with an Application to the Entscheidungsproblem ». In : *Proceedings of the London Mathematical Society. Second Series* 42 (1936), p. 230–265. issn : 0024-6115 (print), 1460-244X (electronic).

[Win93] Glynn Winskel. *The formal semantics of programming languages : an introduction*. Foundations of computing. Cambridge (Mass.), London : MIT Press, 1993. isbn : 0262231697.

VU :
Le directeur de thèse
Thomas Genet

VU :
Le responsable de l'École doctorale

VU pour autorisation de soutenance
Rennes, le

Le Président de l'Université Rennes 1

Guy Cathelineau

VU après soutenance pour autorisation de publication

Le Président de jury
(nom et prénom)