Le calcul de réécriture

Horatiu Cirstea

Nancy-Université
Université Nancy 2

Département de formation doctorale en informatique

École doctorale IAEM Lorraine

Le calcul de réécriture

Habilitation à Diriger des Recherches

présentée et soutenue publiquement le 7 octobre 2010

par

Horatiu Cirstea

Composition du jury

Président : Furio Honsell, Professeur, University of Udine, Udine (Italie)

Rapporteurs : Delia Kesner, Professeur, Université Paris Diderot - Paris 7, Paris
Femke van Raamsdonk, Professeur, Vrije Universiteit, Amsterdam (Pays-Bas)

Examinateurs : Maribel Fernández, Professeur, King's College, London (UK)
Claude Kirchner, Directeur de recherche, INRIA, Bordeaux
Luigi Liquori, Directeur de recherche, INRIA, Sophia Antipolis
Dominique Méry, Professeur, Université Henri Poincaré, Nancy

Laboratoire Lorrain de Recherche en Informatique et ses Applications — UMR 7503

Loria

Sommaire

1

Introduction

La notion de forme ou de motif est fondamentale dans la plupart des activités humaines. Ce concept est connu et inconsciemment employé par chacun d'entre nous depuis l'enfance mais il a été étudié formellement seulement depuis le siècle dernier.

Prenons par exemple un des jouets préférés des petits enfants, la boîte à formes. Ce jeu consiste en un ensemble de pièces de différentes formes géométriques (habituellement, un prisme rectangulaire, un prisme triangulaire, et un cylindre) et une boîte qui a plusieurs trous avec des formes (habituellement, un carré, un triangle, et un cercle) correspondant aux formes des pièces et l'objectif est d'insérer les pièces dans la boîte. D'habitude, la boîte est conçue pour permettre l'insertion d'une seule pièce dans chaque trou mais parfois, une même pièce peut être insérée dans différents trous. La boîte peut également avoir plusieurs trous de la même forme, induisant encore plus de choix sur le trou qui pourrait être choisi. On est donc confronté à un problème de discrimination par rapport à des motifs (ou filtres) ou, autrement dit, à un problème de filtrage. En effet, ce type de jeu a été pour la plupart d'entre nous la première expérience avec le concept de filtrage.

La notion de motif est en fait présente dans tous les aspects de la vie. On utilise les motifs pour analyser les informations que l'on voit ou que l'on entend lorsqu'on lit un texte ou lorsqu'on écoute de la musique, pour prendre des décisions lorsqu'on conduit une voiture ou pour réaliser des raisonnements (par cas) dans les démonstrations mathématiques. Par exemple, lors de la lecture d'un texte, on ne lit pas caractère par caractère mais on identifie des mots en général et même, des morceaux entiers de phrase. De la même façon, l'identification d'une chanson est réalisée en filtrant la mélodie par rapport à des accords et éventuellement par rapport à un thème mais les motifs recherchés sont rarement des notes musicales. Le parcours d'un itinéraire suppose l'identification d'un motif (un plan) obtenu à partir d'une carte ou d'un dispositif GPS dans un paysage ou par rapport à une structure routière concrète.

Les mathématiques utilisent pleinement la notion de motif, certains particulièrement élaborés. Par exemple, pour résoudre l'équation $4u^2+4u+1 = 81$ il est utile de reconnaître l'expression $4u^2 + 4u + 1$ comme une instance de la forme expansée de l'expression $(x+y)^2$. Bien que ce dernier problème soit évidemment plus complexe que la boîte à formes, les mêmes notions de motif et de filtrage sont utilisés dans les deux cas.

En informatique, l'utilisation de motifs permet une représentation proche de notre façon de penser pour les problèmes à résoudre ainsi que pour les calculs et les raisonnements correspondants. Les motifs ont été utilisés initialement dans les éditeurs des textes pour la recherche des expressions régulières à la place des simples recherches par mot [DL67]. Le filtrage sur les séquences (de caractères) a été introduit dans des langages de programmation dédiés au traitement des chaînes de caractères comme SNOBOL [Gim73] et un filtrage Ssur des structures d'arbre a été ajouté aux versions initiales du langage LISP [McB70]. Le filtrage est également utilisé couramment dans la programmation fonctionnelle (par exemple, ML [GMW79, LDG+04], Haskell [HF92]), dans la programmation logique (par exemple, Prolog [Rou75]), dans la programmation basée sur la réécriture (par exemple, ASF+SDF [Kli93], Elan [Vit94], Maude [CELM96, CDE+07], Obj* [GKK+87], Stratego [Vis99]) et plus récemment intégré dans des langages existants (par exemple, Tom [MRV03], JMatch [LM03]).

Le filtrage n'est bien évidement pas un but en lui-même : le filtrage est généralement réalisé lorsqu'on veut effectuer une action. Typiquement, quand on détecte qu'un certain nombre est additionné à zéro ou, autrement dit, quand on trouve un motif de la forme $x + 0$ dans une expression arithmétique, on veut simplifier l'expression en remplaçant toute instance de $x + 0$ par instance correspondante de x. L'action peut affecter l'expression filtrée comme, par exemple, quand $7 + 0$ est remplacé par 7, ou peut avoir juste un « effet de bord ». Par exemple, si le filtrage détecte une expression de la forme $7/0$ alors une exception peut être levée. Ce type de transformation peut être naturellement décrite en utilisant la réécriture qui consiste, intuitivement, à filtrer un objet et à le remplacer (partiellement) par un autre.

La réécriture Une approche naturelle pour la spécification des actions à effectuer sur les objets est l'utilisation de règles de réécriture, c'est-à-dire des entités de la forme $l \to r$ qui indiquent qu'une instance de l devrait être remplacée par l'instance correspondante de r. Des actions multiples peuvent être spécifiées par les systèmes de réécriture qui sont des ensembles de règles de réécriture. Par exemple, le système de réécriture comprenant les règles

$$0 + x \to x \tag{1.1}$$

$$s(y) + x \to s(y + x) \tag{1.2}$$

spécifie d'une façon claire et très proche de la définition originale [Pea89] les actions atomiques définissant l'addition sur les entiers de Peano.

Les règles d'un système de réécriture spécifient explicitement la forme des objets réductibles et la façon dont ils sont réécrits mais ne précisent pas comment gérer les conflits potentiels liés au choix de la règle à appliquer ou de la position d'application. Par exemple, les deux règles ci-dessus peuvent être appliquées à différentes positions du terme $s(0) + (0 + s(0))$ et les résultats des réductions correspondantes sont différents. Il faut donc non seulement décider quelle règle devrait être appliquée et à quelle position mais une autre question se pose naturellement dans ce contexte : quelle action doit être effectuée après l'application d'une règle ? Doit-on considérer que le résultat obtenu est définitif ou essayer d'appliquer la même règle ou les autres règles du système ?

Toutes ces questions sont au cœur de l'étude des systèmes de réécriture où, en général, une relation de réécriture est définie et étudiée [HO80, BN98, KK99, Ter03]. Quand le système de réécriture est confluent (garantissant que les réductions correspondantes sont joignables) et fortement normalisant (garantissant que toute réduction termine), la notion de forme normale permet de donner la sémantique de l'application d'un ensemble de règles de réécriture. Par exemple, les règles ci-dessus donnent la sémantique de l'opérateur + et la forme normale du terme $s(0) + (0 + s(0))$ par rapport à ces règles est $s(s(0))$.

Dans ce formalisme, l'application d'une règle de réécriture est néanmoins implicite et le mécanisme global d'application d'un système de règles devient, bien que tout à fait précis, relativement technique. Par ailleurs, puisque le processus d'application d'une règle n'est pas visible dans le formalisme, il n'est pas possible de donner une description explicite des stratégies de réécriture et les quelques tentatives dans cette direction sont des fonctionnalités supplémentaires ajoutés aux cadres existants comme, par exemple, les stratégies locales d'Obj* ou les stratégies plus générales de Elan, Maude, Stratego ou Tom.

Systèmes d'ordre supérieur L'application est une opération explicite du λ-calcul, le formalisme bien connu et largement étudié introduit par Church dans les années 40 [Chu41]. D'autre part, il est surprenant que le λ-calcul, un des modèles de calcul les plus utilisés, utilise seulement un mécanisme de filtrage trivial et il faut noter que l'absence de mécanismes de filtrage plus élaborés dans le λ-calcul mène à des encodages complexes pour des notions classiques. Par exemple, un nombre entier n peut être représenté par le λ-terme $\lambda f.\lambda x.f(f(\ldots x))$ où f apparaît n fois dans la partie droite de l'abstraction et l'addition peut être encodée par le terme $n+m \triangleq \lambda n.\lambda m.\lambda f.\lambda x.nf(mfx)$. Il est clair qu'il serait intéressant de disposer dans un formalisme comme le λ-calcul d'une spécification explicite des naturels et des opérations arithmétiques exprimées d'une façon plus intuitive.

En effet, S. Peyton-Jones définit une généralisation du λ-calcul avec du filtrage et propose ainsi un paradigme incluant de manière primitive le fil-

trage [PJ87] comme un fondement théorique pour les langages fonctionnels. A partir des années 90, d'autres calculs avec motifs sont introduits soit dans le cadre de la programmation fonctionnelle (le λ-calcul avec motifs [vO90], le calcul basique de filtrage [Kah03], le calcul à motifs purs [Jay04, JK06] et le λ-calcul avec constructeurs [AMR06]) soit pour donner une sémantique aux langages à base de règles (le calcul de réécriture [CK01]).

Le calcul de réécriture Nous avons introduit et étudié le calcul de réécriture, appelé aussi ρ-calcul, un formalisme où les notions de filtrage, de règle de réécriture, d'application de règle et les résultats de ces applications sont définis au même niveau. Nous disposons donc d'un formalisme où les ingrédients de la réécriture et du λ-calcul sont intégrés d'une manière uniforme, précise et atomique.

Dans ma thèse [Cir00] j'ai montré que l'intégration uniforme des mécanismes de bases de la réécriture et du λ-calcul permet une définition précise des stratégies de réécriture utilisées dans les langages basés sur la réécriture. J'ai continué l'étude de ce calcul et les recherches menées en collaboration avec différents chercheurs ont conduit ces dix dernières années à des développements et des résultats très fructueux comprenant :

- la définition des encodages de calculs à objets [CKL01a] et des encodages simplifiés de stratégies classiques de parcours de termes [CHW07] ;
- des systèmes de types dépendants [CKL01b, BCKL03] étudiés dans le contexte des assistants à la démonstration aussi bien que des systèmes de types [CLW03, CKLW06] développés dans le contexte des langages de programmation à base de règles et stratégies ;
- la définition d'un formalisme pour la réécriture d'ordre supérieur et l'étude de la relation avec des formalismes similaires et en particulier avec les CRSs [BCK06] ;
- la définition des extensions avec des mécanismes de filtrage et substitutions explicites [CFK07] ;
- la définition d'une extension avec des graphes [BBCK06] permettant le partage et la définition de structures de données cycliques.

D'autres aspects du calcul de réécriture ont été étudié indépendemment et on peut citer, en particulier, l'intégration des mécanismes d'exceptions dans le calcul [FK02, CKL02], des extensions du calcul avec des traits impératifs [LS04, LS08], des stratégies de réduction pour le calcul de réécriture [FMS06] ainsi que l'étude des notions de modèles généralisant ceux du λ-calcul [Fau07].

Ce manuscrit se focalise sur la présentation du calcul de réécriture comme un formalisme théorique permettant de donner la sémantique dynamique et statique de toute une famille de langages basés sur le filtrage, les règles et les stratégies de réécriture. Le ρ-calcul a plusieurs paramètres ce qui nous fait

parler *du calcul* quand on abstrait par rapport à ses paramètres et donner des noms spécifiques aux calculs obtenus en précisant, par exemple, le filtrage utilisé ou en ajoutant des fonctionnalités supplémentaires comme l'échec explicite de filtrage. Nous présentons ici le calcul de réécriture général ainsi que plusieurs instances et extensions. Nous mettons l'accent sur l'expressivité et sur les propriétés de ces calculs. Plus précisément, nous montrons comment le calcul de réécriture peut être utilisé pour expliciter le processus de réécriture (sous des stratégies) et nous présentons des encodages (typés ou non) de différents formalismes similaires. La propriété qui est montrée pour tous ces calculs est la *confluence* et nous fournissons les schémas des preuves qui deviennent particulièrement élaborées quand le filtrage est non trivial ainsi qu'une preuve de confluence générique qui isole les conditions suffisantes en axiomatisant la manière dont le filtrage est réalisé.

Le chapitre 2 introduit le calcul de réécriture général et illustre son *expressivité*. Nous nous intéressons dans ce chapitre à deux variantes du calcul général et nous présentons les résultats de confluence correspondants. L'expressivité de ces calculs est illustrée à travers des encodages de la réécriture avec stratégies. Nous discutons brièvement les différentes présentations du calcul de réécriture utilisées dans la littérature depuis son introduction.

Dans le chapitre 3 on s'intéresse aux aspects liés au *typage* dans le calcul de réécriture et nous nous focalisons seulement sur les systèmes de types développés dans l'optique d'un formalisme théorique pour le typage dans des langages de programmation à base de règles. En particulier, nous montrons que les encodages présentés dans le chapitre 2 sont bien typés. Nous prouvons également que les propriétés usuelles sont vérifiées par les différents versions typées du calcul. Nous considérons ces systèmes de types comme de bons candidats pour donner la sémantique statique de toute une famille de langages basés sur la réécriture.

Le chapitre 4 introduit progressivement un calcul de réécriture qui manipule *explicitement* les contraintes de filtrage et l'application des substitutions obtenues comme résultat. Le calcul proposé est modulaire dans le sens où il peut être adapté à différentes théories de filtrage et cette modularité se situe non seulement au niveau de la conception mais également au niveau de la preuve de confluence qui est relativement complexe mais isole clairement les parties concernées par le filtrage. Nous considérons que ces calculs avec substitutions explicites sont particulièrement appropriés comme support théorique pour les implantations des langages basés sur la réécriture.

On s'intéresse ensuite dans le chapitre 5 à une extension du calcul de réécriture qui utilise non seulement les contraintes de filtrage du calcul présentées dans le chapitre 4 mais également des contraintes d'unification. Le formalisme ainsi obtenu bénéficie non seulement des avantages du calcul de réécriture général mais permet également la manipulation de *graphes* avec du partage et des cycles. Il généralise le λ-calcul cyclique et permet un encodage des réductions des systèmes de réécriture de termes-graphes. Le formalisme

présenté ici met en évidence des techniques d'optimisation pour des implantations basées sur le calcul de réécriture et suggère des extensions possibles pour améliorer l'expressivité de langages à base de règles.

L'objectif du chapitre 6 est d'établir une méthodologie pour prouver la *confluence* des différentes versions du calcul de réécriture mais également des autres calculs à motifs. Nous introduisons un calcul d'ordre-supérieur avec motifs et nous montrons que les principaux calculs à motifs introduits dans la littérature ont des formulations alternatives dans le formalisme général. La confluence de ce formalisme général est obtenue en imposant certaines conditions de stabilité pour la fonction de filtrage sous-jacente et nous montrons que tous les calculs à motifs étudiés satisfont ces conditions. Nous fournissons ainsi un formalisme général permettant d'exprimer différents calculs à motif et une méthodologie qui met en évidence les points clés permettant d'obtenir la confluence de ces calculs.

Ce document présente seulement une partie de mes travaux de recherche réalisés depuis ma thèse. Pour terminer cette présentation, je discuterai brièvement les autres thèmes abordés pendant ces dix dernières années et je donnerai des perspectives de recherche émergeant de tous ces travaux.

Contexte :

Les notions de filtrage et de réécriture sont omniprésentes en informatique et elles apparaissent dès les fondements théoriques jusqu'aux réalisations logicielles. La réécriture a été utilisée pour définir la sémantique opérationnelle de langages de programmation, mais également pour décrire par des règles d'inférence des logiques, des prouveurs de théorèmes ou des solveurs de contraintes. Il n'est donc pas surprenant que de nombreux langages se sont construits autour de cette notion de réécriture.

Bien que la réécriture soit un formalisme Turing complet, les langages de programmation reposant sur ce concept ont introduit des extensions permettant de les rendre encore plus expressifs. La sémantique complète de ces langages devient donc plus élaborée et est souvent exprimée en utilisant plusieurs formalismes.

Je me suis donc intéressé à l'étude d'un formalisme permettant d'exprimer d'une manière uniforme la sémantique des langages à base de règles et stratégies et j'ai essayé de répondre à plusieurs questions dans ce contexte.

Questions :

- Quel formalisme pour expliciter les mécanismes de la réécriture avec stratégies ?
- Quel est son pouvoir d'expression ?
- Quelles sont ses propriétés ?

Contributions :

J'ai proposé le calcul de réécriture, un formalisme permettant de décrire l'application de règles de réécriture et de représenter les résultats obtenus.

Nous avons montré que la définition des règles de réécriture et des ensembles de résultats au niveau objet de ce calcul nous permet de modéliser l'exécution des règles et stratégies de réécriture. Nous avons également montré que l'expressivité du calcul va au-delà de la sémantique des langages à base de règles et permet, en particulier, un encodage relativement simple de λ-calculs à objets. Nous ne présentons pas dans ce document ces encodages décrits par ailleurs mais nous montrons un encodage des systèmes de réécriture guidés par des stratégies classiques. Ces études d'expressivité du calcul de réécriture ont permis non seulement de donner une sémantique complète au langage Elan mais ont également conduit au développement du langages de stratégies du langage TOM.

Le calcul n'est ni confluent ni terminant quand aucune discipline n'est imposée sur la constructions des termes. Nous avons proposé des restrictions (syntaxiques) permettant d'obtenir la confluence pour le calcul de base et pour des instances permettant une manipulation explicite de l'échec (de filtrage).

2

Le calcul de réécriture de base

Le calcul de réécriture, également appelé ρ-calcul, est un formalisme d'ordre supérieur qui hérite du λ-calcul ses fonctionnalités d'ordre supérieur : le traitement explicite des fonctions et leur application. Pour exprimer des règles de réécriture, la λ-abstraction est généralisée à une abstraction sur des motifs plus élaborés qu'une simple variable et l'évaluation est basée sur le *filtrage sur les motifs* qui peut être réalisé syntaxiquement ou modulo une théorie équationnelle. Dans le dernier cas, elle peut donner plusieurs résultats qui sont groupés, dans le ρ-calcul, dans des *collections de termes*, permettant ainsi un traitement naturel du non-déterminisme intrinsèque à la réécriture.

Les premières présentations du ρ-calcul [CK98, CK99b, Cir00, CK01] se focalisent principalement sur l'encodage de la réécriture et des stratégies. Une version simplifiée [CKL01a] permettant l'encodage de λ-calculs à objets a été introduite par la suite. Plusieurs extension du ρ-calcul ont été également proposées afin de traiter différents problèmes : pour traiter les exceptions [FK02], pour manipuler explicitement le filtrage et les substitutions (chapitre 4), pour disposer de traits impératifs [LS04], pour manipuler des graphes au lieu des termes (chapitre 5).

Nous présentons dans ce chapitre le ρ-calcul de base et deux variantes : le ρ_{stk}-calcul qui permet de gérer explicitement les échecs de filtrage et le ρ_d-calcul avec une application distributive. Nous discutons brièvement les propriétés de confluence du calcul de base et de ses extensions et nous illustrons leur puissance d'expression en montrant des encodages de la réécriture stratégique. Dans ce document nous présentons seulement la syntaxe et la sémantique de la dernière version du calcul ; la dernière section de ce chapitre rappelle brièvement les différences par rapport aux autres versions.

La première version du calcul de réécriture a été réalisée en collaboration avec Claude Kirchner. La syntaxe et la sémantique du calcul de base présenté ici est le résultat de plusieurs travaux réalisés avec Claude Kirchner et Luigi Liquori. Benjamin Wack a joué un rôle important dans le développement des variantes présentées dans ce chapitre, le ρ_d-calcul ayant été proposé avec

$\mathcal{T}, \mathcal{P}$	::=	$\mathcal{X}$	(Variables)
	|	$\mathcal{K}$	(Constantes)
	|	$\mathcal{P} \twoheadrightarrow \mathcal{T}$	(Abstractions)
	|	$\mathcal{T}\ \mathcal{T}$	(Applications)
	|	$\mathcal{T} \wr \mathcal{T}$	(Structures)

FIGURE 2.1 – *Syntaxe du ρ-calcul*

Clément Houtmann dans le cadre de son stage de master.

2.1 La syntaxe du calcul de réécriture de base

Nous considérons les méta-symboles « _ $\twoheadrightarrow$ _ » (opérateur d'abstraction), « _ $\wr$ _ » (opérateur de structure) et l'opérateur « _ _ » d'application. Etant donné un ensemble $\mathcal{K}$ de constantes et un ensemble $\mathcal{X}$ de variables, l'ensemble des termes du ρ-calcul de base est défini dans la figure 2.1.

En ρ-calcul, une λ-abstraction de la forme $\lambda x.A$ est généralisée par une *abstraction sur un motif* (également appelée règle) de la forme $P \twoheadrightarrow A$ où P peut être un motif plus général qu'une simple variable. Le membre gauche de l'abstraction définit les variables liées et une information de contexte. Dans une abstraction de la forme $P \twoheadrightarrow A$ nous appelons le terme P le motif et le terme A le corps.

Les termes peuvent être groupés dans des *structures* de la forme $A \wr B$ qui peuvent être considérées comme des collections de termes. Une structure peut être vue comme une paire, mais on l'utilisera parfois comme un ensemble ou une liste ordonnée, en se donnant une théorie équationnelle appropriée sur le symbole « _ $\wr$ _ ». Comme les règles sont elles-mêmes des termes, la structure permet de représenter non seulement des ensembles de résultats, mais aussi des ensembles de règles, ce qui s'avère très utile lorsqu'on veut encoder la réécriture.

On suppose que, par défaut, l'opérateur d'application est associatif à gauche, alors que les autres opérateurs sont associatifs à droite. L'application est plus prioritaire que « _ $\twoheadrightarrow$ _ » qui est à son tour plus prioritaire que « _ $\wr$ _ ».

On utilisera les symboles $A, B, C, \ldots$ pour dénoter les termes de l'ensemble $\mathcal{T}$, les symboles $x, y, z, \ldots$ pour les variables de l'ensemble $\mathcal{X}$ ($\mathcal{X} \subseteq \mathcal{T}$), les symboles $a, b, c, \ldots, f, g, h, \ldots$ et des chaînes de caractères construites en les utilisant pour des constantes de l'ensemble $\mathcal{K}$ ($\mathcal{K} \subseteq \mathcal{T}$). Finalement, les symboles P, Q dénotent des motifs de l'ensemble $\mathcal{P}$ ($\mathcal{X} \subseteq \mathcal{P} \subseteq \mathcal{T}$). Des vecteurs de termes de la forme $(A_1, \ldots, A_n)$ sont notés par $\overline{A}$. L'identité des termes est notée $\equiv$.

Si une même variable apparaît au plus une fois dans un terme alors le

terme est dit *linéaire*.

Définition 1 (Termes algébriques) *Les termes* algébriques *sont les termes de la forme*

$$(\cdots((f\ A_1)\ A_2)\cdots)\ A_n$$

où f est une constante et $A_1, A_2, \ldots, A_n$ sont des termes algébriques. On utilise habituellement la notation $f(A_1, A_2, \ldots, A_n)$ ou encore $f\overline{A}$ pour ce type de termes.

Les termes algébriques simulent exactement l'union de toutes les algèbres de termes $\mathcal{T}(\mathcal{K}, \mathcal{X})$ que l'on peut construire en assignant tous les choix d'arités possibles aux constantes de $\mathcal{K}$. C'est une des classes de termes que nous utiliserons le plus souvent pour les motifs.

Exemple 1 (Encodage de formules propositionnelles) *Nous considérons les constantes* true, false, not, and, or, xor *(qui dénotent les valeurs booléennes vrai et faux, la négation, la conjonction, la disjonction et la disjonction exclusive). Nous pouvons définir les termes* $\mathsf{and}(x, \mathsf{true})$ *et* $\mathsf{or}(\mathsf{not}(x), \mathsf{not}(y))$ *représentant des formules propositionnelles. Notons que les deux termes sont algébriques.*

Nous pouvons également définir quelques règles pour calculer dans l'algèbre booléenne :

- $\mathsf{and}(x, \mathsf{true}) \rightarrowtriangle x$ *; l'occurrence de la variable x dans le membre droit est liée par le motif* $\mathsf{and}(x, \mathsf{true})$.
- $\mathsf{not}(\mathsf{and}(x, y)) \rightarrowtriangle \mathsf{or}(\mathsf{not}(x), \mathsf{not}(y))$ *; cette règle lie les variables x et y.*
- $\mathsf{xor}(x, x) \rightarrowtriangle \mathsf{false}$ *; une règle non-linéaire.*

L'application de la deuxième règle au terme $\mathsf{not}(\mathsf{and}(\mathsf{true}, \mathsf{false}))$ *est représentée par le terme*

$$\big(\mathsf{not}(\mathsf{and}(x, y)) \rightarrowtriangle \mathsf{or}(\mathsf{not}(x), \mathsf{not}(y))\big)\ \mathsf{not}(\mathsf{and}(\mathsf{true}, \mathsf{false}))$$

et, comme on verra dans la section suivante, ce terme est réduit à $\mathsf{or}(\mathsf{not}(\mathsf{true}), \mathsf{not}(\mathsf{false}))$.

Exemple 2 (Instruction conditionnelle) *Etant donné les constantes* true *et* false *l'encodage de l'instruction conditionnelle est définie comme suit :*

$$\begin{aligned} [\![\mathsf{neg}]\!] &\triangleq (\mathsf{true} \rightarrowtriangle \mathsf{false} \wr \mathsf{false} \rightarrowtriangle \mathsf{true}) \\ [\![\mathsf{if}\ A\ \mathsf{then}\ B\ \mathsf{else}\ C]\!] &\triangleq (\mathsf{true} \rightarrowtriangle [\![B]\!] \wr \mathsf{false} \rightarrowtriangle [\![C]\!])\ [\![A]\!] \end{aligned}$$

Définition 2 (Variables libres) *L'ensemble de variables libres d'un terme A et l'ensemble de variables liées, notés $\mathcal{FV}(A)$ et $\mathcal{BV}(A)$ respectivement, sont définies inductivement par :*

$$\begin{array}{rcl@{\qquad}rcl}
\mathcal{FV}(c) & \triangleq & \emptyset & \mathcal{BV}(c) & \triangleq & \emptyset \\
\mathcal{FV}(x) & \triangleq & \{x\} & \mathcal{BV}(x) & \triangleq & \emptyset \\
\mathcal{FV}(A\ B) & \triangleq & \mathcal{FV}(A) \cup \mathcal{FV}(B) & \mathcal{BV}(A\ B) & \triangleq & \mathcal{BV}(A) \cup \mathcal{BV}(B) \\
\mathcal{FV}(A \wr B) & \triangleq & \mathcal{FV}(A) \cup \mathcal{FV}(B) & \mathcal{BV}(A \wr B) & \triangleq & \mathcal{BV}(A) \cup \mathcal{BV}(B)
\end{array}$$

$$\begin{array}{rcl}
\mathcal{FV}(P \twoheadrightarrow A) & \triangleq & \mathcal{FV}(A) \backslash \mathcal{FV}(P) \\
\mathcal{BV}(P \twoheadrightarrow A) & \triangleq & \mathcal{BV}(A) \cup \mathcal{BV}(P) \cup \mathcal{FV}(P)
\end{array}$$

Quand l'ensemble de variables libres d'un terme est vide on dit que le terme est clos. Sinon, il est ouvert.

Dans la suite de ce document nous travaillons modulo l'α-conversion, *c.-à-d.* on ne fait pas de différence entre deux termes α-convertibles. Nous adoptons la convention d'hygiène de Barendregt [Bar84], *c.-à-d.* pour un terme donné, les ensembles de ses variables libres et liées sont disjoints.

Définition 3 (Substitution) *Une* substitution *est une application de l'ensemble des variables dans l'ensemble de termes. Le domaine d'une substitution σ, noté $\mathcal{D}om(\sigma)$, est l'ensemble de variables tel que $\sigma(x) \neq x$. Si $\mathcal{D}om(\sigma) = \{x_1, \ldots, x_n\}$ et $\sigma(x_i) = A_i$, alors la substitution σ est notée $\{A_1/x_1 \ldots A_n/x_n\}$. Une substitution s'étend à une application de termes dans les termes. Cette application par rapport à un terme A, notée $A\sigma$, est définie inductivement par*

$$\begin{array}{rcll@{\qquad}rcl}
x\sigma & \triangleq & \sigma(x) & \text{if } x \in \mathcal{D}om(\sigma) & (P \twoheadrightarrow B)\sigma & \triangleq & P \twoheadrightarrow (B\sigma) \\
y\sigma & \triangleq & y & \text{if } y \notin \mathcal{D}om(\sigma) & (A_1\ A_2)\sigma & \triangleq & (A_1\sigma)\ (A_2\sigma) \\
c\sigma & \triangleq & c & & (A_1 \wr A_2)\sigma & \triangleq & (A_1\sigma) \wr (A_2\sigma)
\end{array}$$

Le co-domaine d'une substitution σ, noté $\mathcal{R}an(\sigma)$, est défini comme l'union des ensembles $\mathcal{FV}(x\sigma)$ avec $x \in \mathcal{D}om(\sigma)$. L'ensemble des variables d'une substitution σ, noté $\mathcal{V}ar(\sigma)$, est défini par $\mathcal{D}om(\sigma) \cup \mathcal{R}an(\sigma)$.

La substitution identité, *c.-à-d.* la substitution qui envoie toute variable sur elle-même, est notée `id`.

Les substitutions sont appliquées modulo l'α-conversion et donc, les représentants appropriés des termes sont utilisés afin d'éviter des captures éventuelles de variables. Concrètement, dans le cas de l'abstraction, on peut toujours supposer que $\mathcal{D}om(\sigma) \cap \mathcal{FV}(P) = \emptyset$ et $\mathcal{FV}(P) \cap \mathcal{R}an(\theta) = \emptyset$.

La composition de deux substitutions σ et τ est notée $\sigma \circ \tau$ et elle est définie comme d'habitude, *c.-à-d.* $x(\sigma \circ \tau) = (x\tau)\sigma$. La restriction (du domaine) d'une substitution σ à un ensemble de variables θ est notée $\sigma_{|\theta}$.

2.2 La sémantique du calcul de réécriture de base

Le mécanisme d'évaluation du ρ-calcul est basé sur l'opération fondamentale de *filtrage*. Quand une règle est appliquée, cette opération permet

de lier les variables à leurs valeurs correspondantes. Le formalisme général ne précise pas explicitement le type de filtrage utilisé, filtrage qui peut être effectué syntaxiquement ou modulo une théorie équationnelle. L'ensemble des motifs et la congruence modulo laquelle les termes sont filtrés sont ainsi des paramètres du calcul. Le troisième paramètre est la théorie sous-jacente à l'opérateur de structure (typiquement, une combinaison d'associativité, commutativité et idempotence).

Nous donnons d'abord quelques exemples d'algorithmes de filtrage qui peuvent être utilisés par le mécanisme d'évaluation du ρ-calcul et ensuite nous définissons la sémantique opérationnelle du ρ-calcul.

2.2.1 Filtrage

Nous définissons d'abord l'opération de filtrage et donnons ensuite quelques exemples d'algorithmes de filtrage pour différents ensembles de motifs.

Définition 4 (Filtrage) *Etant donnée une théorie $\mathbb{T}$, c.-à-d. un ensemble d'axiomes définissant une relation de congruence $\overset{\mathbb{T}}{=}$, on a :*

1. *Une équation de filtrage (appelée aussi problème de filtrage) est une paire notée $P \lessdot_{\mathbb{T}} A$ avec P un motif et A un terme.*
2. *Une substitution σ est une solution de l'équation $P \lessdot_{\mathbb{T}} A$ ssi $P\sigma \overset{\mathbb{T}}{=} A$.*

L'ensemble des solutions de $P \lessdot_{\mathbb{T}} A$ est noté $\mathcal{S}ol(P \lessdot_{\mathbb{T}} A)$.

Nous considérons souvent $\mathcal{S}ol$ comme une fonction qui, étant donné un motif et un terme, renvoie l'ensemble de substitutions (potentiellement vide) du problème de filtrage correspondant. Si la congruence utilisée pour le filtrage est claire dans le contexte alors elle n'est pas mentionnée explicitement.

Nous verrons dans les sections suivantes que la confluence des différentes instances et extensions du calcul est obtenue en imposant des restrictions sur l'ensemble des motifs et implicitement sur le filtrage utilisé. Notons déjà que dans le cas du filtrage syntaxique, la substitution résultant du filtrage, quand elle existe, est unique et peut être calculée par un algorithme récursif simple donné par exemple par G.Huet [Hue76].

Définition 5 (Filtrage syntaxique) *On considère des contraintes de filtrage de la forme $A \ll^? B$ et des conjonctions potentiellement vides de tels problèmes construites avec l'opérateur commutatif et associatif $\wedge$. L'ensemble suivant de règles confluentes et terminantes peut être utilisé pour résoudre des problèmes de filtrage syntaxique (non-linéaire) :*

$$
\begin{array}{lcl}
A \ll^? A \wedge M & \rightarrow & M \\
(A_1\ A_2 \ll^? B_1\ B_2) \wedge M & \rightarrow & (A_1 \ll^? B_1) \wedge (A_2 \ll^? B_2) \wedge M
\end{array}
$$

$(P \twoheadrightarrow A)\ B$	$\rightarrow_\rho$	$A\sigma_1 \wr \ldots \wr A\sigma_n$ **avec** $\{\sigma_1, \ldots, \sigma_n\} = \mathcal{S}ol(P \lessdot_{\mathbb{T}} B)$
$(A_1 \wr A_2)\ B$	$\rightarrow_\delta$	$A_1\ B \wr A_2\ B$

FIGURE 2.2 – *Sémantique opérationnelle du ρ-calcul*

L'ensemble des solutions d'un problème de filtrage $P \lessdot_\emptyset A$ est obtenu en normalisant la contrainte de filtrage $P \ll^? A$ par rapport aux règles de réécriture ci-dessus et selon le résultat obtenu on a :

- $\mathcal{S}ol(P \lessdot A) = \{\{A_i/x_i\}_{i\in I}\}$ *si le résultat est* $\bigwedge_{i\in I\neq\emptyset}(x_i \ll^? A_i)$ *avec* $A_i = A_j$ *si* $x_i = x_j$ *;*
- $\mathcal{S}ol(P \lessdot A) = \{\mathtt{id}\}$ *si le résultat est la conjonction vide ;*
- $\mathcal{S}ol(P \lessdot A) = \emptyset$ *sinon.*

Quand on manipule des motifs algébriques on peut remplacer la règle de décomposition pour l'application par la règle suivante :

$$f(A_1, \ldots, A_n) \ll^? f(B_1, \ldots, B_n) \wedge M \qquad \rightarrow \qquad \wedge_{i=1}^n (A_i \lessdot B_i) \wedge M$$

L'ensemble de règles ci-dessus peut être utilisé pour résoudre des problèmes de filtrage sur les termes algébriques et il peut être facilement étendu pour manipuler des structures de termes algébriques.

Définition 6 (Structures algébriques) *L'ensemble de structures algébriques est défini par :*

$$\mathcal{P} \ ::= \ \mathcal{X} \mid \mathcal{K} \mid \mathcal{K}\overline{\mathcal{P}} \mid \mathcal{P} \wr \mathcal{P}$$

Si aucune théorie n'est utilisée pour l'opérateur de structure alors cet opérateur peut être considéré comme une constante et on doit ajouter la règle suivante aux algorithmes de filtrage précédents :

$$(P \wr Q \ll^? A_1 \wr A_2) \wedge M \qquad \rightarrow \qquad (P \ll^? A_1) \wedge (Q \ll^? A_2) \wedge M$$

2.2.2 Règles d'évaluation

La sémantique opérationnelle du ρ-calcul est donnée dans la figure 2.2 où $\mathbb{T}$ dénote la congruence modulo laquelle les problèmes de filtrage sont résolus.

La règle (ρ) précise le comportement de l'application des abstractions : pour appliquer une règle $P \twoheadrightarrow A$ à un terme B, on calcule d'abord les solutions potentielles du problème de filtrage entre P et B modulo la congruence $\mathbb{T}$. Si de telles solutions existent (à noter la condition implicite $n > 1$ dans la

règle), les substitutions correspondantes sont appliquées à A et les résultats sont alors groupés en utilisant l'opérateur de structure.

La règle (δ) définit l'application des structures. L'application d'une structure $A_1 \wr A_2$ à un terme B consiste simplement à distribuer l'application sur l'argument par rapport aux éléments de la structure. Quand A_1 et A_2 sont des règles, ce type d'application correspond, intuitivement, à l'application de plusieurs règles d'un système de réécriture au terme B.

La fermeture compatible d'une relation $\mathcal{R}$ serra notée $\mapsto_{\mathcal{R}}$ ou, par abus de notation, simplement $\mathcal{R}$. Sa fermeture transitive et réflexive est notée soit $\mapsto\!\!\!\!\rightarrow_{\mathcal{R}}$ soit $\mathcal{R}^*$. Par exemple, $\mapsto\!\!\!\!\rightarrow_{\rho\delta}$ dénote la fermeture réflexive et transitive de la relation compatible induite par les règles (ρ) et (δ).

Exemple 3 (Réductions)

1. *Si on considère un filtrage syntaxique ($\mathbb{T} = \emptyset$), on a*

$$\begin{array}{rcl}
(x \twoheadrightarrow A)\ B & \mapsto_{\rho} & A\{B/x\} \\
(a \twoheadrightarrow A)\ a & \mapsto_{\rho} & A \\
(a \twoheadrightarrow a)\ b & & \textit{est en forme normale puisque } a \textit{ et } b \textit{ ne filtre pas} \\
(f(x) \twoheadrightarrow g(x))\ f(a) & \mapsto_{\rho} & g(a) \\
(a \twoheadrightarrow b \wr a \twoheadrightarrow c)\ a & \mapsto_{\delta} & (a \twoheadrightarrow b)\ a \wr (a \twoheadrightarrow c)\ a \\
 & \mapsto_{\rho} & b \wr (a \twoheadrightarrow c)\ a \\
 & \mapsto_{\rho} & b \wr c
\end{array}$$

2. *Dans une théorie équationnelle $f\ x\ y =_{\mathbb{T}} f\ y\ x$, nous avons*

$$\begin{array}{rcl}
(x \twoheadrightarrow x)\ f(a,b) & \mapsto_{\rho} & f(a,b) \\
(f(x,y) \twoheadrightarrow x)\ f(a,b) & \mapsto_{\rho} & a \wr b
\end{array}$$

Il faut noter que la fonction $\mathcal{S}ol$ associée à une théorie de filtrage non-unitaire retourne un ensemble de substitutions et pour préserver la nature non ordonnée de cet ensemble la théorie associée à l'opérateur de structure devrait inclure l'associativité et la commutativité.

Exemple 4 (Application d'un système de réécriture) *Dans une première approximation, l'application d'une structure de règles peut être vue comme l'application d'un système de réécriture. Le terme*

$$\Big(\mathsf{and}(x,\mathsf{or}(y,z)) \twoheadrightarrow \mathsf{or}(\mathsf{and}(x,y),\mathsf{and}(x,z)) \wr \mathsf{and}(\mathsf{or}(x,y),z) \twoheadrightarrow \mathsf{or}(\mathsf{and}(x,z),\mathsf{and}(y,z))\Big) \quad \mathsf{and}(\mathsf{or}(\mathsf{true},\mathsf{false}),\mathsf{or}(\mathsf{false},\mathsf{false}))$$

représente l'application du système de réécriture

$$\left\{\begin{array}{rcl}
\mathsf{and}(x,\mathsf{or}(y,z)) & \to & \mathsf{or}(\mathsf{and}(x,y),\mathsf{and}(x,z)) \\
\mathsf{and}(\mathsf{or}(x,y),z) & \to & \mathsf{or}(\mathsf{and}(x,z),\mathsf{and}(y,z))
\end{array}\right.$$

au terme $\mathsf{and}(\mathsf{or}(\mathsf{true},\mathsf{false}),\mathsf{or}(\mathsf{false},\mathsf{false}))$. *On utilise la règle* (δ) *pour distribuer les deux règles et on obtient :*

$$\begin{array}{l}\big(\mathsf{and}(x,\mathsf{or}(y,z)) \twoheadrightarrow \mathsf{or}(\mathsf{and}(x,y),\mathsf{and}(x,z))\big)\ \mathsf{and}(\mathsf{or}(\mathsf{true},\mathsf{false}),\mathsf{or}(\mathsf{false},\mathsf{false})) \wr \\ \big(\mathsf{and}(\mathsf{or}(x,y),z) \twoheadrightarrow \mathsf{or}(\mathsf{and}(x,z),\mathsf{and}(y,z))\big)\ \mathsf{and}(\mathsf{or}(\mathsf{true},\mathsf{false}),\mathsf{or}(\mathsf{false},\mathsf{false}))\end{array}$$

Après deux réductions on obtient :

$$\begin{array}{l}\mathsf{or}(\mathsf{and}(\mathsf{or}(\mathsf{true},\mathsf{false}),\mathsf{false}),\mathsf{and}(\mathsf{or}(\mathsf{true},\mathsf{false}),\mathsf{false}))\ \wr \\ \mathsf{or}(\mathsf{and}(\mathsf{true},\mathsf{or}(\mathsf{false},\mathsf{false})),\mathsf{and}(\mathsf{false},\mathsf{or}(\mathsf{false},\mathsf{false})))\end{array}$$

L'application d'un système de réécriture n'est jamais aussi simple que ça. Ici, nous encodons seulement un (méta) pas de réécriture mais, généralement l'encodage est plus compliqué car on doit encoder non seulement l'application d'une règle de réécriture à la position de tête d'un terme mais également la stratégie de réduction guidant l'application des règles. Nous montrons dans les sections 2.5.2 et 2.6.2 que le problème peut être résolu, par exemple, en utilisant des points fixes (typés) [CKLW03] pour appliquer le système de réécriture récursivement.

2.3 Expressivité

Dans cette section nous donnons quelques exemples de ρ-termes et les réductions correspondantes. Nous commençons par donner un codage naturel des λ-termes en des ρ-termes.

Définition 7 (Encodage du λ-calcul)
La translation de λ-termes en ρ-termes est définie par

$$\begin{array}{lcl}[\![x]\!] & \triangleq & x \\ [\![\lambda x.M]\!] & \triangleq & x \twoheadrightarrow [\![M]\!] \\ [\![M\ N]\!] & \triangleq & [\![M]\!]\ [\![N]\!]\end{array}$$

Quand on traduit des λ-termes en ρ-termes l'abstracteur « λ » est remplacé par l'opérateur d'abstraction (de règle) « $\twoheadrightarrow$ ». Il est facile de voir que la traduction est compatible avec la réduction des deux formalismes : pour chaque β-réduction d'un λ-terme il existe une ρ-réduction correspondante du terme traduit dans le ρ-calcul. La traduction inverse du ρ-calcul vers le λ-calcul est considérablement plus élaborée [Wac05].

Théorème 1 (Simulation du λ-calcul en ρ-calcul) *Si A et B sont deux λ-termes alors*

$$A \mapsto_\beta B \qquad \textit{si et seulement si} \qquad [\![A]\!] \mapsto_\rho [\![B]\!].$$

Plusieurs calculs orientés abject, à savoir le « Lambda Calculus of Objects » de Fisher, Honsell, et Mitchell [FHM94] et le « Object Calculus » de Abadi et Cardelli [AC96a] peuvent également être encodés dans le ρ-calcul [CKL01a].

En ce qui concerne la réécriture, la comparaison est un peu plus élaborée car l'évaluation en ρ-calcul et en réécriture est assez différente :

- en ρ-calcul, les règles sont « consommées » une fois utilisées ;
- en ρ-calcul, la position où une règles (de réécriture) est appliquée est donnée explicitement (par l'opérateur d'application) ;
- en ρ-calcul, une éventuelle stratégie (sur l'ordre) d'application des règles doit être également donnée explicitement.

Les termes algébriques habituels des systèmes de réécriture peuvent être traduits directement en ρ-termes algébriques. Nous considérons que cet encodage trivial est fait par le fonction $[\![\ ]\!]$ et nous obtenons un premier résultat relatif à l'encodage des dérivations de la réécriture dans le ρ-calcul.

Théorème 2 (Représentation d'un chemin de réécriture [CK01])
Etant donné un système de réécriture $\mathcal{R}$ et deux termes algébriques t et t', si $t \mapsto\!\!\!\!\!\rightarrow_{\mathcal{R}} t'$, alors il existe un ρ-terme A tel que

$$A\ [\![t]\!] \mapsto\!\!\!\!\!\rightarrow_{\rho\delta} [\![t']\!].$$

Nous pouvons aller encore plus loin et prouver que le comportement d'une certaine classe de systèmes de réécriture peut être simulé. En particulier, nous montrerons dans les sections 2.5.2 et 2.6.2 que le comportement d'un système de réécriture guidé par une certaine stratégie peut également être simulé [CLW03, CKLW03].

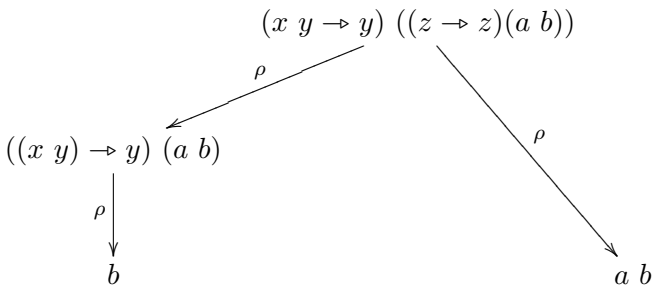
La combinaison de la réécriture de termes de premier ordre avec le λ-calcul a été déjà réalisée soit en enrichissant la réécriture de premier ordre avec des fonctionnalités d'ordre supérieur, comme dans les « Combinatory Reduction Systems » (CRS) [Klo80, KvOvR93], soit en ajoutant au λ-calcul des caractéristiques algébrique [BT88, JO97].

Les divers systèmes de réécriture d'ordre supérieur et le calcul de réécriture partagent des concepts similaires et ont des applications similaires. Il est donc naturel de comparer ces formalismes pour mieux comprendre les différences et les avantages de chaque formalisme. Nous avons montré [BCK06, Ber05] qu'on peut exprimer des dérivations d'un CRS en termes de dérivations du calcul de réécriture. Réciproquement, la version la plus simple du ρ-calcul peut également être codée par un CRS orthogonal [BK07] et, par conséquent, des propriétés importantes tel que la confluence, les développements finis et la standardisation sont déduites directement pour la version correspondante du calcul de réécriture.

2.4 Résultats de confluence

Cette section rappelle les principaux résultats concernant la confluence du ρ-calcul. Une étude plus détaillée de la confluence des calculs à motifs sera faite dans le chapitre 6.

Nous devons tout d'abord préciser que l'utilisation du filtrage syntaxique peut mener à des réductions non-confluentes si nous n'imposons aucune restriction sur la forme des motifs. Les réductions



illustrent le fait qu'en présence de motifs d'ordre supérieur (comme $x\ y$) le filtrage de premier ordre n'est pas approprié.

Par ailleurs, les motifs non-linéaires de premier ordre peuvent mener également à des réductions non-confluentes. La possibilité d'obtenir des paires critiques non-joinables dans les systèmes de réécriture d'ordre supérieur manipulant des motifs non-linéaires a été montrée pour la première fois par J.W.Klop [Klo80]. Le contre-exemple de Klop peut être encodé dans le ρ-calcul [Wac05] montrant ainsi que le ρ-calcul non-linéaire n'est pas confluent si aucune restriction n'est imposée sur les réductions.

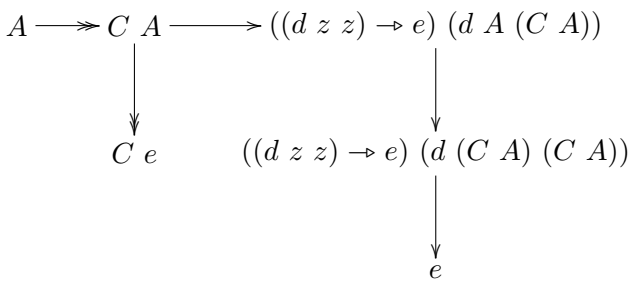
Le combinateur de point fixe classique Y peut être défini comme dans le λ-calcul

$$\mathsf{Y} \triangleq \Big(y \twoheadrightarrow x \twoheadrightarrow (x\ (y\ y\ x))\Big)\ \Big(y \twoheadrightarrow x \twoheadrightarrow (x\ (y\ y\ x))\Big)$$

et on a, pour tout ρ-terme A, la réduction $\mathsf{Y}\ A \mapsto\!\!\!\!\twoheadrightarrow_\rho A\ (\mathsf{Y}\ A)$. On définit alors les termes suivants

$$\begin{array}{lcl} C & \equiv & \mathsf{Y}\ \big(y \twoheadrightarrow x \twoheadrightarrow ((d\ z\ z) \twoheadrightarrow e)\ (d\ x\ (y\ x))\big) \\ C & \mapsto\!\!\!\!\twoheadrightarrow_\rho & \big(y \twoheadrightarrow x \twoheadrightarrow ((d\ z\ z) \twoheadrightarrow e)\ (d\ x\ (y\ x))\big)\ C \\ & \mapsto_\rho & x \twoheadrightarrow ((d\ z\ z) \twoheadrightarrow e)\ (d\ x\ (C\ x)) \\ A & \equiv & \mathsf{Y}\ C \end{array}$$

et on obtient les réductions



La constante e est en forme normale et dans la plus courte réduction de $C\ e$ à e, il faut forcément réduire ses radicaux de tête. Au bout de quelques ρ-réductions on obtient le terme $((d\ z\ z) \twoheadrightarrow e)\ (d\ e\ (C\ e))$, qui ne peut être réduit en tête qu'après une réduction de $C\ e$ à e. Comme on a supposé la minimalité de la réduction, on conclut par l'absurde que $C\ e$ ne se réduit pas à e et donc les deux réductions ne confluent pas.

Pour garantir la confluence, V.van Oostrom propose dans son étude sur le λ-calcul avec motifs [vO90, KvOdV08] une condition de « motifs rigides ». Nous présentons une caractérisation de cette condition ; la définition générale sera discutée dans le chapitre 6.

Définition 8 (Rigid Pattern Condition) *Un motif P respecte la «* rigid pattern condition *» si il est*
- *linéaire (toute variable libre apparaît au plus une fois),*
- *en forme $\rho\delta$-normale,*
- *sans variable active (c.-à-d., pas de sous-termes xA avec x libre).*

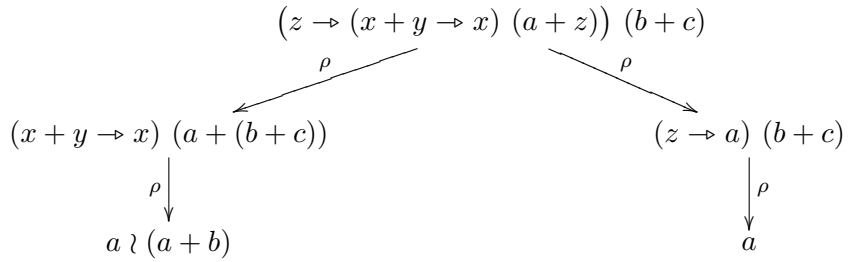
Théorème 3 (Confluence par Rigid Pattern Condition) *La relation $\rho\delta$ est confluente si tous les motifs vérifient la Rigid Pattern Condition.*

Malheureusement, ce résultat est valable dans le cas du filtrage syntaxique mais il est assez facile de voir que même une théorie de filtrage très naturelle peut invalider la confluence du calcul. Comme mentionné précédemment, l'ensemble de solutions d'un problème de filtrage non-unitaire n'est pas ordonné et donc, afin d'éviter des réductions trivialement non-confluentes, la théorie associée à l'opérateur de structure devrait inclure l'associativité et la commutativité. On pourrait imposer une certaine notion de préservation d'ordre pour la liste de substitutions produite par l'algorithme de filtrage pour éviter ce type de problèmes mais la confluence du calcul ne peut pas être obtenue sans imposer des conditions supplémentaires sur la réduction.

Exemple 5 (Confluence avec filtrage non-unitaire) *On considère un symbole $+$ associatif :*

$$x + (y + z) =_{\mathbb{T}} (x + y) + z$$

On a alors les réductions suivantes :



Le radical interne est réduit avant instanciation de z et donc il n'est pas possible de réarranger l'argument $a + z$ pour trouver les deux solutions du problème de filtrage modulo l'associativité qui apparaît dans la réduction de gauche.

Pour obtenir un calcul confluent dans les théories de filtrage non syntaxiques il faut, en général, fixer une *stratégie d'évaluation*, autrement dit un ordre bien précis de choix des radicaux dans un terme donné. Par exemple, on peut autoriser des variables à gauche de l'application dans les motifs tout en conservant un algorithme de filtrage très simple, à condition que l'algorithme garanti que ces variables ne peuvent être instanciées que par des constantes. La forme des arguments autorisés rappelle alors le λ-calcul par valeurs de Plotkin [Plo75], adapté pour tenir compte du filtrage. Une étude poussée de la confluence avec ce genre de motifs a été faite dans le calcul à motifs purs [JK06].

Une *stratégie de réduction* dans le ρ-calcul peut être définie par un prédicat $\mathcal{C}$ sur les ρ-termes tel que la règle (ρ) devient :

$$(P \twoheadrightarrow A)\ B \quad \mapsto_{\rho} \quad A\sigma_1 \wr \ldots \wr A\sigma_n \quad \textbf{si } \mathcal{S}ol(P \ll_{\mathbb{T}} B) = \{\sigma_1, \ldots, \sigma_n\} \textbf{ avec } n > 0 \textbf{ et } B \textbf{ satisfait } \mathcal{C}$$

La condition $\mathcal{C}$ est souvent définie par (l'appartenance à) un ensemble de termes. Par exemple, si on considère qu'un ρ-terme est une *valeur rigide* s'il est clos et en forme normale par rapport à $\rho\delta$ alors la stratégie d'appel par valeurs rigides est la stratégie d'évaluation utilisant l'ensemble de valeurs rigides.

Il n'est pas difficile de remarquer que si la règle (ρ) est appliquée quand les arguments des abstractions ne peuvent plus évoluer on obtient un calcul confluent puisqu'il y a seulement des paires critiques triviales. L'exemple 5 de réductions non-confluentes en présence d'un symbole associatif est maintenant résolu parce que l'évaluation qui mène à a n'est plus possible.

Théorème 4 (Confluence par valeurs rigides [CHW07, Cir00])
Quelle que soit la théorie de filtrage choisie, la relation $\rho\delta$ est confluente sous la stratégie d'appel par valeurs rigides.

$$\begin{array}{lcll} \mathcal{P} & ::= & \mathcal{X} \mid \mathcal{K} \mid \mathsf{stk} \mid \mathcal{K}\overline{\mathcal{P}} & \text{(Motifs)} \\ \mathcal{T} & ::= & \mathcal{X} \mid \mathcal{K} \mid \mathsf{stk} \mid \mathcal{P} \twoheadrightarrow \mathcal{T} \mid \mathcal{T}\ \mathcal{T} \mid \mathcal{T} \wr \mathcal{T} & \text{(Termes)} \end{array}$$

FIGURE 2.3 – *Syntaxe de* ρ_{stk}

2.5 Echec explicite de filtrage : ρ_{stk}

On a vu dans la section précédente que si le problème de filtrage correspondant à l'application d'une abstraction ne réussit pas alors aucune réduction n'est exécutée. Par exemple, le terme $(a \twoheadrightarrow b)\ c$ est en forme normale. En revanche, on n'a pour l'instant aucun moyen dans le calcul pour identifier de tels échecs. De plus, il nous est impossible de différencier les échecs temporaires des échecs définitifs : par exemple, dans $(x \twoheadrightarrow ((a \twoheadrightarrow b)\ x))\ a$, le sous-terme $(a \twoheadrightarrow b)\ x$ est en forme normale, mais après instanciation de x par le radical externe, il devient le radical $(a \twoheadrightarrow b)\ a$ qui se réduit à a.

Nous nous concentrons ici sur une version du ρ-calcul qui utilise une constante spéciale stk pour représenter les échecs de filtrage. Cette extension du ρ-calcul de base, appelée ici ρ_{stk}-calcul, peut être utilisée pour encoder des systèmes de réécriture et pour donner une sémantique aux langages de programmation fonctionnelle [CLW03].

Nous présentons ici la version non typée de ce calcul et nous illustrons son pouvoir d'expression. Dans le calcul disposant d'un système de types approprié [CLW03] on peut non seulement encoder mais également typer des points fixe (orientés objet) et des systèmes de réécriture de termes.

2.5.1 Syntaxe et sémantique de ρ_{stk}

La syntaxe du ρ_{stk}-calcul est donnée dans la figure 2.3. Comme pour le calcul de base, dans le cas général, on n'impose aucune restriction sur les motifs, mais nous nous concentrons ici sur les motifs algébriques construits en utilisant les constantes de la signature ainsi que la constante spéciale stk. On verra plus tard que ceci nous permet de garantir la confluence du calcul.

Comme nous l'avons dit, la constante stk représente les applications de règle qui ne réussissent jamais en raison d'échecs de filtrage permanents. Nous devons donc décider quand un problème de filtrage associé à l'application d'une règle n'a pas de solution et n'en aura jamais une. Ceci correspond à dire que le filtrage n'aura pas de solution quelque soient les instanciations et les réductions des termes impliqués dans le filtrage ou, en d'autres termes,

$$\frac{\forall \sigma_1, \sigma_2,\ \forall B',\ B\sigma_1 \mapsto\!\!\!\!\twoheadrightarrow_{\beta\delta} B'\ \Rightarrow\ P\sigma_2 \not\equiv B'}{(P \twoheadrightarrow A)\ B \mapsto_{\mathsf{stk}} \mathsf{stk}}$$

Cependant, une telle règle de réduction pose problème dans la mesure où la condition de réduction semble indécidable. C'est pourquoi nous introduisons

$$
\begin{array}{rcl}
(P \rightarrowtriangle A)\ B & \rightarrow_{\rho} & A\sigma_1 \wr \ldots \wr A\sigma_n \\
 & & \textbf{avec } \{\sigma_1, \ldots, \sigma_n\} = \mathcal{S}ol(P \prec\!\!\!\prec_{\mathbb{T}} B) \\
(A_1 \wr A_2)\ B & \rightarrow_{\delta} & A_1\ B \wr A_2\ B \\
 & & \\
(P \rightarrowtriangle A)\ B & \rightarrow_{stk} & \mathsf{stk} \\
 & & \textbf{si } P \not\sqsubseteq B \\
\mathsf{stk} \wr A & \rightarrow_{stk} & A \\
A \wr \mathsf{stk} & \rightarrow_{stk} & A \\
\mathsf{stk}\ A & \rightarrow_{stk} & \mathsf{stk}
\end{array}
$$

FIGURE 2.4 – *Sémantique de* ρ_{stk}

une relation $\not\sqsubseteq$ de superposition entre (des motifs et) des termes dont le but est de caractériser une classe d'équations de filtrage qui n'ont pas et n'auront jamais une solution (*c.-à-d.* indépendamment des instanciations et des réductions éventuelles).

Définition 9 (Echecs définitifs) *La relation* $\not\sqsubseteq$ *est définie comme suit :*

$$
\begin{array}{lcll}
\mathsf{stk} & \not\sqsubseteq & g(B_1, \ldots, B_n) & \text{if } g \not\equiv \mathsf{stk} \\
\mathsf{stk} & \not\sqsubseteq & Q \rightarrowtriangle B & \\
f(P_1, \ldots, P_m) & \not\sqsubseteq & g(B_1, \ldots, B_n) & \text{if } f \not\equiv g \text{ or } n \neq m \text{ or } \exists i,\ P_i \not\sqsubseteq B_i \\
f(P_1, \ldots, P_m) & \not\sqsubseteq & \mathsf{stk} & \\
f(P_1, \ldots, P_m) & \not\sqsubseteq & Q \rightarrowtriangle B & \\
f(P_1, \ldots, P_m) & \not\sqsubseteq & (Q \rightarrowtriangle A)\ B & \text{if } Q \not\sqsubseteq B \text{ or } f(P_1, \ldots, P_m) \not\sqsubseteq A
\end{array}
$$

Le ρ_{stk}-calcul réalise un traitement uniforme des échecs de filtrage et les élimine quand ils ne sont pas significatifs pour l'évaluation. La sémantique du ρ_{stk}-calcul est donnée dans la figure 2.4.

Comme mentionné précédemment, ces règles sont utilisées pour propager ou éliminer les termes correspondants à des échecs définitif. Les structures peuvent être vues comme des collections de résultats (intéressants) et donc on veut identifier tous les échecs (de filtrage) et les éliminer de ces collections ; ceci est matérialisé par les deux règles d'évaluations décrivant le comportement de stk par rapport à la structure. D'autre part, un terme stk peut être vu comme un ensemble vide de règles ; la dernière règle correspond alors à la règle (δ) pour les structures vides et donc, à une propagation de échec.

La fermeture compatible de la relation induite par les règles de ρ_{stk} est notée $\mapsto^{\mathsf{stk}}_{\rho\delta}$. Sa fermeture transitive et réflexive est notée comme d'habitude.

Théorème 5 (Confluence du ρ_{stk}-calcul [CLW03]) *Si tous les patterns sont linéaires, la relation* $\mapsto^{\mathsf{stk}}_{\rho\delta}$ *est confluente.*

2.5.2 Encodage des systèmes de réécriture convergents

Nous commençons par montrer un exemple de point fixe contenant un système de réécriture qui définit l'addition et nous donnerons ensuite une méthode systématique de transformation d'un système de réécriture en un ρ-terme. On définit :

$$\begin{array}{lcl} \mathit{plus} & \triangleq & (\mathit{rec}\ z) \twoheadrightarrow \left(\begin{array}{l} (\mathit{add}\ 0\ y) \twoheadrightarrow y \\ \wr (\mathit{add}\ (S\ x)\ y) \twoheadrightarrow S\ \ (z\ (\mathit{rec}\ z)\ (\mathit{add}\ x\ y)) \end{array} \right) \\ \mathit{addition} & \triangleq & n \twoheadrightarrow m \twoheadrightarrow \big(\mathit{plus}\ (\mathit{rec}\ \mathit{plus})\ (\mathit{add}\ n\ m)\big) \end{array}$$

Comme pour le combinateur de point fixe usuel, la variable z va propager une copie de *plus* au fur et à mesure des réductions et permettre ainsi des « appels récursifs ». Dans la dérivation ci-dessous on peut voir comment ce terme calcule l'addition de deux entiers de Peano ; les expressions $\overline{m}$, $\overline{m+n}$ et $\overline{m-n}$ sont des abréviations pour les termes $S\ (\ldots (S\ 0)\ldots)$ avec le nombre de S correspondant :

$$\begin{array}{ll} & \mathit{addition}\ \overline{n}\ \overline{m} \\ \mapsto_{\rho} & \mathit{plus}(\mathit{rec}\ \mathit{plus})(\mathit{add}\ \overline{n}\ \overline{m}) \\ \mapsto_{\rho\delta} & ((\mathit{add}\ 0\ y) \twoheadrightarrow y)(\mathit{add}\ \overline{n}\ \overline{m}) \\ & \wr\ ((\mathit{add}\ (S\ x)\ y) \twoheadrightarrow S\ (\mathit{plus}(\mathit{rec}\ \mathit{plus})(\mathit{add}\ x\ y)))(\mathit{add}\ \overline{n}\ \overline{m}) \\ \mapsto_{\rho} & ((\mathit{add}\ 0\ y) \twoheadrightarrow y)(\mathit{add}\ \overline{n}\ \overline{m}) \\ & \wr\ S(\mathit{plus}(\mathit{rec}\ \mathit{plus})(\mathit{add}\ \overline{n-1}\ \overline{m})) \\ \mapsto_{\mathsf{stk}} & S(\mathit{plus}(\mathit{rec}\ \mathit{plus})(\mathit{add}\ \overline{n-1}\ \overline{m})) \\ \mapsto_{\rho\delta} & S \left(\begin{array}{l} ((\mathit{add}\ 0\ y) \twoheadrightarrow y)(\mathit{add}\ \overline{n-1}\ \overline{m}) \\ \wr\ ((\mathit{add}(S\ x)\ y) \twoheadrightarrow S(\mathit{plus}(\mathit{rec}\ \mathit{plus})(\mathit{add}\ x\ y)))(\mathit{add}\ \overline{n-1}\ \overline{m}) \end{array} \right) \\ & \vdots \\ \mapsto_{\rho\delta}^{\mathsf{stk}} & S(\ldots(S \left(\begin{array}{l} ((\mathit{add}\ 0\ y) \twoheadrightarrow y)(\mathit{add}\ 0\ \overline{m})\ \wr \\ ((\mathit{add}(S\ x)\ y) \twoheadrightarrow S(\mathit{plus}(\mathit{rec}\ \mathit{plus})(\mathit{add}\ x\ y)))(\mathit{add}\ 0\ \overline{m}) \end{array} \right)\ldots) \\ \mapsto_{\rho\delta} & S(\ldots(S \left(\begin{array}{l} \overline{m}\ \wr \\ ((\mathit{add}(S\ x)\ y) \twoheadrightarrow S(\mathit{plus}(\mathit{rec}\ \mathit{plus})(\mathit{add}\ x\ y)))(\mathit{add}\ 0\ \overline{m}) \end{array} \right)\ldots) \\ \mapsto_{\mathsf{stk}} & S(\ldots(S\overline{m})) \\ \equiv & \overline{m+n} \end{array}$$

Le terme $\mathit{plus}(\mathit{rec}\ \mathit{plus})\ (\mathit{add}\ \overline{n}\ \overline{m})$ admet évidemment des réductions infinies, mais la réduction ci-dessus montre qu'il est faiblement normalisant. La réduction $\mapsto_{\mathsf{stk}}$ élimine les $n-1$ premières tentatives d'application de la règle $(\mathit{add}\ 0\ y) \twoheadrightarrow y$ car $0 \not\sqsubseteq \overline{n}$ tant que $n > 0$; en fin de réduction $\mapsto_{\mathsf{stk}}$ détecte l'échec $S\ x \not\sqsubseteq 0$ ce qui permet d'éliminer le sous-terme non terminant. La forme normale $\overline{m+n}$ (unique par confluence) est bien le résultat attendu.

Pour encoder le système de réécriture, il est suffisant dans ce cas de réappliquer $z\ (rec\ z)$ à un seul sous-terme dans les membres droits des règles. Par contre, en général, on ne sait pas quels sont les sous-termes des membres droits susceptibles d'être encore réécrits et il faut donc prévoir un mécanisme qui « essaie » des règles sur un terme et le laisse tel quel si aucune règle ne convient.

On peut définir un tel mécanisme grâce à stk. En effet, si $A_1, A_2, \ldots, A_n$ sont des termes, on définit

$$\begin{array}{l}\mathsf{first}(A_1, A_2, \ldots, A_n) \triangleq \\ \quad x \rightarrowtriangle ((\mathsf{stk} \rightarrowtriangle A_n\ x \wr y \rightarrowtriangle y)\ (\ldots\ (\mathsf{stk} \rightarrowtriangle A_2\ x \wr y \rightarrowtriangle y)\ (A_1 x)))\end{array}$$

Alors on vérifie que, pour tout terme B, le terme $\mathsf{first}(A_1, \ldots, A_n)\ B$ s'évalue comme $A_i\ B$ si

$$\left\{\begin{array}{lllll} \forall j < i, & A_j\ B & \mapsto^{\mathsf{stk}}_{\rho\delta} & \mathsf{stk} \\ & A_i\ B & \mapsto^{\mathsf{stk}}_{\rho\delta} & f\ \overline{B} \end{array}\right.$$

En effet, pour tout i on a :

$$\begin{array}{lll} (\mathsf{stk} \rightarrowtriangle A_{i+1}\ B \wr y \rightarrowtriangle y)\ (A_i\ B) & \mapsto^{\mathsf{stk}}_{\rho\delta} & (\mathsf{stk} \rightarrowtriangle A_{i+1}\ B)\ \mathsf{stk} \wr (y \rightarrowtriangle y)\ \mathsf{stk} \\ & \mapsto^{\mathsf{stk}}_{\rho\delta} & A_{i+1}\ B \wr \mathsf{stk} \\ & \mapsto_{\mathsf{stk}} & A_{i+1}\ B \end{array}$$

si $A_i\ B$ s'évalue à stk et

$$\begin{array}{lll} (\mathsf{stk} \rightarrowtriangle A_{i+1}\ B \wr y \rightarrowtriangle y)\ (A_i\ B) & \mapsto^{\mathsf{stk}}_{\rho\delta} & (\mathsf{stk} \rightarrowtriangle A_{i+1} B)\ (f\overline{B}) \wr (y \rightarrowtriangle y)\ (f\ \overline{B}) \\ & \mapsto^{\mathsf{stk}}_{\rho\delta} & \mathsf{stk} \wr f\ \overline{B} \\ & \mapsto_{\mathsf{stk}} & f\ \overline{B} \end{array}$$

si $A_i\ B \mapsto^{\mathsf{stk}}_{\rho\delta} f\ \overline{B}$ puisque $\mathsf{stk} \not\sqsubseteq f\ \overline{B}$.

En particulier, on utilisera souvent $\mathsf{first}(A_1, \ldots, A_n, y \rightarrowtriangle y)\ B$, qui essaye les différents $A_i\ B$ et renvoie son argument B à l'identique dans le cas où tous les $A_i\ B$ échouent.

Nous sommes maintenant en mesure de donner un encodage d'un système de réécriture $\mathcal{R}$ quelconque.

Définition 10 (Encodage d'un système de réécriture) *Etant donné le système de réécriture $\mathcal{R} = \{l_i \rightarrow r_i \mid i = 1..n\}$ sur la signature $\Sigma = \{a_1, \ldots, a_m\}$ et une variable z non utilisée dans $\mathcal{R}$, le terme $(\!|\mathcal{R}|\!)$ est*

défini comme suit (avec la représentation usuelle des termes algébriques) :

$$
\begin{array}{rcl}
(\!|\mathcal{R}|\!) & \triangleq & (rec\ z) \twoheadrightarrow \mathsf{first} \left(\begin{array}{l} l_1 \twoheadrightarrow z\ (rec\ z)\ r_1, \\ \ldots \\ l_n \twoheadrightarrow z\ (rec\ z)\ r_n, \\ (a_1\overline{x}) \twoheadrightarrow z\ (Rec\ z)\ (a_1\overline{z\ (rec\ z)\ x}), \\ \ldots \\ (a_m\overline{x}) \twoheadrightarrow z\ (Rec\ z)\ (a_m\overline{z\ (rec\ z)\ x}), \end{array} \right) \\
 & & \wr(Rec\ z) \twoheadrightarrow \mathsf{first} \left(\begin{array}{l} l_1 \twoheadrightarrow z\ (rec\ z)\ r_1, \\ \ldots \\ l_n \twoheadrightarrow z\ (rec\ z)\ r_n, \\ y \twoheadrightarrow y \end{array} \right)
\end{array}
$$

Pour évaluer un terme t selon le système $\mathcal{R}$, on évalue le ρ-terme $(\!|\mathcal{R}|\!)\ (rec\ (\!|\mathcal{R}|\!))\ t$. Notons que cette représentation applique en fait une stratégie de réécriture bien précise. Dans les langages comme Maude [CDE+07] ou Obj* [FN96], elle correspond à la stratégie locale `f (0 1 2 ...n)` pour toute constante f d'arité n, doublée d'un ordre sur les règles.

En effet, les règles sont d'abord essayées en tête des termes puisque $rec\ z$ est le seul motif qui filtre $rec\ (\!|\mathcal{R}|\!)$. Si aucune règle ne convient, les termes $(a_i\overline{x}) \twoheadrightarrow \ldots$ déclenchent l'évaluation dans les sous-termes immédiats. Ensuite, quand ces sous-termes ne sont plus réductibles en tête, l'identité leur est appliquée et le terme $(\!|\mathcal{R}|\!)\ (Rec\ (\!|\mathcal{R}|\!))$ relance une réduction de tête. Quand le terme est en forme normale, l'identité s'applique et tous les $(\!|\mathcal{R}|\!)$ s'éliminent. Nous avons montré [CLW03, Wac05] que cet encodage est correct et complet pour tout système de réécriture convergent (autrement dit confluent et terminant).

Théorème 6 (Correction et complétude de l'encodage)

1. *Pour tout système de réécriture $\mathcal{R}$, pour tous termes algébriques t et t',*

$$(\!|\mathcal{R}|\!)\ (rec\ (\!|\mathcal{R}|\!))\ (\!|t|\!) \mapsto^{\mathsf{stk}}_{\rho\delta} (\!|t'|\!) \quad \Rightarrow \quad t \twoheadrightarrow_{\mathcal{R}} t' \ \textit{et}\ t' \ \textit{est en forme}\ \mathcal{R}\textit{-normale.}$$

2. *Pour tout système de réécriture convergent $\mathcal{R}$, pour tous termes algébriques t et t' tels que t' est en forme $\mathcal{R}$-normale,*

$$t \twoheadrightarrow_{\mathcal{R}} t' \quad \Rightarrow \quad (\!|\mathcal{R}|\!)\ (rec\ (\!|\mathcal{R}|\!))\ (\!|t|\!) \mapsto^{\mathsf{stk}}_{\rho\delta} (\!|t'|\!)$$

En particulier, la seconde propriété assure que $(\!|\mathcal{R}|\!)\ (rec\ (\!|\mathcal{R}|\!))\ (\!|t|\!)$ est faiblement normalisant pour tout terme algébrique t.

Pour accompagner cette démonstration, on peut montrer des contre-exemples à la complétude dans le cas où le système de réécriture $\mathcal{R}$ n'est pas convergent.

Pour le cas où $\mathcal{R}$ n'est pas confluent on peut considérer le système $\mathcal{R} = \{\mathsf{random} \to 0, \mathsf{random} \to 1\}$ et dans ce cas on a la réduction $(\!|\mathcal{R}|\!)\ (rec\ (\!|\mathcal{R}|\!))\ \mathsf{random} \mapsto^{\mathsf{stk}}_{\rho\delta} 0$ mais, par contre, la réduction $\mathsf{random} \to_{\mathcal{R}} 1$ n'est pas représentée dans le calcul.

Pour le cas où $\mathcal{R}$ n'est pas terminant on peut considérer le système

$$\mathcal{R} = \left\{ \begin{array}{lcl} g(x) & \to & g(Sx) \\ g(x) & \to & 0 \\ f(0) & \to & 0 \end{array} \right.$$

et dans ce cas on a

$$f(g(0))A \to_{\mathcal{R}} \ldots \to_{\mathcal{R}} f(g(S^n 0)) \to_{\mathcal{R}} f(0) \to_{\mathcal{R}} 0$$

mais

$(\!|\mathcal{R}|\!)\ (rec\ (\!|\mathcal{R}|\!))\ (f(g\ 0)) \mapsto^{\mathsf{stk}}_{\rho\delta}$
$\ldots \mapsto^{\mathsf{stk}}_{\rho\delta} (\!|\mathcal{R}|\!)\ (rec\ (\!|\mathcal{R}|\!))\ f((\!|\mathcal{R}|\!)\ (rec\ (\!|\mathcal{R}|\!))\ g(S^n 0)) \mapsto^{\mathsf{stk}}_{\rho\delta} \ldots$

Nous avons utilisé l'opérateur first de façon assez restrictive, mais des combinaisons plus subtiles des diverses règles du système de réécriture permettent d'encoder d'autres stratégies.

On peut modifier $(\!|\mathcal{R}|\!)$ pour simuler différentes stratégies de réduction. Par exemple, en plaçant les règles de propagation $a_i\overline{x} \twoheadrightarrow z\ (Rec\ z)\ (a_i\ z\ \overline{(rec\ z)\ x})$ au début du premier first, on obtient une stratégie *innermost* : les règles sont d'abord appliquées sur les sous-termes les plus profonds. Si au lieu de mettre toutes les règles dans un unique first, on hiérarchise un peu plus leur application, on peut obtenir un contrôle très fin sur l'application de $\mathcal{R}$.

2.6 Application distributive : ρ_d

Les systèmes de réécriture de termes et les stratégies de réécriture classiques ont été codés dans la version originale du calcul de réécriture étendu avec un opérateur de choix [CK01] et, comme montré dans la section précédente, d'une manière plus directe dans ρ_{stk}. Le premier codage est plutôt élaboré tandis que ce dernier est plus simple mais limité à des systèmes de réécriture de termes convergents. Afin d'étendre ce codage à des systèmes de réécriture arbitraires, une nouvelle règle d'évaluation qui enrichit la sémantique de l'opérateur de structure est ajoutée et une stratégie d'évaluation est imposée en forçant une certaine discipline pour l'application des règles d'évaluation. Cette stratégie est définie syntaxiquement en utilisant une notion spécifique de *valeur* et est utilisée afin de garantir la confluence du calcul qui n'est pas vérifiée dans le cas général. Le calcul obtenu est appelé le ρ-calcul distributif ou encore le ρ_d-calcul. Ce calcul a été introduit pour la première fois dans [CHW07].

$$
\begin{array}{rcll}
\mathcal{P} & ::= & \mathcal{X} \mid \mathcal{K} \mid \mathsf{stk} \mid \mathcal{K}\overline{\mathcal{P}} & \text{(Motifs)} \\
\mathcal{T} & ::= & \mathcal{X} \mid \mathcal{K} \mid \mathsf{stk} \mid \mathcal{P} \twoheadrightarrow \mathcal{T} \mid \mathcal{T}\ \mathcal{T} \mid \mathcal{T} \wr \mathcal{T} & \text{(Termes)} \\
\mathcal{V} & ::= & \mathcal{X} \mid \mathcal{K} \mid \mathcal{K}\overline{\mathcal{V}} \mid \mathcal{P} \twoheadrightarrow \mathcal{T} & \text{(Valeurs)} \\
\mathcal{V}^{\gamma} & ::= & \mathcal{V} \mid \mathcal{V}^{\gamma} \wr \mathcal{V}^{\gamma} & \text{(Valeurs structure)} \\
\mathcal{V}^{\rho\delta} & ::= & \mathcal{V} \mid \mathsf{stk} & \text{(Valeurs stuck)}
\end{array}
$$

FIGURE 2.5 – *La syntaxe de ρ_d*

2.6.1 Syntaxe et sémantique de ρ_d

Ce formalisme partage la syntaxe de ρ_{stk} et utilise la notion de *valeur* qui représente intuitivement les termes qui n'ont pas besoin d'être évalués. Les valeurs classiques comprenant des termes algébriques et des abstractions peuvent être étendues à des *valeurs structure* et à des *valeurs stuck* qui seront utilisées pour contraindre l'application des règles d'évaluation. La syntaxe du ρ_d-calcul est donnée dans la figure 2.5.

Dans cette section, on utilisera le symbole V pour dénoter les valeurs, le symbole V^{γ} pour dénoter les valeurs structure de l'ensemble $\mathcal{V}^{\gamma}$, le symbole $V^{\rho\delta}$ pour dénoter les valeurs stuck de l'ensemble $\mathcal{V}^{\rho\delta}$. Comme d'habitude, tous ces symboles peuvent être indexés.

Comme pour le ρ-calcul général, le mécanisme d'évaluation du ρ_d-calcul est basé sur l'opération fondamentale de filtrage et dans le cas général le filtrage peut être effectué modulo une congruence sur les termes. Dans cette section nous nous limitons au filtrage syntaxique, dans quel cas la substitution résultant du filtrage, quand elle existe, est unique et peut être calculée, par exemple, en utilisant l'algorithme donné dans la section 2.2.

La sémantique opérationnelle du ρ_d-calcul est donnée dans la figure 2.6. Les conditions implicites imposant que les arguments d'une application sont des valeurs sont essentiellement liées à la confluence du calcul [CHW07].

La différence principale par rapport à ρ_{stk} est la règle γ qui est la contrepartie de la règle (δ) et qui définit la distributivité à gauche de l'application par rapport à la structure. Bien que cette règle semble très naturelle et non dangereuse, elle mène à des réductions non confluentes si des restrictions sur les valeurs structure ne sont pas imposées [CHW07].

La fermeture compatible de la relation induite par les règles de ρ_d est notée $\mapsto^{\mathsf{stk}}_{\rho\delta\gamma}$. Sa fermeture transitive et réflexive est notée comme d'habitude.

Théorème 7 (Confluence de ρ_d [CHW07])
Si tous les patterns sont linéaires, la relation $\mapsto^{\mathsf{stk}}_{\rho\delta\gamma}$ est confluente.

$$
\begin{array}{rll}
(P \twoheadrightarrow A)\ V^{\rho\delta} & \to_{\rho} & A\sigma \\
 & & \textbf{avec } \{\sigma\} = \mathcal{S}ol(P \lessdot V^{\rho\delta}) \\
(A_1 \wr A_2)\ V^{\rho\delta} & \to_{\delta} & A_1\ V^{\rho\delta} \wr A_2\ V^{\rho\delta} \\
A\ (V_1^{\gamma} \wr V_2^{\gamma}) & \to_{\gamma} & A\ V_1^{\gamma} \wr A\ V_2^{\gamma} \\
(P \twoheadrightarrow A)\ B & \to_{stk} & \mathsf{stk} \\
 & & \textbf{si } P \not\sqsubseteq A \\
\mathsf{stk} \wr A & \to_{\mathsf{stk}} & A \\
A \wr \mathsf{stk} & \to_{stk} & A \\
\mathsf{stk}\ A & \to_{stk} & \mathsf{stk}
\end{array}
$$

FIGURE 2.6 – *La sémantique de ρ_d*

2.6.2 Encodage des systèmes de réécriture

Grâce à la règle (γ), qui définit la distributivité à droite de l'application par rapport à la structure, nous pouvons encoder les systèmes de réécriture potentiellement non-confluents dans le ρ_d-calcul. Plus précisément, étant donné un système de réécriture $\mathcal{R}$ nous pouvons construire les termes $\Omega^1_{\mathcal{R}}$ et $\Omega_{\mathcal{R}}$ tel que

- $\Omega^1_{\mathcal{R}}\ m$ représente (*c.-à-d.* est réduit à) le réduit en un pas de m par rapport à $\mathcal{R}$,
- $\Omega_{\mathcal{R}}\ m$ représente les formes normales de m par rapport à $\mathcal{R}$ (si elles existent).

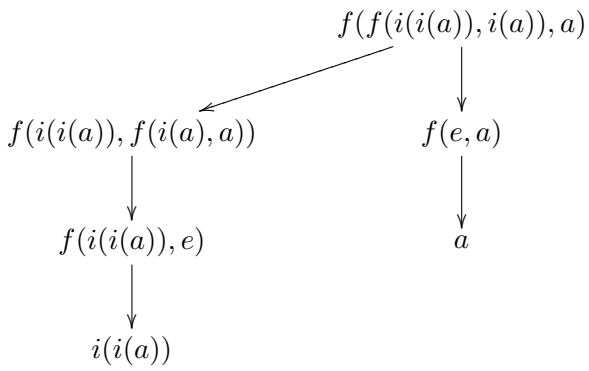
Nous montrons sur un exemple comment le ρ_d-calcul peut être utilisé pour encoder des systèmes de réécriture de termes ; plus de détails sur la méthode systématique de traduction et sur les propriétés correspondantes peuvent être trouvés dans [CHW07].

Exemple 6 (Groupes) *On considère les axiomes de la théorie des groupes orientés comme suit :*

$$
\left\{\begin{array}{l} f(x,e) \to x \\ f(e,x) \to x \end{array}\right. \qquad \left\{\begin{array}{l} f(x,i(x)) \to e \\ f(i(x),x) \to e \end{array}\right.
$$

$$
f(f(x,y),z) \to f(x,f(y,z))
$$

Ce système de réécriture n'est pas confluent car



Les termes $\omega^1_{\mathcal{R}}$, $\Omega^1_{\mathcal{R}}$, $\omega_{\mathcal{R}}$ et $\Omega_{\mathcal{R}}$ sont définis par :

$$\omega^1_{\mathcal{R}} \triangleq \pi \twoheadrightarrow \left(\begin{array}{l} f(x,e) \twoheadrightarrow x \\ f(e,x) \twoheadrightarrow x \\ f(x,i(x)) \twoheadrightarrow e \\ f(i(x),x) \twoheadrightarrow e \\ f(f(x,y),z) \twoheadrightarrow f(x,f(y,z)) \\ i(x) \twoheadrightarrow (nop(z) \twoheadrightarrow z)\ (\mathsf{first} \left(\begin{array}{l} \mathsf{stk} \twoheadrightarrow nop(\mathsf{stk}) \\ y \twoheadrightarrow nop(i(y)) \end{array}\right)\ ((\pi\ \pi)\ x)) \\ f(x_1,x_2) \twoheadrightarrow (nop(z) \twoheadrightarrow z)\ (\mathsf{first} \left(\begin{array}{l} \mathsf{stk} \twoheadrightarrow nop(\mathsf{stk}) \\ y \twoheadrightarrow nop(f(y,x_2)) \end{array}\right)\ ((\pi\ \pi)\ x_1)) \\ f(x_1,x_2) \twoheadrightarrow (nop(z) \twoheadrightarrow z)\ (\mathsf{first} \left(\begin{array}{l} \mathsf{stk} \twoheadrightarrow nop(\mathsf{stk}) \\ y \twoheadrightarrow nop(f(x_1,y)) \end{array}\right)\ ((\pi\ \pi)\ x_2)) \end{array}\right)$$

$$\Omega^1_{\mathcal{R}} \triangleq \omega^1_{\mathcal{R}}\ \omega^1_{\mathcal{R}}$$

$$\omega_{\mathcal{R}} \triangleq s \twoheadrightarrow x \twoheadrightarrow \mathsf{first}\ (\mathsf{stk} \twoheadrightarrow x)\ (z \twoheadrightarrow (s\ s)\ z)\ (\Omega^1_{\mathcal{R}}\ x)$$

et

$$\Omega_{\mathcal{R}} \triangleq \omega_{\mathcal{R}}\ \omega_{\mathcal{R}}$$

Alors on a les encodages suivants pour les réductions en un pas :

$$\begin{array}{lll} \Omega^1_{\mathcal{R}}\ f(f(i(i(a)),i(a)),a) & \overset{\mathsf{stk}}{\underset{\rho\delta\gamma}{\longmapsto\!\!\!\!\twoheadrightarrow}} & f(i(i(a),f(i(a),a)) \wr f(e,a) \\ \Omega^1_{\mathcal{R}}\ f(i(i(a)),f(i(a),a)) & \overset{\mathsf{stk}}{\underset{\rho\delta\gamma}{\longmapsto\!\!\!\!\twoheadrightarrow}} & f(i(i(a)),e) \\ \Omega^1_{\mathcal{R}}\ f(i(i(a)),e) & \overset{\mathsf{stk}}{\underset{\rho\delta\gamma}{\longmapsto\!\!\!\!\twoheadrightarrow}} & i(i(a)) \\ \Omega^1_{\mathcal{R}}\ i(i(a)) & \overset{\mathsf{stk}}{\underset{\rho\delta\gamma}{\longmapsto\!\!\!\!\twoheadrightarrow}} & \mathsf{stk} \\ \Omega^1_{\mathcal{R}}\ f(e,a) & \overset{\mathsf{stk}}{\underset{\rho\delta\gamma}{\longmapsto\!\!\!\!\twoheadrightarrow}} & a \\ \Omega^1_{\mathcal{R}}\ a & \overset{\mathsf{stk}}{\underset{\rho\delta\gamma}{\longmapsto\!\!\!\!\twoheadrightarrow}} & \mathsf{stk} \end{array}$$

et pour une réduction en forme normale on a

$$\Omega_{\mathcal{R}}\ f(f(i(i(a)),i(a)),a) \qquad \overset{\mathsf{stk}}{\underset{\rho\delta\gamma}{\longmapsto\!\!\!\!\twoheadrightarrow}} \qquad i(i(a)) \wr a$$

Cette dernière réduction reflète bien les réductions non confluentes du terme $f(f(i(i(a)),i(a)),a)$ par rapport au système de réécriture $\mathcal{R}$ puisque le résultat $i(i(a)) \wr a$ représente les deux formes normales.

2.7 Un peu d'histoire

La syntaxe du calcul de réécriture a changé plusieurs fois depuis sa première version proposée en 1998 et nous décrivons brièvement ici les différentes présentations qui peuvent être trouvées dans la littérature. Les différences entre ces présentations concernent essentiellement les notations pour l'application et les opérateurs de structure et le comportement de la structure par rapport aux autres opérateurs.

Dans la première version du ρ-calcul [CK01] l'opérateur binaire d'application était noté « $[\_](\_)$ ». La notation et la sémantique classiques d'ensemble étaient utilisées pour les structures. L'ensemble des ρ-termes de base était défini inductivement par la grammaire suivante :

$$\mathcal{T} \quad ::= \quad \mathcal{X} \mid f(\mathcal{T},\ldots,\mathcal{T}) \mid \{\mathcal{T},\ldots,\mathcal{T}\} \mid [\mathcal{T}](\mathcal{T}) \mid \mathcal{T} \twoheadrightarrow \mathcal{T}$$

Puisque les ensembles donnent dans ce cas la sémantique des résultats, les symboles « $\{\}$ » et « $\emptyset$ » représentent tous les deux l'ensemble vide et dans les termes de la forme $\{A_1,\ldots,A_n\}$ la virgule est supposée associative, commutative et idempotente.

Les règles d'évaluation ci-dessous ont été utilisées dans cette première version de calcul (on suppose donnée une théorie $\mathbb{T}$ sur les ρ-termes avec un problème de filtrage décidable) :

$$\begin{array}{rll}
[P \twoheadrightarrow A](B) & \to_{Fire} & \{A\sigma_1,\ldots,A\sigma_n\} \\
 & \text{avec} & \{\sigma_1,\ldots,\sigma_n\} = \mathcal{S}ol(P \lessdot_{\mathbb{T}} B) \\
[f(A_1,\ldots,A_n)](f(B_1,\ldots,B_n)) & \to_{Cong} & \{f([A_1](B_1),\ldots,[A_n](B_n))\} \\
[f(A_1,\ldots,A_n)](g(B_1,\ldots,B_m)) & \to_{Cong_{fail}} & \emptyset \\
[\{A_1,\ldots,A_n\}](B) & \to_{Distrib} & \{[A_1](B),\ldots,[A_n](B)\} \\
[B](\{A_1,\ldots,A_n\}) & \to_{Batch} & \{[B](A_1),\ldots,[B](A_n)\} \\
\{A_1,\ldots,A_n\} \twoheadrightarrow B & \to_{Switch_L} & \{A_1 \twoheadrightarrow B,\ldots,A_n \twoheadrightarrow B\} \\
A \twoheadrightarrow \{B_1,\ldots,B_n\} & \to_{Switch_R} & \{A \twoheadrightarrow B_1,\ldots,A \twoheadrightarrow B_n\} \\
f(B_1,\ldots,\{A_1,\ldots,A_m\},\ldots,B_n) & \to_{OpOnSet} & \\
 & & \{f(B_1,\ldots,A_1,\ldots,B_n),\ldots,f(B_1,\ldots,A_m,\ldots,B_n)\} \\
\{A_1,\ldots,\{B_1,\ldots,B_n\},\ldots,A_m\} & \to_{Flat} & \{A_1,\ldots,B_1,\ldots,B_n,\ldots,A_m\}
\end{array}$$

La règle *Fire* correspond à la règle (ρ) avec un filtrage non unitaire : l'application d'une règle de réécriture à la position de tête d'un terme est réalisée en filtrant le membre gauche de la règle de réécriture contre l'argument et en retournant le membre droit instancié correctement. Quand le filtrage mène

à un échec représenté par un ensemble vide de substitutions, le résultat de l'application de la règle *Fire* est l'ensemble vide.

Dans cette version du calcul, comme en λ-calcul, l'application peut toujours être évaluée mais, par rapport au λ-calcul, l'ensemble de résultats peut être vide. Plus généralement, quand on filtre modulo une théorie $\mathbb{T}$, l'ensemble de substitutions obtenu comme résultat peut être vide, un singleton (comme pour la théorie vide), un ensemble fini (comme pour l'associativité-commutativité) ou infini (voir [FH83]). Nous avions ainsi choisi de représenter le résultat de l'application d'une règle de réécriture à un terme par un ensemble. Un ensemble vide signifie que la règle de réécriture $P \twoheadrightarrow A$ ne s'applique pas à B (à cause d'un échec de filtrage entre P et B).

Afin de propager l'application des règles de réécriture plus profondément dans des termes, les deux règles d'évaluation *Cong* et $Cong_{fail}$ ont été introduites. Quand les deux termes de l'application $[A](B)$ ont le même symbole de tête, les arguments du terme A sont appliqués sur ceux du terme B un à un. Si les symboles de tête ne sont pas identiques, un ensemble vide est obtenu.

Les règles *Distrib* et *Batch* décrivent l'interaction entre l'application et les opérateurs d'ensemble, les règles $Switch_L$ et $Switch_R$ décrivent l'interaction entre l'abstraction et les opérateurs d'ensemble, la règle *OpOnSet* décrit l'interaction entre les symboles de la signature et les opérateurs d'ensemble.

On s'intéresse habituellement à l'ensemble de résultats obtenu par la réduction des radicaux et non pas à la trace exacte de la réduction menant à ces résultats. La règle d'évaluation *Flat* est utilisée pour aplatir les ensembles et éliminer les symboles d'ensemble (imbriqués). Notez que ceci implique que l'échec (*c.-à-d.* l'ensemble vide) n'est pas propagé strictement par rapport aux ensembles.

Dans cette présentation du calcul, les motifs algébriques étaient des termes de la forme $f(A_1, \ldots, A_n)$ construits à partir d'un symbole f d'arité fixe n tandis que dans les versions suivantes du calcul on représente les termes algébriques d'une façon *curryfiée* en utilisant l'opérateur d'application et donc, le terme ci-dessus correspond au terme $(((f\ A_1)\ A_2) \cdots\ A_n)$ dans la version courante du calcul.

Dans les versions suivantes du calcul les ensembles ont été généralisés [CKL01a] à des structures génériques dénoté d'abord par « , » et ensuite par « ≀ » et dont la sémantique est décrite par une théorie appropriée qui dépend du type de résultats que l'on veut formaliser. Typiquement, nous obtenons la sémantique initiale des ensembles de résultats en prenant une sémantique associative-commutative et idempotente pour l'opérateur de structure. Si on préfère des listes ou des multi-ensembles de résultats, alors la formalisation correspondante de l'opérateur de structure doit être spécifiée.

La deuxième version du calcul [CKL01a] utilise une notation différente pour l'application (« • ») et la structure vide est dénotée par « *null* ». La

syntaxe de ce ρ-calcul simplifié est définie par :

$$\mathcal{T} \quad ::= \quad \mathcal{X} \mid \mathcal{K} \mid \mathcal{T} \twoheadrightarrow \mathcal{T} \mid \mathcal{T}\bullet\mathcal{T} \mid \mathcal{T},\mathcal{T} \mid \mathit{null}$$

La sémantique opérationnelle est définie par les règles :

$$\begin{array}{rcl}
(A_1 \twoheadrightarrow A_2)\bullet A_3 & \to_\rho & \begin{cases} \mathit{null} & \text{si } A_1 \lessdot_{\mathbb{T}} A_3 \text{ n'a pas de solution} \\ A_2\sigma_1, \ldots, A_2\sigma_n & \text{si } \sigma_i \in \mathcal{S}ol(A_1 \lessdot_{\mathbb{T}} A_3) \end{cases} \\
(A_1, A_2)\bullet A_3 & \to_\delta & A_1\bullet A_3, A_2\bullet A_3 \\
\mathit{null}\bullet t & \to_\nu & \mathit{null}
\end{array}$$

Cette sémantique est une simplification de la précédente qui élimine les règles d'évaluation dont le comportement peut être simulé par des constructions appropriées. Bien que la notation $f(A_1, \ldots, A_n)$ soit toujours utilisée parfois pour dénoter le terme $(((f\bullet A_1)\bullet A_2)\cdots\bullet A_n)$ les règles *OpOnSet* et $Cong_{fail}$ ne sont plus nécessaires. L'aplatissement des structures est maintenant contrôlé par la théorie sous-jacente de la structure et la distributivité de l'opérateur de structure par rapport à l'application est définie par les règles (δ) et (ν) (correspondant à l'ancienne règle *Distrib*).

Le symbole « *null* » représente l'échec (de filtrage) et afin d'obtenir une granularité plus fine permettant de faire la différence entre différents échecs, les contraintes explicites ont été introduites dans la syntaxe [CKL02]. Cette construction peut également être vue comme un premier pas vers une version explicite du calcul [CFK04]. La syntaxe de cette version du ρ-calcul est définie par :

$$\mathcal{T} \quad ::= \quad \mathcal{X} \mid \mathcal{K} \mid \mathcal{T} \twoheadrightarrow \mathcal{T} \mid \mathcal{T}[\mathcal{T} \ll \mathcal{T}] \mid \mathcal{T}\ \mathcal{T} \mid \mathcal{T},\mathcal{T}$$

Les termes de la forme $A_3[A_1 \ll A_2]$ [1] dénotent des *contraintes de filtrage en attente* et représentent une évolution importante en ce qui concerne la syntaxe et le comportement du calcul. L'application d'une abstraction $A_1 \twoheadrightarrow A_3$ à un terme A_2 est toujours déclenchée et produit le terme $A_3[A_1 \ll A_2]$ qui représente un terme contraint où l'équation de filtrage « est mise sur la pile ». Si une solution σ du filtrage entre A_1 et A_2 existe alors le terme contraint peut être évalué à $A_3\sigma$.

La sémantique de ce calcul est définie par les règles de réduction suivantes :

$$\begin{array}{rcll}
(A_1 \twoheadrightarrow A_2)A_3 & \to_\rho & A_2[A_1 \ll A_3] & \\
A_2[A_1 \ll A_3] & \to_\sigma & A_2\sigma_1, \ldots, A_2\sigma_n & \text{if } \sigma_i \in \mathcal{S}ol(A_1 \lessdot_{\mathbb{T}} A_3) \\
(A_1, A_2)\,A_3 & \to_\delta & A_1\,A_3, A_2\,A_3 &
\end{array}$$

1. la notation $[A_1 \ll A_2].A_3$ a été utilisée quand cette construction a été présentée pour la première fois

Puisque les échecs (de filtrage) sont conservés tels quels dans cette version du calcul, la règle (ρ) devient plus simple et la règle (ν) n'est plus nécessaire. D'autre part, la règle (σ) est ajoutée pour décrire la sémantique des contraintes de filtrage en attente. On peut noter que si on considère une théorie vide de filtrage cette présentation est moralement identique à celle présentée au début de ce chapitre ; les règles (ρ) et (σ) sont fusionnées dans une seule règle (ρ).

2.8 Conclusions

Les travaux décrits dans ce chapitre ont eu pour but la conception d'un formalisme permettant d'expliciter les mécanismes de la réécriture avec stratégies et, en particulier, d'exprimer d'une manière uniforme la sémantique des langages à base de règles et stratégies (ASF+SDF [vD96], Elan [BKKM02], Maude [CDE$^+$07], Obj* [FN96], Stratego [Vis01], Tom [BBK$^+$07], *etc.*). Ce travail a débuté à la fin des années '90 quand j'avais proposé en collaboration avec Claude Kirchner la première version du calcul de réécriture. Le formalisme a évolué pour arriver à sa présentation actuelle en 2003. Nous nous sommes focalisé dans ce chapitre sur cette dernière version du calcul de réécriture et sur deux versions qui utilisent une constante spéciale stk pour représenter les échecs de filtrage.

Par rapport aux versions précédentes du calcul présentées brièvement dans la dernière section de ce chapitre, la version actuelle est plus simple mais a exactement le même pouvoir d'expression. L'encodage originel des systèmes de réécriture utilisait un traitement plus atomique des échecs de filtrage basé sur l'introduction d'un opérateur primitif appelé first. Dans la version actuelle nous ne disposons pas d'un tel opérateur et, de plus, les règles de congruence et de distributivité ne sont plus disponibles. Nous avons montré que le comportement de l'opérateur first peut être néanmoins encodé cette fois-ci au niveau objet du calcul (grâce à la la constante stk). De plus, nous allons voir que ce nouvel encodage est typable.

Puisque le calcul est une extension conservative du λ-calcul et de la réécriture, les problèmes de confluence liés à l'ordre supérieur et à la non-linéarité sont présents ici aussi. Pour récupérer la confluence, nous avions déjà proposé des stratégies d'évaluation appropriées. Les mêmes conditions sur l'application des règles d'évaluation peuvent être utilisées pour les calculs présentés ici et nous avons montré que, dans certains cas, ces conditions peuvent être remplacées par des restrictions syntaxiques sur les motifs utilisés. Les conditions syntaxique utilisées pour le calcul général ne sont plus suffisantes dans le cas du ρ_d-calcul où une notion de valeur est utilisée pour exprimer une stratégie de type appel par valeurs.

Contexte :

Le calcul de réécriture permet la manipulation des règles de réécriture et la définition des stratégies guidant leur application au niveau objet du calcul, et fournit ainsi une sémantique uniforme aux langages basées sur les règles et les stratégies. L'encodage des programmes Elan proposé initialement dans [CK01] ainsi que la version simplifiée présentée dans le chapitre précédent sont réalisées dans des versions non typée du ρ-calcul mais il est bien connu qu'une analyse statique basée sur des systèmes de types appropriés impose une discipline de programmation plus stricte.

En effet, les langages à basé règles sont traditionnellement typés et ils cherchent à garantir un bon niveau de sûreté sans sacrifier leur expressivité. Ces objectifs sont en fait les motivations principales pour la recherche des formalismes qui offrent un bon compromis entre la flexibilité et l'expressivité d'une approche non typée, et la rigueur d'une approche typée.

Le premier système de types proposé pour le calcul de réécriture [CK00] s'inscrit partiellement dans cette philosophie puisqu'il garantit la consistance au niveau des types mais ne permet pas le typage des programmes potentiellement non-terminants.

Nous avons essayé d'améliorer cette première approche pour traiter les encodages de la réécriture stratégique et plusieurs questions sont apparues naturellement dans ce contexte.

Questions :

- Quels systèmes de types dans cette optique de programmation ?
- Des décorations de type explicites ou des types inférés automatiquement ?
- Quelles sont les propriétés des systèmes de types ainsi obtenus ?

Contributions :

Nous avons tout d'abord proposé un système de types de premier ordre pour le ρ_{stk}-calcul. Dans ce formalisme, le filtrage permet la construction et le typage de termes non-normalisant utilisés pour la définition d'« opérateurs de récursion fonctionnels ». Par ailleurs, le système de types proposé est suffisamment puissant pour assurer un typage correct pour les équations de filtrage obtenues au cours de la réduction, c'est-à-dire l'instanciation des paramètres formels est conforme à la discipline des types. Nous considérons donc ce système de types pour le ρ-calcul comme un bon candidat pour donner la sémantique statique de toute une famille de langages basés sur la réécriture.

Nous avons également proposé un ρ-calcul typé qui dispose d'une forme restreinte de polymorphisme qui correspond à une programmation avec des règles et des stratégies où l'utilisateur peut écrire des programmes dans le langage non typé, et des types sont automatiquement inférés au moment de la compilation.

Nous avons montré que les versions avec types simples et polymorphisme vérifient la plupart des propriétés usuelles : préservation du type, unicité et décidabilité du typage.

3

Systèmes de types

Nous avons montré [CK01] que le calcul de réécriture peut être utilisé comme une sémantique opérationnelle pour des langages à base de règles. Les encodages proposés utilisent des opérateurs de point fixe inspirés des ceux du λ-calcul qui ne sont pas typables dans la version simplement typé [CK00] du calcul de réécriture initial.

L'analyse statique par l'intermédiaire d'un système de types approprié impose néanmoins une discipline de programmation plus stricte. Les systèmes de types proposés dans le cadre du λ-calcul et des langages fonctionnels ont été utilisés comme point de départ pour le premier système de types pour le ρ-calcul mais les restrictions plutôt strictes imposées aux termes concernés ne permettent pas un traitement complet des encodages des règles et des stratégies de réécriture. Nous présentons ici une version typée du ρ_{stk}-calcul avec des types de premier ordre et des termes auto-réproductibles [CLW03].

Ce système de types a été étendu [LW05] pour obtenir un ρ-calcul *à la* Church avec des types polymorphes de second ordre. Dans ce calcul de réécriture de second ordre, les types des variables liées sont spécifiés dans les termes, conduisant ainsi à une reconstruction et une vérification de type relativement simple. Dans le calcul polymorphe *à la* Curry [LW05, Wac05], puisque l'information de type n'est pas donnée dans le terme et que le système de types n'est pas entièrement dirigé par la syntaxe on obtient une discipline de types particulièrement flexible. Malheureusement, comme pour le λ-calcul [Wel99], la vérification et l'inférence de types dans le système polymorphe *à la* Curry sont des problèmes indécidables.

Nous présentons dans la deuxième partie de ce chapitre une restriction du système général, appelée $\rho^{\forall}_{\ll}$ [CKLW06], où les types polymorphes sont clairement séparés des schémas de type polymorphes. Nous montrons son expressivité et nous présentons un algorithme d'inférence de types. Ce système peut être vu comme un bon candidat pour donner la sémantique statique d'une famille de langages de programmation basés sur le filtrage et la réécriture.

$$
\begin{array}{rcll}
\mathcal{T}^\forall & ::= & \mathcal{K}^\forall \mid \mathcal{T}^\forall \to \mathcal{T}^\forall & \text{(Types simples)} \\
\Delta & ::= & \emptyset \mid \Delta, \mathcal{X}{:}\mathcal{T}^\forall \mid \Delta, \mathcal{K}{:}\mathcal{T}^\forall & \text{(Contextes)} \\
\mathcal{P} & ::= & \mathcal{X} \mid \mathcal{K} \mid \mathsf{stk} \mid \mathcal{K}\overline{\mathcal{P}} & \text{(Motifs)} \\
\mathcal{T} & ::= & \mathcal{X} \mid \mathcal{K} \mid \mathsf{stk} \mid \mathcal{P} \twoheadrightarrow_\Delta \mathcal{T} \mid \mathcal{T}\ \mathcal{T} \mid \mathcal{T} \wr \mathcal{T} & \text{(Termes)}
\end{array}
$$

FIGURE 3.1 – *Syntaxe de ρ_{stk} typé*

Les travaux décrits dans ce chapitre ont été réalisés en collaboration avec Claude Kirchner, Luigi Liquori et Benjamin Wack.

3.1 Types simples

Nous présentons ici la version simplement typée du ρ_{stk}-calcul introduit dans la Sections 2.5. Dans ce calcul disposant d'un système de types approprié on peut non seulement encoder mais également typer des points fixes (orientés objet) et des systèmes de réécriture de termes.

3.1.1 Système de types à la Church pour ρ_{stk}

Le premier système de types que nous allons définir est une adaptation directe des types simples du λ-calcul [Bar92] au ρ-calcul. La syntaxe du calcul typé, appelée $\rho^{\twoheadrightarrow}_{\mathsf{stk}}$, est donnée dans la figure 3.1.

On considère des contextes qui associent un unique type simple à chaque variables de $\mathcal{X}$ et à chaque constante de $\mathcal{K}$ (ce qui est plus précis que les arités utilisées pour les termes algébriques). En λ-calcul, à chaque λ-abstraction on associe le type de la variable correspondante mais comme ici on a un motif plus élaboré, il faut associer à toute abstraction un contexte Δ qui précise les types des variables sur lesquelles on abstrait.

On utilise le symbole τ pour dénoter les types de l'ensemble $\mathcal{T}^\forall$, le symbole ι pour les constantes de type de l'ensemble $\mathcal{K}^\forall$ ($\mathcal{K}^\forall \subseteq \mathcal{T}^\forall$) et les symboles $\Delta, \Gamma, \ldots$ pour les contextes.

Exemple 7 *Dans la règle* $\mathsf{not}(\mathsf{and}(x,y)) \twoheadrightarrow \mathsf{or}(\mathsf{not}(x),\mathsf{not}(y))$ *du ρ-calcul non-typé les variables x et y sont liées. Dans le calcul typé, ce terme peut s'écrire* $\mathsf{not}(\mathsf{and}(x,y)) \twoheadrightarrow_{x:\tau_1,y:\tau_2} \mathsf{or}(\mathsf{not}(x),\mathsf{not}(y))$.

Les règles de typage sont présentées dans la figure 3.2.

(Var), (Const) : Le type des variables et des constantes est déterminé à partir d'un contexte (qui ne peut pas contenir deux déclarations pour la même variable ou constante).

$$\frac{x{:}\tau \in \Gamma}{\Gamma \vdash x : \tau}\,(\mathsf{Var}) \qquad \frac{f{:}\tau \in \Gamma}{\Gamma \vdash f : \tau}\,(\mathsf{Const}) \qquad \frac{\Gamma \vdash A : \tau \quad \Gamma \vdash B : \tau}{\Gamma \vdash A \wr B : \tau}\,(\mathsf{Struct})$$

$$\frac{\tau \in \mathcal{T}^{\forall}}{\Gamma \vdash \mathsf{stk} : \tau}\,(\mathsf{Stuck}) \qquad \frac{\Gamma \vdash A : \tau_1 \to \tau_2 \quad \Gamma \vdash B : \tau_1}{\Gamma \vdash A\ B : \tau_2}\,(\mathsf{Appl})$$

$$\frac{\Gamma, \Delta \vdash P : \tau_1 \quad \Gamma, \Delta \vdash A : \tau_2 \quad \mathcal{D}om(\Delta) = \mathcal{FV}(P)}{\Gamma \vdash P \twoheadrightarrow_{\Delta} A : \tau_1 \to \tau_2}\,(\mathsf{Abs})$$

FIGURE 3.2 – *Système de types simples pour* ρ_{stk}

(Struct) : Cette règle impose que tous les éléments d'une même structure aient le même type. Ceci correspond à une interprétation des structures comme des collections de résultats et donc, si une fonction peut retourner plusieurs résultats on voudrait qu'ils aient tous le même type. Alternativement, on pourrait utiliser une règle plus générale

$$\frac{\Gamma \vdash A : \tau_2 \quad \Gamma \vdash B : \tau_1}{\Gamma \vdash A \wr B : \tau_1 \wedge \tau_2}\,(\mathsf{Struct}')$$

et une notion de sous-typage permettant d'éliminer les $\wedge$. Cette notion de *type conjonction* ne doit pas être confondue avec les types intersections [BCD83], dans lesquels τ_1 et τ_2 doivent être deux types admissibles pour un même terme. Une notion basique de type conjonction pour le ρ-calcul a été étudiée dans [LS04].

(Stuck) : Étant donné que stk peut apparaître dans toute structure, la préservation du type n'est possible que si le terme stk peut avoir tous les types possibles. On discutera brièvement par la suite des alternatives permettant un typage plus fins de l'échec.

(Appl) : Comme dans le λ-calcul, on utilise directement l'information donnée dans le type « fonctionnel » du terme à appliquer et on vérifie statiquement que son argument a le type τ_1 attendu.

(Abs) : La règle pour l'abstraction ne se contente plus de copier le type de la variable abstraite à gauche de la flèche mais elle doit calculer un type pour le motif P et charger tout Δ dans le contexte. Comme mentionné précédemment, les types de toutes les variables liées par l'abstraction doivent être donnés dans le contexte correspondant, *c.-à-d.* $\mathcal{D}om(\Delta) = \mathcal{FV}(P)$.

Exemple 8 (Dérivations simples) *La règle* (Appl) *est utilisée pour le typage des applications d'abstraction mais également pour les termes*

algébriques. Etant donné $\Gamma \triangleq f{:}\iota \to \iota, a{:}\iota$ *on a*

$$\frac{\Gamma \vdash f : \iota \to \iota \quad \Gamma \vdash a : \iota}{\Gamma \vdash f\ a : \iota}\,(\mathsf{Appl})$$

et si $\Gamma \triangleq f{:}\tau_1 \to \tau_2, g{:}\tau_1 \to \iota, a{:}\tau_1$ *alors*

$$\frac{\dfrac{\Gamma, x{:}\tau_1 \vdash f\ x : \tau_2 \quad \Gamma, x{:}\tau_1 \vdash g\ x : \iota}{\Gamma \vdash f\ x \twoheadrightarrow_{(x:\tau_1)} g\ x : \tau_2 \to \iota}\,(\mathsf{Abs}) \qquad \Gamma \vdash f\ a : \tau_2}{\Gamma \vdash (f\ x \twoheadrightarrow_{(x:\tau_1)} g\ x)\ (f\ a) : \iota}\,(\mathsf{Appl})$$

Comme mentionné précédemment, la règle pour les structures est importante quand on les considère comme des collections des résultats. Si $\Gamma \triangleq 3{:}\iota, 4{:}\iota$ *alors*

$$\frac{\dfrac{}{\Gamma \vdash 3 : \iota}\,(\mathsf{Const}) \qquad \dfrac{}{\Gamma \vdash 4 : \iota}\,(\mathsf{Const})}{\Gamma \vdash 3 \wr 4 : \iota}\,(\mathsf{Struct})$$

Ce système de types a été conçu comme une discipline de types pour un langage de programmation : son but est de s'assurer que les arguments d'une fonction ont les mêmes types que les paramètres formels correspondants. La notion de motif bien typé utilisée ici est cruciale puisqu'elle permet de garantir que l'instanciation des variables du motif sera correcte par rapport aux types (*c.-à-d.* les substitutions obtenues comme résultat du filtrage sont *bien typées*) même si aucune vérification de type n'est effectuée dans l'algorithme de filtrage.

La sémantique de ρ_{stk} est celle donnée dans la section 2.5 mais dans le cas typé nous devons considérer des substitutions *bien typées* et identifier des théories de filtrage que nous pourrons utiliser en accord avec les types.

Définition 11 (Substitution bien typée) *Dans un contexte* Γ*, une substitution* θ *est dite* bien typée relativement à un contexte Δ *si*

1. $\mathcal{D}om(\theta) = \mathcal{D}om(\Delta)$ *;*
2. *pour tout* $(x{:}\tau) \in \Delta$*, on a* $\Gamma \vdash x\theta : \tau$*.*

Définition 12 (Théorie bien typée) *Une théorie équationnelle* $\mathbb{T}$ *est une théorie de filtrage* bien typée *si, pour tous termes* P *et* A *tels que* $\Gamma, \Delta \vdash P : \tau$ *et* $\Gamma \vdash A : \tau$ *avec* $\mathcal{D}om(\Delta) = \mathcal{FV}(P)$*, toute solution de l'équation de filtrage* $P \lessdot_{\mathbb{T}} A$ *est bien typée relativement à* Δ*.*

Dans ce chapitre et plus généralement dans le cadre des ρ-calculs typés, nous ne considérerons que des théories de filtrage bien typées et finitaires (*i.e.* tout problème de filtrage admet un nombre fini de solutions).

Proposition 1 (Filtrage syntaxique et typage [Wac05]) *La théorie syntaxique est une théorie de filtrage finitaire et bien typée.*

3.1.2 Propriétés du calcul simplement typé

Nous regardons dans cette section les propriétés vérifiées par le système de types simple. Le premier lemme indique que le typage est dirigé par la syntaxe, *c.-à-d.* que toute dérivation de typage dépend de la forme du terme à typer. Ce résultat est obtenu par simple inspection des règles de typage et il est utilisé pour montrer les autres propriétés du calcul.

Lemme 1 (Lemme de génération) *Dans une dérivation de typage ayant pour conclusion* $\Gamma \vdash A : \tau$ *la dernière règle utilisée dépend uniquement de la forme de* A *; si*

($A \equiv x$) *alors la dernière règle utilisée est* (Var) *et* $(x{:}\tau) \in \Gamma$ *;*

($A \equiv f$) *alors la dernière règle utilisée est* (Const) *et* $(f{:}\tau) \in \Gamma$ *;*

($A \equiv A_1 \wr A_2$) *alors la dernière règle utilisée est* (Struct) *avec* $\Gamma \vdash A_1 : \tau$ *et* $\Gamma \vdash A_2 : \tau$ *;*

($A \equiv A_1\ A_2$) *alors la dernière règle utilisée est* (App) *et il existe un type* τ' *tel que* $\Gamma \vdash A_1 : \tau' \to \tau$ *et* $\Gamma \vdash A_2 : \tau'$.

($A \equiv P \twoheadrightarrow_\Delta A$) *alors la dernière règle utilisée est* (Abs) *et* $\tau \equiv \tau_1 \to \tau_2$ *avec* $\Gamma, \Delta \vdash P : \tau_1$ *et* $\Gamma, \Delta \vdash A : \tau_2$ *;*

($A \equiv \mathsf{stk}$) *alors la dernière règle utilisée est* (Stuck).

Un autre résultat important obtenu par récurrence sur la structure des termes est la clôture du typage par substitution :

Lemme 2 (Lemme de substitution) *Pour tous contextes* Γ, Δ, *pour tout terme* A *et tout type* τ, *si* θ *est une substitution bien typée relativement à* Δ *et si* $\Gamma, \Delta \vdash A : \tau$, *alors* $\Gamma \vdash A\theta : \tau$.

En utilisant ce dernier lemme, on peut démontrer (par récurrence sur la structure des termes) la préservation du type, une propriété fondamentale qui assure qu'un objet ne change pas de type au cours de ses réductions.

Théorème 8 (Préservation du type [CLW03]) *Si* $\Gamma \vdash A : \tau$ *et si* $A \overset{\mathsf{stk}}{\underset{\rho^\delta}{\mapsto}} A'$ *alors* $\Gamma \vdash A' : \tau$.

La préservation du type dans le cas d'un échec de filtrage est obtenue grâce au typage de stk qui peut avoir tous les types possibles. Alternativement, on pourrait considérer une *famille* de constantes stk_τ qui garde l'information du type du terme qui a été à l'origine de l'échec mais dans ce cas il faudrait faire des vérifications de type dans les règles d'évaluation $\to_{stk}$.

Une seconde propriété importante pour les questions d'inférence et de vérification de types est l'unicité du type. Cette propriété est montrée par

récurrence sur la structure des termes et en utilisant le lemme 1 de génération. Il faut noter que les cas de base des variables et des constantes sont déterminés de façon unique par le contexte correspondant. Puisque stk peut avoir tous les types possibles, cette propriété n'est pas vérifiée pour le terme stk.

Théorème 9 (Unicité du type [CLW03]) *Pour tout contexte* Γ*, pour tout terme* A *sans* stk*, si* $\Gamma \vdash A : \tau_1$ *et* $\Gamma \vdash A : \tau_2$ *alors* $\tau_1 \equiv \tau_2$*.*

Ce théorème garanti l'unicité du type pour les termes qui ne contiennent pas un sous-terme stk. En fait, d'autres conditions moins restrictives peuvent être imposées pour obtenir l'unicité du type en présence du stk. Par exemple, si stk n'apparaît que dans des structures dont au moins un membre n'est pas stk, alors le type de stk dans cette structure est imposé par les autres termes et donc le typage reste unique. Cette condition est respectée par tous les termes que nous avons utilisés dans la section 2.5 et que nous utiliseront par la suite. Il faut néanmoins noter que cette condition n'est pas préservée par la réduction.

On peut maintenant énoncer les résultats de décidabilité des problèmes de vérification et d'inférence de types.

Théorème 10 (Décidabilité du typage [CLW03]) *Les deux problèmes suivants sont décidables :*

1. *Étant donnés un contexte* Γ*, un type* τ *et un terme* A *sans* stk*, peut-on dériver* $\Gamma \vdash A : \tau$ *?*
2. *Étant donnés un contexte* Γ *et un terme* A *sans* stk*, existe-t-il un type* τ *tel que* $\Gamma \vdash A : \tau$ *?*

Pour montrer la décidabilité du problème de vérification il suffit de se ramener au problème d'inférence et montrer sa décidabilité. Plus précisément, pour décider si $\Gamma \vdash A : \tau$, il suffit de chercher un type τ' tel que $\Gamma \vdash A : \tau'$. Si un tel type n'existe pas, alors τ ne peut pas être un type valide pour A. Si τ' existe alors, par unicité, τ est un type valide pour A si et seulement si $\tau \equiv \tau'$.

On peut donc se contenter de donner des algorithmes d'inférence de types. Nous donnons un algorithme possible permettant de construire τ' (ou retournant false s'il n'existe pas) et nous montrerons dans la section 3.2 les propriétés de l'algorithme généralisé pour le cas polymorphe.

$\mathsf{Type}(A;\Gamma) \triangleq$

$$
\begin{array}{lll}
A \equiv x & \Rightarrow & \tau \\
 & & \text{ssi } (x{:}\tau) \in \Gamma \\
A \equiv f & \Rightarrow & \tau \\
 & & \text{ssi } (f{:}\tau) \in \Gamma \\
A \equiv A_1 \wr A_2 & \Rightarrow & \mathsf{Type}(A_1;\Gamma) \\
 & & \text{ssi } \mathsf{Type}(A_1;\Gamma) = \mathsf{Type}(A_2;\Gamma) \\
A \equiv P \twoheadrightarrow_\Delta A_1 & \Rightarrow & \mathsf{Type}(P;\Gamma,\Delta) \to \mathsf{Type}(A_1;\Gamma,\Delta) \\
 & & \text{ssi } \mathsf{Type}(P;\Gamma,\Delta) \neq \mathsf{false} \neq \mathsf{Type}(A_1;\Gamma,\Delta) \\
A \equiv A_1\ A_2 & \Rightarrow & \tau_2 \\
 & & \text{ssi } \mathsf{Type}(A_1;\Gamma) = \tau_1 \to \tau_2 \ \wedge\ \mathsf{Type}(A_2;\Gamma) = \tau_1 \\
\text{sinon} & \Rightarrow & \mathsf{false}
\end{array}
$$

Nous pouvons maintenant utiliser cet algorithme pour résoudre le premier problème :

$$\mathsf{Typecheck}(A;\Gamma;\tau) \quad \triangleq \quad \text{si } \mathsf{Type}(A;\Gamma) = \tau \text{ alors Vrai else Faux}$$

3.1.3 Typage des encodages des systèmes de réécriture

Il est évident que le ρ_{stk}-calcul typé est une extension conservative du λ-calcul typé : un λ-terme représenté en ρ-calcul admet un type τ si et seulement s'il admet le même type dans le système de types correspondant en λ-calcul. On doit néanmoins noter que la normalisation forte, l'une des propriétés essentielles du λ-calcul typé, n'est pas conservée. Même si les termes habituellement rejetés par les systèmes de types du λ-calcul sont également rejetés par le système de types de ρ_{stk}, il existe des termes non normalisants propres au ρ-calcul qui sont acceptés par le système de types de ρ_{stk}.

Par exemple, le terme $(x \twoheadrightarrow x\ x)\ (x \twoheadrightarrow x\ x)$ correspondant au classique λ-terme $\omega\omega$ n'admet pas une dérivation de type dans ρ_{stk}. Ceci est due au fait qu'on essaye d'appliquer x à lui-même, et donc que x (en tant qu'argument) devrait être dans son propre domaine (en tant que fonction). Au niveau du système de types ceci reviendrait, intuitivement, à avoir des types $\tau = \tau \to \tau'$, ce qui est impossible car aucun type τ ne peut être un de ses sous-termes.

Pour typer un terme de cette forme, il faudrait donc trouver un moyen pour que x accepte x lui-même comme argument. En λ-calcul la solution classique est celle des *types récursifs* qui permettent de typer tout λ-terme [Cop85]. Dans le ρ-calcul, grâce au filtrage, on peut obtenir un résultat similaire sans utiliser des types récursifs : prenons une variable x de type $\iota \to \kappa$; pour obtenir un terme de type ι, il nous suffit de considérer

une constante f de type $(\iota \to \kappa) \to \iota$. On a alors les dérivations de typage suivantes, où $\Delta \triangleq (x{:}\iota \to \kappa)$ et $\Gamma \triangleq (f{:}(\iota \to \kappa) \to \iota)$:

$$\dfrac{\dfrac{\Gamma \vdash f : (\iota \to \kappa) \to \iota \quad \Delta \vdash x : \iota \to \kappa}{\Gamma, \Delta \vdash f\ x : \iota}\,(\mathsf{Appl}) \qquad \dfrac{\Delta \vdash x : \iota \to \kappa \quad \dfrac{\vdots}{\Gamma, \Delta \vdash f\ x : \iota}}{\Gamma, \Delta \vdash x\ (f\ x) : \kappa}\,(\mathsf{Appl})}{\Gamma \vdash f\ x \twoheadrightarrow_\Delta x\ (f\ x) : \iota \to \kappa}\,(\mathsf{Abs})$$

Si on définit $\omega_f \triangleq f\ x \twoheadrightarrow_\Delta x\ (f\ x)$ alors on a la dérivation :

$$\dfrac{\dfrac{\vdots}{\Gamma \vdash \omega_f : \iota \to \kappa} \qquad \dfrac{\dfrac{}{\Gamma \vdash f : (\iota \to \kappa) \to \iota} \quad \dfrac{\vdots}{\Gamma \vdash \omega_f : \iota \to \kappa}}{\Gamma \vdash f\ \omega_f : \iota}\,(\mathsf{Appl})}{\Gamma \vdash \omega_f\ (f\ \omega_f) : \kappa}\,(\mathsf{Appl})$$

On peut maintenant remarquer que ce dernier terme admet une chaîne infinie de réductions :

$$\begin{array}{rcl} \omega_f\ (f\ \omega_f) & \equiv & ((f\ x) \twoheadrightarrow_\Delta x\ (f\ x))\ (f\ \omega_f) \\ & \mapsto^{\mathsf{stk}}_{\rho\delta} & x\ (f\ x)\{\omega_f/x\} \\ & = & \omega_f\ (f\ \omega_f) \\ & \mapsto^{\mathsf{stk}}_{\rho\delta} & \ldots \end{array}$$

Des types similaires peuvent être utilisés pour typer les ρ-termes permettant l'encodage des calculs à objets et en particulier le $\varsigma\mathcal{O}bj$ [AC96b]. Nous avons montré que l'utilisation du filtrage permet d'améliorer l'encodage proposé dans le ρ-calcul non-typé [CKL01a] et que les termes correspondants peuvent être typés en utilisant le système de types présenté dans cette section [CLW03]. Dans notre approche, une méthode peut être encodée par le terme $(m\ \mathcal{S}) \twoheadrightarrow \mathcal{T}_m$, où la constante m représente le nom de la méthode, la variable $\mathcal{S}$ joue le rôle du mot-clé this (contenant une copie de l'objet) et $\mathcal{T}_m$ est le terme encodant le corps de la méthode. Un objet *obj* est alors une structure contenant des méthodes. La méthode concrète *meth* est appelée en utilisant l'application de Kamin [Kam88] $obj.meth \triangleq obj\ (meth\ obj)$ et dans ce cas on obtient :

$$\begin{array}{rcl} obj\ (meth\ obj) & \equiv & (\ldots \wr\ meth\ \mathcal{S} \twoheadrightarrow \mathcal{T}_{meth} \wr \ldots)\ (meth\ obj) \\ & \mapsto^{\mathsf{stk}}_{\rho\delta} & \ldots \wr\ \mathcal{T}_{meth}\{obj/\mathcal{S}\} \wr \ldots \\ & \mapsto^{\mathsf{stk}}_{\rho\delta} & \mathcal{T}_{meth}\{obj/\mathcal{S}\} \end{array}$$

puisque les autres méthodes échouent à cause du filtrage et les stk obtenus sont éliminés successivement.

Les termes correspondant aux objets et aux appels de méthode peuvent être typés par notre système. Pour des raison de simplicité on considère un objet $obj \triangleq meth\ \mathcal{S} \twoheadrightarrow \mathcal{T}_{meth}$ qui contient une seule méthode $meth$ avec le corps $\mathcal{T}_{meth}$. Si κ est la constante de type des étiquettes et si on considère $\Delta = (\mathcal{S}{:}\kappa \twoheadrightarrow \tau)$ et $\Gamma = (meth{:}(\kappa \twoheadrightarrow \tau) \to \kappa, \ldots)$ alors on a :

$$\frac{\Gamma, \Delta \vdash meth\ \mathcal{S} : \kappa \quad \Gamma, \Delta \vdash \mathcal{T}_{meth} : \tau}{\Gamma \vdash meth\ \mathcal{S} \twoheadrightarrow_{\Delta} \mathcal{T}_{meth} : \kappa \twoheadrightarrow \tau}\ (\mathsf{Abs})$$

Il faut noter que puisque $\mathcal{S}$ correspond à this il est naturel que $\mathcal{S}$ et obj aient le même type.

Pour l'application de la méthode $obj.meth \triangleq obj\ (meth\ obj)$ on obtient :

$$\frac{\Gamma \vdash obj : \kappa \twoheadrightarrow \tau \quad \Gamma \vdash meth\ obj : \kappa}{\Gamma \vdash obj\ (meth\ obj) : \tau}$$

En utilisant la même idée que pour le terme ω_f on peut définir le terme $\Omega \triangleq (loop\ \mathcal{S}) \twoheadrightarrow_{\Delta} \mathcal{S}.loop$ qui n'est rien d'autre qu'une version orientée objet du terme $\omega\omega$. Ce terme est bien typée :

$$loop : (\kappa \twoheadrightarrow \tau) \to \kappa \vdash (loop\ \mathcal{S}) \twoheadrightarrow_{\Delta} \mathcal{S}.loop : \kappa \twoheadrightarrow \tau$$

mais peut conduire à des réductions divergentes :

$$\begin{array}{rcl} \Omega.loop & \equiv & (loop\ \mathcal{S} \twoheadrightarrow_{\Delta} \mathcal{S}.loop)\ (loop\ \Omega) \\ & \mapsto\!\!\!\!\twoheadrightarrow_{\rho\mathsf{w}\delta} & \Omega.loop \\ & \mapsto\!\!\!\!\twoheadrightarrow_{\rho\mathsf{w}\delta} & \ldots \end{array}$$

Nous reprenons maintenant les différents ρ-termes utilisés dans la section 2.5.2 pour définir des encodages de systèmes de réécriture et nous montrons que tous ces termes sont typables dans le système de types simples de ρ_{stk}-calcul, à condition de bien choisir les types des constantes utilisées.

On considère le terme introduit dans la section 2.5.2 pour encoder un système de réécriture qui définit l'addition mais cette fois-ci en précisant les types des variables liées :

$$plus \quad \triangleq \quad (rec\ z) \twoheadrightarrow_{\Delta} \left(\begin{array}{l} (add\ 0\ y) \twoheadrightarrow_{\Delta} y \\ \wr(add\ (S\ x)\ y) \twoheadrightarrow_{\Delta} S\ \ (z\ (rec\ z)\ (add\ x\ y)) \end{array} \right)$$

avec $\Delta = (x{:}\iota, y{:}\iota, z{:}\kappa \to \iota \to \iota)$.

Si on considère les types suivants pour les constantes de ce terme

$$\Gamma = \{0{:}\iota, S{:}\iota \to \iota, add{:}\iota \to \iota \to \iota, rec{:}(\kappa \to \iota \to \iota) \to \kappa\}$$

alors on peut facilement vérifier par applications successives de la règle (App) que $\Gamma, \Delta \vdash add\ x\ y : \iota$, $\Gamma, \Delta \vdash rec\ z : \kappa$, $\Gamma, \Delta \vdash S\ (z\ (rec\ z)\ (add\ x\ y)) : \iota$ et

que $\Gamma, \Delta \vdash add\ 0\ y : \iota$, $\Gamma, \Delta \vdash add\ (S\ x)\ y : \iota$. En appliquant les règles (Abs) et (Struct) nous obtenons le type $\iota \to \iota$ pour la structure et finalement

$$\Gamma \vdash plus : \kappa \to \iota \to \iota$$

Pour le terme qui encode l'addition par rapport à deux entiers passés en paramètre

$$addition \triangleq n \twoheadrightarrow_{(n:\iota)} m \twoheadrightarrow_{(m:\iota)} \big(plus\ (rec\ plus)\ (add\ n\ m)\big)$$

on a

$$\Gamma \vdash addition : \iota \to \iota \to \iota$$

Cette approche peut être appliquée à la méthode générale d'encodage de TRS et on peut noter que l'encodage $(\!|\mathcal{R}|\!)$ introduit dans la définition 10 est typable si le TRS $\mathcal{R}$ est bien typé, autrement dit si les membres gauches et droits de toutes les règles de $\mathcal{R}$ sont typables et ont le même type.

Tout d'abord, on peut remarquer qu'au terme appelé first dans la section 2.5.2 correspond une famille de terme :

$$\begin{array}{l}\mathsf{first}(A_1, A_2, \ldots, A_n) \triangleq \\ \quad x \twoheadrightarrow_\Delta ((\mathsf{stk} \twoheadrightarrow A_n\ x \wr y \twoheadrightarrow_{\Delta'} y)\ (\ldots\ (\mathsf{stk} \twoheadrightarrow A_2\ x \wr y \twoheadrightarrow_{\Delta'} y)\ (A_1 x)))\end{array}$$

avec des annotations de type pour la variable liée x potentiellement différentes. Si on considère que les A_i ont tous le même type $\tau \to \tau$ (correspondant au type des règles du TRS encodé) dans un contexte Γ alors on doit prendre $\Delta = (x{:}\tau)$ et $\Delta' = (y{:}\tau)$ et dans ce cas on a $\Gamma \vdash \mathsf{first}(A_1, A_2, \ldots, A_n) : \tau \to \tau$.

Maintenant, si $\mathcal{R}$ manipule des données de type τ alors on considère des constantes rec et Rec de type $(\kappa \to \tau \to \tau) \to \kappa$ et on obtient :

$$\Gamma \vdash (\!|\mathcal{R}|\!) : \kappa \to \tau \to \tau$$

3.2 Types polymorphes

On peut étendre ce système avec des types polymorphes qui comportent des variables de type. Dans le calcul $\rho_\forall$ « *à la* Church » [LW05] les types des variables liées sont spécifiés dans le terme et dans ce cas les propriétés du calcul simplement typé restent valides.

Dans le contexte des langages de programmation fonctionnelle, le polymorphisme est souhaitable dans la mesure où il permet de définir des fonctions très générales. Cependant, le fait de devoir manipuler des types explicitement peut rendre un tel langage inutilisable en pratique. On peut donc considérer une version dans le style « *à la* Curry », où les types n'apparaissent pas du tout au niveau des termes [Mil78]. Si on considère le ρ-calcul comme un formalisme de base pour les langages à base de règles, cette approche correspond à une version des langages comme Elan, Maude, Obj* ou

$$
\begin{array}{rcll}
\mathcal{S} & ::= & * & \\
\mathcal{T}^{\forall} & ::= & \mathcal{X}^{\forall} \mid \mathcal{K}^{\forall}(\overline{\mathcal{T}^{\forall}}) \mid \mathcal{T}^{\forall} \to \mathcal{T}^{\forall} & \text{(Types)} \\
\mathfrak{S} & ::= & \mathcal{T}^{\forall} \mid \forall \mathcal{X}^{\forall}.\mathfrak{S} & \text{(Schémas)} \\
\Delta & ::= & \emptyset \mid \Delta, \mathcal{X}{:}\mathfrak{S} \mid \Delta, \mathcal{K}{:}\mathfrak{S} & \text{(Contextes)} \\
\mathcal{P} & ::= & \mathcal{X} \mid \mathcal{K} \mid \mathcal{K}\overline{\mathcal{P}} & \text{(Motifs)} \\
\mathcal{T} & ::= & \mathcal{X} \mid \mathcal{K} \mid \mathcal{P} \twoheadrightarrow \mathcal{T} \mid \mathcal{T}\ \mathcal{T} \mid \mathcal{T} \wr \mathcal{T} \mid \mathsf{let}\ \mathcal{P} \ll \mathcal{T}\ \mathsf{in}\ \mathcal{T} & \text{(Termes)}
\end{array}
$$

FIGURE 3.3 – *Syntaxe de* $\rho^{\forall}_{\ll}$

ASF+SDF où l'utilisateur peut écrire des programmes dans un langage non typé, et des types sont automatiquement inférés au moment de la compilation. Malheureusement, comme pour le λ-calcul [Wel99] la vérification et l'inférence de types dans le système polymorphe *à la* Curry sont des problèmes indécidables.

Nous présentons dans cette section une restriction du système général où les types polymorphes sont clairement séparés des schémas de type polymorphes.

3.2.1 Le système de types

La syntaxe du calcul ainsi obtenu qu'on appelle ici $\rho^{\forall}_{\ll}$ est donnée dans la figure 3.3. On retrouve les termes du calcul non-typé sans aucune annotation de type dans les abstractions avec une construction $\mathsf{let}\ P \ll B\ \mathsf{in}\ A$ similaire aux « contraintes de filtrage » présentes dans différentes versions explicites [CFK04, BBCK06] ou non [CKL01a] du ρ-calcul. Cette construction permet de lier des variables par un motif P tout en sachant déjà quel sera l'argument B à filtrer selon P et on verra que ceci autorise une forme restreinte de polymorphisme.

Nous disposons maintenant de types polymorphes qui comportent des variables de type potentiellement liées en utilisant le lieur « $\forall$ ». Si aucune discipline n'est imposée sur la formation des types, les propriétés usuelles des systèmes de types ne sont pas préservées.

Par exemple, si on considère un contexte $\Gamma = \{1{:}\iota, \mathsf{f}{:}\forall\alpha.(\alpha \to \iota)\}$, alors pour tout terme typable B, dans une version naïve *à la* Curry de $\rho_{\forall}$ on aurait une dérivation de type de la forme [Wac05] :

$$
\dfrac{\dfrac{\dfrac{}{\Gamma \vdash \mathsf{f} : \forall\alpha.(\alpha \to \iota)}}{\Gamma \vdash \mathsf{f} : \tau \to \iota} \qquad \dfrac{\vdots}{\Gamma \vdash B : \tau}}{\Gamma \vdash \mathsf{f}\ B : \iota}
$$

et par conséquent

$$\dfrac{\dfrac{\dfrac{\vdots}{x{:}\kappa \vdash \mathsf{f}\ x : \iota} \quad \dfrac{}{x{:}\kappa \vdash x : \kappa}}{\vdash \mathsf{f}\ x \twoheadrightarrow x : \iota \to \kappa} \quad \dfrac{\vdots}{\vdash \mathsf{f}\ 1 : \iota}}{\vdash (\mathsf{f}\ x \twoheadrightarrow x)\ (\mathsf{f}\ 1) : \kappa}$$

Ce terme a donc un type κ arbitraire et il se réduit à 1 qui a pour type ι.

Pour éviter ce problème, le type de chaque constante doit faire apparaître toutes les variables de type liées dans le type de retour. Avec cette contrainte, le type proposé pour f dans l'exemple précédent n'est pas admissible car α n'apparaît pas dans ι. Les constantes de type de $\rho^{\forall}_{\ll}$ sont ainsi « paramétrable » par un ensemble de variables et donc, le type de la constante f de l'exemple précédent pourrait être $\forall\alpha.(\alpha \to \iota(\alpha))$.

L'indécidabilité du typage dans le calcul polymorphe *à la* Curry appelé $\rho_{\forall C}$ dans [Wac05] est due au langage de types trop important. Dans ce calcul la liaison de variables peut apparaître n'importe où dans un type, ce qui empêche de contrôler l'utilisation des règles de typage concernant les types quantifiés. Nous avons adopté une approche inspirée de la solution utilisée en ML et on s'est restreint à des *schémas de type* avec tous les lieurs se trouvant en tête, *c.-à-d.* de la forme $\forall\overline{\alpha}.\tau$ avec $\overline{\alpha}$ un ensemble de variables et τ un terme monomorphe. Les contextes associent maintenant un (unique) schéma de type à chaque variable et constante mais, comme on verra par la suite, on ne typera les termes qu'avec des types simples.

On utilisera, comme avant, le symbole τ pour dénoter les types de l'ensemble $\mathcal{T}^{\forall}$ et le symbole ι pour les constantes de type de l'ensemble $\mathcal{K}^{\forall}$. Les symboles α, β seront utilisés pour désigner des variables de type de l'ensemble $\mathcal{X}^{\forall}$ et on utilisera la notation $\iota_{\overline{\alpha}}$ pour les constantes de type dépendant des variables $\overline{\alpha}$. Finalement, le symbole σ dénote des schémas de type.

On a maintenant une notion de variable *de type* libre ou liée (qu'on notera $\mathcal{FV}$ et $\mathcal{BV}$), qui s'étend aux contextes :

$$\begin{array}{rcl@{\qquad}rcl}
\mathcal{FV}(\iota) & \triangleq & \emptyset & \mathcal{BV}(\iota) & \triangleq & \emptyset \\
\mathcal{FV}(\alpha) & \triangleq & \{\alpha\} & \mathcal{BV}(\alpha) & \triangleq & \emptyset \\
\mathcal{FV}(\sigma \to \tau) & \triangleq & \mathcal{FV}(\sigma) \cup \mathcal{FV}(\tau) & \mathcal{BV}(\sigma \to \tau) & \triangleq & \mathcal{BV}(\sigma) \cup \mathcal{BV}(\tau) \\
\mathcal{FV}(\forall\alpha.\tau) & \triangleq & \mathcal{FV}(\tau) \setminus \{\alpha\} & \mathcal{BV}(\forall\alpha.\tau) & \triangleq & \mathcal{BV}(\tau) \cup \{\alpha\} \\
\mathcal{FV}(\Gamma, x{:}\tau) & \triangleq & \mathcal{FV}(\Gamma) \cup \mathcal{FV}(\tau) & \mathcal{BV}(\Gamma, x{:}\tau) & \triangleq & \mathcal{BV}(\Gamma) \cup \mathcal{BV}(\tau)
\end{array}$$

On a également $\mathcal{FV}(\mathsf{let}\ P \ll A\ \mathsf{in}\ B) \triangleq \mathcal{FV}((P \twoheadrightarrow B)\ A)$.

On associe également une règle de réduction très similaire à (ρ) pour la nouvelle construction $\mathsf{let}\ P \ll B\ \mathsf{in}\ A$

$$\mathsf{let}\ P \ll B\ \mathsf{in}\ A \to_{\sigma} A\sigma_1 \wr \ldots \wr A\sigma_n \qquad \textbf{avec}\ \{\sigma_1, \ldots, \sigma_n\} = \mathcal{S}ol(P \lessdot_{\mathbb{T}} B)$$

Les nouvelles règles de typage sont présentées dans la figure 3.4 ; elles permettent de prouver des jugements de la forme :

$$\Gamma \vdash_{\mathsf{U}} ok \quad \text{et} \quad \Gamma \vdash_{\mathsf{U}} \tau : * \quad \text{et} \quad \Gamma \vdash_{\mathsf{U}} U : \tau$$

correspondant à la vérification de la bonne formation de contextes, de types et de termes respectivement.

Comme mentionné précédemment, la formation des schémas de type doit respecter une certaine discipline : la condition utilisant l'opérateur Lab garantit que toutes les variables (de type) liées doivent apparaître dans le type de retour.

Pour passer d'un schéma à un type dans les règles pour les constantes et les variables ((Const) et (Var)), on définit une relation $\leq$ exprimant qu'un type τ est une instance d'un schéma de type σ.

$$\sigma \leq \tau \text{ ssi } \sigma \triangleq \forall\overline{\alpha_i}.\tau' \wedge \exists\overline{\tau_i},\ \tau \triangleq \tau'\{\overline{\tau_i/\alpha_i}\}.$$

Dans la règle (Abs) le contexte Δ doit être inféré mais il peut assigner seulement des types (et non des schémas de type) aux variables de P.

Les contraintes de filtrage de la forme $\mathsf{let}\ P \ll B\ \mathsf{in}\ A$ permettent de lier des variables par un motif P tout en sachant déjà quel sera l'argument B à filtrer selon P. Dans la règle (Match), cette connaissance supplémentaire autorise une forme restreinte de polymorphisme dans la mesure où on peut généraliser certaines variables de type apparaissant dans les types des variables libres de P. La fonction auxiliaire $\mathsf{Gen}(\Delta;\Gamma)$ transforme les types des variables de Δ en schémas de types, en évitant les variables libres dans Γ pour ne pas créer d'incohérences.

Exemple 9 (Polymorphisme [Wac05]) *Si on considère le contexte* $\Gamma = \{a{:}\iota, b{:}\kappa, f{:}\iota \to \kappa \to \iota\}$*, le terme* $\big(x \twoheadrightarrow f\ (x\ a)\ (x\ b)\big)(y \twoheadrightarrow y)$ *n'est pas typable car lors du typage de la première abstraction, il faut fixer un type pour* x*, qui ne peut être à la fois* $\iota \to \iota$ *et* $\kappa \to \kappa$*.*

Le terme $\mathsf{let}\ x \ll y \twoheadrightarrow y\ \mathsf{in}\ f\ (x\ a)\ (x\ b)$ *admet les mêmes réductions et il est typable :*

$$\frac{\vdash_{\mathsf{U}} y \twoheadrightarrow y{:}\alpha \to \alpha \quad x{:}\alpha \to \alpha \vdash_{\mathsf{U}} x{:}\alpha \to \alpha \qquad \dfrac{\vdots}{\mathsf{Gen}(x{:}\alpha \to \alpha;\emptyset) \vdash_{\mathsf{U}} f\ (x\ a)\ (x\ b){:}\iota}}{\Gamma \vdash_{\mathsf{U}} \mathsf{let}\ x \ll y \twoheadrightarrow y\ \mathsf{in}\ f\ (x\ a)\ (x\ b){:}\iota}$$

puisque $\mathsf{Gen}(x{:}\alpha \to \alpha;\emptyset)$ *affecte le schéma de type* $\forall\alpha.(\alpha \to \alpha)$ *à* x *et donc il peut être appliqué à la fois à* a *et à* b *dans* $f\ (x\ a)\ (x\ b)$*.*

On peut vérifier que la préservation du type est toujours vérifiée dans le cas polymorphe.

Contextes bien formés

$$\frac{}{\emptyset \vdash_{\mathsf{U}} ok}\,(\mathsf{Ctx{\cdot}Empty}) \qquad \frac{\Gamma \vdash_{\mathsf{U}} ok \quad \iota \notin \mathcal{D}om(\Gamma)}{\Gamma, \iota{:}* \vdash_{\mathsf{U}} ok}\,(\mathsf{TypeConst})$$

$$\frac{\Gamma \vdash_{\mathsf{U}} ok \quad x \notin \mathcal{D}om(\Gamma)}{\Gamma, x{:}\sigma \vdash_{\mathsf{U}} ok}\,(\mathsf{Var}) \qquad \frac{\Gamma \vdash_{\mathsf{U}} ok \quad \alpha \notin \mathcal{D}om(\Gamma)}{\Gamma, \alpha{:}* \vdash_{\mathsf{U}} ok}\,(\mathsf{Var}^{\forall})$$

$$\frac{\Gamma \vdash_{\mathsf{U}} ok \quad \Gamma \vdash_{\mathsf{U}} \sigma{:}* \quad f \notin \mathcal{D}om(\Gamma)}{\Gamma, f{:}\sigma \vdash_{\mathsf{U}} ok}\,(\mathsf{Const})$$

(Schémas de) types bien formés

$$\frac{\alpha \in \mathsf{Lab}(\sigma) \quad \Gamma, \alpha{:}* \vdash_{\mathsf{U}} \sigma : *}{\Gamma \vdash_{\mathsf{U}} \forall\alpha.\sigma : *}\,(\mathsf{TypeScheme}^{\forall}) \qquad \frac{\Gamma \vdash_{\mathsf{U}} ok}{\Gamma \vdash_{\mathsf{U}} \tau : *}\,(\mathsf{Type})$$

où

$$\mathsf{Lab}(\iota_{\overline{\alpha}}) = \overline{\alpha} \qquad \mathsf{Lab}(\tau_1 \to \tau_2) = \mathsf{Lab}(\tau_2)$$
$$\mathsf{Lab}(\alpha) = \{\alpha\} \quad \mathsf{Lab}(\forall\beta.\sigma) = \mathsf{Lab}(\sigma)$$

Termes bien formés

$$\frac{\Gamma, x{:}\sigma \vdash_{\mathsf{U}} ok \quad \sigma \leq \tau}{\Gamma, x{:}\sigma \vdash_{\mathsf{U}} x : \tau}\,(\mathsf{Var}) \qquad \frac{\Gamma, f{:}\sigma \vdash_{\mathsf{U}} ok \quad \sigma \leq \tau}{\Gamma, f{:}\sigma \vdash_{\mathsf{U}} f : \tau}\,(\mathsf{Const})$$

$$\frac{\Gamma \vdash_{\mathsf{U}} A : \tau \quad \Gamma \vdash_{\mathsf{U}} B : \tau}{\Gamma \vdash_{\mathsf{U}} A \wr B : \tau}\,(\mathsf{Struct}) \qquad \frac{\Gamma \vdash_{\mathsf{U}} A : \tau_1 \to \tau_2 \quad \Gamma \vdash_{\mathsf{U}} B : \tau_1}{\Gamma \vdash_{\mathsf{U}} A\ B : \tau_2}\,(\mathsf{Appl})$$

$$\frac{\Gamma, \Delta \vdash_{\mathsf{U}} P : \tau_1 \quad \Gamma, \Delta \vdash_{\mathsf{U}} A : \tau_2 \quad \mathcal{D}om(\Delta) = \mathcal{FV}(P) \quad \mathcal{BV}(\Delta) = \emptyset}{\Gamma \vdash_{\mathsf{U}} P \twoheadrightarrow A : \tau_1 \to \tau_2}\,(\mathsf{Abs})$$

$$\frac{\begin{array}{c}\Gamma, \Delta \vdash_{\mathsf{U}} P : \tau_1 \quad \Gamma, \mathsf{Gen}(\Delta; \Gamma) \vdash_{\mathsf{U}} A : \tau_2 \quad \Gamma \vdash_{\mathsf{U}} B : \tau_1 \\ \mathcal{D}om(\Delta) = \mathcal{FV}(P) \quad \mathcal{BV}(\Delta) = \emptyset\end{array}}{\Gamma \vdash_{\mathsf{U}} \mathsf{let}\ P \ll B\ \mathsf{in}\ A : \tau_2}\,(\mathsf{Match})$$

où

$$\mathsf{Gen}(\tau; \Gamma) \triangleq \forall\overline{\alpha}.\tau \quad \text{avec } \overline{\alpha} = \mathcal{FV}(\tau) \setminus \mathcal{FV}(\Gamma)$$
$$\mathsf{Gen}(x{:}\tau, \Delta; \Gamma) \triangleq x{:}\mathsf{Gen}(\tau; \Gamma), \mathsf{Gen}(\Delta; \Gamma)$$

FIGURE 3.4 – *Système de types de* $\rho^{\forall}_{\ll}$

3.2.2 Inférence de types

Nous allons présenter maintenant un raffinement de l'algorithme W de L. Damas et R. Milner [DM82] pour un système polymorphe à la Curry dans le λ-calcul qui infère un type principal étant donnés un terme A et un contexte Γ. Notre algorithme $\mathsf{W}^{\ll}$ donné dans la figure 3.5 suit la présentation utilisée par F. Pottier pour l'inférence de types dans Caml [Pot]. Il prend en argument le terme à typer A, l'environnement de typage Γ et un ensemble de variables de type fraîches $\mathcal{V}$ utilisées pour créer de nouveaux types et vérifie si A est typable, dans quel cas l'algorithme renvoie un type τ dont il suffit d'abstraire les variables libres pour obtenir un schéma de type principal, une valuation θ des variables de types libres dans Γ (qui servira notamment à instancier les variables prises dans $\mathcal{V}$) et le sous-ensemble $\mathcal{V}'$ des variables non encore utilisées dans $\mathcal{V}$.

L'algorithme utilise une fonction auxiliaire Inst qui transforme un schéma de type en un type dont on connaît les variables libres ainsi qu'une fonction auxiliaire mgu qui calcule un unificateur plus général [JK91]. Il faut noter que l'échec (false) doit être propagé strictement, et doit notamment être renvoyé quand la fonction mgu échoue.

Nous pouvons d'abord montrer par récurrence sur la structure des termes un lemme de substitution pour les variables de type.

Lemme 3 (Lemme de substitution pour les variables de type) *Pour tout contexte Γ, pour tout terme A et tout type τ, si $\Gamma \vdash_{\mathsf{U}} A : \tau$, alors pour toute substitution θ telle que $\mathcal{D}om(\theta) \subseteq \mathcal{X}^{\forall}$ on a $\Gamma\theta \vdash_{\mathsf{U}} A : \tau\theta$.*

Nous pouvons montrer la correction de l'algorithme par récurrence sur la structure des termes et en utilisant la définition de Inst et le lemme de substitution sur les variables de type.

Théorème 11 (Correction de $\mathsf{W}^{\ll}$ [CKLW06])
Si $\mathsf{W}^{\ll}(\Gamma; A; \mathcal{V}) = (\tau; \theta; \mathcal{V}')$, alors $\Gamma\theta \vdash_{\mathsf{U}} A : \tau$.

En ce qui concerne la complétude, il faut considérer une égalité particulière sur les substitutions permettant de vérifier leur comportement sur les nouvelles variables de type que l'on introduit. On définit donc une égalité par rapport à un ensemble de variables :

Définition 13 *Deux substitutions θ_1 et θ_2* coïncident hors de $\mathcal{V}$, *noté $\theta_1 \overset{\mathcal{V}}{=} \theta_2$, si $\alpha\theta_1 = \alpha\theta_2$ pour tout $\alpha \notin \mathcal{V}$.*

Nous pouvons maintenant montrer en utilisant une récurrence simultanée la complétude et la principalité de $\mathsf{W}^{\ll}$.

Théorème 12 (Complétude et principalité [CKLW06]) *Pour tous $\mathcal{V}$ et Γ tels que $\mathcal{V} \cap \mathcal{FV}(\Gamma) = \emptyset$, si $\Gamma\phi \vdash_{\mathsf{U}} A : \tau'$ pour une substitutions ϕ, alors :*

$$
\begin{array}{rcl}
\mathsf{W}^{\lll}(\Gamma; A; \mathcal{V}) & \triangleq & \\
A \equiv x & \Rightarrow & (\tau\,;\mathtt{id}\,;\mathcal{V}') \\
 & & \text{avec } (\tau\,;\mathcal{V}') = \mathsf{Inst}(\Gamma(x)\,;\mathcal{V}) \\
 & & \text{ssi } x \in \mathcal{D}om(\Gamma) \\
A \equiv f & \Rightarrow & (\tau\,;\mathtt{id}\,;\mathcal{V}') \\
 & & \text{avec } (\tau\,;\mathcal{V}') = \mathsf{Inst}(\Gamma(f)\,;\mathcal{V}) \\
 & & \text{ssi } f \in \mathcal{D}om(\Gamma) \\
A \equiv A_1 \wr A_2 & \Rightarrow & (\tau_2\mu\,;\mu\circ\theta_2\circ\theta_1\,;\mathcal{V}_2) \\
 & & \text{avec } (\tau_1;\theta_1;\mathcal{V}_1) = \mathsf{W}^{\lll}(\Gamma; A_1; \mathcal{V}) \\
 & & \qquad (\tau_2;\theta_2;\mathcal{V}_2) = \mathsf{W}^{\lll}(\Gamma\theta_1; A_2; \mathcal{V}_1) \\
 & & \qquad \mu = \mathsf{mgu}(\tau_1\theta_2 = \tau_2) \\
A \equiv P \twoheadrightarrow A_1 & \Rightarrow & (\tau_1\theta_2 \to \tau_2\,;\theta_2\circ\theta_1\,;\mathcal{V}_2) \\
 & & \text{avec } \overline{x} = \mathcal{FV}(P) \\
 & & \qquad \overline{\alpha_x} \in \mathcal{V}_1 \\
 & & \qquad (\tau_1;\theta_1;\mathcal{V}_1) = \mathsf{W}^{\lll}(\Gamma, \overline{x{:}\alpha_x}\,;P\,;\mathcal{V}\setminus\{\overline{\alpha_x}\}) \\
 & & \qquad (\tau_2;\theta_2;\mathcal{V}_2) = \mathsf{W}^{\lll}(\Gamma\theta_1, \overline{x{:}\alpha_x\theta_1}\,;A_1\,;\mathcal{V}_1) \\
A \equiv A_1\ A_2 & \Rightarrow & (\alpha\mu\,;\mu\circ\theta_2\circ\theta_1\,;\mathcal{V}_2\setminus\{\alpha\}) \\
 & & \text{avec } (\tau_1;\theta_1;\mathcal{V}_1) = \mathsf{W}^{\lll}(\Gamma; A_1\,;\mathcal{V}) \\
 & & \qquad (\tau_2;\theta_2;\mathcal{V}_2) = \mathsf{W}^{\lll}(\Gamma\theta_1\,;A_2\,;\mathcal{V}_1) \\
 & & \qquad \alpha \in \mathcal{V}_2 \\
 & & \qquad \mu = \mathsf{mgu}(\tau_1\theta_2 = \tau_2 \to \alpha) \\
A \equiv \mathsf{let}\ P \ll A_2\ \mathsf{in}\ A_1 & \Rightarrow & (\tau_3\,;\theta_3\circ\mu\circ\theta_2\circ\theta_1\,;\mathcal{V}_3) \\
 & & \text{avec } (\tau_1;\theta_1;\mathcal{V}_1) = \mathsf{W}^{\lll}(\Gamma; A_2\,;\mathcal{V}) \\
 & & \qquad \overline{x} = \mathcal{FV}(P) \\
 & & \qquad \overline{\alpha_x} \in \mathcal{V}_1 \\
 & & \qquad (\tau_2;\theta_2;\mathcal{V}_2) = \mathsf{W}^{\lll}(\Gamma\theta_1, \overline{x{:}\alpha_x}\,;P\,;\mathcal{V}_1\setminus\{\overline{\alpha_x}\}) \\
 & & \qquad \mu = \mathsf{mgu}(\tau_1\theta_2 = \tau_2) \\
 & & \qquad (\tau_3;\theta_3;\mathcal{V}_3) = \mathsf{W}^{\lll}(\Gamma\theta_1\theta_2\mu, \overline{x{:}\mathsf{Gen}(\alpha_x\theta_2\mu\,;\,\Gamma\theta_1\theta_2\mu)}\,;A_1\,;\mathcal{V}_2) \\
 & \text{sinon} & \Rightarrow\ \mathsf{false}
\end{array}
$$

$\mathsf{Inst}(\forall\overline{\alpha}.\tau\,;\,\mathcal{V}) \triangleq (\tau\{\overline{\beta/\alpha}\}\,;\,\mathcal{V}\setminus\{\overline{\beta}\})$ où les $\overline{\beta}$ sont des variables distinctes prises dans $\mathcal{V}$

FIGURE 3.5 – *L'algorithme* $\mathsf{W}^{\lll}$

1. $\mathsf{W}^{\lll}(\Gamma; A; \mathcal{V}) \neq \mathsf{false}$;
2. *il existe* τ, θ *et* $\mathcal{V}'$ *tels que* $\mathsf{W}^{\lll}(\Gamma; A; \mathcal{V}) = (\tau; \theta; \mathcal{V}')$;
3. $\tau' = \tau\psi$ *et* $\phi \overset{\mathcal{V}}{=} \psi \circ \theta$ *pour une certaine substitution* ψ.

L'algorithme $\mathsf{W}^{\lll}$ peut également être utiliser pour démontre la décidabilité du typage.

Théorème 13 (Décidabilité du typage [CKLW06]) *Les deux problèmes suivants sont décidables dans le système de types de* $\rho^{\forall}_{\ll}$:

1. *Étant donnés un contexte* Γ, *un type* τ *et un terme* A, *peut-on dériver* $\Gamma \vdash_{\mathsf{U}} A : \tau$?
2. *Étant donnés un contexte* Γ *et un terme* A, *existe-t-il un type* τ *tel que* $\Gamma \vdash_{\mathsf{U}} A : \tau$?

Puisque l'algorithme $\mathsf{W}^{\lll}$ est correct et complet, il permet de répondre directement au second problème. Pour le premier, par principalité, on peut se ramener au problème

$$\mathsf{W}^{\lll}(\Gamma; U; \mathcal{V}) = (\tau'; \theta; \mathcal{V}') \ \wedge \ \tau = \tau'\psi \text{ avec } \mathcal{D}om(\psi) \subseteq \mathcal{V}$$

et donc à un problème de filtrage (modulo le renommage des variables de type liées) décidable.

3.3 Conclusions

Nous avons présenté un système de types de premier ordre pour le ρ_{stk}-calcul et nous avons montré que les encodages présentés dans la section 2.5 sont bien typés. On doit noter que si la discipline de types est renforcée avec des types dépendants [CKL01b, BCKL03] alors tous les programmes présentés ici sont statiquement rejetés, *c.-à-d.* bloqués par le système de types. Les systèmes de types présentés ici ne s'inscrivent pas dans la philosophie de [BCKL03] où les systèmes de types (dépendants) ont été étudiés dans une perspective logique où la normalisation forte des termes typables est essentielle, mais dans l'optique d'un formalisme théorique pour le typage dans des langages de programmation disposant des mécanismes de filtrages expressifs et puissants. Par conséquent, le système de types de $\rho^{\rightarrow}_{\mathsf{stk}}$ est bien adapté pour l'analyse statique des programmes, *c.-à-d.* pour garantir que les arguments des fonctions ont le type attendu, mais ne garantit pas la terminaison des termes bien typés qui correspondent à des programmes potentiellement cycliques.

Des réductions non terminantes similaires peuvent apparaître dans d'autres formalismes et, en particulier, N.P. Mendler [Men87] a montré que l'utilisation de définitions récursives dans le λ-calcul typé peut mener à des

réduction non terminantes si une condition de « positivité » n'est pas imposée. Ce type de condition est utilisé également dans le calcul des constructions inductives [Wer94] qui est à la base de l'assistant à la preuve Coq. Le même problème peut apparaître dans des langages de programmation également : par exemple, dans ML on peut définir des fonctions récursives sans utiliser letrec.

Nous avons également présenté $\rho^{\forall}_{\ll}$, un calcul de réécriture typé qui dispose d'une forme restreinte de polymorphisme avec des restrictions très proches de celles adoptées dans ML. Nous avons montré que les versions avec types simples et polymorphisme vérifient la plupart des propriétés usuelles : préservation du type, unicité et décidabilité du typage.

Nous avons adapté l'algorithme W de Damas-Milner pour le calcul présenté et nous avons prouvé que les propriétés classiques sont satisfaites : la correction et la complétude de l'inférence de types ainsi que la décidabilité du typage et l'existence d'un type principal pour tout terme.

L'algorithme d'inférence de types que nous avons présenté fournit un critère de correction pour les ρ-termes de la section 2.5 et par conséquent pour les systèmes de réécriture encodés. Ce système de types peut être vu également comme un point de départ pour l'intégration du polymorphisme et donc pour l'amélioration de l'expressivité des systèmes de types habituellement utilisés dans des langages à base de règles.

Contexte :

Le filtrage ainsi que l'application de substitutions sont des opérations fondamentales des calculs présentés dans les chapitres précédents mais elle sont réalisées au méta-niveau de ces calculs. Dans des implantations concrètes ces deux opérations doivent être séparées et doivent interagir entre elles et avec les mécanismes d'évaluation du calcul concerné.

Dans le contexte du λ-calcul, on utilise un formalisme intermédiaire qui réalise la décomposition de l'application de substitutions d'ordre supérieur en des étapes plus atomiques et permet de capturer le comportement calculatoire du système original. Les systèmes avec substitutions explicites ont été largement étudiés. Ils jouent évidement un rôle central dans certaines implantations de langages fonctionnels et logiques mais fournissent également des outils intéressants pour l'unification d'ordre supérieur ou pour représenter les preuves incomplètes dans la théorie des types.

Les calculs permettant de trouver la solution d'un problème de filtrage sont importants d'un point de vue conceptuel et calculatoire pour toutes les théories de filtrage, que ce soit de simples théories syntaxiques ou des théories équationnelles ou d'ordre supérieur.

Je me suis donc intéressé à des approches permettant un traitement au niveau objet du calcul de réécriture pour les opérations de filtrage et d'application de substitutions qui en résultent.

Questions :

- Peut-on identifier les formalismes permettant une définition explicite du filtrage et de l'application de substitutions ?
- Quelles sont les propriétés du calcul explicite et de ses sous-calculs ?

Contributions :

Les mécanismes d'application de substitutions du calcul de réécriture sont hérités du λ-calcul et nous avons déjà proposé, dans la première version du calcul, une version avec substitutions explicites basée sur une approche avec des indices de de Bruijn similaire à celle utilisée pour le $\lambda\sigma_{\Uparrow}$-calcul. Nous présentons dans ce document une approche plus modulaire permettant d'intégrer d'une manière indépendante des (sous-)calculs explicites pour calculer et appliquer les solutions d'un problème de filtrage.

Nous présentons le calcul avec filtrage explicite ainsi que le calcul explicite mais nous nous focalisons sur une version qui combine les parties explicites de ces deux calculs et qui permet la composition de substitutions. La preuve de confluence n'est pas triviale mais elle est particulièrement intéressante car elle permet d'identifier les points clés pour obtenir la confluence si un des sous-calculs est remplacé par un calcul alternatif.

4

Substitution et filtrage explicites

La possibilité de paramétrer le ρ-calcul général par une théorie de filtrage ouvre de nombreuses perspectives et mène à un calcul très expressif. Néanmoins, tous les calculs relatifs à la théorie de filtrage considérée sont toujours effectués au méta-niveau. La même situation apparaît dans Rogue [SDK$^+$03], un langage de programmation basé sur une version non typé du ρ-calcul et destiné principalement à l'implantation de procédures de décision. La sémantique opérationnelle de Rogue comme les règles du ρ-calcul utilisent des fonctions auxiliaires qui sont en effet des calculs implicites. Ces calculs sont importants d'un point de vue conceptuel et calculatoire pour toutes les théories de filtrage, que ce soit de simples théories syntaxiques ou des théories telles que l'associativité et la commutativité [Eke95]. Il est donc naturel d'envisager l'intégration explicite de ces calculs dans le formalisme de base.

Un premier pas vers une manipulation explicite des calculs liés au filtrage a été l'introduction des problèmes de filtrage explicitement dans la syntaxe du ρ-calcul [CKL02]. Les contraintes de filtrage sont résolues et les substitutions qui en résultent sont appliquées en une étape. Dans des implantations concrètes ces opérations doivent être séparées et doivent interagir avec d'autres calculs et, en particulier, on veut que les calculs sur des contraintes ainsi que l'application de contraintes résolues (*c.-à-d.* les substitutions) soient explicites.

Les λ-calculs avec substitutions explicites [ACCL91a, Les94, Mel95, Mel96, Ros96, DG01, Kes07] ont été largement étudiés. Ils jouent un rôle central dans certaines implantations de ML [LN04] et fournissent des outils intéressants pour l'unification d'ordre supérieur [DHK00] ou pour représenter les preuves incomplètes dans la théorie des types [Muñ97].

Dans tous les calculs avec substitutions explicites, les substitutions peuvent être retardées grâce à la règle β qui transforme un β-radical $(\lambda x.A)\,B$ en l'application *explicite* sur A de la substitution qui remplace x par B. Dans le ρ-calcul explicite, l'application des substitutions est retardée au moment où la contrainte de filtrage originale est résolue. Ainsi, le rôle de la règle β

est joué par la règle ρ qui transforme l'application d'une règle de réécriture en l'application d'une contrainte de filtrage et par les règles d'évaluation utilisées pour résoudre le problème de filtrage obtenu. Ceci mène à un calcul comparable à λ_x [Ros96] simple et sans composition de substitutions.

Ce calcul [CFK04] manipule au niveau objet non seulement les problèmes de filtrage mais également l'application des substitutions obtenues. Néanmoins, ces deux sous-calculs sont considérés séparément et ils sont gérés différemment.

Nous présentons deux versions du ρ-calcul, l'une avec substitutions explicites et l'autre avec filtrage explicite, ainsi qu'une version qui combine les deux et considère les problèmes d'efficacité et plus précisément la composition des substitutions. Ceci nous permet d'isoler les caractéristiques absolument nécessaires dans les deux cas et d'analyser les problèmes liés aux deux approches. Le résultat est un calcul qui a les bonnes propriétés habituelles des substitutions explicites (conservativité, terminaison) et qui est confluent. Le ρ-calcul et, en particulier, les ρ-calculs avec substitutions explicites sont donc particulièrement appropriés comme support théorique pour les implantations des langages basés sur la réécriture.

Nous nous focalisons ici sur les aspects opérationnels de ces calculs à filtrage et substitutions explicites et non sur les aspects logiques liés à ce type de formalisme. Le lien entre les règles de réduction d'un calcul avec motifs et substitutions explicites et les pas de normalisation des preuves en logique a été étudié dans [CK99a]. Plus précisément, Serenella Ceritto et Delia Kesner ont proposé le « Typed Pattern Calculus with Explicit Substitutions » [CK99a, CK04], un calcul typé avec motifs et substitutions explicites inspiré par l'isomorphisme dit de Curry-Howard entre les preuves dans la déduction naturelle et les termes du lambda-calcul simplement typé. Les règles de typage et les règles de réduction de ce calcul sont fondées sur le calcul des séquents ; les termes correspondent à des preuves du calcul des séquents et les règles de réduction à l'élimination des coupures. Toujours dans un contexte logique, les substitutions explicites ont été utilisées dans les encodages de systèmes d'ordre supérieur en déduction modulo [DHK02] et plus récemment par Guillame Burel pour l'encodage des systèmes de type purs fonctionnels en déduction surnaturelle [Bur08].

Les travaux décrits dans ce chapitre ont été réalisés en collaboration avec Germain Faure et Claude Kirchner.

4.1 Substitutions explicites : le $\rho_{\mathbf{s}}$-calcul

Nous présentons dans cette section une généralisation de λ_x [Ros96], appelé ρ-calcul avec substitutions explicites, qui utilise des abstractions non seulement sur des simples variables mais également sur des motifs contenant éventuellement plusieurs variables. D'autre part, le ρ-calcul avec sub-

stitutions explicites, appelé également $\rho_{\mathtt{s}}$-calcul, peut être vu comme une extension du ρ-calcul de base avec des substitutions explicites.

Dans le ρ-calcul de base, pour évaluer l'application d'une règle à un terme, le filtrage entre le membre gauche de la règle et le terme est résolu et la substitution obtenue est appliquée au membre droit de la règle au méta-niveau du calcul. Ceci signifie que, en une étape d'évaluation, on calcule la substitution solution du problème de filtrage et on l'applique au terme correspondant.

Cette réduction peut évidemment être décomposée en deux étapes, une calculant la substitution et l'autre décrivant l'application de cette substitution. Cette décomposition ne signifie pas que les calculs liés au filtrage et à l'application des substitutions sont explicites mais juste qu'ils sont clairement séparés. Dans cette section nous faisons un premier pas vers une version explicite du ρ-calcul en proposant une version du calcul où l'application des substitutions est exécutée explicitement tandis que les problèmes de filtrage sont encore résolus au méta-niveau.

4.1.1 La syntaxe des substitutions explicites

La syntaxe du ρ-calcul de base donné dans le chapitre 2 est étendue avec un opérateur explicite de substitution et elle est donnée dans la figure 4.1. Par la suite, nous suivons les conventions de notation et de nommage mentionnées dans les chapitres précédents.

L'opérateur d'*application de substitutions* est une généralisation du même opérateur dans λ_x. Dans le λ-calcul, une λ-abstraction lie seulement une variable et une substitution explicite considère ainsi une unique variable. Dans le ρ-calcul, l'abstraction est réalisée sur un motif et puisqu'elle peut lier un nombre arbitraire de variables, la définition d'une substitution est étendue en conséquence. Le symbole `id` représente la substitution identité. Il faut préciser que dans ce contexte le symbole « = » n'est pas symétrique.

On utilisera les symboles $\phi, \psi, \ldots$ pour dénoter les substitutions de l'ensemble Φ. Un terme est appelé *pur* s'il ne contient aucune application de substitution explicite.

Pour simplifier la lecture, nous adoptons la notation suivante pour l'application explicite de substitutions

$$B\left[x_i = A_i\right]_{i=1}^{n} \quad \triangleq \quad \begin{cases} B\left[\bigwedge_{i=1\ldots n} x_i = A_i\right] & \text{si } n > 0 \\ B\left[\mathtt{id}\right] & \text{sinon} \end{cases}$$

De la même façon, nous adoptons la notation suivante pour l'application de

Termes

$\mathcal{T}, \mathcal{P}$	::=	$\mathcal{X}$	(Variables)
	|	$\mathcal{K}$	(Constantes)
	|	$\mathcal{P} \twoheadrightarrow \mathcal{T}$	(Abstractions)
	|	$\mathcal{T}\ \mathcal{T}$	(Applications fonctionnelle)
	|	$\mathcal{T} \wr \mathcal{T}$	(Structures)
	|	$\mathcal{T}\,[\phi]$	(Applications de substitution)

Substitutions

Φ	::=	`id`	(Identité)
	|	$\mathcal{X} = \mathcal{T}$	(Equations)
	|	$\Phi \wedge \Phi$	(Conjonctions)

avec $\wedge$ associatif et `id` son élément neutre

FIGURE 4.1 – *Syntaxe du ρ_s-calcul*

substitutions au méta-niveau

$$B\{C_i/x_i\}_{i=1}^{n} \quad \triangleq \quad \begin{cases} B\{C_1/x_1, \ldots, C_n/x_n\} & \text{si } n > 0 \\ B & \text{sinon} \end{cases}$$

Le *domaine* d'une substitution $\phi = \bigwedge_{i=1\ldots n} x_i = A_i$, noté $\mathcal{D}om(\phi)$, est l'ensemble de variables $\{x_i\}_{i=1}^{n}$.

Dans ce chapitre, nous nous limitons à des motifs qui sont des structures algébriques définis dans la section 2.2.1. La définition formelle de l'ensemble de variables libres est obtenue en restreignant la définition donnée dans la section 4.3 pour le calcul avec filtrage et substitutions explicites.

4.1.2 Application des substitutions explicites

La sémantique opérationnelle du ρ-calcul avec substitutions explicites est donnée dans la figure 4.2. Nous suivons les conventions utilisées dans les chapitres précédents et, en particulier, nous considérons les termes modulo l'α-conversion. Les règles de réduction données dans la figure 4.2 sont séparées en deux catégories : les règles décrivant l'application des structures et des abstractions sur les ρ-termes et les règles définissant l'application des substitutions.

La règle (ρ) est utilisée pour réduire l'application d'une abstraction à un terme en filtrant le membre gauche de l'abstraction contre l'argument de la règle (en utilisant, par exemple, l'algorithme de filtrage donné dans la section 2.2.1) et en déclenchant l'application de la substitution obtenue au membre droit de l'abstraction. Dans le cas particulier où la fonction $\mathcal{S}ol$

$(P \twoheadrightarrow A)\ B$	$\rightarrow_{\rho}$	$A\,[x_i = C_i]_{i=1}^{n}$ **avec** $\mathcal{S}ol(P \lessdot B) = \{C_i/x_i\}_{i=1}^{n}$
$(A_1 \wr A_2)\ B$	$\rightarrow_{\delta}$	$A_1\ B \wr A_2\ B$
$A\,[\mathtt{id}]$	$\rightarrow_{Identity}$	A
$x\,[\phi \wedge (x = A) \wedge \psi]$	$\rightarrow_{Replace}$	A
$y\,[\phi]$	$\rightarrow_{Var}$	y **si** $y \notin \mathcal{D}om(\phi)$
$c\,[\phi]$	$\rightarrow_{Const}$	c
$(P \twoheadrightarrow A)\,[\phi]$	$\rightarrow_{Abs}$	$P\,[\phi] \twoheadrightarrow A\,[\phi]$
$(A\ B)\,[\phi]$	$\rightarrow_{App}$	$A\,[\phi]\ B\,[\phi]$
$(A \wr B)\,[\phi]$	$\rightarrow_{Struct}$	$A\,[\phi] \wr B\,[\phi]$

FIGURE 4.2 – *Sémantique opérationnelle du ρ_s-calcul*

renvoie la substitution identité (représentée par $\{Q_i/x_i\}_{i=1}^{0}$) le résultat est l'application explicite de `id` (représenté par $A[x_i = Q_i]_{i=1}^{0}$). S'il n'existe pas de solution pour le problème de filtrage correspondant, alors la règle n'est pas appliquée. La règle (δ) est héritée du ρ-calcul de base.

Les règles concernant l'application de substitutions distribue l'application par rapport aux différents opérateurs jusqu'à atteindre une variable ou une constante. Si la variable est dans le domaine de la substitution alors le terme correspondant la remplace ; une substitution appliquée à une variable qui n'est pas dans son domaine ou à une constante est ignorée. Puisque on considère des classes de termes modulo l'α-conversion, les représentants appropriés sont toujours choisis afin d'éviter les éventuelles captures de variables (potentiellement introduites par la règle (Abs)).

Nous définissons les relations σ et ρ_s induites par les règles manipulant les substitutions (*c.-à-d.* (Identity), (Replace), (Var), (Const), (Abs), (App), (Struct)) et par l'ensemble de toutes les règles dans la figure 4.2, respectivement. Il faut préciser que dans le ρ_s-calcul et dans les autres calculs introduits dans les sections suivantes toutes les étapes d'évaluation sont effectuées modulo la théorie sous-jacente (associativité ou associativité avec élément neutre) pour la conjonction (voir le Définition 15 pour une définition précise de la réécriture modulo).

On peut noter qu'en remplaçant le motif P par une variable dans la règle (ρ) nous retrouvons la règle (β) des λ-calculs avec substitutions explicites.

Exemple 10 (Application d'une règle de réécriture) *Pour calculer la forme normale disjonctive d'une formule propositionnelle, nous utilisons la*

règle de réécriture

$$\mathsf{not}(\mathsf{and}(x, y)) \twoheadrightarrow \mathsf{or}(\mathsf{not}(x), \mathsf{not}(y)).$$

L'application de cette règle au terme $\mathsf{not}(\mathsf{and}(\mathsf{true}, \mathsf{false}))$ *est définie en* ρ_s *par le terme* $\big(\mathsf{not}(\mathsf{and}(x, y)) \twoheadrightarrow \mathsf{or}(\mathsf{not}(x), \mathsf{not}(y))\big)\ \mathsf{not}(\mathsf{and}(\mathsf{true}, \mathsf{false}))$ *avec la réduction suivante*

$$\begin{array}{ll} & \big(\mathsf{not}(\mathsf{and}(x, y)) \twoheadrightarrow \mathsf{or}(\mathsf{not}(x), \mathsf{not}(y))\big)\ \mathsf{not}(\mathsf{and}(\mathsf{true}, \mathsf{false})) \\ \mapsto_{\rho} & \mathsf{or}(\mathsf{not}(x), \mathsf{not}(y))\,[x = \mathsf{true} \wedge y = \mathsf{false}] \\ \mapsto\!\!\!\!\rightarrow_{App} & (\mathsf{or}\,[x = \mathsf{true} \wedge y = \mathsf{false}])(\mathsf{not}(x\,[x = \mathsf{true} \wedge y = \mathsf{false}]), \\ & \qquad\qquad \mathsf{not}(y\,[x = \mathsf{true} \wedge y = \mathsf{false}])) \\ \mapsto_{Const} & \mathsf{or}(\mathsf{not}(x\,[x = \mathsf{true} \wedge y = \mathsf{false}]), \mathsf{not}(y\,[x = \mathsf{true} \wedge y = \mathsf{false}])) \\ \mapsto\!\!\!\!\rightarrow_{Replace} & \mathsf{or}(\mathsf{not}(\mathsf{true}), \mathsf{not}(\mathsf{false})) \end{array}$$

Exemple 11 (Applications multiples) *Puisque les substitutions ne peuvent pas être composées, l'application de chaque substitution est faite indépendamment et peut impliquer beaucoup de pas d'évaluation pour accéder aux feuilles du terme (vu comme un arbre) :*

$$\begin{array}{ll} & (g(x) \twoheadrightarrow (f(y) \twoheadrightarrow h(x, y))\ f(a))\ g(b) \\ \mapsto_{\rho} & ((f(y) \twoheadrightarrow h(x, y))\ f(a))\,[x = b] \\ \mapsto_{\rho} & h(x, y)\,[y = a]\,[x = b] \\ \mapsto\!\!\!\!\rightarrow_{\sigma} & h(x, y\,[y = a])\,[x = b] \\ \mapsto\!\!\!\!\rightarrow_{\sigma} & h(x, a)\,[x = b] \\ \mapsto\!\!\!\!\rightarrow_{\sigma} & h(x\,[x = b], a) \\ \mapsto\!\!\!\!\rightarrow_{\sigma} & h(b, a) \end{array}$$

Comme on verra plus tard, si les substitutions sont composées alors les pas relatifs au parcours du terme peuvent être factorisés (voir exemple 16).

4.2 Filtrage explicite : le $\rho_{\mathtt{m}}$-calcul

Dans cette section nous nous concentrons sur les problèmes de filtrage intrinsèques au ρ-calcul et, plus précisément, nous voulons rendre explicites les calculs liés au filtrage exécutés pendant la ρ-évaluation. Nous proposons ici le ρ-calcul avec filtrage explicite, appelé aussi $\rho_{\mathtt{m}}$-calcul, qui étend le ρ-calcul de base avec des règles d'évaluation traitant le filtrage (syntaxique). Dans ce calcul les problèmes de filtrage obtenus sont résolus explicitement tandis que l'application des substitutions est réalisée au méta-niveau.

4.2.1 Termes avec contraintes de filtrage

La syntaxe du ρ-calcul avec filtrage explicite est donnée dans la figure 4.3. Les substitutions explicites du $\rho_{\mathtt{s}}$-calcul qui peuvent être considérées comme

Termes

$\mathcal{T}, \mathcal{P}$	::=	$\mathcal{X}$	(Variables)
	|	$\mathcal{K}$	(Constantes)
	|	$\mathcal{P} \twoheadrightarrow A$	(Abstractions)
	|	$\mathcal{T}\ \mathcal{T}$	(Applications fonctionnelle)
	|	$\mathcal{T} \wr \mathcal{T}$	(Structures)
	|	$\mathcal{T}\,[\mathcal{C}]$	(Applications de contrainte)

Contraintes

$\mathcal{C}$	::=	`id`	(Identité)
	|	$\mathcal{P} \ll \mathcal{T}$	(Equations de filtrage)
	|	$\mathcal{C} \wedge \mathcal{C}$	(Conjonctions de contraintes)

avec $\wedge$ associatif et `id` son élément neutre

FIGURE 4.3 – *Syntaxe du ρ_m-calcul*

des équations de filtrage en forme résolue sont remplacées par les contraintes de filtrage qui seront résolues pendant le processus d'évaluation.

On utilisera les symboles $\mathfrak{C}, \mathfrak{D}, \mathfrak{E} \ldots$ pour dénoter des ensembles (potentiellement vides) de contraintes. Le symbole `id` représente ici la contrainte identité tandis que dans le ρ_s-calcul le même symbole était utilisé pour représenter la substitution vide. On considère qu'une conjonction vide et `id` représentent le même objet.

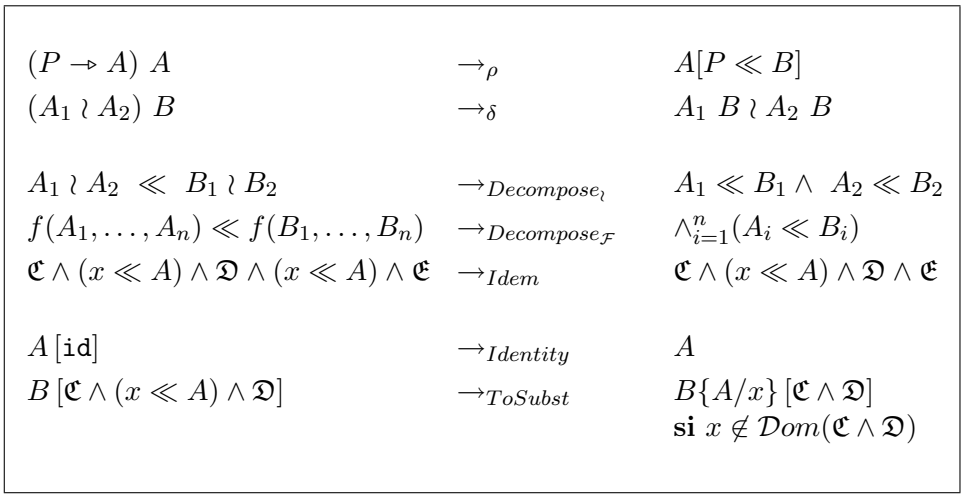
Le domaine d'une contrainte $\mathfrak{C}$, noté $\mathcal{D}om(\mathfrak{C})$, est intuitivement le même que le domaine de la substitution qui résout tous les problèmes de filtrage correspondants et est calculé en prenant l'union des ensembles de variable libres des motifs de toutes les équations de filtrage dans $\mathfrak{C}$. La définition formelle est donnée dans la section 4.3.1.

4.2.2 Résolution de contraintes de filtrage

La sémantique opérationnelle du ρ-calcul avec filtrage explicite consiste en deux partie présentées dans la figure 4.4.

Comme pour le ρ_s-calcul, la première partie décrit l'application des structures et des abstractions sur les ρ-termes. Cette fois, la règle (ρ) s'applique toujours et réduit l'application d'une abstraction à un terme contraint par le problème de filtrage correspondant.

Les problèmes de filtrage sont traités par la deuxième partie des règles d'évaluation ; ces règles sont clairement inspirées par celles présentées dans la section 2.2.1. Une contrainte de filtrage est simplifiée en utilisant les règles de décomposition qui sont fortement liées à la théorie de filtrage considérée. Comme nous l'avons déjà mentionné, nous considérons seulement des structures de termes algébriques comme motifs et nous nous limitons à une



$(P \rightarrowtriangle A)\ A$	$\rightarrow_{\rho}$	$A[P \ll B]$
$(A_1 \wr A_2)\ B$	$\rightarrow_{\delta}$	$A_1\ B \wr A_2\ B$
$A_1 \wr A_2 \ll B_1 \wr B_2$	$\rightarrow_{Decompose_{\wr}}$	$A_1 \ll B_1 \wedge A_2 \ll B_2$
$f(A_1, \ldots, A_n) \ll f(B_1, \ldots, B_n)$	$\rightarrow_{Decompose_{\mathcal{F}}}$	$\wedge_{i=1}^{n}(A_i \ll B_i)$
$\mathfrak{C} \wedge (x \ll A) \wedge \mathfrak{D} \wedge (x \ll A) \wedge \mathfrak{E}$	$\rightarrow_{Idem}$	$\mathfrak{C} \wedge (x \ll A) \wedge \mathfrak{D} \wedge \mathfrak{E}$
$A\,[\mathtt{id}]$	$\rightarrow_{Identity}$	A
$B\,[\mathfrak{C} \wedge (x \ll A) \wedge \mathfrak{D}]$	$\rightarrow_{ToSubst}$	$B\{A/x\}\,[\mathfrak{C} \wedge \mathfrak{D}]$ **si** $x \notin \mathcal{D}om(\mathfrak{C} \wedge \mathfrak{D})$

FIGURE 4.4 – *Sémantique opérationnelle de* $\rho_{\mathtt{m}}$

théorie de filtrage décidable et unitaire et, plus précisément, au filtrage syntaxique. Par conséquent, nous ne considérons pas les symboles d'ordre supérieur (comme, par exemple, « $\rightarrowtriangle$ » et « $\ll$ ») et nous décomposons seulement les motifs restreints. Les deux règles de décomposition données dans la figure 4.4 sont ainsi exécutés par rapport à une théorie de filtrage syntaxique pour l'opérateur de structure et pour les constantes. Puisqu'une conjonction vides et `id` représentent le même objet alors la règle ($Decompose_{\mathcal{F}}$) peut être utilisée pour des constantes (*c.-à-d.* $n = 0$) et dans ce cas le résultat est l'identité.

Quand une partie de la contrainte est résolue et indépendante du reste de la contrainte, la substitution correspondante peut être appliquée au méta-niveau. Afin d'éviter les éventuelles captures de variables, nous considérons que la méta-application de la substitution $\{A/x\}$ au terme B, notée $B\{A/x\}$ est d'ordre supérieur et donc exécutée modulo l'α-conversion. La condition dans la règle (ToSubst) garantit que le problème de filtrage est (partiellement) résolu et ainsi (une partie de) la substitution correspondante peut être appliquée seulement s'il y a qu'une assignation possible pour la variable « résolue » dans la contrainte respective. Les problèmes de filtrage non linéaires peuvent mener à des contraintes de filtrage qui associent différents termes à une même variable et qui représentent, intuitivement, un échec (voir l'exemple 14). Les doublons dans une contrainte de filtrage sont éliminés avec la règle (Idem).

Il faut noter que dans la règle (ToSubst), grâce à la convention d'hygiène, l'intersection entre l'ensemble de variables libres de A et $\mathcal{D}om(\mathfrak{C} \wedge \mathfrak{D})$ est vide et donc, les variables dans A ne peuvent pas être capturées dans le membre droit de la règle.

Le ρ-calcul (et en particulier le $\rho_{\mathtt{m}}$-calcul) est bien adapté pour traiter

les échecs (de filtrage), représentées par des contraintes sans solution, *c.-à-d.*, des contraintes qui ne représentent pas des substitutions. Selon l'utilisation prévue du calcul nous pouvons envisager ou non la propagation de tels (contraintes d') échecs. Si les échecs sont propagés, l'origine de l'erreur est perdue et le résultat final serait un terme avec des contraintes sans solution appliquées sur chaque feuille du terme (considéré comme un arbre). L'information contenue dans un tel terme semble inutile quand on veut analyser l'erreur et, pour ce type de raison, on veut pas perdre l'origine de l'erreur. C'est pourquoi les échecs ne sont pas propagés tels quels mais seulement la substitution correspondante (si elle existe) est propagée.

Exemple 12 (Application d'une règle de réécriture) *Le terme présenté dans l'exemple 10 est réduit en ρ-calcul avec filtrage explicite comme suit*

$$
\begin{array}{ll}
 & \big(\mathsf{not}(\mathsf{and}(x,y)) \multimap \mathsf{or}(\mathsf{not}(x),\mathsf{not}(y))\big)\ \mathsf{not}(\mathsf{and}(\mathsf{true},\mathsf{false})) \\
\mapsto_{\rho} & \mathsf{or}(\mathsf{not}(x),\mathsf{not}(y))\,[\mathsf{not}(\mathsf{and}(x,y)) \ll \mathsf{not}(\mathsf{and}(\mathsf{true},\mathsf{false}))] \\
\mapsto\!\!\!\!\rightarrow_{Decompose_{\mathcal{F}}} & \mathsf{or}(\mathsf{not}(x),\mathsf{not}(y))\,[x \ll \mathsf{true} \wedge y \ll \mathsf{false}] \\
\mapsto\!\!\!\!\rightarrow_{ToSubst} & \mathsf{or}(\mathsf{not}(\mathsf{true}),\mathsf{not}(\mathsf{false}))\,[\mathtt{id}] \\
\mapsto_{Identity} & \mathsf{or}(\mathsf{not}(\mathsf{true}),\mathsf{not}(\mathsf{false}))
\end{array}
$$

Exemple 13 (Application d'une règle de réécriture non-linéaire) *Quand des motifs non-linéaires sont utilisés, la règle (Idem) peut fusionner les problèmes de filtrage résolus identiques :*

$$
\begin{array}{ll}
 & (\mathsf{xor}(x,x) \multimap \mathsf{false})\ \mathsf{xor}(\mathsf{true},\mathsf{true}) \\
\mapsto_{\rho} & \mathsf{false}\,[\mathsf{xor}(x,x) \ll \mathsf{xor}(\mathsf{true},\mathsf{true})] \\
\mapsto_{Decompose_{\mathcal{F}}} & \mathsf{false}\,[x \ll \mathsf{true} \wedge x \ll \mathsf{true}] \\
\mapsto_{Idem} & \mathsf{false}\,[x \ll \mathsf{true}] \\
\mapsto\!\!\!\!\rightarrow_{ToSubst} & \mathsf{false}\,[\mathtt{id}] \\
\mapsto_{Identity} & \mathsf{false}
\end{array}
$$

Exemple 14 (Règle de réécriture non-linéaire avec échec) *Bien évidemment, l'application d'une règle de réécriture non-linéaire peut mener à des échecs dus à des conflits de fusion. Les conflits de fusion ne sont pas réduits mais gardés comme un échec de l'application de la contrainte.*

$$
\begin{array}{ll}
 & (\mathsf{xor}(x,x) \multimap \mathsf{false})\ \mathsf{xor}(\mathsf{true},\mathsf{false}) \\
\mapsto_{\rho} & \mathsf{false}\,[\mathsf{xor}(x,x) \ll \mathsf{xor}(\mathsf{true},\mathsf{false})] \\
\mapsto_{Decompose_{\mathcal{F}}} & \mathsf{false}\,[x \ll \mathsf{true} \wedge x \ll \mathsf{false}]
\end{array}
$$

L'exemple suivant illustre l'utilité du filtrage explicite quand nous voulons dépister la source (ou autrement dit, la cause) de l'échec.

Exemple 15 (Erreur au run-time : échec de filtrage) *Considérons la règle suivante qui vérifie si deux personnes sont de frères, c.-à-d. si elles ont le même père :*

$$\texttt{Brother}(\texttt{Person}(\texttt{Name}(x),\texttt{Father}(z)),\texttt{Person}(\texttt{Name}(y),\texttt{Father}(z))) \twoheadrightarrow \mathsf{true}$$

Quand on vérifie si deux personnes (Alice et Bob) sont des frères en appliquant cette règle au terme correspondant :

$$\texttt{Brother}(\texttt{Person}(\texttt{Name}(\mathit{Alice}),\texttt{Father}(\mathit{John})),\ \texttt{Person}(\texttt{Name}(\mathit{Bob}),\texttt{Father}(\mathit{Jim})))$$

on obtient comme résultat le terme

$$\mathsf{true}\,[z \ll \mathit{John} \wedge z \ll \mathit{Jim}]$$

indiquant que la variable z correspondant au père ne peut pas être correctement instanciée, c.-à-d., que le père des deux personnes n'est pas le même.

Ceci est en contraste avec le ρ-calcul avec substitutions explicites où le fait que l'application de la règle au terme est en forme normale indique que le filtrage n'a aucune solution mais ne donne aucune information sur la source de cet échec.

4.3 Le calcul avec substitution et filtrage explicites : le $\rho^{\circ}_{\mathtt{x}}$-calcul

La combinaison des deux calculs présentés précédemment, le $\rho_{\mathtt{s}}$-calcul et le $\rho_{\mathtt{m}}$-calcul, mène à une version du calcul qui manipule explicitement la résolution des contraintes de filtrage aussi bien que l'application des substitutions. Ce calcul, introduit dans [CFK04], ne considère pas la composition des substitutions, une question clé quand on veut obtenir des implantations efficaces.

Dans cette section nous montrons comment la composition peut être réalisée dans ce contexte et nous définissons le $\rho^{\circ}_{\mathtt{x}}$-calcul. Nous étudions ensuite les propriétés de ce calcul.

4.3.1 Syntaxe de $\rho^{\circ}_{\mathtt{x}}$-calcul

La syntaxe des $\rho^{\circ}_{\mathtt{x}}$-termes présentée dans la figure 4.5 est une fusion des syntaxes du $\rho_{\mathtt{s}}$-calcul et du $\rho_{\mathtt{m}}$-calcul. Par la suite, nous faisons référence aux trois catégories de $\rho^{\circ}_{\mathtt{x}}$-termes en les appelant simplement *termes*, *substitutions* et *contraintes* respectivement.

On peut noter que l'opérateur de conjonction $\wedge$ est surchargé et il est utilisé pour construire des substitutions aussi bien que des contraintes. Puisque les deux types de conjonctions sont disjoints, par la suite, on dénotera généralement les identités correspondantes id_s et id_m par le même symbole $\mathtt{id}$.

Termes

$\mathcal{T}, \mathcal{P}$	::=	$\mathcal{X}$	(Variables)
	|	$\mathcal{K}$	(Constantes)
	|	$\mathcal{P} \twoheadrightarrow \mathcal{T}$	(Abstractions)
	|	$\mathcal{T}\ \mathcal{T}$	(Applications fonctionnelle)
	|	$\mathcal{T} \wr \mathcal{T}$	(Structures)
	|	$\mathcal{T}\,[\phi]$	(Applications de substitution)
	|	$\mathcal{T}\,[\mathcal{C}]$	(Applications de contrainte)

Substitutions

Φ	::=	id_s	(Identité)
	|	$\mathcal{X} = \mathcal{T}$	(Equations)
	|	$\Phi \wedge \Phi$	(Conjonctions d'équations)

Contraintes

$\mathcal{C}$	::=	id_m	(Identité)
	|	$\mathcal{P} \ll \mathcal{T}$	(Equations de filtrage)
	|	$\mathcal{C} \wedge \mathcal{C}$	(Conjonctions de contraintes)

avec $\wedge$ associatif
et id_s et id_m ses éléments neutres

FIGURE 4.5 – *Syntaxe du $\rho^{\circ}_{\boldsymbol{x}}$-calcul*

Nous supposons que les opérateurs d'application fonctionnelle, de substitution et de contrainte associent à gauche, alors que les autres opérateurs associent à droite. L'application de substitution est plus prioritaire que l'application de contrainte qui est à son tour plus prioritaire que l'application fonctionnelle. L'application a une priorité plus élevée que « _ $\twoheadrightarrow$ _ » qui est d'une priorité plus élevée que « _ $\wr$ _ ». Les opérateurs d'équation sont d'une priorité plus élevée que les opérateurs de conjonction. Les opérateurs d'équation et de conjonction ont une priorité plus faible que les autres opérateurs.

Définition 14 (Variables libres et domaines des contraintes)
L'ensemble de variables libres et le domaine d'une contrainte (resp. substitution) sont définis par :

$$\begin{array}{lcllcl}
\mathcal{FV}(x) & = & \{x\} & & & \\
\mathcal{FV}(c) & = & \emptyset & \mathcal{FV}(P \twoheadrightarrow A) & = & \mathcal{FV}(A) \backslash \mathcal{FV}(P) \\
\mathcal{FV}(A\ B) & = & \mathcal{FV}(A) \cup \mathcal{FV}(B) & \mathcal{FV}(A \wr B) & = & \mathcal{FV}(A) \cup \mathcal{FV}(B) \\
 & & \mathcal{FV}(B\,[\mathfrak{C}]) = \mathcal{FV}(\mathfrak{C}) \cup \big(\mathcal{FV}(B) \backslash \mathcal{D}om(\mathfrak{C})\big) & & & \\
 & & \mathcal{FV}(B\,[\phi]) = \mathcal{FV}(\phi) \cup \big(\mathcal{FV}(B) \backslash \mathcal{D}om(\phi)\big) & & &
\end{array}$$

$$\begin{array}{lcllcl}
\mathcal{FV}(\mathfrak{C} \wedge \mathfrak{D}) & = & \mathcal{FV}(\mathfrak{C}) \cup \mathcal{FV}(\mathfrak{D}) & \mathcal{FV}(\phi \wedge \psi) & = & \mathcal{FV}(\phi) \cup \mathcal{FV}(\psi) \\
\mathcal{FV}(x = A) & = & FV(A) & \mathcal{FV}(\mathtt{id}) = \emptyset & & \\
\mathcal{FV}(P \ll A) & = & \mathcal{FV}(A) & & &
\end{array}$$

$$\begin{array}{lcllcl}
\mathcal{D}om(P \ll A) & = & \mathcal{FV}(P) & \mathcal{D}om(\mathfrak{C} \wedge \mathfrak{D}) & = & \mathcal{D}om(\mathfrak{C}) \cup \mathcal{D}om(\mathfrak{D}) \\
\mathcal{D}om(\mathtt{id}) & = & \emptyset & & & \\
\mathcal{D}om(x = A) & = & \{x\} & \mathcal{D}om(\phi \wedge \psi) & = & \mathcal{D}om(\phi) \wedge \mathcal{D}om(\psi)
\end{array}$$

4.3.2 Sémantique du $\rho^{\circ}_{\mathtt{x}}$-calcul

Les règles d'évaluation du $\rho^{\circ}_{\mathtt{x}}$-calcul sont présentées dans la figure 4.6 et consistent en celles utilisées pour le $\rho_{\mathtt{s}}$-calcul et pour le $\rho_{\mathtt{m}}$-calcul ainsi qu'une règle de composition.

L'application des abstractions et des structures aussi bien que la décomposition des contraintes de filtrage sont héritées du $\rho_{\mathtt{m}}$-calcul. Comme dans le $\rho_{\mathtt{m}}$-calcul, quand une partie de la contrainte est résolue et indépendante du reste de la contrainte, la substitution correspondante peut être appliquée. Dans le $\rho^{\circ}_{\mathtt{x}}$-calcul l'application de la substitution est juste déclenchée (comme dans le $\rho_{\mathtt{s}}$-calcul) dans la règle (ToSubst) et les règles héritées du $\rho_{\mathtt{s}}$-calcul réalisent son application. La règle (Identity) représente en fait deux règles, une pour la substitution identité et une seconde pour la contrainte identité. Quand la règle n'est pas clairement identifiable dans le contexte, la première est appelée ($Identity_s$) tandis que la dernière est appelée ($Identity_c$).

La nouvelle règle (Constraint) distribue les substitutions dans les contraintes. La règle (Compose) définit la composition des substitutions. La condition utilisée pour ces deux règles indique que la contrainte et respectivement la substitution ne peuvent pas être des identités. Pour la simplicité, nous avons utilisé un abus de notation dans la règle (Compose) où $B[\phi \wedge x_i = A_i\,[\phi]]_{i=1}^{n}$ dénote le terme $B\,[\phi \wedge x_1 = A_1\,[\phi] \wedge \ldots \wedge x_n = A_n\,[\phi]]$.

Exemple 16 (Applications multiples) *La composition des substitutions mène à des évaluations plus efficaces. L'évaluation suivante est obtenue pour le terme considéré dans l'exemple 11.*

$$\begin{array}{ll}
 & (g(x) \twoheadrightarrow (f(y) \twoheadrightarrow h(x,y))\ f(a))\ g(b) \\
\mapsto\!\!\!\!\twoheadrightarrow & ((f(y) \twoheadrightarrow h(x,y))\ f(a))\,[x = b] \\
\mapsto\!\!\!\!\twoheadrightarrow & h(x,y)\,[y = a]\,[x = b] \\
\mapsto_{Compose} & h(x,y)\,[x = b \wedge y = a\,[x = b]] \\
\mapsto\!\!\!\!\twoheadrightarrow_{App} & h(x\,[x = b \wedge y = a], y\,[x = b \wedge y = a]) \\
\mapsto\!\!\!\!\twoheadrightarrow_{Replace} & h(b,a)
\end{array}$$

Tous les pas d'évaluation concernant le parcours du terme $h(x,y)$ sont faits seulement une fois cette fois-ci puisque seulement une substitution doit être propagée.

$(P \twoheadrightarrow A)\ B$	$\rightarrow_{\rho}$	$A[P \ll B]$
$(A_1 \wr A_2)\ B$	$\rightarrow_{\delta}$	$A_1\ B \wr A_2\ B$
$A_1 \wr A_2 \ll B_1 \wr B_2$	$\rightarrow_{Decompose_{\wr}}$	$A_1 \ll B_1 \wedge A_2 \ll B_2$
$f(A_1, \ldots, A_n) \ll f(B_1, \ldots, B_n)$	$\rightarrow_{Decompose_{\mathcal{F}}}$	$\wedge_{i=1}^{n}(A_i \ll B_i)$
$\mathfrak{C} \wedge (x \ll A) \wedge \mathfrak{D} \wedge (x \ll A) \wedge \mathfrak{E}$	$\rightarrow_{Idem}$	$\mathfrak{C} \wedge (x \ll A) \wedge \mathfrak{D} \wedge \mathfrak{E}$
$B\,[\mathfrak{C} \wedge (x \ll A) \wedge \mathfrak{D}]$	$\rightarrow_{ToSubst}$	$B\,[x = A]\,[\mathfrak{C} \wedge \mathfrak{D}]$ **si** $x \notin \mathcal{D}om(\mathfrak{C} \wedge \mathfrak{D})$
$A\,[\mathtt{id}]$	$\rightarrow_{Identity}$	A
$x\,[\phi \wedge (x = A) \wedge \psi]$	$\rightarrow_{Replace}$	A
$y\,[\phi]$	$\rightarrow_{Var}$	y **si** $y \notin \mathcal{D}om(\phi)$
$c\,[\phi]$	$\rightarrow_{Const}$	c
$(P \twoheadrightarrow A)\,[\phi]$	$\rightarrow_{Abs}$	$P\,[\phi] \twoheadrightarrow A\,[\phi]$
$(A\ B)\,[\phi]$	$\rightarrow_{App}$	$A\,[\phi]\ B\,[\phi]$
$(A \wr B)\,[\phi]$	$\rightarrow_{Struct}$	$A\,[\phi] \wr B\,[\phi]$
$(A[P_i \ll B_i]_{i=1}^{n})\,[\phi]$	$\rightarrow_{Constraint}$	$(A\,[\phi])[P_i\,[\phi] \ll B_i\,[\phi]]_{i=1}^{n}$ **si** $n > 0$
$(B[x_i = A_i]_{i=1}^{n})\,[\phi]$	$\rightarrow_{Compose}$	$B[\phi \wedge x_i = A_i\,[\phi]]_{i=1}^{n}$ **si** $n > 0$

FIGURE 4.6 – *Sémantique opérationnelle du $\rho^{\circ}_{\mathtt{x}}$-calcul*

L'évaluation non efficace donnée dans l'exemple 11 est toujours possible mais tandis que c'était la seule alternative pour le $\rho_{\mathtt{s}}$-calcul, le problème disparaît pour le $\rho^{\circ}_{\mathtt{x}}$-calcul si on applique les règles d'évaluation avec une stratégie [BCDK09] qui donne la priorité la plus élevée à la règle de composition, une manière canonique de limiter les parcours de terme.

Comme mentionné au début de cette section, un ρ-calcul avec un traitement explicite de la résolution de contraintes et de l'application de substitutions a été présenté pour la première fois dans [CFK04]. Dans ce calcul, appelé $\rho_{\mathtt{x}}$-calcul, il n'y avait aucun mécanisme pour la composition des substitutions (*c.-à-d.* pas de règle (Compose)) et par conséquent, pas de conjonction de substitutions. Par conséquent, nous pouvons considérer le $\rho_{\mathtt{x}}$-calcul comme une restriction du $\rho^{\circ}_{\mathtt{x}}$-calcul où toutes les substitutions sont des équations simples et la règle (Compose) n'est pas disponible.

4.4 Sur la confluence du $\rho^{\circ}_{\mathtt{x}}$-calcul

Nous avons étudié les propriétés de ces nouveaux calculs et nous avons montré, en particulier, que le calcul explicite est confluent dans les mêmes conditions que le calcul général. Nous avons utilisé une technique de preuve basée sur le lemme de Yokouchi-Hikita [YH90] (que nous avons étendu pour le cas de la réécriture modulo) qui nous a permis non seulement de montrer la confluence du calcul mais également de mettre en évidence les propriétés des sous-calculs (de substitution et de filtrage) explicites. Nous présentons ici juste les grandes lignes de la preuve et les différents résultats intermédiaires ; une preuve plus détaillée est disponible dans [CFK07].

Comme mentionné précédemment, toutes les étapes d'évaluation sont exécutées modulo la théorie de la conjonction (dans le $\rho^{\circ}_{\mathtt{x}}$-calcul, modulo l'associativité et les éléments neutres) et ainsi, la réécriture est effectuée modulo un ensemble d'axiomes. Il faut donc tout d'abord préciser la relation de réécriture modulo correspondante et ensuite montrer la confluence modulo cette théorie.

Définition 15 (R/A) *Etant donné un système de réécriture R et un ensemble d'axiomes A, le terme t est (R/A)-réécrit en t', noté $t \to_{R/A} t'$ s'il existe une règle $l \to r \in R$, un terme u, une occurrence p dans u et une substitution σ tels que $t \overset{*}{\longleftrightarrow}_A u[p \hookleftarrow \sigma(l)]$ et $t' \overset{*}{\longleftrightarrow}_A u[p \hookleftarrow \sigma(r)]$ où $u[p \hookleftarrow v]$ dénote le terme u avec le sous-terme à la position p remplacé par v.*

Les propriétés de cette relation doivent être prouvées mais l'évaluation ci-dessous est plus opérationnelle et elle est habituellement utilisée dans les preuves et dans les implantations. La première définition est évidemment plus générale et dans certains cas les deux sont équivalentes ; pour une présentation détaillée de la réécriture modulo, nous nous référons à [Hue80, JK86, KK99, Ohl98].

Définition 16 (R, A) *Etant donné un système de réécriture R et un ensemble d'axiomes A, le terme t est (R, A)-réécrit en t', noté $t \to_{R,A} t'$ s'il existe une règle $l \to r \in R$, un terme u, une occurrence p dans u et une substitution σ tels que $t_{|p} \overset{*}{\longleftrightarrow}_A \sigma(l)$ (c.-à-d. le sous-terme de t à la position p est équivalent à $\sigma(l)$), et $t' = t[p \hookleftarrow \sigma(r)]$ (c.-à-d. t' est égal à t où le sous-terme à la position p est remplacé par $\sigma(r)$).*

Comme mentionné dans le chapitre 2, les motifs non linéaires peuvent mener à des réductions non-confluentes. Ceci justifie la définition suivante.

Définition 17 ($\rho^{\circ}_{\mathtt{x}}$-calcul linéaire) *Un motif est* linéaire *s'il ne contient pas deux occurrences de la même variable. Une substitution $(x_i = A_i)_{i=1}^n, n > 0$, est* linéaire *si toutes les variables x_i sont différentes. Une contrainte de*

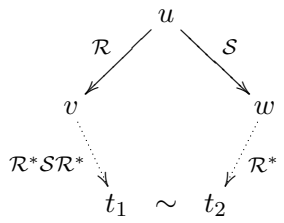
filtrage $(P_i \ll A_i)_{i=1}^n, n > 0$, *est* linéaire *si pour tous motifs* P_i, P_j *avec* $1 \leq i \neq j \leq n$ *on a* $\mathcal{FV}(P_i) \cap \mathcal{FV}(P_j) = \emptyset$. id_s *et* id_m *sont linéaires.*

Le $\rho^{\circ}_{\mathtt{x}}$-calcul linéaire *est le* $\rho^{\circ}_{\mathtt{x}}$-calcul *où tous les motifs, substitutions et contraintes de filtrage sont linéaires.*

Le schéma général de la preuve de confluence du $\rho^{\circ}_{\mathtt{x}}$-calcul linéaire suit la preuve de confluence utilisée pour le $\lambda\sigma_{\Uparrow}$-calcul et présentée dans [CHL96] et utilise une variante du lemme de Yokouchi-Hikita [YH90] (énoncé dans la section 6.2.1) :

Lemme 4 (Yokouchi-Hikita modulo [CFK07]) *Etant données deux relations* $\mathcal{R}$ *et* $\mathcal{S}$ *définies sur le même ensemble* $\mathcal{T}$ *et* $\sim$ *une relation d'équivalence telles que*

- $\mathcal{R}$ *est fortement normalisante,*
- $\mathcal{R}$ *est confluente modulo* $\sim$ *: pour tous* u, v, w *dans* $\mathcal{T}$ *tels que* $u\mathcal{R}^*v$ *et* $u\mathcal{R}^*w$ *il existe* t_1, t_2 *dans* $\mathcal{T}$ *tels que* $v\mathcal{R}^*t_1$, $w\mathcal{R}^*t_2$ *et* $t_1 \sim t_2$,
- $\mathcal{S}$ *est fortement confluente : pour tous* u, v, w *dans* $\mathcal{T}$ *tels que* $u\mathcal{S}v$ *et* $u\mathcal{S}w$ *il existe* t *dans* $\mathcal{T}$ *tel que* $v\mathcal{S}t$ *et* $w\mathcal{S}t$,
- $\mathcal{R}$ *et* $\mathcal{S}$ *sont cohérentes modulo* $\sim$ *: pour tous* u, v, w *tels que* $u \sim v$, $u\mathcal{R}w$ *(resp.* $u\mathcal{S}w$*) il existe un* t *tel que* $v\mathcal{R}t$ *(resp.* $v\mathcal{S}t$*) et* $w \sim t$,
- *le diagramme (de Yokouchi-Hikita modulo) suivant est satisfait :*



Alors la relation $\mathcal{R}^*\mathcal{S}\mathcal{R}^*$ *est confluente modulo* $\sim$.

On partage donc les règles d'évaluation du $\rho^{\circ}_{\mathtt{x}}$-calcul en deux relations correspondant aux relations $\mathcal{R}$ et $\mathcal{S}$ du lemme précédent. Un choix naturel pour la relation $\mathcal{R}$ est la relation de réécriture modulo induite par toutes les règles d'évaluation à l'exception de (ρ) et (δ) ; cette relation est notée $\mapsto_{\mathcal{C}}$ par la suite. En fait, les règles traitant les contraintes et les substitutions explicites devraient être fortement normalisantes et confluentes (dans des calculs de substitutions explicites on demande toujours ces propriétés pour la partie explicite). La relation $\mathcal{S}$ ne peut pas être la relation de réécriture induite par les règles (ρ) et (δ) puisque cette relation ne vérifie ni la confluence forte (la règle (δ) duplique des radicaux) ni le diagramme de Yokouchi-Hikita modulo (la relation $\mapsto_{\mathcal{C}}$ duplique également des radicaux). C'est pourquoi nous utilisons une parallélisation des règles (ρ) et (δ).

Nous montrons d'abord la terminaison et la confluence de $\mapsto_{\mathcal{C}}$ prenant en considération le fait que les règles peuvent être appliquées modulo l'associativité et les éléments neutres (id_s, id_m) pour les conjonctions. Nous définissons ensuite formellement la parallélisation de (ρ) et (δ) et nous montrons qu'elle est fortement confluente. Finalement, nous prouvons le diagramme de Yokouchi-Hikita modulo et on termine la preuve de confluence du $\rho^{\circ}_{\mathtt{x}}$-calcul linéaire en observant que la parallélisation et la relation originale ont la même fermeture transitive.

On dénote par la suite par $\mathtt{A1}$ l'ensemble d'axiomes définissant l'associativité et les éléments neutres id_m et id_s pour la conjonction et par $\sim_{\mathtt{A1}}$ la relation d'équivalence induite par ces axiomes. $\mapsto_{\mathcal{C}}$ dénote donc la relation $\mapsto_{k,\mathtt{A1}}$ où k est la relation de réécriture induite par toutes les règles d'évaluation à l'exception de (ρ) et (δ). On dénote par $\varrho^{\circ}_{\mathtt{x}}$ le système de réécriture donné dans la figure 4.6, et on obtient ainsi les relations $\mapsto_{\varrho^{\circ}_{\mathtt{x}},\mathtt{A1}}$ et $\mapsto_{\varrho^{\circ}_{\mathtt{x}}/\mathtt{A1}}$.

4.4.1 Terminaison et confluence des calculs explicites

Pour montrer que $\mapsto_{\mathcal{C}}$ normalise fortement nous avons utilisé le produit lexicographique de deux ordres. Le premier ordre est une mesure (une taille) sur les termes qui représente, pour tout terme, (une borne supérieure de) la taille du terme pur correspondant où toutes les substitutions en attente ont été appliquées. La taille d'un terme $B\,[x = A]$ est la taille de B plus la taille de A multipliée par le nombre d'occurrences de x dans B (appelé ici la multiplicité de x dans B), prenant de ce fait en considération les instanciations possibles de x par A dans B. En ce qui concerne les contraintes de filtrage, on doit prendre en considération le fait qu'elles peuvent (potentiellement) devenir des substitutions. Nous faisons ainsi une approximation et considérons que pour les termes de la forme $B\,[P \ll A]$, chaque variable de P est (potentiellement) instanciée par un sous-terme de A qui est approximé par A. Cette mesure est une sur-approximation qui est préservée par les règles duplicatives (par exemple, le terme $(x \wr x)\,[\phi]$ et son réduit $x\,[\phi] \wr x\,[\phi]$ par rapport à (Struct) ont la même taille indépendamment de ϕ) et qui décroît sur les autres règles. Le deuxième ordre est basé sur une interprétation polynomiale qui décroît sur toutes les règles dont la taille est préservée par le premier ordre.

Lemme 5 *La relation $\mapsto_{\mathcal{C}}$ est fortement normalisante.*

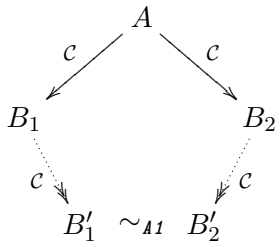
En ce qui concerne la confluence, nous prouvons d'abord que les substitutions explicites correspondent aux substitutions réalisées au niveau méta.

Lemme 6 (Correction des substitutions explicites) *Pour tout $n \geq 1$, pour tous termes A_i et B, il existe un terme B' tel que*

$$B\,[x_i = A_i]_{i=1}^{n} \quad \mapsto\!\!\!\!\!\rightarrow_{\mathcal{C}} \quad B'\{A_i/x_i\}_{i=1}^{n} \qquad \textit{et} \qquad B \mapsto\!\!\!\!\!\rightarrow_{\mathcal{C}} B'$$

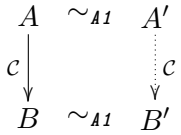
En utilisant ce résultat, on peut prouver par induction la confluence locale :

Lemme 7 (Confluence locale modulo `A1`) *Pour tous termes A, B_1, B_2 tels que $A \mapsto_{\mathcal{C}} B_1$ et $A \mapsto_{\mathcal{C}} B_2$, il existe deux termes B'_1 et B'_2 tels que $B_1 \mapsto\!\!\!\!\to_{\mathcal{C}} B'_1$ et $B_2 \mapsto\!\!\!\!\to_{\mathcal{C}} B'_2$ avec $B'_1 \sim_{\mathtt{A1}} B'_2$:*



Nous pouvons montrer par analyse de cas que deux termes équivalents modulo `A1` peuvent être réduits à deux termes équivalents :

Lemme 8 (Cohérence modulo `A1`) *Pour tous termes A, B, A' tels que $A \mapsto_{\mathcal{C}} B$ et $A \sim_{\mathtt{A1}} A'$, il existe B' tel que $B \sim_{\mathtt{A1}} B'$ et $A' \mapsto_{\mathcal{C}} B'$.*



En utilisant la normalisation forte, la confluence locale modulo `A1` et la cohérence modulo `A1`, nous obtenons, conformément à [Ohl98], que $\mapsto_{\mathcal{C}}$ est Church-Rosser modulo `A1`. Nous obtenons comme conséquence immédiate la confluence modulo `A1` de $\mapsto_{\mathcal{C}}$.

Lemme 9 *La relation $\mapsto_{\mathcal{C}}$ est confluente modulo `A1`.*

4.4.2 Théorème de confluence

La parallélisation des règles ρ et δ définie inductivement dans la figure 4.7 est une extension naturelle de la réduction parallèle de Tait et de Martin-Löf. Le nombre important de règles de congruence conduit, dans notre cas, à une définition un peu plus élaborée que celle utilisée pour la règle β du λ-calcul.

On note que la relation $\mapsto_{\rho\delta_{\|}}$ est la parallélisation d'un système orthogonal et nous obtenons ainsi sa confluence forte.

Lemme 10 (Confluence forte de $\mapsto_{\rho\delta_{\|}}$) *Pour tous termes A, B_1, B_2 tels que $A \mapsto_{\rho\delta_{\|}} B_1$ et $A \mapsto_{\rho\delta_{\|}} B_2$, il existe un terme A' tel que $B_1 \mapsto_{\rho\delta_{\|}} A'$ et $B_2 \mapsto_{\rho\delta_{\|}} A'$.*

Nous pouvons montrer par analyse de cas sur les réductions explicite que le diagramme de Yokouchi-Hikita modulo est satifait.

$$\frac{A \mapsto_{\rho\delta_{\|}} A' \quad B \mapsto_{\rho\delta_{\|}} B'}{(P \twoheadrightarrow A)\ B \mapsto_{\rho\delta_{\|}} A'[P \ll B']}$$

$$\frac{A'_1 \mapsto_{\rho\delta_{\|}} A'_1 \quad A_2 \mapsto_{\rho\delta_{\|}} A'_2 \quad B \mapsto_{\rho\delta_{\|}} B'}{(A_1 \wr A_2)\ B \mapsto_{\rho\delta_{\|}} A'_1\ B' \wr A'_2\ B'}$$

$$\frac{}{A \mapsto_{\rho\delta_{\|}} A} \qquad \frac{A \mapsto_{\rho\delta_{\|}} A'}{P \twoheadrightarrow A \mapsto_{\rho\delta_{\|}} P \twoheadrightarrow A'}$$

$$\frac{A \mapsto_{\rho\delta_{\|}} A' \quad B \mapsto_{\rho\delta_{\|}} B'}{A\ B \mapsto_{\rho\delta_{\|}} A'\ B'} \qquad \frac{A \mapsto_{\rho\delta_{\|}} A' \quad B \mapsto_{\rho\delta_{\|}} B'}{A \wr B \mapsto_{\rho\delta_{\|}} A' \wr B'}$$

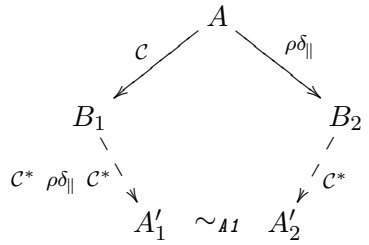
$$\frac{\mathfrak{C} \mapsto_{\rho\delta_{\|}} \mathfrak{C}' \quad A \mapsto_{\rho\delta_{\|}} A'}{A\,[\mathfrak{C}] \mapsto_{\rho\delta_{\|}} A'\,[\mathfrak{C}']} \qquad \frac{\phi \mapsto_{\rho\delta_{\|}} \phi' \quad A \mapsto_{\rho\delta_{\|}} A'}{A\,[\phi] \mapsto_{\rho\delta_{\|}} A'\,[\phi']}$$

$$\frac{B \mapsto_{\rho\delta_{\|}} B'}{(P \ll B) \mapsto_{\rho\delta_{\|}} (P \ll B')} \qquad \frac{B \mapsto_{\rho\delta_{\|}} B'}{(x = B) \mapsto_{\rho\delta_{\|}} (x = B')}$$

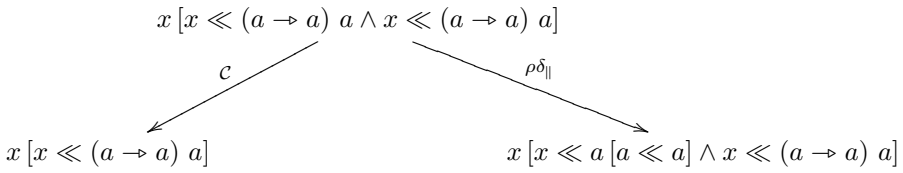
$$\frac{\mathfrak{C} \mapsto_{\rho\delta_{\|}} \mathfrak{C}' \quad \mathfrak{D} \mapsto_{\rho\delta_{\|}} \mathfrak{D}'}{\mathfrak{C} \wedge \mathfrak{D} \mapsto_{\rho\delta_{\|}} \mathfrak{C}' \wedge \mathfrak{D}'} \qquad \frac{\phi \mapsto_{\rho\delta_{\|}} \phi' \quad \psi \mapsto_{\rho\delta_{\|}} \psi'}{\phi \wedge \psi \mapsto_{\rho\delta_{\|}} \phi' \wedge \psi'}$$

FIGURE 4.7 – *Version parallèle de $\rho\delta$*

Lemme 11 (Diagramme Yokouchi-Hikita modulo) *Pour tous termes A, B_1 et B_2 dans le ρ°_{x}* linéaire*, il existe les termes A'_1, A'_2 tels que le diagramme suivant est satisfait :*



Il faut noter que la condition de linéarité est essentielle ici puisqu'elle assure que la règle (Idem) n'est jamais utilisée. Si cette condition n'est pas imposée et on permet des termes non-linéaires, alors le diagramme suivant ne peut pas être fermé et donne ainsi un contre-exemple au diagramme Yokouchi-Hikita modulo.



Toutes les hypothèses du lemme de Yokouchi-Hikita modulo (Lemme 4) sont prouvées par les lemmes précédents et donc, nous obtenons la confluence modulo $\mathtt{A1}$ de la relation $\twoheadrightarrow_{\mathcal{C}} \mapsto_{\rho\delta_{\|}} \twoheadrightarrow_{\mathcal{C}}$ et puisque $\mapsto_{\varrho^{\circ}_{\mathtt{x}},\mathtt{A1}} \subseteq \twoheadrightarrow_{\mathcal{C}} \mapsto_{\rho\delta_{\|}} \twoheadrightarrow_{\mathcal{C}} \subseteq \twoheadrightarrow_{\varrho^{\circ}_{\mathtt{x}},\mathtt{A1}}$ la confluence modulo $\mathtt{A1}$ de la relation initiale.

Lemme 12 *La relation $\mapsto_{\varrho^{\circ}_{\boldsymbol{x}},\boldsymbol{A1}}$ est confluente modulo $\boldsymbol{A1}$.*

La confluence de la relation $\mapsto_{\varrho^{\circ}_{\mathtt{x}}/\mathtt{A1}}$ et donc du calcul est obtenue comme une conséquence [Ohl98] des lemmes précédents et de la cohérence de $\mapsto_{\varrho^{\circ}_{\mathtt{x}},\mathtt{A1}}$ modulo $\sim_{\mathtt{A1}}$.

Théorème 14 *Le $\rho^{\circ}_{\boldsymbol{x}}$-calcul linéaire est confluent.*

On peut facilement remarquer qu'un terme pur dont tous les patterns sont linéaires ne peut pas être réduit en un terme non-linéaire au sens de la définition 17 et donc, on peut dire, comme pour le calcul non-explicite, que le $\rho^{\circ}_{\mathtt{x}}$-calcul est confluent pour tout terme pur avec tous les patterns linéaires.

4.5 Conclusions

Nous avons proposé un calcul de réécriture qui manipule explicitement les contraintes de filtrage (syntaxiques) et l'application des substitutions obtenues comme résultat. Ce calcul a de bonnes propriétés et, en particulier, nous avons montré qu'il est confluent et que le sous-calcul qui manipulent les différents types d'équations est confluent et terminant.

Ce travail s'inscrit dans la continuité des travaux sur les λ-calculs avec substitutions explicites en prenant en compte le filtrage, un ingrédient principal du calcul de réécriture et des langages à base de règles. La manipulation explicite du filtrage a été proposée pour la première fois dans le PSA-Calcul [BKK98] qui peut être vu comme l'ancêtre du ρ-calcul. Dans ce calcul l'application d'une règle de réécriture est explicite mais les stratégies ne peuvent pas être définies au niveau objet du calcul comme dans le ρ-calcul.

Nous avons également proposé un calcul de réécriture avec des substitutions explicites, appelé $\rho\sigma$-calcul [CK99b, Cir00], mais dans ce cas le traitement des problèmes de filtrage est effectué au niveau méta du calcul.

Par rapport à ces différents formalismes, le calcul que nous avons présenté ici est modulaire et peut être adapté à différentes théories de filtrage. La modularité se situe non seulement au niveau de la conception mais également au niveau de la technique de preuve qui isole clairement les parties concernées par le filtrage.

Nous avons considéré ici seulement des filtrages unitaires dans la théorie vide mais dans la pratique, il est intéressant d'avoir la possibilité de raisonner modulo des théories équationnelles. Dans ce cas, la règle de décomposition $Decompose_{\mathcal{F}}$ doit être adaptée selon la théorie que l'on veut traiter et, par exemple, si on veut manipuler des symboles commutatifs il suffit de construire dans le membre droit de cette règle une structure contenant les conjonctions correspondant aux différentes alternatives [CFK07]. Si on veut disposer du filtrage associatif ou associatif avec élément neutre (filtrage sur les listes) utilisé, par exemple, dans le langage Tom, alors il faut intégrer les règles que nous avons déjà formalisées dans [CKKM10]. Il est évident que si la théorie de l'opérateur de structure n'est pas vide alors la règle $Decompose_{\wr}$ doit être modifiée également.

Il faudra maintenant étudier la possibilité d'améliorer notre calcul en utilisant d'autres approches basées sur des substitutions explicites. Une première piste consiste à étudier une extension des travaux sur les calculs génériques de substitutions explicites [Ste00] pour prendre en compte la composition des substitutions. Il faudrait également comprendre comment l'approche proposée dans [KL05, Kes07] et, en particulier, la permutation des substitutions peut être appliquée pour simplifier la notion de substitution du ρ-calcul. L'utilisation des indices deBruijn [dB78] comme dans le $\rho\sigma$-calcul est également envisageable.

Contexte :

Le calcul de réécriture est un modèle de calcul particulièrement adapté pour l'étude de la sémantique des langages basés sur les notions de motif et de filtrage. Certains aspects pratiques liés aux implantations concrètes sont étudiés dans le calcul de réécriture avec substitutions explicites mais les technique utilisées dans ce cas sont essentiellement liés au traitement explicite et efficace des opérations de filtrage et d'application des substitutions correspondantes. L'intégration d'autres techniques d'optimisation et, en particulier, d'une approche basée sur (la réécriture) de graphes déjà étudiées dans le contexte des langages déclaratifs [PJ87], est clairement envisageable dans un calcul utilisé pour la modélisation de ce type de langage.

L'intérêt d'une représentation au niveau objet du calcul des structures de graphes va au-delà des aspects liés à l'efficacité des implantations et concerne également une amélioration de l'expressivité d'un formalisme à base de termes qui a montré ses limites pour la modélisation des système complexes comme les systèmes chimiques et biologiques. Ce type de système peut être modélisé avec de simples termes [BCC+03] mais les structures naturelles pour ce type de formalisme sont les graphes [AK07].

Je me suis donc intéressé à une extension du calcul de réécriture permettant une représentation et une manipulation explicite des structures de graphes.

Questions :

- Comment représenter le partage et les cycles en calcul de réécriture ?
- Quelles sont les propriétés du formalisme obtenu ?
- Quel lien avec les formalismes similaires ?

Contributions :

Le calcul de réécriture explicite permet une représentation au niveau objet des substitutions et donc une certaine forme de partage ainsi qu'un traitement explicite du filtrage syntaxique sur des termes sans partage. Nous introduisons dans ce chapitre un calcul qui étend les contraintes de filtrage à des contraintes d'unification permettant ainsi d'exprimer non seulement le partage mais également des structures cycliques. Le filtrage est toujours effectué explicitement dans ce calcul mais cette fois-ci il est réalisé sur des termes-graphes éventuellement cycliques.

Nous avons montré que ce calcul généralise le calcul de réécriture de base en permettant la manipulation de termes-graphes ainsi que le λ-calcul cyclique en permettant des motifs plus élaborés que des simples variables. Il permet également un encodage des réductions des systèmes de réécriture de termes-graphes. Le calcul introduit ici représente donc un support théorique pour les implantations efficaces et expressives des langages basés sur la réécriture.

La preuve de confluence de ce calcul est encore plus élaborée que pour les autres calculs de réécriture mais elle reste modulaire et peut être clairement étendue pour des éventuelles extensions du filtrage sous-jacent.

5

Calcul de réécriture de graphes

Dans le ρ-calcul de base ainsi que dans les versions présentées dans les chapitres précédents les termes sont vus comme des arbres et l'évaluation est réalisée en remplaçant des sous-arbres. Afin d'améliorer l'efficacité des implantations correspondantes, il est fondamental de penser et d'implanter les ρ-termes par des graphes [BvEG$^+$87]. Dans ce cas, la possibilité de partager des sous-termes permet d'améliorer la gestion de l'espace (en utilisant des pointeurs multiples au même sous-terme au lieu de dupliquer le sous-terme) et l'efficacité en temps (un radical apparaissant dans un sous-terme partagé sera réduit tout au plus une fois et des tests d'égalité peuvent être faits en temps constant quand le partage est maximal).

La réécriture de graphes est une technique utile pour l'optimisation des implantations des langages fonctionnels et déclaratifs [PJ87]. D'ailleurs, la possibilité de définir des cycles mène à une puissance accrue de l'expressivité et permet, en particulier, de représenter facilement des structures de données infinies régulières (*c.-à-d.* avec un nombre fini de sous-structures différentes) comme, par exemple, le graphe cyclique représentant la liste circulaire $ones = 1 \oplus ones$, où « $\oplus$ » dénote l'opérateur de concaténation. La réécriture de termes-graphes cycliques a été largement étudiée, aussi bien d'un point de vue opérationnel [BvEG$^+$87, AK96] que d'un point de vue catégorique/logique [CG99] (voir [SPvE93] pour une présentation de la réécriture de termes-graphes).

Dans ce contexte, un modèle abstrait généralisant le λ-calcul en ajoutant des cycles et des mécanismes de partage a été proposé par Z.M. Ariola et J.W. Klop [AK97]. Leur approche consiste en un formalisme équationnel qui modélise le λ-calcul étendu avec de la récursion explicite. Un λ-graphe est traité comme un système d'équations récursives contenant éventuellement des λ-termes et la réécriture est décrite comme une séquence de transformations équationnelles. Cette approche permet la combinaison des structures de graphes avec les mécanismes d'ordre supérieur du λ-calcul. Un dernier ingrédient important manque toujours : le filtrage (sur les motifs).

Termes

$\mathcal{G}, \mathcal{P}$	::=	$\mathcal{X}$	(Variables)
	|	$\mathcal{K}$	(Constantes)
	|	$\mathcal{P} \twoheadrightarrow \mathcal{G}$	(Abstractions)
	|	$\mathcal{G}\ \mathcal{G}$	(Applications fonctionnelle)
	|	$\mathcal{G} \wr \mathcal{G}$	(Structures)
	|	$\mathcal{G}\,[\mathcal{C}]$	(Applications de contrainte)

Contraintes

$\mathcal{C}$	::=	ϵ	(Identité)
	|	$\mathcal{X} = \mathcal{G}$	(Equations)
	|	$\mathcal{P} \ll \mathcal{G}$	(Equations de filtrage)
	|	$\mathcal{C} \wedge \mathcal{C}$	(Conjonctions de contraintes)

avec $\wedge$ associatif, commutatif et idempotent
et ϵ son élément neutre

FIGURE 5.1 – *Syntaxe de* ρ_{g}

Dans ce chapitre nous présentons un nouveau formalisme, appelé ρ_{g}-calcul, qui généralise le λ-calcul cyclique de la même manière que le ρ-calcul généralise le λ-calcul classique. Le ρ_{g}-calcul manipule des termes avec des variables liées et peut exprimer le partage et les cycles en utilisant des listes d'équations récursives. Comme la variante du ρ-calcul présentée dans la section 4.2, dans le ρ_{g}-calcul les contraintes de filtrage sont explicites et les calculs liés au filtrage sont exécutés au niveau objet du calcul. Pour une présentation complète du calcul et de ses propriétés le lecteur peut se référer à [BBCK06].

Les travaux décrits dans ce chapitre ont été réalisés en collaboration avec Paolo Baldan, Clara Bertolissi et Claude Kirchner.

5.1 Termes avec contraintes généralisées

La syntaxe du ρ_{g}-calcul présentée dans la figure 5.1 étend la syntaxe du ρ-calcul de base avec un nouvel opérateur d'application pour les contraintes noté par « _ [_] » et déjà utilisé dans la version explicite du calcul donnée dans la section 4.3. Comme le ρ-calcul avec filtrage explicite, le ρ_{g}-calcul manipule explicitement les contraintes de filtrage mais il introduit une nouvelle sorte de contrainte, les équations récursives.

Une contrainte de filtrage est générée quand on réduit l'application d'une règle de réécriture à un terme comme, par exemple, l'application $(f(x) \twoheadrightarrow g(x))\ f(a)$ qui est réduite à $g(x)\ [f(x) \ll f(a)]$. Une équation récursive est une contrainte de la forme $\mathcal{X} = \mathcal{G}$ qui peut être vue comme un

environnement associé à un terme. Elle est utilisée pour définir des termes avec partage et cycles et peut être également vue comme une substitution en attente. Par exemple, dans le terme $G = h(x, x)\ [x = s(0)]$ le terme $s(0)$ est considéré partagé et nous pouvons penser à $s(0)$ comme à la valeur associée à la variable x dans le terme G. En résumant, les contraintes du ρ_{g}-calcul sont des conjonctions (construites en utilisant l'opérateur « $\_ \wedge \_$ ») d'équations de filtrage et d'équations récursives. La contrainte vide est notée ϵ. L'opérateur « $\_ \wedge \_$ » est supposé associatif, commutatif et idempotent, avec ϵ comme élément neutre.

Ceci signifie que nous considérons égaux deux ρ_{g}-termes qui diffèrent seulement par l'ordre des équations dans leurs contraintes comme, par exemple, $x\ [x = f(y) \wedge y = g(x)]$ et $x\ [y = g(x) \wedge x = f(y)]$ puisque, intuitivement, ils représentent le même terme-graphe. De la même façon, nous supposons l'opérateur « $\_ \wedge \_$ » associatif. Par exemple, $x\ [\big(x = f(y) \wedge y = g(z)\big) \wedge z = a]$ est équivalent à $x\ [x = f(y) \wedge \big(y = g(z) \wedge z = a\big)]$ et donc, habituellement, les parenthèses sont omises. L'axiome d'idempotence évite la duplication de contraintes identiques et donc $x\ [x \ll f(y) \wedge x \ll f(y)]$ est équivalent à $x\ [x \ll f(y)]$. La duplication peut se produire pendant la réduction de termes contenant des motifs non-linéaires comme, par exemple, pour le terme $(h(x, x) \twoheadrightarrow x)\ h(a, a)$. Dans cet exemple, la contrainte de filtrage $h(x, x) \ll h(a, a)$ produite peut être décomposée en deux contraintes identiques et on obtient le terme $x\ [x \ll a \wedge x \ll a]$ qui est simplifié, par idempotence, en $x\ [x \ll a]$.

Comme dans les chapitres précédents, on suppose que l'opérateur d'application associe à gauche, tandis que les autres opérateurs associent à droite. Les priorités qu'on considère pour les opérateurs du ρ_{g}-calcul sont également similaires à celles dans les chapitres précédents. Voici la liste des opérateurs ordonnés de la plus haute priorité à la plus basse : application « $\_\ \_$ », « $\_ \twoheadrightarrow \_$ », « $\_ \wr \_$ », « $\_\ [\_]$ », « $\_ \ll \_$ », « $\_ = \_$ » et « $\_ \wedge \_$ ». Les ρ_{g}-termes seront généralement dénotés par les symboles $G, H, \ldots$ et les ensembles de contraintes par les symboles $\mathfrak{C}, \mathfrak{D}, \ldots$.

On dit qu'un ρ_{g}-graphe est *bien formé* si chaque variable apparaît tout au plus une fois en tant que membre gauche d'une équation récursive. Tous les ρ_{g}-graphes considérés par la suite sont implicitement bien formés.

On appelle *algébriques* les ρ_{g}-graphes définis par la grammaire suivante :

$$\mathcal{A} \ ::= \ \mathcal{X} \ \mid \ \mathcal{K} \ \mid \ (((\mathcal{K}\ \mathcal{A})\ \mathcal{A})\ldots)\ \mathcal{A} \ \mid \ \mathcal{A}\ [\mathcal{X} = \mathcal{A} \wedge \ldots \wedge \mathcal{X} = \mathcal{A}]$$

Comme pour les ρ-termes, un terme algébrique de la forme $(((f\ G_1)\ G_2)\ldots)\ G_n$ est généralement écrit $f(G_1, G_2, \ldots, G_n)$.

Un ensembles de contraintes dont les équations récursives ont des références mutuelles entre elles, comme dans $x\ [x = f(y) \wedge y = g(x)]$, est utilisée pour représenter les termes cycliques. Nous définissons formellement la notion de cycle en utilisant la notation $\mathsf{Ctx}[\Box]$ pour un contexte avec exactement un trou noté $\Box$. Il faut noter que l'insertion dans un contexte est une

substitution qui peut mener à des captures de variables, *c.-à-d.* des variables qui apparaissent libres dans G peuvent devenir liées dans $\mathsf{Ctx}[G]$.

Définition 18 (Order, cycle) *On dénote par $\leq$ le plus petit pré-ordre sur les variables de récursion tel que $x \geq y$ si $\mathsf{Ctx}_1\{x\} \lll \mathsf{Ctx}_2\{y\}$ pour des contextes $\mathsf{Ctx}_i\{\Box\}$, $i = 1, 2$, où le symbole $\lll$ peut être l'opérateur de récursion $=$ ou l'opérateur de filtrage $\ll$. L'équivalence induite par ce pré-ordre est notée $\equiv$ et on dit que x et y sont cycliquement équivalent ($x \equiv y$) si $x \geq y \geq x$. On écrit $x > y$ si $x \geq y$ et $x \not\equiv y$.*

On dit qu'un ρ_{g}-graphe est acyclique *si $\geq$ est un ordre partiel (et $\equiv$ est ainsi l'identité sur les variables).*

Il faut noter qu'un ρ_{g}-graphe cyclique contient un *cycle*, *c.-à-d.* une séquence de contraintes de la forme $\mathsf{Ctx}_0\{x_0\} \lll \mathsf{Ctx}_1\{x_1\} \wedge \mathsf{Ctx}_2\{x_1\} \lll \mathsf{Ctx}_3\{x_2\} \wedge \ldots \wedge \mathsf{Ctx}_m\{x_n\} \lll \mathsf{Ctx}_{m+1}\{x_0\}$, avec $n, m \in \mathbb{N}$, où $x_0 \equiv x_1 \equiv \ldots \equiv x_n$.

Nous dénotons par $\bullet$ (trou noir) une constante, déjà introduite par Z.M.Ariola et J.W.Klop [AK96] dans une approche équationnelle et également par A.Corradini [Cor93] dans une approche catégorique, pour donner un nom à des termes « inexistants » qui correspondent à l'expression $x\ [x = x]$ (un terme qui boucle sur lui-même). La notation $x =_\circ x$ est une abréviation pour la séquence $x = x_1 \wedge \ldots \wedge x_n = x$.

Nous utilisons parfois une représentation graphique des ρ_{g}-graphes sans contraintes de filtrage pour donner l'intuition de ces termes. Il est clair que n'importe quel terme sans contraintes peut être représenté comme un arbre tandis qu'un ρ_{g}-graphe $G\ [x_1 = G_1 \wedge \ldots \wedge x_n = G_n]$ peut être vu comme une construction $\texttt{letrec}\ x_1 = G_1, \ldots, x_n = G_n\ \texttt{in}\ G$ et représenté par une structure avec partage et cycles. Dans notre approche, la correspondance entre une variable dans le membre gauche d'une règle et son occurrence liée dans le membre droit est représentée en utilisant les mêmes noms pour les variables respectives (au lieu d'utiliser des pointeurs vers l'arrière). La correspondance entre un terme et sa représentation graphique peut être étendue à des ρ_{g}-graphes généraux, contenant éventuellement des contraintes de filtrage, comme décrit dans [Ber05]. Dans ce chapitre, nous utiliserons cette correspondance seulement informellement et pour des ρ_{g}-graphes simples ne contenant pas des équations de filtrage.

Exemple 17 (ρ_{g}-termes) *La représentation graphique des termes ci-dessous est donnée dans la figure 5.2.*

1. *Dans la règle $(2 * x) \rightarrowtriangle ((y + y)\ [y = f(x)])$ le partage dans le membre droit évite la copie de l'objet qui instancie $f(x)$ quand la règle est appliquée à un ρ_{g}-terme.*
2. *Le ρ_{g}-terme $cons(head(x), x)\ [x = cons(0, x)]$ représente une liste infinie de zéros. On peut noter que la variable de récursion x lie l'occurrence de x dans le membre droit de l'équation ainsi que celles dans le terme $cons(head(x), x)$ auquel la contrainte est appliquée.*

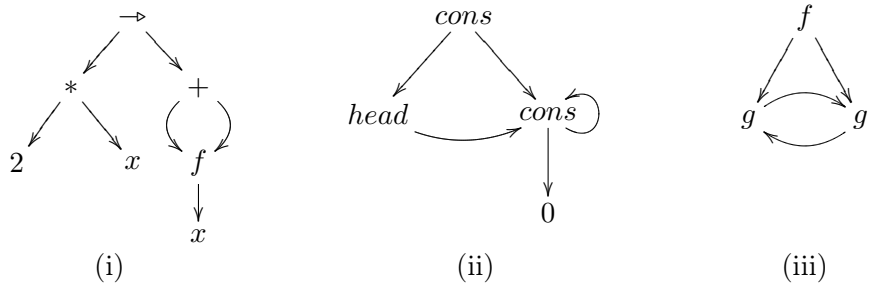


FIGURE 5.2 – *Des ρ_{g}-termes*

3. *Le ρ_{g}-terme $f(x,y)$ $[x = g(y) \wedge y = g(x)]$ est un exemple de partage qui peut être exprimé en utilisant la construction* `letrec`*. On a $x \geq y$ et $y \geq x$ et par conséquent $x \equiv y$.*

Les notions de variable libre et liée des ρ_{g}-termes prennent en considération les trois lieurs du calcul : l'abstraction, la récursion et le filtrage. En particulier, pour simplifier la définition, nous introduisons également le domaine d'une contrainte $\mathfrak{C}$, noté $\mathcal{DV}(\mathfrak{C})$, comme l'ensemble de variables (potentiellement) définies par les équations récursives et de filtrage qu'elle contient. L'ensemble $\mathcal{DV}(\mathfrak{C})$ contient donc, pour toute équation récursive $x = G$ dans $\mathfrak{C}$ la variable x et pour toute équation de filtrage $G_1 \ll G_2$ dans $\mathfrak{C}$ l'ensemble de variables libres de G_1.

Définition 19 (Variables libres, liées et définies) *Etant donne un ρ_{g}-terme G, ses variables libres, notées $\mathcal{FV}(G)$, et ses variables liées, notées $\mathcal{BV}(G)$, sont définies récursivement par :*

$$\begin{array}{rcl@{\qquad}rcl}
\mathcal{FV}(c) & \triangleq & \emptyset & \mathcal{BV}(c) & \triangleq & \emptyset \\
\mathcal{FV}(x) & \triangleq & \{x\} & \mathcal{BV}(x) & \triangleq & \emptyset \\
\mathcal{FV}(A\ B) & \triangleq & \mathcal{FV}(A \wr B) & \mathcal{BV}(A\ B) & \triangleq & \mathcal{BV}(A \wr B) \\
\mathcal{FV}(A \wr B) & \triangleq & \mathcal{FV}(A) \cup \mathcal{FV}(B) & \mathcal{BV}(A \wr B) & \triangleq & \mathcal{BV}(A) \cup \mathcal{BV}(B) \\
\mathcal{FV}(P \twoheadrightarrow A) & \triangleq & \mathcal{FV}(A) \backslash \mathcal{FV}(P) & \mathcal{BV}(P \twoheadrightarrow A) & \triangleq & \mathcal{BV}(A) \cup \mathcal{BV}(P) \cup \mathcal{FV}(P)
\end{array}$$

$$\begin{array}{rcl}
\mathcal{FV}(G\ [\mathfrak{C}]) & \triangleq & (\mathcal{FV}(G) \cup \mathcal{FV}(\mathfrak{C})) \setminus \mathcal{DV}(\mathfrak{C}) \\
\mathcal{BV}(G\ [\mathfrak{C}]) & \triangleq & \mathcal{BV}(G) \cup \mathcal{BV}(\mathfrak{C})
\end{array}$$

Pour une contrainte $\mathfrak{C}$, les variables libres, notées $\mathcal{FV}(\mathfrak{C})$, les variables liées, notées $\mathcal{BV}(\mathfrak{C})$, et les variables définies, notées $\mathcal{DV}(\mathfrak{C})$, sont définies par :

$$\begin{array}{rcl@{\qquad}rcl}
\mathcal{FV}(\epsilon) & \triangleq & \emptyset & \mathcal{DV}(\epsilon) & \triangleq & \emptyset \\
\mathcal{FV}(x = G) & \triangleq & \mathcal{FV}(G) & \mathcal{DV}(x = G) & \triangleq & \{x\} \\
\mathcal{FV}(G_1 \ll G_2) & \triangleq & \mathcal{FV}(G_2) & \mathcal{DV}(G_1 \ll G_2) & \triangleq & \mathcal{FV}(G_1) \\
\mathcal{FV}(\mathfrak{C}_1 \wedge \mathfrak{C}_2) & \triangleq & \mathcal{FV}(\mathfrak{C}_1) \cup \mathcal{FV}(\mathfrak{C}_2) & \mathcal{DV}(\mathfrak{C}_1 \wedge \mathfrak{C}_2) & \triangleq & \mathcal{DV}(\mathfrak{C}_1) \cup \mathcal{DV}(\mathfrak{C}_2)
\end{array}$$

$$\begin{array}{rcl} \mathcal{BV}(\epsilon) & \triangleq & \emptyset \\ \mathcal{BV}(x = G) & \triangleq & \{x\} \cup \mathcal{BV}(G) \\ \mathcal{BV}(G_1 \ll G_2) & \triangleq & \mathcal{FV}(G_1) \cup \mathcal{BV}(G_1) \cup \mathcal{BV}(G_2) \\ \mathcal{BV}(\mathfrak{C}_1 \wedge \mathfrak{C}_2) & \triangleq & \mathcal{BV}(\mathfrak{C}_1) \cup \mathcal{BV}(\mathfrak{C}_2) \end{array}$$

La notion d'α-conversion utilisée dans le ρ-calcul peut être naturellement étendue pour être utilisée pour les termes du $\rho_{\mathtt{g}}$-calcul. Il faut noter que l'ensemble des variables liées dans les sous-termes G d'une application de contrainte $G\ [\mathfrak{C}]$ est l'union entre le domaine de $\mathfrak{C}$ et l'ensemble de variables liées de G. Par exemple, le terme $x\ [x \ll a \wedge x \ll b]$ est équivalent modulo α-conversion au terme $y\ [y \ll a \wedge y \ll b]$ (intuitivement, ce terme est similaire à $y\ [f(y,y) \ll f(a,b)]$).

Il faut noter également que la visibilité d'une variable de récursion est limitée aux $\rho_{\mathtt{g}}$-termes de la contrainte où la variable de récursion est définie et au $\rho_{\mathtt{g}}$-terme auquel cette contrainte est appliquée. Par exemple, dans le terme $f(x,y)\ [x = g(y)\ [y = a]]$ la variable y définie dans l'équation récursive lie son occurrence dans $g(y)$ mais pas dans $f(x,y)$. En fait, le terme ne satisfait pas la condition de nommage puisque y a des occurrences libres et liées.

Ces conventions de nommage nous permettent d'ignorer certains termes (voir les exemples ci-dessous) et d'appliquer ainsi des remplacements (comme pour les règles d'évaluation dans la figure 5.3) naïvement puisqu'aucune capture de variable n'est possible.

Exemple 18 (Les variables libres et liées ont des noms différents) *Etant donné le $\rho_{\mathtt{g}}$-terme $z\ [z = x \twoheadrightarrow y \wedge y = x + x]$, on pourrait envisager de remplacer naïvement la variable y par $x + x$ dans le membre droit de l'abstraction, ce qui conduirait à une capture de variable. Ceci pourrait se produire à cause du fait que le terme précédent ne respecte pas nos conventions de nommage : la capture de variable n'est plus possible si on considère le $\rho_{\mathtt{g}}$-terme $z\ [z = x_1 \twoheadrightarrow y \wedge y = x + x]$ obtenu après α-conversion et qui respecte ces conventions. Si on souhaite lier les occurrences de la variable x apparaissant dans la dernière équation par l'abstraction, alors on doit utiliser une contrainte imbriquée comme dans le $\rho_{\mathtt{g}}$-terme $z\ [z = x \twoheadrightarrow (y\ [y = x + x])]$.*

Exemple 19 (Les variables liées différentes ont des noms différents) *Conformément aux notions de variable libre et liée, dans un $\rho_{\mathtt{g}}$-terme il ne peut pas y avoir de partage entre le membre gauche d'une règle de réécriture et le reste du terme. En d'autres termes, le membre gauche d'une règle de réécriture est « self-contained ». Le partage à l'intérieur du membre gauche est possible et aucune restriction n'est imposée pour le membre droit. Par exemple, dans le terme $f(y, g(y) \twoheadrightarrow a)\ [y = x]$ la première occurrence de y est liée par la variable de récursion, alors que la portée de y dans l'abstraction*

« _ ⇸ _ » est limitée au membre droit de cette abstraction. Le ρ_g-terme devrait, en fait, être réécrit (par α-conversion) en $f(y, g(z) \twoheadrightarrow a)\ [y = x]$.

On doit noter également qu'il n'est pas possible d'exprimer le partage entre les membres gauches et droits d'une abstraction. Par exemple, dans le terme $(x \twoheadrightarrow x)\ [x = a]$ la variable x dans le membre droit de l'abstraction est liée par le x dans le membre gauche et donc le terme peut être α-converti en $(z \twoheadrightarrow z)\ [x = a]$, terme qui vérifie la convention de nommage.

Dans ce chapitre nous nous limitons à des *motifs* (utilisés comme membre gauche des abstractions et des équations de filtrage) qui sont des ρ_g-graphes acycliques algébriques. Par exemple, le ρ_g-graphe $(f(y)\ [y = g(y)] \twoheadrightarrow a)$ n'est pas permit puisque l'abstraction a un membre gauche cyclique.

5.2 Sémantique du calcul de réécriture de graphes

Dans le ρ-calcul de base, quand on réduit l'application d'une règle à un terme, le problème de filtrage correspondant est résolu et la substitution obtenue est appliquée au méta-niveau du calcul. Dans la version explicite du ρ-calcul présentée dans la section 4.3, cette réduction est décomposée en deux phases clairement séparées et toutes les deux traitées explicitement, une calculant les substitutions et l'autre décrivant l'application de ces substitutions.

Dans le ρ_g-calcul, le calcul des substitutions solutions des contraintes de filtrage est effectué explicitement et, si le calcul est réussi, le résultat est une équation récursive ajoutée à la contrainte du terme. Ceci signifie que la substitution n'est pas appliquée forcement immédiatement au terme mais elle est gardée dans l'environnement pour une application ultérieure.

Les règles d'évaluation du ρ_g-calcul présentées dans la figure 5.3 peuvent être séparées en trois catégories : les règles décrivant l'application des abstractions et des structures sur les ρ_g-termes - Basic rules (Règles de base), les règles décrivant la résolution des équations de filtrage - Matching rules (Règles de filtrage) et les règles décrivant les remplacements et le nettoyage (« garbage collection ») - Graph rules (Règles de graphes).

On rappelle que, formellement, les réductions ont lieu par rapport à des classes d'équivalence de ρ_g-termes définies modulo la théorie spécifiée pour l'opérateur de conjonction pour des contraintes, *c.-à-d.* associativité, commutativité, idempotence et axiome d'élément neutre pour ϵ. Les règles d'évaluation sont ainsi appliquées à des ρ_g-termes en utilisant un filtrage modulo la théorie de l'opérateur de conjonction. On verra plus tard que ceci introduit certaines difficultés techniques pour la preuve de confluence du calcul.

Les deux premières règles (ρ) et (δ) viennent du ρ-calcul. La règle (δ) décrit la distributivité de l'application sur les structures construites avec l'opérateur « _ ≀ _ » tandis que la règle (ρ) déclenche l'application d'une règle de réécriture à un ρ_g-terme en appliquant la contrainte de filtrage

BASIC RULES :

$(P \rightarrowtriangle G_2)\ G_3$	$\rightarrowtriangle_\rho$	$G_2\ [P \ll G_3]$
$(P \rightarrowtriangle G_2)\ [\mathfrak{C}]\ G_3$	$\rightarrowtriangle_\rho$	$G_2\ [P \ll G_3 \wedge \mathfrak{C}]$
$(G_1 \wr G_2)\ G_3$	$\rightarrowtriangle_\delta$	$G_1\ G_3 \wr G_2\ G_3$
$(G_1 \wr G_2)\ [\mathfrak{C}]\ G_3$	$\rightarrowtriangle_\delta$	$(G_1\ G_3 \wr G_2\ G_3)\ [\mathfrak{C}]$

MATCHING RULES :

$P \ll (G\ [\mathfrak{C}])$	$\rightarrowtriangle_{propagate}$	$P \ll G \wedge \mathfrak{C}$ **si** $P \notin \mathcal{X}$
$K(G_1, \ldots G_n) \ll K(G'_1, \ldots G'_n)$	$\rightarrowtriangle_{decompose}$	$G_1 \ll G'_1, \ldots, G_n \ll G'_n$ **avec** $n \geq 0$
$x \ll G \wedge \mathfrak{C}$	$\rightarrowtriangle_{solved}$	$x = G \wedge \mathfrak{C}$ **si** $x \notin \mathcal{DV}(\mathfrak{C})$

GRAPH RULES :

$\mathsf{Ctx}[y]\ [y = G \wedge \mathfrak{C}]$	$\rightarrowtriangle_{ext-sub}$	$\mathsf{Ctx}[G]\ [y = G \wedge \mathfrak{C}]$
$G\ [P \lll \mathsf{Ctx}[y] \wedge y = G_1 \wedge \mathfrak{C}]$	$\rightarrowtriangle_{acyc-sub}$	$G\ [P \lll \mathsf{Ctx}[G_1] \wedge y = G_1 \wedge \mathfrak{C}]$ **si** $x > y,\ \forall x \in \mathcal{FV}(P)$ **avec** $\lll \in \{=, \ll\}$
$G\ [\mathfrak{C} \wedge x = G']$	$\rightarrowtriangle_{gc}$	$G\ [\mathfrak{C}]$ **si** $x \notin \mathcal{FV}(\mathfrak{C}) \cup \mathcal{FV}(G)$
$G\ [\epsilon]$	$\rightarrowtriangle_{gc}$	G
$\mathsf{Ctx}[x]\ [x =_\circ x \wedge \mathfrak{C}]$	$\rightarrowtriangle_{bh}$	$\mathsf{Ctx}[\bullet]\ [x =_\circ x \wedge \mathfrak{C}]$
$G\ [P \lll \mathsf{Ctx}[y] \wedge y =_\circ y \wedge \mathfrak{C}]$	$\rightarrowtriangle_{bh}$	$G\ [P \lll \mathsf{Ctx}[\bullet] \wedge y =_\circ y \wedge \mathfrak{C}]$ **si** $x > y,\ \forall x \in \mathcal{FV}(P)$

FIGURE 5.3 – *Sémantique de* ρ_{g}

appropriée au membre droit de la règle. Pour chacune de ces règles une règle supplémentaire prenant en compte l'existence éventuelle de contraintes est ajoutée. Sans ces règles, l'application de ρ_{g}-termes abstraction comme $f(y) \rightarrowtriangle x\ f(y)\ [x = f(y) \rightarrowtriangle x\ f(y)]\ f(a)$ (qui peut encoder une application récursive comme dans l'exemple 22) ne peut pas être réduite. Alternativement, des règles appropriées de distributivité pourraient être utilisées mais cette approche n'est pas considérée dans ce document.

Les MATCHING RULES et, en particulier, la règle (decompose) sont fortement liées à la théorie modulo laquelle on veut calculer les solutions du filtrage. Dans cette première version du ρ_{g}-calcul, nous avons choisi de présenter le ρ_{g}-calcul avec une théorie vide, théorie connue pour être décidable

et unitaire, mais des extensions à des théories plus élaborées sont évidement possibles. Due aux restrictions imposées aux membres gauches des règles de réécriture, on doit seulement décomposer les termes algébriques. Le but de cet ensemble de règles est de produire une contrainte de la forme $x_1 = G_1 \wedge \ldots \wedge x_n = G_n$ à partir d'une équation de filtrage. Ceci est possible quand les membres gauches et droits de l'équation de filtrage sont algébriques mais certains remplacements pourraient être nécessaires (comme décrit par les GRAPH RULES) dès que les termes comporte du partage.

Une équation de filtrage contenant des contraintes est réduite (par la règle propagate) à une contrainte contenant la même équation de filtrage sans les contraintes qui sont propagées au niveau juste au-dessus. Puisque les membres gauche des équations de filtrage sont acycliques, il n'y a pas besoin d'une règle d'évaluation propageant les contraintes du membre gauche de l'équation de filtrage ; les éventuelles contraintes de ce côté de l'équation de filtrage peuvent être propagées plus bas dans le terme en utilisant les règles de substitution et de nettoyage. Les termes algébriques sont décomposés et les équations triviales sont éliminées. Une contrainte de filtrage $x \ll G_1$ est transformée en une équation récursive $x = G_1$ s'il n'existe aucune autre contrainte de la forme $x = G_2$ ou $x \ll G_2$ dans la liste de contraintes. Par exemple, la contrainte $x \ll a \wedge x \ll b$ ne peut pas être réduite indiquant ainsi que le problème de filtrage (non linéaire) original n'a aucune solution.

Les règles GRAPH RULES sont héritées du λ-calcul cyclique de Ariola et Klop. Les deux premières règles font une copie d'un ρ_{g}-terme associé à une variable récursive dans un terme qui est dans la portée de la contrainte correspondante. Ceci est important quand un radical doit être rendu explicite (par exemple, dans $x\ a\ [x = a \twoheadrightarrow b]$) ou quand une équation de filtrage doit être résolue (par exemple, dans $a\ [a \ll x \wedge x = a]$). L'ordre sur les variables des ρ_{g}-termes permet de réaliser les copies seulement vers le haut. Sans cette condition la confluence est perdue : le ρ_{g}-terme $z_1\ [z_1 = x \twoheadrightarrow z_2\ s(x) \wedge z_2 = y \twoheadrightarrow z_1\ s(y)]$ peut être réduit soit à $z_1\ [z_1 = x \twoheadrightarrow z_1\ s(s(x))]$ soit à $z_1\ [z_1 = x \twoheadrightarrow z_2\ s(x) \wedge z_2 = y \twoheadrightarrow z_2\ s(s(y))]$ (voir [AK97] pour le contre-exemple complet). Comme on verra par la suite, en imposant certaines restrictions sur la forme des règles de réécriture, cet ordre est un des ingrédients essentiels pour obtenir la confluence du ρ_{g}-calcul.

Les règles (gc) éliminent les équations récursives qui représentent des parties non reliées du ρ_{g}-terme. On n'élimine pas les contraintes de filtrage, gardant de ce fait la trace des échecs de filtrage pendant une réduction non réussie. Les règles (bh) liées aux trous noirs remplacent les ρ_{g}-termes non définis par la constante $\bullet$.

La relation induite par les règles du ρ_{g}-calcul est notée $\mapsto_{\not\rho\mathsf{g}}$.

Exemple 20 (Non-linéarité) *Le filtrage impliquant des motifs non-linéaires peut mener à une forme normale qui est soit une contrainte contenant seulement des équations récursives (qui représente un filtrage*

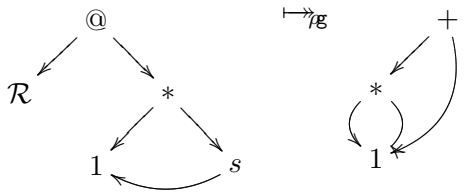


FIGURE 5.4 – *Exemple de réductions*

réussi) soit une contrainte qui contient un certain nombre d'équations de filtrage (représentant un échec de filtrage).

$$
\begin{array}{lll}
 & f(y,y) \ll f(a,a) & f(y,y) \ll f(a,b) \\
\mapsto_{decompose} & y \ll a \quad (par\ idempotence) & \mapsto_{decompose} \quad y \ll a \wedge y \ll b \\
\mapsto_{s} & y = a &
\end{array}
$$

Il serait intéressant d'étudier les stratégies appropriées qui retardent l'application des règles de substitution (externes et acycliques) (ext-sub) et (acyc-sub) pour garder l'information de partage aussi longtemps que possible. Une idée, illustrée dans le prochain exemple, consiste à appliquer les règles de substitution seulement si nécessaire pour produire de nouveaux radicaux pour les règles de base ou de filtrage. De plus, les règles de substitutions sont utilisées pour « éliminer » les équations récursives triviales de la forme $x = y$.

Définition 20 *La stratégie* SharingStrat *exécute une étape de réduction en appliquant les règles d'évaluation (ext-sub) or (acyc-sub) à un ρ_{g}-terme G seulement si :*

- *une variable en position active est instanciée par une abstraction ou une structure, ou*
- *une variable dans une équation de filtrage bloquée est instanciée, ou*
- *une variable est instanciée par une variable.*

Exemple 21 (Multiplication) *Si on utilise une notation infixée pour la constante « $*$ » alors le ρ_{g}-terme suivant correspond à l'application de la règle de réécriture $\mathcal{R} = x * s(y) \rightarrowtriangle (x * y + x)$ au terme $1 * s(1)$ où la constante 1 est partagée. Le résultat est montré sous forme graphique dans la figure 5.4.*

$$
\begin{array}{ll}
 & (x * s(y) \rightarrowtriangle (x * y + x))\ (z * s(z)\ [z = 1]) \\
\mapsto_{p} & x * y + x\ [x * s(y) \ll (z * s(z)\ [z = 1])] \\
\mapsto_{propagate} & x * y + x\ [x * s(y) \ll z * s(z), z = 1] \\
\mapsto\!\!\!\!\!\rightarrow_{decompose} & x * y + x\ [x \ll z \wedge y \ll z \wedge z = 1] \\
\mapsto\!\!\!\!\!\rightarrow_{solved} & x * y + x\ [x = z \wedge y = z \wedge z = 1] \\
\mapsto\!\!\!\!\!\rightarrow_{es} & (z * z + z)\ [x = z \wedge y = z \wedge z = 1] \\
\mapsto\!\!\!\!\!\rightarrow_{gc} & (z * z + z)\ [z = 1]
\end{array}
$$

L'exemple suivant montre comment le combinateur de point fixe du λ-calcul peut être représenté par un ρ_{g}-terme cyclique.

Exemple 22 *On considère la règle de réécriture $R_Y = Y\ x \twoheadrightarrow x\ (Y\ x)$ qui exprime le comportement du combinateur de point fixe Y du λ-calcul. Etant donné un terme t, on obtient la séquence de réécriture infinie :*

$$Y\ t\ \twoheadrightarrow_{R_Y}\ t\ (Y\ t)\ \twoheadrightarrow_{R_Y}\ t\ (t\ (Y\ t))\ \twoheadrightarrow_{R_Y}\ \ldots$$

qui, dans un certain sens ([KKSdV91, Cor93]), converge vers le terme infini $t\ (t\ (t\ (\ldots)))$.

On peut représenter le combinateur Y dans le ρ_{g}-calcul par le terme :

$$Y \triangleq x_0\ [x_0 = x \twoheadrightarrow x\ (x_0\ x)].$$

Si on prend $R = x \twoheadrightarrow x\ (x_0\ x)$, alors on a la réduction suivante :

$$\begin{array}{ll} & Y\ G \\ \mapsto_{ext-sub} & (x \twoheadrightarrow x\ (x_0\ x))\ [x_0 = R]\ G \\ \mapsto_{p} & x\ (x_0\ x)\ [x \ll G \wedge x_0 = R] \\ \mapsto_{s} & x\ (x_0\ x)\ [x = G \wedge x_0 = R] \\ \mapsto\!\!\!\!\twoheadrightarrow_{es} & G\ (x_0\ G)\ [x = G \wedge x_0 = R] \\ \mapsto_{gc} & G\ (x_0\ G)\ [x_0 = R] \\ \mapsto\!\!\!\!\twoheadrightarrow_{\mathsf{g}} & G(G \ldots (x_0\ G))\ [x_0 = R] \\ \mapsto\!\!\!\!\twoheadrightarrow_{\mathsf{g}} & \ldots \end{array}$$

En continuant la réduction, on « converge » vers le terme de la figure 5.5(a).

On peut obtenir une implantation plus efficace de la réduction du même terme en utilisant une méthode proposée par D.A.Turner [Tur79] où la règle R_Y est modélisée en utilisant le terme cyclique représenté dans la figure 5.5(b). Ceci correspond en ρ_{g}-calcul au ρ_{g}-terme

$$Y_T \triangleq x \twoheadrightarrow (z\ [z = x\ z])$$

On obtient dans ce cas la réduction suivante :

$$\begin{array}{ll} & Y_T\ G \\ \mapsto_{p} & z\ [z = x\ z]\ [x \ll G] \\ \mapsto_{s} & z\ [z = x\ z]\ [x = G] \\ \mapsto_{ext-sub} & z\ [z = G\ z]\ [x = G] \\ \mapsto\!\!\!\!\twoheadrightarrow_{gc} & z\ [z = G\ z] \end{array}$$

Le ρ_{g}-terme qui en résulte est représenté dans la figure 5.5(c). Si on déplie de façon intuitive ce ρ_{g}-terme cyclique alors on obtient le terme infini montré dans la figure 5.5(a).

Cette réduction montre qu'une séquence finie de réécritures sur des ρ_{g}-termes cycliques peut correspondre à une réduction infinie de termes.

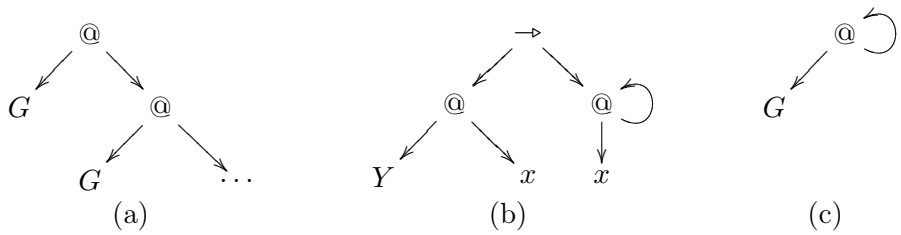


FIGURE 5.5 – *Exemples de réductions*

5.3 Résultats de confluence

Puisque le ρ_{g}-calcul généralise le ρ-calcul et le λ-calcul, il n'est pas surprenant qu'il n'est pas terminant. En plus des réductions infinies obtenues pour des ρ_{g}-termes correspondant à des λ-termes comme $\Omega = (\lambda x.xx)(\lambda x.xx)$, dans le ρ_{g}-calcul on peut avoir des réductions non-terminantes induites par la présence de cycles dans les termes, comme dans

$$x\ [x = f(x)] \mapsto_{ext-sub} f(x)\ [x = f(x)] \mapsto_{ext-sub} \dots$$

Nous montrons que pour un ρ_{g}-calcul avec certaines restrictions de linéarité on obtient la confluence sur des classes d'équivalence de ρ_{g}-termes. La méthode originale de preuve proposée ici généralise la preuve de la confluence du λ-calcul cyclique [AK97] pour un formalisme permettant de traiter des termes contenant des motifs et des équations de filtrage et où la réécriture est réalisée modulo une théorie équationnelle [Ohl98]. La preuve utilise le concept de « développement » et la propriété de « développements finis » déjà définis dans la théorie classique du λ-calcul. Nous présentons ici seulement les grandes lignes de la preuve ; la preuve complète est disponible dans [BBCK06].

La confluence pour les systèmes d'ordre supérieur utilisant du filtrage non-linéaire est un problème délicat à cause des paires critiques non joignable obtenues habituellement pour ce type de formalisme. Ceci a été montré par J.W.Klop dans le contexte du λ-calcul [Klo80] et, comme on a vu dans la section 2.4, le contre-exemple de Klop peut être encodé dans le ρ-calcul [Wac05]. Le contre-exemple reste valide dans le ρ_{g}-calcul et donc, comme pour $\rho^{\circ}_{\mathsf{x}}$, nous considérons une version du ρ_{g}-calcul avec certaines conditions de linéarité.

Définition 21 (ρ_{g}-calcul linéaire) *La classe de* motifs linéaires *(algébriques) est définie par :*

$$\mathcal{L} \ ::= \ \mathcal{X} \ \mid \ \mathcal{K} \ \mid \ (((\mathcal{K}\ \mathcal{L}_0)\ \mathcal{L}_1)\dots)\ \mathcal{L}_n \ \mid \ \mathcal{L}_0\ [\mathcal{X}_1 = \mathcal{L}_1 \wedge \dots \wedge \mathcal{X}_n = \mathcal{L}_n]$$

où on suppose que $\mathcal{FV}(\mathcal{L}_i) \cap \mathcal{FV}(\mathcal{L}_j) = \emptyset$ pour $i \neq j$. Une contrainte de la forme $[L_1 \lll G_1 \wedge \dots \wedge L_n \lll G_n]$, où $\lll \in \{=, \ll\}$, est linéaire *si pour tous motifs L_i, L_j avec $1 \leq i \neq j \leq n$ on a $\mathcal{FV}(L_i) \cap \mathcal{FV}(L_j) = \emptyset$.*

Le $\rho_{\mathtt{g}}$-calcul linéaire *est le* $\rho_{\mathtt{g}}$*-calcul où tous les motifs et toutes les contraintes sont linéaires.*

Comme mentionné précédemment, l'opérateur « $\_ \wedge \_$ » est supposé associatif, commutatif et idempotent, avec ϵ comme élément neutre. Cependant, dans le $\rho_{\mathtt{g}}$-calcul linéaire, l'idempotence n'est pas nécessaire puisque les contraintes de la forme $x \ll G \ \wedge \ x \ll G$ ne sont pas permises (et ne peuvent pas être obtenues par réduction). Par conséquent, dans le $\rho_{\mathtt{g}}$-calcul, la réécriture agit sur des classes d'équivalence de $\rho_{\mathtt{g}}$-termes par rapport à $\sim_{\mathtt{AC1}}$, la relation de congruence générée par les axiomes d'associativité et commutativité pour l'opérateur « $\_ \wedge \_$ », et l'axiome d'élément neutre pour ϵ. $\mapsto_{\not\mathtt{g}}$ dénote la relation induite par les règles du $\rho_{\mathtt{g}}$-calcul et donc la relation induite sur des classes d'`AC1`-équivalence est notée $\mapsto_{\rho_{\mathtt{g}/\mathtt{AC1}}}$ (voir définition 15).

Concrètement, dans la plupart des preuves nous utiliserons la relation de réécriture modulo `AC1`, notée $\mapsto_{\rho_{\mathtt{g}},\mathtt{AC1}}$. Rappelons tout d'abord que dans ce cas les règles de réécriture agissent sur des termes à la place des classes d'équivalence de termes et le filtrage modulo `AC1` est effectué à chaque étape de la réduction (voir définition 16). Il faut mentionner que le filtrage modulo `AC1` peut mener à un nombre infini de solutions, mais l'ensemble complet de solutions est finitaire et a comme représentant canonique la solution dans laquelle les termes sont normalisés par rapport à l'élément neutre [Kir90]. D'un point de vue calculatoire, la réécriture modulo `AC1` est plus convenable que la réécriture sur les classes d'équivalences `AC1`. En fait, dans ce dernier cas la classe complète (et donc potentiellement infinie) doit être explorée quand on recherche les termes réductible. D'autre part, comme on verra dans la section suivante, puisque des conditions de cohérence (similaires à celle du $\rho^{\circ}_{\mathtt{x}}$-calcul) sont satisfaites par notre calcul, la confluence de la relation $\mapsto_{\rho_{\mathtt{g}},\mathtt{AC1}}$ implique la confluence de la relation $\mapsto_{\rho_{\mathtt{g}/\mathtt{AC1}}}$.

5.3.1 Résumé de la preuve

La preuve de confluence est assez élaborée et nous l'avons décomposée en un certain nombre de lemmes pour obtenir le résultat final. Sa complexité est principalement due à la non-terminaison du système et au fait que l'équivalence modulo `AC1` sur les $\rho_{\mathtt{g}}$-graphes doit être considérée tout le long de la preuve.

Nous commençons par prouver un lemme fondamental de cohérence qui montre que la relation de réécriture du $\rho_{\mathtt{g}}$-calcul à le bon comportement par rapport à la relation de congruence `AC1`. Ce lemme assure que s'il existe un pas de réécriture à partir d'un $\rho_{\mathtt{g}}$-graphe G, alors le « même » pas peut être exécuté à partir de n'importe quel terme `AC1`-équivalent à G.

Lemme 13 (Cohérence modulo AC1) *Pour tous termes A, B, A' tels que $A \mapsto_{\rho_g,AC1} B$ et $A \sim_{AC1} A'$, il existe B' tel que $B \sim_{AC1} B'$ et $A' \mapsto_{\rho_g,AC1} B'$.*

$$\begin{array}{ccc} A & \sim_{AC1} & A' \\ {\scriptstyle \rho_g,AC1}\downarrow & & \vdots\,{\scriptstyle \rho_g,AC1} \\ B & \sim_{AC1} & B' \end{array}$$

Nous précisons que, puisque la cohérence est vérifiée pour toutes les règles du ρ_g-calcul, alors elle est également vérifiée pour n'importe quel sous-ensemble de règles d'évaluation du ρ_g-calcul et, en particulier, pour la sémantique complète du ρ_g-calcul.

Pour prouver la confluence de $\mapsto\!\!\!\!\!\rightarrow_{\rho_g,\mathtt{AC1}}$ nous utilisons une technique inspirée par celle adoptée pour montrer la confluence du λ-calcul cyclique [AK97]. Le nombre plus grand de règles d'évaluation du ρ_g-calcul et le traitement explicite de la relation de congruence sur les ρ_g-graphes rendent la preuve pour le ρ_g-calcul beaucoup plus élaborée.

L'idée principale de la preuve est de séparer les règles en deux sous-ensembles, de montrer séparément leur confluence et de prouver ensuite la confluence de l'union en utilisant un lemme de commutation pour les deux ensembles de règles. L'ensemble de règles du ρ_g-calcul est ainsi partagé en les deux sous-ensembles suivants :

- les *règles* Σ, contenant les règles de substitution (ext-sub) et (acyc-sub), plus la règle (δ) ;
- les *règles* τ, contenant toutes les autres règles.

Les règles Σ incluent les règles de substitution qui représentent « la part non-terminante » du ρ_g-calcul. La règle (δ) est également incluse dans les règles Σ bien qu'elle pourrait être ajoutée sans risque au règles τ en gardant cet ensemble de règles terminant. Ce choix est motivé par le fait que, pour des raisons de non-linéarité, ajouter la règle (δ) au règles τ aurait posé des problèmes dans la preuve du lemme final de commutation.

La preuve de confluence de $\mapsto\!\!\!\!\!\rightarrow_{\rho_g,\mathtt{AC1}}$ est structurée en trois parties :

- la confluence modulo AC1 de la relation induite par les règles τ,
- la confluence modulo AC1 de la relation induite par les règles Σ,
- la confluence générale modulo AC1.

Pour prouver la confluence modulo AC1 de la relation induite par les règles τ nous utilisons le fait qu'une relation fortement normalisante et localement confluente modulo AC1 est confluente modulo AC1 si la propriété de cohérence est vérifiée (ceci suit immédiatement par [Ohl98]). Pour prouver la normalisation forte, une interprétation polynomiale des opérateurs du ρ_g-calcul est utilisée.

La confluence locale modulo AC1 est prouvée par analyse de paires critiques. Nous pouvons conclure ainsi la confluence modulo AC1 de la relation induite par les règles τ en utilisant la propriété de cohérence du Lemme 13.

Lemme 14 *La relation $\mapsto_\tau$ est confluente modulo* `AC1`*.*

La preuve de confluence modulo `AC1` de la relation induite par les règles Σ est la partie la plus complexe de la preuve. Les difficultés dans cette preuve résultent du fait que la relation de réécriture induite par les règles Σ n'est pas fortement normalisante. En particulier, on peut noter que les relations de réécriture induites par les règles de substitution sont toutes les deux non terminantes en présence de cycles :

$x\ [x = f(y) \wedge y = g(y)] \mapsto_{acyc-sub} x\ [x = f(g(y)) \wedge y = g(y)] \mapsto_{acyc-sub} \cdots$

$y\ [y = g(y)] \mapsto_{ext-sub} g(y)\ [y = g(y)] \mapsto_{ext-sub} \cdots$

Par conséquent, les techniques utilisées pour la relation τ ne s'appliquent pas pour prouver la confluence dans ce dernier cas. En s'inspirant de la preuve de confluence du λ-calcul cyclique [AK97], nous utilisons la *méthode de développements complets* du λ-calcul en l'adaptant à la relation $\mapsto_\Sigma$.

Plus précisément, nous utilisons la *méthode de développements complets* du λ-calcul en définissant une version terminante de la relation induite par les règles Σ (le développement), et en utilisant ses propriétés pour déduire la confluence de la relation originale de réécriture. D'abord, nous définissons une nouvelle relation de réécriture $\mapsto_{Cpl}$ avec la même fermeture transitive que $\mapsto_\Sigma$. Intuitivement, un pas de réécriture $\mapsto_{Cpl}$ sur un terme G représente la réduction complète d'un ensemble de radicaux fixes au départ dans G. En d'autres termes, certains radicaux sont soulignés dans G et un développement complet de ces radicaux est effectué par la relation $\mapsto_{Cpl}$.

Définition 22 (Relation $\mapsto_{Cpl}$) *Etant donnés les termes G_1 et G_2, on a $G_1 \mapsto_{Cpl} G_2$ ssi il existe une version soulignée G'_1 de G_1 telle que $G'_1 \mapsto\!\!\!\!\rightarrow_{\underline{\Sigma}} G_2$ et G_2 est en forme normale par rapport à la relation $\mapsto_{\underline{\Sigma}}$ induite par les règles*

$(G_1 \underline{\wr} G_2)\ G_3$	$\rightarrowtriangle_{\underline{\delta}}$	$G_1\ G_3 \wr G_2\ G_3$
$(G_1 \underline{\wr} G_2)\ [\mathfrak{C}]\ G_3$	$\rightarrowtriangle_{\underline{\delta}}$	$(G_1\ G_3 \wr G_2\ G_3)\ [\mathfrak{C}]$
$\mathsf{Ctx}[\underline{y}]\ [\underline{y} = G \wedge \mathfrak{C}]$	$\rightarrowtriangle_{\underline{ext-sub}}$	$\mathsf{Ctx}[G]\ [y = G \wedge \mathfrak{C}]$
$G\ [P \lll \mathsf{Ctx}[\underline{y}] \wedge y = G_1, \mathfrak{C}]$	$\rightarrowtriangle_{\underline{acyc-sub}}$	$G\ [P \lll \mathsf{Ctx}[G_1] \wedge y = G_1 \wedge \mathfrak{C}]$ ***si*** $x > \underline{y},\ \forall x \in \mathcal{FV}(P)$ ***avec*** $\lll \in \{=, \ll\}$

Nous prouvons ensuite que la relation $\mapsto_{\underline{\Sigma}}$ est faiblement normalisante ainsi que sa confluence locale modulo `AC1` et nous obtenons finalement la confluence forte modulo `AC1` de la relation $\mapsto_{Cpl}$. Notez que si la relation $\mapsto_{Cpl}$ est fortement confluente modulo `AC1` alors sa fermeture transitive l'est aussi. La confluence de la relation $\mapsto_\Sigma$ suit alors facilement en montrant que la relation $\mapsto_\Sigma$ et la relation $\mapsto_{Cpl}$ ont la même fermeture transitive. Nous pouvons conclure ainsi la confluence modulo `AC1` de la relation induite par les règles Σ.

Lemme 15 *La relation $\mapsto_\Sigma$ est confluente modulo* `AC1`*.*

Finalement, nous considérons l'union des relations induites par les sous-ensembles de règles τ et Σ. La confluence modulo AC1 générale est montrée en utilisant les résultats précédents et le fait que la commutation modulo AC1 des relations $\mapsto_\tau$ et $\mapsto_\Sigma$ peut être montrée en analysant les paires critiques.

Lemme 16 *La relation $\mapsto_{\rho_g, AC1}$ est confluente modulo AC1.*

Comme mentionné au début de cette section, ce que nous visons est un résultat plus général sur la réécriture sur des classes d'AC1-équivalence de ρ_g-termes. Grâce à la propriété de cohérence des règles d'évaluation du ρ_g-calcul avec l'AC1, la propriété de Church-Rosser sur les classes d'AC1-équivalence pour la relation de réécriture du ρ_g-calcul peut être facilement dérivée du dernier lemme. Nous obtenons comme conséquence immédiate la confluence modulo AC1 du ρ_g-calcul.

Théorème 15 *Le ρ_g-calcul linéaire est confluent.*

Le ρ_g-calcul est donc confluent dans les mêmes conditions que le ρ°_x-calcul et le ρ-calcul.

5.4 Pouvoir d'expression

Nous montrons dans cette section que le ρ_g-calcul est un formalisme tout à fait expressif qui permet de simuler non seulement le ρ-calcul de base mais aussi le λ-calcul cyclique, fournissant ainsi un cadre homogène pour le filtrage sur les motifs et les structures de graphe d'ordre supérieur [BBCK05].

Par ailleurs, puisqu'on a la possibilité de représenter directement des structures avec des cycles et du partage, la question qui se pose naturellement est si la réécriture de termes-graphes de premier ordre peut être simulée dans ce formalisme. Dans cette section nous donnons une réponse positive ; les preuves complètes peuvent être trouvées dans [BBCK06].

5.4.1 Calcul de réécriture de base

L'ensemble de termes du ρ-calcul est un sous-ensemble strict de l'ensemble de termes du ρ_g-calcul (modulo quelques conventions syntactiques). La différence principale pour les ρ-termes est la restriction pour certaines versions du ρ-calcul de l'ensemble de contraintes à une contrainte simple nécessairement de la forme « _ $\ll$ _ » (contrainte de filtrage retardée).

Tout d'abord nous avons prouvé par induction structurelle que les règles de filtrage MATCHING RULES du ρ_g-calcul sont cohérentes par rapport à un algorithme de filtrage classique limité aux motifs algébriques (comme celui présenté dans la section 2.2).

Lemme 17 *Etant donné un ρ-terme algébrique T avec $\mathcal{FV}(T) = \{x_1, \ldots, x_n\}$ et un problème de filtrage syntaxique $T \prec\!\!\!\prec U$ avec la solution $\sigma = \{U_1/x_1, \ldots, U_n/x_n\}$, on a $T \ll U \mapsto\!\!\!\!\!\rightarrow_{\mathcal{M}} x_1 = U_1, \ldots, x_n = U_n$.*

En utilisant ce dernier résultat, nous avons montré que pour chaque étape de réduction dans le ρ-calcul on a une séquence d'étapes de réductions correspondante dans le ρ_{g}-calcul et donc, qu'une réduction en ρ-calcul peut être simulée dans le ρ_{g}-calcul.

Théorème 16 (Complétude) *Etant donnés les ρ-termes T et T', s'il existe une réduction $T \mapsto\!\!\!\!\!\rightarrow_{\rho\delta} T'$ dans le ρ-calcul alors $T \mapsto\!\!\!\!\!\rightarrow_{\mathsf{fg}} T'$ dans le ρ_{g}-calcul.*

Dans le cas des échecs de filtrage, les deux calculs manipulent les erreurs d'une manière légèrement différente, même si, dans les deux cas, les échecs de filtrage ne sont pas réduits à une constante spéciale comme dans ρ_{stk}. En particulier, on peut avoir une décomposition d'un problème de filtrage sans solution dans le ρ_{g}-calcul et donc, il est possible qu'un ρ-terme en forme normale puisse être encore réduit dans le ρ_{g}-calcul. La discussion de la section 4.2 sur les avantages de décomposer des problèmes de filtrage par rapport au traçage de l'origine des erreurs (de filtrage) s'applique également pour le ρ_{g}-calcul.

Exemple 23 (Echec de filtrage dans le ρ-calcul et le ρ_{g}-calcul) *Le terme $(f(a) \twoheadrightarrow b)\ f(c)$ est en forme normale en ρ-calcul tandis que dans le ρ_{g}-calcul l'origine de l'échec de filtrage peut être identifiée avec précision :*

$$\begin{array}{ll} & (f(a) \twoheadrightarrow b)\ f(c) \\ \mapsto_{\rho} & b\ [f(a) \ll f(c)] \\ \mapsto_{decompose} & b\ [a \ll c] \end{array}$$

On peut noter que dans le ρ-calcul, puisque l'algorithme de filtrage ne peut pas calculer une substitution solution de l'équation $f(a) \ll f(c)$, alors la règle (ρ) ne peut pas être appliquée et la réduction est ainsi bloquée. D'autre part, dans le ρ_{g}-calcul les règles de filtrage peuvent décomposer partiellement l'équation de filtrage jusqu'à ce que l'échec $a \ll c$ soit atteint.

5.4.2 λ-calcul cyclique

Le λ-calcul cyclique proposé par Z.M.Ariola et J.W.Klop est un cadre équationnel pour la réécriture de termes-graphes avec des cycles. Il étend le λ-calcul en ajoutant une construction `letrec` de telle manière que les nouveaux termes, appelés λ-graphes, sont représentés comme des systèmes d'équations récursives (potentiellement imbriquées) sur des λ-termes standard. Si le système est utilisé sans aucune restrictions sur les règles, alors la confluence est perdue. Cette propriété peut être néanmoins obtenue en contrôlant les opérations sur les équations récursives. Le calcul qui en résulte, appelé $\lambda\phi$ [AK97],

(β)	$(\lambda x.t_1)\ t_2$	$\rightarrow_\beta$	$\langle t_1 \mid x = t_2 \rangle$
(external sub)	$\langle \mathsf{Ctx}[y] \mid y = t, E \rangle$	$\rightarrow_{es}$	$\langle \mathsf{Ctx}[t] \mid y = t, E \rangle$
(acyclic sub)	$\langle t_1 \mid y = \mathsf{Ctx}[x], x = t_2, E \rangle$	$\rightarrow_{ac}$	$\langle t_1 \mid y = \mathsf{Ctx}[t_2], x = t_2, E \rangle$ *si* $y > x$
(black hole)	$\langle \mathsf{Ctx}[x] \mid x =_\circ x, E \rangle$	$\rightarrow_\bullet$	$\langle \mathsf{Ctx}[\bullet] \mid x =_\circ x, E \rangle$
	$\langle t \mid y = \mathsf{Ctx}[x], x =_\circ x, E \rangle$	$\rightarrow_\bullet$	$\langle t \mid y = \mathsf{Ctx}[\bullet], x =_\circ x, E \rangle$ *si* $y > x$
(garbage collect)	$\langle t \mid E, E' \rangle$	$\rightarrow_{gc}$	$\langle t \mid E \rangle$ *si* $E' \neq \epsilon$ *et* $E' \perp (E, t)$
	$\langle t \mid \epsilon \rangle$	$\rightarrow_{gc}$	t

FIGURE 5.6 – *Règles d'évaluation du $\lambda\phi$-calcul*

est suffisamment puissant pour encoder le λ-calcul classique [Bar84] ainsi que le $\lambda\mu$-calcul [Par92] et le $\lambda\sigma$-calcul avec des noms [ACCL91b] étendu avec du partage horizontal et vertical. La syntaxe de $\lambda\phi$ est la suivante :

$$t \quad ::= \quad x \mid f(t_1, \ldots t_n) \mid t_0\ t_1 \mid \lambda x.t \mid \langle t_0 \mid \ x_1 = t_1, \ldots, x_n = t_n \rangle$$

L'ensemble de $\lambda\phi$-termes est composé de λ-termes classiques (*c.-à-d.* variables, fonctions d'arité fixe, applications, abstractions) et de nouveaux termes construits en utilisant la construction `letrec` : $\langle t_0 \mid \ x_1 = t_1, \ldots, x_n = t_n \rangle$, où on suppose que les variables de récursion x_i, $i = 1, \ldots, n$, sont toutes distinctes. On dénote par E une séquence non-ordonnée d'équations $x_1 = t_1, \ldots, x_n = t_n$ et par ϵ la séquence vide. Les variables sont liées soit par la lambda abstraction soit par une équation récursive. On dénote par $\leq$ le plus petit pré-ordre sur les variables de récursion tels que $x \geq y$ si $x = \mathsf{Ctx}[y]$ pour un certain contexte $\mathsf{Ctx}[\Box]$. On écrit $x > y$ si $x \geq y$ et $x \not\equiv y$, où $\equiv$ est l'équivalence induite par le pré-ordre, *c.-à-d.* $x \equiv y$ si $x \geq y \geq x$ (les variables x et y apparaissent dans un cycle). On écrit $E \perp (E', t)$, E est orthogonal à une séquence d'équations E' et un terme t, si les variables de récursion liées de E n'intersectent pas l'ensemble de variables libres de E' et t. La notation $x =_\circ x$ est une abréviation pour la séquence d'équations récursives $x = x_1, \ldots, x_n = x$.

Les règles de réduction du $\lambda\phi$-calcul de base sont données dans la figure 5.6. Certaines extensions de cet ensemble de règles de base peuvent être considérées [AK97] en ajoutant des règles de distribution ($\lambda\phi_1$) ou des règles de fusionnement et d'élimination ($\lambda\phi_2$). Par la suite nous concentrerons notre attention sur le système de base de la figure 5.6. Dans la règle β, la variable x liée par λ devient liée par une équation récursive après la réduction. Les deux règles de substitution sont employées pour faire une copie d'un graphe associé à une variable de récursion. La restriction sur l'ordre des variables de récursion est introduite pour assurer la confluence dans le

cas des configurations cycliques de lambda radicaux. La condition $y > x$ dans les règles (acyclic sub) et (black hole) est nécessaire afin d'assurer la confluence du système. La condition $E' \neq \epsilon$ dans la règle (garbage collect) évite des réductions non-terminantes triviales. On dénote par $\mapsto_{\lambda\phi}$ la relation de réécriture induite par l'ensemble de règles de la figure 5.6 et par $\mapsto\!\!\!\!\rightarrow_{\lambda\phi}$ sa fermeture réflexive et transitive.

Les termes de $\lambda\phi$ peuvent être facilement traduits en termes du ρ_{g}-calcul. La différence principale entre $\lambda\phi$ et le ρ_{g}-calcul est la restriction de l'ensemble de contraintes à une liste d'équations récursives. Les contraintes de filtrage ne sont pas nécessaires puisqu'en λ-calcul le filtrage est toujours trivialement satisfait.

Définition 23 (Translation) *La translation d'un $\lambda\phi$-terme t en un ρ_{g}-terme, notée $[\![t]\!]$, est définie inductivement par :*

$$\begin{array}{rcl} [\![x]\!] & \triangleq & x \\ [\![\lambda x.t]\!] & \triangleq & x \rightarrowtriangle [\![t]\!] \\ [\![t_0\ t_1]\!] & \triangleq & [\![t_0]\!]\ [\![t_1]\!] \\ [\![f(t_1,\ldots,t_n)]\!] & \triangleq & f([\![t_1]\!],\ldots,[\![t_n]\!]) \\ [\![\langle t_0|\ x_1 = t_1,\ldots,x_n = t_n\rangle]\!] & \triangleq & [\![t_0]\!]\ [x_1 = [\![t_1]\!] \wedge \ldots \wedge x_n = [\![t_n]\!]] \end{array}$$

Les règles d'évaluation du ρ_{g}-calcul peuvent être vues comme une généralisation des celles du $\lambda\phi$-calcul. La règle β peut être simulée en utilisant les règles Basic rules du ρ_{g}-calcul. Les autres règles peuvent être simulées en utilisant les règles correspondantes dans le sous-ensemble Graph rules du ρ_{g}-calcul.

Nous pouvons montrer, en analysant chaque règle de réduction de $\lambda\phi$, qu'une réduction en $\lambda\phi$-calcul peut être simulée dans le ρ_{g}-calcul :

Théorème 17 (Complétude) *Etant donnés deux $\lambda\phi$-termes t_1 et t_2 et une réduction $t_1 \mapsto\!\!\!\!\rightarrow_{\lambda\phi} t_2$ dans le λ-calcul cyclique, il existe une réduction correspondante $[\![t_1]\!] \mapsto\!\!\!\!\rightarrow_{\mathsf{g}} [\![t_2]\!]$ dans le ρ_{g}-calcul.*

5.4.3 Réécriture de graphes

Plusieurs présentations ont été proposées pour les *systèmes de réécriture de termes-graphes* (*Term Graph Rewrite Systems* - TGRS) (voir [SPvE93]). Nous nous focalisons ici sur une présentation équationnelle dans le style de [AK96], qui est plus proche de l'approche utilisée dans le ρ_{g}-calcul. Etant donné un ensemble de variables $\mathcal{X}$ et une signature de premier ordre $\mathcal{F}$ avec des symboles d'arité fixe, un terme-graphe sur $\mathcal{X}$ et $\mathcal{F}$ est un système d'équations de la forme

$$G = \{x_1 \mid x_1 = t_1, \ldots, x_n = t_n\}$$

où $t_1, \ldots, t_n$ sont des termes de premier ordre sur $\mathcal{X}$ et $\mathcal{F}$ et les variables de récursion x_i sont mutuellement distinctes. La variable x_1 à gauche représente la racine du terme-graphe. On appelle *corps* du terme-graphe la liste d'équations et on le dénote par E_G, ou simplement E, quand le graphe G est clair dans le contexte. La liste vide est dénotée par ϵ. Chacune des variables $x_1, \ldots, x_n$ est liée dans le terme-graphe par l'équation récursive associée. Les autres variables apparaissant dans le terme-graphe G sont libres et l'ensemble de variables libres est noté $\mathcal{FV}(G)$. Un terme-graphe sans variables libres est clos. On dénote l'ensemble de variables apparaissant dans G par $\mathcal{V}ar(G)$. Deux graphes α-équivalents *c.-à-d.* deux graphes qui diffèrent seulement par les noms des variables liées, sont considérés égaux. Des cycles peuvent apparaître dans le système d'équations et les cycles dégénérés, *c.-à-d.* les équations de la forme $x = x$, sont remplacés par $x = \bullet$ (trou noir).

Un terme-graphe est en *forme aplatie* si toutes ses équations récursives sont de la forme $x = f(x_1, \ldots, x_n)$, où les variables $x, x_1, \ldots, x_n$ ne sont pas nécessairement distinctes. Par la suite nous considérerons seulement des termes-graphes en forme aplatie et sans équations inutiles (« garbage ») qui sont enlevées automatiquement pendant la réécriture. Un terme-graphe en forme aplatie peut être facilement interprété et représenté comme un graphe avec l'ensemble de variables comme noeuds. Dans les exemples, nous utiliserons souvent la représentation graphique pour donner l'intuition.

La réécriture est réalisée au moyen de règles de réécriture de termes-graphes. Une *règle de réécriture de termes-graphes* est une paire de termes-graphes (L, R) telle que L et R ont la même racine, L n'est pas une simple variable et $\mathcal{FV}(R) \subseteq \mathcal{FV}(L)$. On dit qu'une règle de réécriture est *linéaire à gauche* si L est un arbre. Par la suite nous nous limiterons aux règles de réécriture linéaires à gauche.

Définition 24 (Substitution de variables) *Une substitution* $\sigma = \{y_1/x_1, \ldots, y_n/x_n\}$ *est une fonction des variables vers les termes-graphes. Son application à un terme-graphe* G*, noté* $\sigma(G)$*, est définie par :*

$$\begin{array}{rcl} \sigma(x) & \triangleq & \begin{cases} y_i & \text{if } x = x_i \in \{x_1, \ldots, x_n\} \\ x & \text{sinon} \end{cases} \\ \sigma(f(G_1, \ldots, G_n)) & \triangleq & f(\sigma(G_1), \ldots, \sigma(G_n)) \\ \sigma(\{x_1 \mid x_1 = G_1, \ldots, x_n = G_n\}) & \triangleq & \{\sigma(x_1) \mid \sigma(x_1) = \sigma(G_1), \ldots, \sigma(x_n) = \sigma(G_n)\} \end{array}$$

Une règle de réécriture peut être appliquée à un terme-graphe G s'il existe un filtre entre son membre gauche et le graphe. Formellement, un *homomorphisme (filtre)* d'un terme-graphe L à un terme-graphe G est une substitution σ telle que $\sigma(L) \subseteq G$ où l'inclusion signifie que toutes les équations récursives de $\sigma(L)$ sont présentes également dans G. Notez que dans le cas des termes-graphes en forme aplatie, l'homomorphisme σ est simplement une substitution de variables.

Un *radical* dans un terme-graphe G est une paire $((L, R), \sigma)$ où (L, R) est une règle et σ est un homomorphisme du membre gauche L de la règle à G. Si x est la racine de L, on appelle $\sigma(x)$ la tête du radical.

Définition 25 (Chemin, position) *Un* chemin *dans un graphe clos G est une séquence de symboles de fonction intercalés avec des entiers $p = f_1 i_1 f_2 \dots i_{n-1} f_n$ telle que f_{j+1} est le i_j-ème argument de f_j, pour tout $j = 0 \dots n$. La séquence d'entiers $i_1 \dots i_{n-1}$ s'appelle la* position *du noeud étiqueté f_n et par un petit abus de notation elle peut également être notée p.*

Nous utilisons la notation $G_{|p}$ pour le sous-graphe de G à la position p dans G, tandis que la notation $G_{\lceil G' \rceil_p}$ indique que G contient un terme-graphe G' à la position p. Dans la même situation, si z est la racine de G' et $z = t$ est l'équation correspondante alors on écrit également $G_{\lceil z=t \rceil_p}$. Avec la notation $G_{[G']_p}$ on dénote le terme-graphe G où le sous-graphe $G_{|p}$, ou plus précisément l'équation définissant la racine de $G_{|p}$, a été remplacé par G'.

Les notions de chemin et de position sont utilisées pour définir un pas de réécriture. Soit $((L, R), \sigma)$ un radical apparaissant dans G à la position p. Un *pas de réécriture* consiste à enlever l'équation spécifiée par la tête du radical et à la remplacer par le corps de $\sigma(R)$ avec ses variables liées remplacées par des variables fraîches. En utilisant la notation introduite précédemment on écrit $G_{\lceil \sigma(x)=t \rceil_p} \rightarrowtriangle G_{[\sigma(E_R)]_p}$.

Les termes-graphes et l'ensemble de règles peuvent être traduits au niveau d'objet du ρ_{g}-calcul, *c.-à-d.* peuvent être traduits en des ρ_{g}-graphes. En utilisant le cadre équationnel, l'ensemble des termes-graphes d'un TGRS est un sous-ensemble strict de l'ensemble des termes du ρ_{g}-calcul, modulo quelques conventions syntactiques évidentes. En particulier, par abus de notation, on confond parfois les deux notations $\{x \mid E\}$ et $x\ [E]$. Une règle de réécriture (L, R) est traduite en un ρ_{g}-graphe $L \rightarrowtriangle R$. On rappelle que nous considérons seulement des règles de réécriture de termes-graphes linéaire à gauche.

Nous avons prouvé [BBCK06] que le filtrage dans le ρ_{g}-calcul est correct par rapport à la notion d'homomorphisme de termes-graphes. Plus précisément, nous avons montré que s'il existe un homomorphisme, *c.-à-d.* un renommage de variables, entre deux termes-graphes alors, dans le ρ_{g}-calcul on obtient ce renommage de variables (sous la forme d'un ensemble d'équations récursives) comme résultat de l'évaluation du problème de filtrage généré à partir des deux graphes. En d'autres termes, ceci signifie que si une règle de réécriture peut être appliquée à un terme-graphe, l'application est encore possible au niveau du ρ_{g}-calcul.

En utilisant ce résultat, nous avons étudié la relation entre les dérivations d'un système de réécriture de termes-graphes et les réductions dans le ρ_{g}-calcul. Etant donné un terme-graphe G et une dérivation de G par rapport à un ensemble de règles de réécriture de termes-graphes, nous avons prouvé

qu'il est possible de construire un $\rho_{\mathtt{g}}$-graphe qui est réduit dans le $\rho_{\mathtt{g}}$-calcul à un terme correspondant au terme-graphe final de la réduction originale.

Théorème 18 *Etant donné un terme-graphe G et une règle de réécriture linéaire à gauche (L, R) de racine z tels que $G_{\lceil \sigma(z)=t \rceil_p} \twoheadrightarrow G_{\lceil \sigma(R) \rceil_p} = G_1$, on peut construire le $\rho_{\mathtt{g}}$-graphe H tel que pour tout terme G' (dont la forme aplatie est) égal modulo le renommage des variables à G il existe une réduction $(H\ G') \mapsto\!\!\!\!\!\rightarrow_{\not\rho\mathtt{g}} G'_1$ telle que (la forme aplatie de) G'_1 est égal modulo le renommage des variables à G_1.*

Le $\rho_{\mathtt{g}}$-terme final que nous obtenons n'est pas exactement identique au terme-graphe résultant de la réduction dans le TGRS et ceci est dû à quelques étapes de « unsharing » qui peuvent avoir lieu dans la réduction. Généralement, on a un homomorphisme entre les deux graphes et ceci correspond au fait que, en présence de cycles, le $\rho_{\mathtt{g}}$-terme est potentiellement plus « unraveled » que le terme-graphe G'.

Exemple 24 (Addition) *On considère la règle de réécriture (L, R) avec $L = \{x_1 \mid x_1 = add(x_2, y_2),\ x_2 = s(y_1)\}$ et $R = \{x_1 \mid x_1 = s(x_2),\ x_2 = add(y_1, y_2)\}$ décrivant l'addition des nombres naturels. On applique cette règle au terme-graphe $G = \{z \mid z = s(z_0), z_0 = add(z_1, z_2), z_1 = s(z_2), z_2 = 0\}$ en utilisant le renommage de variables $\sigma = \{x_1/z_0, x_2/z_1, y_1/z_2, y_2/z_2\}$ et on obtient $G' = \{z \mid z = s(z_0), z_0 = s(z'_1), z'_1 = add(z_2, z_2), z_2 = 0\}$. Pour une représentation graphique voir la figure 5.7.*

La réduction correspondante dans le $\rho_{\mathtt{g}}$-calcul est obtenue comme suit. Tout d'abord, puisque la règle n'est pas appliquée à la position de tête de G, on a besoin de définir le $\rho_{\mathtt{g}}$-terme $H = y \twoheadrightarrow s(x) \twoheadrightarrow s(y\ x)$ qui propage l'application de la règle de réécriture à la position d'application correspondante c.-à-d. au niveau du symbole s. On considère le ρ-terme correspondant $G = z\ [z = s(z_0) \wedge z_0 = add(z_1, z_2) \wedge z_1 = s(z_2) \wedge z_2 = 0] = z\ [E_G]$ et on obtient la réduction

$$
\begin{array}{ll}
 & (y \twoheadrightarrow s(x) \twoheadrightarrow s(y\ x))\ (L \twoheadrightarrow R)\ G \\
\mapsto\!\!\!\!\!\rightarrow_{\not\rho\mathtt{g}} & (s(x) \twoheadrightarrow s((L \twoheadrightarrow R)\ x))\ G \\
\mapsto_{\rho} & s((L \twoheadrightarrow R)\ x)\ [s(x) \ll G] \\
\mapsto_{p} & s((L \twoheadrightarrow R)\ x)\ [s(x) \ll z \wedge E_G] \\
\mapsto\!\!\!\!\!\rightarrow_{\not\rho\mathtt{g}} & s((L \twoheadrightarrow R)\ z_0)\ [E_G] \\
\mapsto_{\rho} & s(R\ [L \ll z_0])\ [E_G] \\
\mapsto\!\!\!\!\!\rightarrow_{\not\rho\mathtt{g}} & s(R\ [y_1 = z_2 \wedge y_2 = z_2])\ [E_G] \\
= & s(x_1\ [x_1 = s(x_2) \wedge x_2 = add(y_1, y_2)]\ [y_1 = z_2 \wedge y_2 = z_2])\ [E_G] \\
\mapsto\!\!\!\!\!\rightarrow_{\not\rho\mathtt{g}} & s(x_1\ [x_1 = s(x_2) \wedge x_2 = add(z_1, z_2)])\ [E_G] = G''
\end{array}
$$

Une soi-disant forme canonique $\overline{G''}$ de G'' est alors obtenue en enlevant les équations récursives inutiles dans E_G et en fusionnant les ensembles de contraintes. On obtient $\overline{G''} = x\ [x = s(x_1) \wedge x_1 = s(x_2) \wedge x_2 = add(z_1, z_2) \wedge$

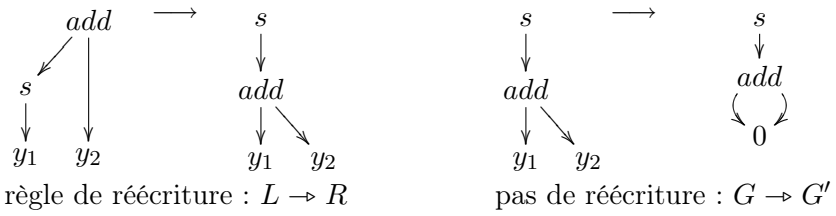


FIGURE 5.7 – *Exemple de réécritures dans un TGRS*

$z_0 = 0]$. *Le graphe* $\overline{G''}$ *est homomorphe (dans ce cas-ci même égal modulo le renommage de variables) au terme-graphe* G'.

5.5 Conclusions

Nous avons introduit le $\rho_{\mathtt{g}}$-calcul, une extension du ρ-calcul qui utilise non seulement les contraintes de filtrage du $\rho^{\circ}_{\mathtt{x}}$-calcul mais également des contraintes d'unification. Nous obtenons ainsi un calcul qui bénéficie des avantages du $\rho^{\circ}_{\mathtt{x}}$-calcul et qui permet également la manipulation de graphes avec du partage et des cycles (qui peuvent être utilisés pour représenter les structures de données infinies régulières).

Nous avons prouvé que le $\rho_{\mathtt{g}}$-calcul est confluent si des conditions de linéarités appropriées sont imposées aux motifs ; ces conditions sont en fait celles imposées au calcul de base mais étendues pour prendre en compte les nouvelles constructions du $\rho_{\mathtt{g}}$-calcul.

Le $\rho_{\mathtt{g}}$-calcul est un calcul très expressif, capable de simuler le λ-calcul cyclique aussi bien que la réécriture de termes-graphes. La différence principale entre le $\rho_{\mathtt{g}}$-calcul et les TGRS est la même qu'entre le ρ-calcul et les systèmes de réécriture de termes : dans les systèmes de réécriture (de termes ou de graphes) la stratégie d'application des règle de réécriture est soit implicite soit exprimée au niveau méta du formalisme tandis que l'application (des règles) est définie au niveau objet du calcul de réécriture. Nous avons montré que toute réduction dans les TGRS correspond à un $\rho_{\mathtt{g}}$-terme mais il serait certainement intéressant de définir d'une manière générique, au niveau objet du $\rho_{\mathtt{g}}$-calcul, des stratégies d'itération et de parcours de $\rho_{\mathtt{g}}$-graphes permettant la simulation de la réécriture de graphes guidée par une stratégie de réduction donnée. Nous conjecturons que les approches présentées dans le chapitre 2 pour l'encodage des systèmes de réécriture de termes utilisant une ρ-structure contenant les règles de réécriture correspondantes encapsulées dans un itérateur qui permet l'application répétitive des règles peut être adaptée et généralisée pour manipuler des termes-graphes et simuler ainsi des réductions de termes-graphes.

Nous nous sommes focalisés ici, comme pour le $\rho^{\circ}_{\mathtt{x}}$-calcul, sur des pro-

blèmes de filtrage syntaxique mais cette fois-ci sur des termes-graphes. Nous avons utilisé une règle de décomposition qui est déclenchée seulement quand le membre gauche d'un problème de filtrage est un terme pur mais il est évident que d'autres approches alternatives peuvent être envisagées. Tout d'abord, au lieu d'appliquer les contraintes dans le membre gauche, on pourrait les distribuer dans les sous-problèmes obtenus mais dans la perspective d'une implantation concrète cette approche pourrait être relativement coûteuses en espace et en temps. Alternativement, on pourrait avoir une règle symétrique à la règle propagate qui propage les contraintes présentes dans le membre gauche d'un problème de filtrage. Dans ce dernier cas, nous devrions traiter certains problèmes de liaison de variables qui pourraient apparaître à cause des variables liées (localement) dans le membre gauche et qui sont propagées dans la contrainte globale menant ainsi à des captures de variables. Ce type de problème peut être résolu avec une approche similaire à celle déjà utilisée dans d'autres calculs à motifs [BCKL03, JK06] et qui consiste à associer explicitement à chaque opérateur d'abstraction et de filtrage la liste de variables liées par l'opérateur respectif ; cette approche sera présentée dans le chapitre suivant pour un calcul sans cycles et sans partage.

Contexte :

Le λ-calcul est le modèle théorique utilisé comme support pour les langages de programmation fonctionnelle et, de manière relativement surprenante, bien que dans la pratique les langages fonctionnelles [All78, Pey03, LDG+04, Pic07] reposent fortement sur le filtrage, le λ-calcul utilise seulement un filtrage trivial. La conception des langages de programmation fonctionnelle [Pey87] a conduit à l'émergence de plusieurs formalismes qui intègrent le filtrage sur les motifs dans le λ-calcul. On peut ainsi mentionner le λ-calcul avec motifs [vO90], le calcul de réécriture [CK01], le calcul à motifs purs [JK06] et le λ-calcul avec constructeurs [AMR06].

La différence principale entre ces calculs à motifs se situe au niveau du traitement du filtrage et leur confluence est perdue quand aucune restriction n'est imposée aux algorithmes de filtrage correspondants. Les conditions imposées pour récupérer la confluence concernent généralement la forme des motifs qui peut être restreinte syntaxiquement ou sémantiquement ainsi que la forme des termes filtrés qui peuvent être limités à certaines valeurs spécifiques. Les méthodes utilisées pour montrer la confluence sont différentes mais elles partagent généralement la même structure et les propriétés intermédiaires qu'on doit établir au cours de la preuve sont généralement les mêmes. Nous avons ainsi essayé de donner une réponse positive à plusieurs questions.

Questions :

- Quel formalisme pour généraliser les différents calculs à motifs ?
- Quelles sont les conditions nécessaires pour garantir la confluence ?
- Ces conditions sont-elles satisfaites par les différents calculs à motifs ?

Contributions :

Nous avons proposé un calcul à motifs dynamiques d'ordre supérieur, c'est-à-dire, disposant des motifs qui peuvent être instanciés et réduits, et nous avons prouvé que les divers calculs à motifs de la littérature peuvent être vus comme des instances de ce formalisme. Le lien entre chacun de ces calculs et le formalisme présenté est réalisé par la fonction de filtrage qui est générique dans notre formalisme et instanciée d'une manière spécifique pour chaque calcul.

Nous avons proposé une preuve de confluence générique qui isole les conditions suffisantes en axiomatisant la manière dont le filtrage est réalisé. Intuitivement, les conditions suffisantes pour garantir la confluence imposent la stabilité par substitution et par réduction de la fonction de filtrage. Cette approche ne fournit pas des preuves de confluence gratuites mais elle établit une méthodologie de preuve en isolant les points clés qui rendent les calculs confluents. Elle peut également mettre en évidence certains algorithmes de filtrage qui semblent consistants mais qui peuvent mener à des réductions non confluentes dans le calcul correspondant.

6

Confluence de λ-calculs à motifs

Nous avons déjà discuté de l'importance du filtrage sur des motifs pour la modélisation du traitement de l'information et particulièrement pour les langages de programmation et nous avons déjà évoqué différents formalismes avec motifs qui ont émergé dès la fin des années 90 [PJ87, vO90, CK01, Kah03, Jay04, AMR06]. Les différences entre ces calculs se situent essentiellement au niveau de la façon dont les motifs sont définis et de la façon dont les abstractions à base de motifs sont appliquées.

Ainsi, les motifs peuvent être de simples variables comme dans le λ-calcul, des termes algébriques comme dans le ρ-calcul algébrique [CLW03], des motifs (statiques) spéciaux qui satisfont certaines conditions (sémantiques ou syntaxiques) comme dans λ-calcul avec motifs [KvOdV08] ou des motifs dynamiques qui peuvent être instanciés et potentiellement réduits comme dans le calcul à motifs purs [JK09] et certaines versions du ρ-calcul [BCKL03]. La théorie de filtrage sous-jacente dépend fortement de la forme des motifs et peut être syntaxique, équationnelle ou plus élaborée.

La confluence de ces formalismes est généralement perdue quand les abstractions sont réalisées sur des motifs à la place de simples variables et aucune restriction n'est imposée sur la réduction correspondante. Plusieurs approches sont alors utilisées pour récupérer la confluence. Une de ces techniques consiste à limiter syntaxiquement l'ensemble de motifs et montrer ensuite que la relation de réduction est confluente pour le sous-ensemble choisi. Cette approche est utilisée, par exemple, dans le λ-calcul avec motifs et dans le ρ-calcul (avec des motifs algébriques). La deuxième technique est basée sur une restriction de la relation de réduction initiale (c'est-à-dire, une stratégie) pour garantir que le calcul est confluent sur l'ensemble complet de termes. Cette technique est utilisée, par exemple, dans le calcul à motifs purs où l'algorithme de filtrage est une fonction partielle (tandis que n'importe quel terme est un motif).

On peut néanmoins noter que dans la pratique les méthodes de preuve partagent la même structure et que chaque variation sur la façon dont les

abstractions (sur les motifs) sont appliquées nécessite une preuve différente de confluence. Il y a ainsi un besoin pour une approche plus abstraite et plus modulaire dans l'esprit de [Mel02, Kes00]. Une manière possible d'avoir une approche unifiée pour prouver la confluence est l'application des résultats généraux et puissants sur la confluence des systèmes de réécriture d'ordre supérieur [KvOvR93, MN98, Ter03]. Bien que ces résultats ont été déjà appliqués pour certains calculs à motifs [BK07], l'encodage semble plutôt complexe pour certains calculs et en particulier pour le formalisme général proposé ici. Par ailleurs, il serait intéressant d'avoir un cadre où l'expressivité et les propriétés (de confluence) des différents calculs à motif peuvent être comparées.

Nous montrons dans ce chapitre que tous les calculs à motifs peuvent être exprimés dans un formalisme général paramétrés par une fonction qui définit l'algorithme de filtrage sous-jacent et ainsi la manière dont les abstractions sont appliquées. Cette fonction peut être instanciée (implantée) par un algorithme de filtrage unitaire comme dans [CK01, JK06] mais également par un algorithme de filtrage sur les anti-motifs [KKM07] ou il peut être encore plus général [LHL07]. Nous proposons une preuve générique de confluence où les abstractions sont appliquées de manière axiomatisée. Intuitivement, les conditions suffisantes pour assurer la confluence garantissent que la fonction (de filtrage) est stable par substitution et par réduction.

Nous appliquons notre approche à plusieurs calculs à motifs classiques, à savoir le λ-calcul avec motifs, le calcul de réécriture, le λ-calcul avec constructeurs et une version simplifiée du calcul à motifs purs. Pour tous ces calculs, nous donnons les encodages dans le cadre général et nous obtenons des preuves de confluence. Cette approche ne fournit pas des preuves de confluence gratuites mais elle établit une méthodologie de preuve et isole les points clés qui rendent les calculs confluents. Elle peut également mettre en évidence certains algorithmes de filtrage qui bien que naturels à la première vue peuvent mener à des réductions non confluentes dans le calcul correspondant.

Les travaux décrits dans ce chapitre ont été réalisés en collaboration avec Germain Faure.

6.1 Un calcul avec motifs dynamiques

Dans cette section, nous définissons d'abord la syntaxe et la sémantique opérationnelle du λ-calcul à motifs dynamiques de base. Nous donnons ensuite la définition générale du λ-calcul à motifs dynamiques.

6.1.1 Syntaxe

La syntaxe du λ-calcul à motifs dynamiques de base est définie dans la figure 6.1. On utilise les conventions syntaxiques des chapitres précédents. En particulier, les variables sont notées $x, y, z, \ldots$, les constantes sont notées

$$
\begin{array}{rcll}
\mathcal{T}, \mathcal{P} & ::= & \mathcal{X} & \text{(Variables)} \\
& | & \mathcal{K} & \text{(Constantes)} \\
& | & \mathcal{T} \twoheadrightarrow_\theta \mathcal{T} & \text{(Abstractions)} \\
& | & \mathcal{T}\ \mathcal{T} & \text{(Applications)}
\end{array}
$$

FIGURE 6.1 – *Syntaxe de λ-calcul à motifs dynamiques (de base)*

$a, b, c, d, e, f, \ldots$ et on utilise parfois une notation algébrique $f(A_1, \ldots, A_n)$ pour le terme $((f\ A_1) \ldots)\ A_n)$. Dans une abstraction $A \twoheadrightarrow_\theta B$, θ est un sous-ensemble de l'ensemble de variables de A et représente les variables liées par l'abstraction. Cet ensemble est souvent omis quand c'est exactement l'ensemble de variables libres du motif. Les notions de variables libres et liées sont donc légèrement différentes par rapport aux approches classiques :

Définition 26 (Variables libres et liées) *Les ensembles des variables libres et liées d'un terme A, notés $\mathcal{FV}(A)$ et $\mathcal{BV}(A)$, sont définis inductivement par :*

$$
\begin{array}{rclcrcl}
\mathcal{FV}(c) & \triangleq & \emptyset & & \mathcal{BV}(c) & \triangleq & \emptyset \\
\mathcal{FV}(x) & \triangleq & \{x\} & & \mathcal{BV}(x) & \triangleq & \emptyset \\
\mathcal{FV}(A\ B) & \triangleq & \mathcal{FV}(A) \cup \mathcal{FV}(B) & & \mathcal{BV}(A\ B) & \triangleq & \mathcal{BV}(A) \cup \mathcal{BV}(B) \\
\mathcal{FV}(A \wr B) & \triangleq & \mathcal{FV}(A) \cup \mathcal{FV}(B) & & \mathcal{BV}(A \wr B) & \triangleq & \mathcal{BV}(A) \cup \mathcal{BV}(B)
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{FV}(A \twoheadrightarrow_\theta B) & \triangleq & (\mathcal{FV}(A) \cup \mathcal{FV}(B)) \backslash \theta \\
\mathcal{BV}(A \twoheadrightarrow_\theta B) & \triangleq & \mathcal{BV}(A) \cup \mathcal{BV}(B) \cup \theta
\end{array}
$$

On abstrait ainsi non seulement sur des variables mais sur des termes généraux et l'ensemble de variables liées par une abstraction n'est pas nécessairement le même que l'ensemble de variables (libres) du motif correspondant. On dit ainsi que les motifs sont dynamiques puisqu'ils peuvent être instanciés et potentiellement réduits.

Comme d'habitude, on dit qu'un terme est clos quand l'ensemble de ses variables libres est vide et on travaille modulo l'α-conversion et modulo la *convention d'hygiène* de Barendregt. Les substitutions et l'application de substitutions sont définies comme dans le chapitre 2.

6.1.2 Sémantique

La sémantique opérationnelle du λ-calcul à motifs dynamiques de base est donnée par une seule règle de réduction qui définit la manière dont les abstractions sur des motifs sont appliquées. Cette règle, donnée dans la figure 6.2, est paramétrée par une fonction partielle, notée $\mathcal{S}ol(A \prec\!\!\!\prec_\theta B)$, qui prend comme paramètres deux termes A et B et un ensemble de variables θ et renvoie une substitution. Pour simplifier la présentation nous

$$(A \twoheadrightarrow_\theta B)\ C \qquad \to_\rho \qquad B\sigma \quad \textbf{avec } \sigma = \mathcal{S}ol(A \lessdot_\theta C)$$

FIGURE 6.2 – *Sémantique du λ-calcul à motifs dynamiques de base*

nous focalisons donc sur une version du calcul avec un filtrage unitaire et nous considérons que la fonction $\mathcal{S}ol$ retourne une substitution et non un ensemble (un singleton dans le cas unitaire) de substitutions comme dans le ρ-calcul.

Différentes instances du λ-calcul à motifs dynamiques de base sont obtenus en donnant des définitions concrètes pour la fonction $\mathcal{S}ol$. Par exemple, le λ-calcul peut être vu comme le λ-calcul à motifs dynamiques de base tel que

$$\sigma = \mathcal{S}ol(A \lessdot_\theta C) \text{ ssi } A \text{ est une variable } x,\ \theta = \{x\} \text{ et } A\sigma \equiv C$$

On peut également considérer une fonction qui calcule les solutions du problème de filtrage syntaxique correspondant en modifiant la définition 5 pour prendre en compte les variables liées explicitement dans le λ-calcul à motifs dynamiques de base.

Définition 27 (Filtrage syntaxique) *L'ensemble des solutions d'un problème de filtrage $A \lessdot_\theta B$ est obtenu en normalisant la contrainte de filtrage $A \ll^? B$ par rapport aux règles de la définition 5 et selon le résultat obtenu $\mathcal{S}ol(A \lessdot_\theta B)$ renvoie*

- *$\{A_i/x_i\}_{i\in I}$ si $\mathcal{FV}(A) = \theta$ et le résultat est $\bigwedge_{i\in I\neq\emptyset}(x_i \ll^? A_i)$ avec $A_i = A_j$ si $x_i = x_j$;*
- *`id` si le résultat est la conjonction vide ;*
- *rien (c.a.d. n'est pas définie) dans les autres cas.*

Exemple 25 (Branchements) *On considère un calcul à motifs avec une construction de branchement notée « | » (correspondant à l'opérateur `match` des langages de programmation fonctionnelle). Elle peut être codée dans le λ-calcul à motifs dynamiques de base comme suit :*

$$(A_1 \mapsto B_1 | \ldots | A_n \mapsto B_n)\ C \triangleq \big((A_1 \hookrightarrow B_1 / \ldots / A_n \hookrightarrow B_n) \twoheadrightarrow_x x\big)\ C$$

où x est une variable fraîche, les symboles $\hookrightarrow$ et / sont des constantes du λ-calcul à motifs dynamiques de base (notation infixée) et la fonction $\mathcal{S}ol$ peut être définie par

$$\mathcal{S}ol\big((A_1 \hookrightarrow B_1 / \ldots / A_n \hookrightarrow B_n) \lessdot_x A_i\sigma\big) \triangleq \{B_i\sigma/x\}$$

Certains calculs à motifs disposent de fonctionnalités additionnelles et ne peuvent pas être exprimés comme des instances du λ-calcul à motifs dynamiques de base. Par exemple, le calcul à motifs purs [JK06] et quelques versions du calcul de réécriture (voir la section 2.5) réduisent l'application d'une abstraction à un terme spécial quand le problème correspondant de filtrage n'a pas et n'aura jamais une solution. Dans le calcul de réécriture on a également l'opérateur de structure et la règle correspondante de distribution par rapport aux applications qui ne sont pas directement exprimables en λ-calcul à motifs dynamiques de base.

Nous définissons ainsi le λ-calcul à motifs dynamiques comme le λ-calcul à motifs dynamiques de base étendu par un ensemble de règles d'évaluation. De la même manière que nous avons procédé pour $\mathcal{S}ol$, cet ensemble, noté ξ, n'est pas précisé à priori et peut être considéré comme un paramètre du calcul. Il peut inclure, par exemple, des règles pour réduire des applications d'abstractions particulières à une constante spéciale représentant un échec définitif de filtrage ou des règles supplémentaires décrivant la distributivité de certains symboles (comme l'opérateur de structure du ρ-calcul) par rapport aux applications.

Par la suite, $\mapsto_\rho$ dénote la fermeture compatible de la relation induite par la règle ρ et $\mapsto\!\!\!\!\!\rightarrow_\rho$ dénote la fermeture transitive de $\mapsto_\rho$. On dénote par $\mapsto_{\rho\cup\xi}$ la fermeture compatible de la relation induite par les règles ρ et ξ. $\mapsto\!\!\!\!\!\rightarrow_{\rho\cup\xi}$ dénote la fermeture transitive de $\mapsto_{\rho\cup\xi}$.

Dans les sections suivantes nous présentons les résultats de confluence du λ-calcul à motifs dynamiques et montrons comment on peut les appliquer pour différents calculs à motifs. Les versions courtes des preuves correspondantes sont disponible dans [CF07] et des versions plus détaillées dans [Fau07].

6.2 Résultats de confluence

Nous avons déjà vu que le ρ-calcul n'est pas confluent quand aucune restriction n'est imposée à la fonction $\mathcal{S}ol$ et ceci est le cas pour le λ-calcul à motifs dynamiques également. Les exemples donnés dans la section 2.4 sont directement applicables au λ-calcul à motifs dynamiques et donc, on peux obtenir des réductions non joignables si on utilise un filtrage naïf impliquant des termes d'ordre supérieur et en particulier, quand on permet la décomposition des applications contenant des variables libres actives ou quand on manipule des motifs non linéaires. Néanmoins, la confluence est rétablie pour certaines définitions spécifiques de $\mathcal{S}ol$ comme, par exemple, celle utilisée dans la section 6.1.2 pour définir le λ-calcul comme un λ-calcul à motifs dynamiques de base.

Dans cette section nous donnons des conditions suffisantes qui permettent de garantir la confluence du λ-calcul à motifs dynamiques de base. Intuitive-

ment, les hypothèses présentées dans la section 6.2.1 et utilisées pour prouver la confluence garantissent la cohérence entre la fonction $\mathcal{S}ol$ et la relation sous-jacente au calcul. Les résultats obtenus peuvent être ensuite généralisés pour un λ-calcul à motifs dynamiques avec un ensemble ξ de règles étendu qui satisfait certaines conditions classiques de cohérence.

La preuve utilise la méthode introduite par Tait et Martin-Löf [Bar84] qui consiste à définir une réduction parallèle qui, intuitivement, peut réduire tous les radicaux présents initialement dans le terme et qui est fortement confluente (même si la réduction en un pas ne l'est pas) sous certaines hypothèses.

Définition 28 (Réduction parallèle) *La réduction parallèle est définie inductivement sur l'ensemble de termes par :*

$$\frac{}{A \ \Rightarrow_\rho \ A} \qquad \frac{A \ \Rightarrow_\rho \ A' \qquad B \ \Rightarrow_\rho \ B'}{A \ B \ \Rightarrow_\rho \ A' \ B'} \qquad \frac{A \ \Rightarrow_\rho \ A' \qquad B \ \Rightarrow_\rho \ B'}{(A \twoheadrightarrow_\theta B) \ \Rightarrow_\rho \ (A' \twoheadrightarrow_\theta B')}$$

$$\frac{A \ \Rightarrow_\rho \ A' \qquad B \ \Rightarrow_\rho \ B' \qquad C \ \Rightarrow_\rho \ C'}{(A \twoheadrightarrow_\theta B) \ C \ \Rightarrow_\rho \ B'\sigma'} \text{ si } \sigma' \in \mathcal{S}ol(A' \prec\!\!\!\prec_\theta C')$$

Il faut noter également que la définition de la réduction parallèle donnée ne coïncide pas avec la notion classique de développements. Par exemple, si nous utilisons une fonction $\mathcal{S}ol$ qui calcule la substitution solution du filtrage entre ses deux arguments (par exemple, celle introduite dans la définition 27) alors on a

$$\frac{f \ x \ \Rightarrow_\rho \ f \ x \qquad x \ \Rightarrow_\rho \ x \qquad (y \twoheadrightarrow f \ y) \ a \ \Rightarrow_\rho \ f \ a}{(f \ x \twoheadrightarrow x) \ ((y \twoheadrightarrow f \ y) \ a) \ \Rightarrow_\rho \ a}$$

La substitution $\{a/x\}$ est solution du filtrage syntaxique entre les termes $f \ x$ et $f \ a$ et donc, même si le terme initial ne contient aucun radical de tête (parce que l'argument $(y \twoheadrightarrow f \ y) \ a$ doit être réduit avant) il peut néanmoins être réduit en utilisant la réduction parallèle.

On étend la définition de la réduction parallèle aux substitutions ayant le même domaine et on écrit $\sigma \ \Rightarrow_\rho \ \sigma'$ dès que pour tout x dans le domaine de σ, on a $x\sigma \ \Rightarrow_\rho \ x\sigma'$.

6.2.1 Conditions suffisantes à la confluence

Comme nous l'avons déjà dit, les conditions suffisantes pour assurer la confluence garantissent que la fonction (de filtrage) $\mathcal{S}ol$ est stable par substitution et par réduction. Dans le reste de cette section nous discutons ces conditions et nous énonçons les résultats de confluence.

$\forall A, C, A', C'$

$\mathcal{H}_0$:

$$\mathcal{S}ol(A \prec\!\!\!\prec_\theta C) = \sigma \quad \text{implique} \quad \begin{cases} \mathcal{D}om(\sigma) = \theta \\ \mathcal{R}an(\sigma) \subseteq \mathcal{FV}(C) \cup (\mathcal{FV}(A) \backslash \theta) \end{cases}$$

$\mathcal{H}_1$:

$$\mathcal{S}ol(A \prec\!\!\!\prec_\theta C) = \sigma \quad \text{implique} \quad \begin{cases} \forall \tau \text{ s.t. } \mathcal{V}ar(\tau) \cap \theta = \emptyset, \text{ on a} \\ \mathcal{S}ol(A\tau \prec\!\!\!\prec_\theta C\tau) = (\tau \circ \sigma)_{|\theta} \end{cases}$$

$\mathcal{H}_2$:

$$\begin{cases} \mathcal{S}ol(A \prec\!\!\!\prec_\theta C) = \sigma \\ A \Rightarrow_\rho A' \quad C \Rightarrow_\rho C' \end{cases} \quad \text{implique} \quad \begin{cases} \mathcal{S}ol(A' \prec\!\!\!\prec C') = \sigma' \\ \sigma \Rightarrow_\rho \sigma' \end{cases}$$

FIGURE 6.3 – *Conditions pour garantir la confluence du λ-calcul à motifs dynamiques de base*

Préservation des variables libres Tout d'abord, quand on définit un calcul d'ordre supérieur il est naturel de demander que l'ensemble de variables libres soit préservé par réduction (des variables libres peuvent être perdues mais aucune variable libre ne peut apparaître pendant la réduction). Par exemple, les variables libres du terme $(A \rightarrowtriangle_\theta B)\ C$ devraient inclure celles du terme $B\sigma$ avec $\sigma = \mathcal{S}ol(A \prec\!\!\!\prec_\theta C)$. Ainsi, la substitution σ devrait instancier toutes les variables liées (par l'abstraction) dans B, *c.-à-d.*, toutes les variables dans θ. De plus, les variables libres de σ devraient déjà être présentes dans C ou libres dans A. Ces conditions sont imposées par l'hypothèse $\mathcal{H}_0$ dans la figure 6.3.

Si on voit $\mathcal{S}ol$ comme un algorithme de filtrage unitaire, les exemples qui ne vérifient pas $\mathcal{H}_0$ sont souvent des algorithmes particuliers (par exemple, la fonction qui renvoie la substitution $\{y/x\}$ pour tout problème). Quand on considère du filtrage non unitaire (non traité ici), on peut exhiber plusieurs exemples qui ne vérifient pas $\mathcal{H}_0$. Par exemple, les algorithmes qui permettent de résoudre des problèmes de filtrage d'ordre supérieur ou des problèmes de filtrage dans des théories non-régulières (par exemple, tel que $x \times 0 = 0$) ne vérifient pas $\mathcal{H}_0$.

Stabilité par substitution Dans le λ-calcul à motifs dynamiques de base, une abstraction peut être appliquée à un argument contenant des variables. On peut attendre que l'argument soit complètement instancié et seulement ensuite calculer la substitution correspondante (si elle existe) et réduire l'application. D'autre part, on pourrait ne pas vouloir procéder de cette façon

mais effectuer la réduction aussitôt que possible. Néanmoins, le même résultat devrait être obtenu pour les deux stratégies de réduction. Ceci est imposé par l'hypothèse $\mathcal{H}_1$ dans la figure 6.3.

Si on considère que $\mathcal{S}ol$ implante un algorithme naïf de filtrage qui ne prend pas en considération les variables de θ et tel que $\mathcal{S}ol(a \lessdot_\emptyset b)$ n'a aucune solution et $\mathcal{S}ol(x \lessdot_\emptyset y) = \{y/x\}$, alors l'hypothèse $\mathcal{H}_1$ n'est clairement pas vérifiée (il suffit de prendre $\tau = \{a/x, b/x\}$).

Stabilité par réduction Quand on applique une abstraction, l'argument peut également ne pas être entièrement réduit. De nouveau, si $\mathcal{S}ol$ réussit et produit une substitution σ alors les réductions alternatives ne devraient pas mener à un échec définitif pour $\mathcal{S}ol$. De plus, la substitution qui est obtenue doit être dérivable de σ. Ceci est formellement défini dans l'hypothèse $\mathcal{H}_2$.

La fonction $\mathcal{S}ol$ introduite dans la définition 27 ne satisfait pas cette hypothèse. Si on considère $I \triangleq (y \twoheadrightarrow y)$ alors $\mathcal{S}ol(f(x,x) \lessdot_x f(I\ I, I\ I)) = \{I\ I/x\}$ mais $\mathcal{S}ol(f(x,x) \lessdot_x f(I\ I, I))$ n'a aucune solution. De même, cette hypothèse n'est pas satisfaite par un algorithme de filtrage qui permet la décomposition des applications contenant une variable libre *active* (*c.-à-d.* variable en position applicative). Par exemple, on peut avoir $\mathcal{S}ol(x\ a \lessdot_x (y \twoheadrightarrow y)\ a) = \{y \twoheadrightarrow y/x\}$ mais $\mathcal{S}ol(x\ a \lessdot_x a)$ n'a aucune solution pour l'algorithme classique de filtrage (de premier ordre).

La preuve de confluence du λ-calcul à motifs dynamiques sous les hypothèses $\mathcal{H}_0, \mathcal{H}_1, \mathcal{H}_2$ est réalisée en utilisant les techniques standard de réduction parallèle introduites par Tait et Martin-Löf. Nous montrons d'abord que la fermeture réflexive et transitive des réductions parallèle et en un pas sont identiques. Ensuite, nous prouvons que la réduction parallèle est fortement confluente et nous en déduisons la confluence de la réduction en un pas.

Les trois hypothèses données dans la section précédente sont utilisées pour montrer la confluence forte de la réduction parallèle et en particulier le lemme 19. D'autre part, nous pouvons prouver que la fermeture réflexive et transitive de $\Rightarrow_\rho$ est égale à $\mapsto\!\!\!\!\!\rightarrow_\rho$ indépendamment des propriétés de $\mathcal{S}ol$.

Lemme 18 *Les inclusions suivantes sont vérifiées :* $\mapsto_\rho \subseteq \Rightarrow_\rho \subseteq \mapsto\!\!\!\!\!\rightarrow_\rho$.

La preuve de la confluence forte de la réduction parallèle utilise le lemme suivant :

Lemme 19 (Lemme fondamental) *Pour tous les termes C et C' et pour toutes les substitutions σ et σ', tels que $C \Rightarrow_\rho C'$ et $\sigma \Rightarrow_\rho \sigma'$ on a $C\sigma \Rightarrow_\rho C'\sigma'$.*

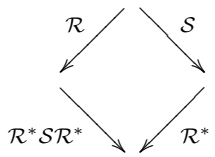
Lemme 20 (Confluence forte de $\Rightarrow_\rho$) *La relation $\Rightarrow_\rho$ est fortement confluente, c.-à-d. pour tous les termes A, B et C, si $A \Rightarrow_\rho B$ et $A \Rightarrow_\rho C$ alors il existe un terme D tel que $B \Rightarrow_\rho D$ et $C \Rightarrow_\rho D$.*

Théorème 19 (Confluence) *Le λ-calcul à motifs dynamiques de base avec $\mathcal{S}ol$ satisfaisant $\mathcal{H}_0$, $\mathcal{H}_1$ et $\mathcal{H}_2$ est confluent.*

Comme nous l'avons déjà dit, la plupart des calculs à motifs étendent la règle β de base (ou son équivalent) par un ensemble de règles. Nous énoncerons ici les conditions qui devraient être imposées afin de prouver la confluence du λ-calcul à motifs dynamiques, des conditions qui s'avèrent être satisfaites par la plupart des différents calculs qui peuvent être exprimés comme instances du λ-calcul à motifs dynamiques.

Nous avons prouvé qu'il est facile à déduire la confluence des extensions du λ-calcul à motifs dynamiques de base avec un ensemble approprié de règles en utilisant le lemme de Yokouchi-Hikita [YH90] (voir également [CHL96]).

Lemme 21 (Yokouchi-Hikita) *Etant données deux relations $\mathcal{R}$ et $\mathcal{S}$ définies sur le même ensemble de termes $\mathcal{T}$ telles que*
- *$\mathcal{R}$ est fortement normalisante et confluente,*
- *$\mathcal{S}$ est fortement confluente,*
- *le diagramme (de Yokouchi-Hikita) suivant est satisfait :*



Alors la relation $\mathcal{R}^*\mathcal{S}\mathcal{R}^*$ est confluente.

Théorème 20 *Le λ-calcul à motifs dynamiques est confluent quand*
- *$\mathcal{S}ol$ satisfait $\mathcal{H}_0, \mathcal{H}_1, \mathcal{H}_2$,*
- *l'ensemble de règles de réduction ξ est fortement normalisant et confluent,*
- *la relation $\Rightarrow_\rho$ et la relation induite par ξ satisfont le diagramme de Yokouchi-Hikita.*

En fait, ce théorème dit que n'importe quel calcul à motifs défini comme un λ-calcul à motifs dynamiques avec une fonction $\mathcal{S}ol$ particulière qui satisfait $\mathcal{H}_0$, $\mathcal{H}_1$ et $\mathcal{H}_2$ est confluent.

6.2.2 Problèmes de confluence en présence de motifs linéaires

Les conditions imposées pour obtenir les différents résultats de la section précédente peuvent paraître relativement restrictives. Ces conditions sont néanmoins respectées par la plupart des calculs à motifs que nous avons explorés et en les relaxant on obtient des contre-exemples classiques pour la confluence.

Par exemple, si le filtrage est réalisé syntaxiquement sur des variables actives alors des réductions non-confluentes peuvent être obtenues dans le lambda-calcul avec motifs [vO90, KvOdV08] ainsi que dans le calcul de réécriture [CLW03]. De la même façon, les motifs non-linéaires mènent à des réductions non-confluentes qui sont des variations du contre-exemple de Klop [Klo80] pour les systèmes d'ordre supérieur utilisant du filtrage non-linéaire.

Quand on utilises des motifs dynamiques contenant des variables qui ne sont pas liées dans l'abstraction les hypothèses de confluence doivent être soigneusement vérifiées. Plus précisément, le comportement des règles non-linéaires peut être encodé en utilisant des motifs *linéaires* et *dynamiques*. Par conséquent, le contre-exemple de Klop peut être encodé dans le calcul correspondant qui est donc non-confluent.

On peut considérer, par exemple, une fonction $\mathcal{S}ol$ telle que pour tous les termes A et B et pour une certaine constante d

$\mathcal{S}ol(A \prec\!\!\!\prec_{\emptyset} A) = \mathtt{id}$
$\mathcal{S}ol(x \prec\!\!\!\prec_{x} A) = \{A/x\}$
$\mathcal{S}ol(d(x,y) \prec\!\!\!\prec_{x,y} d(A,B)) = \{A/x, B/y\}$

Dans ce cas, on peut encoder des motifs non linéaires en utilisant des motifs linéaires dynamiques et en particulier pour la règle $(d\ z\ z) \twoheadrightarrow e$ utilisée dans la section 2.4 on peut utiliser l'encodage

$$(d\ x\ x) \twoheadrightarrow_x e \qquad \triangleq \qquad (d\ x\ y) \twoheadrightarrow_{x,y} (x \twoheadrightarrow_{\emptyset} e)\ y$$

où e est une constante mais peut être remplacée par un terme arbitraire.

Il est ensuite facile d'observer que l'instance correspondante du λ-calcul à motifs dynamiques de base n'est pas confluente. Ceci est montré en utilisant une adaptation de l'encodage du contre-exemple de Klop dans le ρ-calcul donné dans la section 2.4.

Nous utilisons les mêmes deux constantes d et e et on définit le combinateur de point fixe Y comme dans le λ-calcul par (nous omettons l'ensemble θ de variables liées dans une abstraction quand il coïncide avec l'ensemble de variables libres du motif) :

$$\mathsf{Y} \triangleq \Big(y \twoheadrightarrow x \twoheadrightarrow (x\ (y\ y\ x))\Big)\ \Big(y \twoheadrightarrow x \twoheadrightarrow (x\ (y\ y\ x))\Big)$$

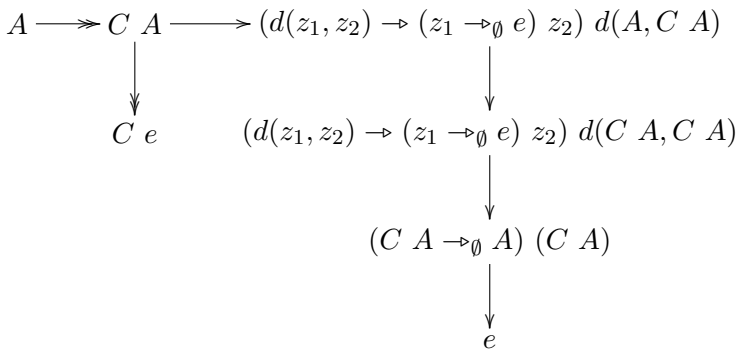
Comme d'habitude, pour tout terme A on a $\mathsf{Y}\ A \mapsto\!\!\!\!\twoheadrightarrow_h A\ (\mathsf{Y}\ A)$. On définit les termes suivant :

$$\begin{aligned} C' &\equiv \big(y \twoheadrightarrow x \twoheadrightarrow (d(z_1,z_2) \twoheadrightarrow (z_1 \twoheadrightarrow_{\emptyset} e)\ z_2)\ d(x, y\ x)\big) \\ C &\equiv \mathsf{Y}\ C' \\ A &\equiv \mathsf{Y}\ C \end{aligned}$$

Si on dénote par $\mapsto\!\!\!\!\rightarrow_h$ les réductions en tête alors nous avons les réductions suivantes :

$$\begin{array}{rcl} C & \mapsto\!\!\!\!\rightarrow_h & \big(y \twoheadrightarrow x \twoheadrightarrow (d(z_1, z_2) \twoheadrightarrow (z_1 \twoheadrightarrow_\emptyset e)\ z_2)\ d(x, y\ x)\big)\ C \\ & \mapsto_h & x \twoheadrightarrow (d(z_1, z_2) \twoheadrightarrow (z_1 \twoheadrightarrow_\emptyset e)\ z_2)\ d(x, C\ x) \end{array}$$

et



La constante e est en forme normale. Considérons la plus petite réduction de $C\ e$ à e. Dans cette réduction, les radicaux en tête doivent être réduits :

$$\begin{array}{rcl} C\ e & \equiv & (\mathsf{Y}\ C')\ e \\ & \mapsto\!\!\!\!\rightarrow_h & C'\ (\mathsf{Y}\ C')\ e \\ & \mapsto\!\!\!\!\rightarrow_h & (d(z_1, z_2) \twoheadrightarrow (z_1 \twoheadrightarrow_\emptyset e)\ z_2)\ d(e, C\ e) \\ & \mapsto_h & (e \twoheadrightarrow_\emptyset e)(C\ e) \end{array}$$

Mais le terme $(e \twoheadrightarrow_\emptyset e)\ (C\ e)$ peut être réduit en tête seulement après avoir réduit $C\ e$ à e. Ceci contredit la minimalité de la réduction et nous concluons ainsi que $C\ e$ ne se réduit pas à e.

Comme conséquence nous obtenons qu'aucun calcul à motifs défini en utilisant une fonction $\mathcal{S}ol$ qui remplit les conditions ci-dessus n'est confluent. Ceci est relativement étonnant puisque les deux derniers calculs sont clairement satisfaits par n'importe quel algorithme classique de filtrage syntaxique et le premier semble un choix raisonnable. D'autre part, ignorer l'information fournie par l'ensemble θ de variables liées quand on effectue le filtrage peut mener à des comportements étranges et, en particulier, ceci permet un encodage de l'égalité des termes.

Il y a eu des tentatives [BCKL03] pour développer des calculs qui combinent ces trois conditions mais, naturellement, elles ne sont pas complètement satisfaisantes. En fait, la solution proposée dans [BCKL03] valide la première condition mais ne valide ni la deuxième condition ni la troisième : la restriction syntaxique ne permet pas dans une abstraction $(A \twoheadrightarrow_\theta B)$ des motifs avec des variables dans θ et n'est ainsi pas conservative par rapport au λ-calcul.

$$(\rho_{\lambda_P}) \qquad (A \twoheadrightarrow B)(A\sigma) \qquad \rightarrow \qquad B\sigma$$

FIGURE 6.4 – *Sémantique opérationnelle du λ-calcul avec motifs*

6.3 Instances du calcul général

Dans cette section, nous analysons quelques instanciations du λ-calcul à motifs dynamiques. Pour des raison de simplicité, nous donnons seulement les points clés pour chacun des calculs à motifs concernés. Tous ces calculs ont été prouvés confluents sous des conditions appropriées ; nous donnons ici des preuves de confluence basées sur ces conditions et en utilisant la preuve de confluence générale pour le λ-calcul à motifs dynamiques. Cette approche ne fournit pas des preuves de confluence automatiques mais donne une méthodologie de preuve qui permet de se focaliser sur les propriétés fondamentales du filtrage sous-jacent qui peut être ainsi vu comme le point clé des calculs à motifs.

6.3.1 λ-calcul avec motifs

Le λ-calcul avec motifs a été introduit dans [vO90, KvOdV08] comme une extension du λ-calcul avec des possibilités de filtrage sur les motifs. L'ensemble de termes est paramétré par un ensemble de motifs Φ sur lesquels on peut abstraire.

La syntaxe du λ-calcul avec motifs est ainsi celle du λ-calcul à motifs dynamiques de base mais les motifs appartiennent dans ce cas à un ensemble donné et les abstractions lient toujours toutes les variables (libres) du motif. Sa sémantique opérationnelle est donnée par la règle ρ_{λ_P} dans la figure 6.4.

Au lieu de considérer des restrictions syntaxiques, nous pouvons, d'une manière équivalente, considérer qu'un problème de filtrage $A \lessdot_\theta A\sigma$ a une solution seulement quand $A \in \Phi$. Le λ-calcul avec motifs peut être ainsi vu comme une instance du λ-calcul à motifs dynamiques de base.

Le calcul n'est pas confluent en général [vO90] mais certaines restrictions peuvent être imposées à l'ensemble de motifs pour récupérer la confluence. Cette restriction s'appelle la condition de motif rigide (*Rigid Pattern Condition* - RPC) et peut être définie en utilisant la réduction parallèle du calcul.

Définition 29 (RPC) *Un terme P satisfait la RPC si*

$$\forall\sigma,\ \forall A,\ P\sigma \ \Rightarrow_\rho\ A \qquad \textit{implique} \qquad A \equiv P\sigma' \textit{ avec } \sigma \ \Rightarrow_\rho\ \sigma'$$

La définition permet des motifs qui sont extensionnellement mais pas intensionnellement rigide comme, par exemple, le terme $\Omega\ x\ y$ avec $\Omega = (x \twoheadrightarrow x\ x)\ (x \twoheadrightarrow x\ x)$. Nous utilisons une caractérisation syntaxiques (moins

générale) [vO90, KvOdV08] que nous appelons également RPC et qui exclut ces cas pathologiques.

Définition 30 (RPC) *L'ensemble de termes satisfaisant RPC est l'ensemble de tous les termes du λ-calcul qui*

- *sont linéaires (chaque variable libres apparaît au plus une fois),*
- *sont en forme normale,*
- *n'ont pas de variables actives (c.-à-d., pas de sous-termes de la forme xA avec x libre).*

Il faut noter que reformuler $\mathcal{H}_2$ dans le cas particulier d'une fonction $\mathcal{S}ol$ qui réalise du filtrage sur des motifs clos mène à une condition proche de la RPC originale (la réduction parallèle utilisée dans la définition de la RPC est légèrement différente). Néanmoins, l'hypothèse $\mathcal{H}_2$ permet au filtrage d'être exécuté sur des motifs qui ne sont pas en forme normale (ou pas réductible à eux-mêmes) alors que ce n'est pas le cas pour la RPC.

Exemple 26 *Les paires et les projections peuvent être encodées dans le λ-calcul avec motifs en filtrant directement sur l'encodage des paires.*

$$\begin{array}{lll} \big((z \twoheadrightarrow (z\ x)\ y) \twoheadrightarrow x\big)\big(z \twoheadrightarrow (z\ A)\ B\big) & \mapsto_{\rho_{\lambda_P}} & x\{A/x,\ B/y\} \\ & \equiv & A \end{array}$$

On peut noter que les hypothèses $\mathcal{H}_0$ et $\mathcal{H}_1$ sont vérifiées pour le λ-calcul avec motifs. Pour $\mathcal{H}_2$, on peut remarquer que si $P\sigma \Rightarrow_\rho B$ avec $P \in$ RPC alors $\exists B', \sigma'$ tel que $B' \equiv P\sigma'$ avec $\sigma \Rightarrow_\rho \sigma'$. Ceci montre qu'un radical ne peut pas se superposer avec P dans $P\sigma$ si $P \in$ RPC et donc que la condition $\mathcal{H}_2$ est satisfaite. Nous pouvons ainsi utiliser le théorème 19 et conclure à la confluence du λ-calcul avec motifs.

Proposition 2 *Le λ-calcul avec motifs est confluent si les motifs appartient à l'ensemble de termes défini par la RPC.*

6.3.2 Calculs de réécriture

Nous regardons maintenant les preuves de confluence des extensions du ρ-calcul présentées dans les sections 2.5 et 2.6, à savoir le ρ_{stk}-calcul et le ρ_d-calcul.

Pour ρ_{stk}, on peut considérer $\wr$ et stk comme des constantes du λ-calcul à motifs dynamiques et on peut voir ainsi la syntaxe de ρ_{stk} comme une instance de la syntaxe du λ-calcul à motifs dynamiques. La règle (ρ) du ρ_{stk}-calcul avec filtrage syntaxique correspond à la règle (ρ) du λ-calcul à motifs dynamiques avec la fonction $\mathcal{S}ol$ implantée en utilisant du filtrage linéaire de premier ordre à la Huet (voir la définition 27) et telle que $\mathcal{S}ol(P \lessdot B)$ renvoie une solution seulement si P est un motif.

Puisque l'ensemble de motifs de ρ_{stk} est un sous-ensemble des motifs satisfaisant RPC et le filtrage est réalisé de la même façon dans le ρ_{stk}-calcul et dans le λ-calcul avec motifs alors, on peut monter comme précédemment que les hypothèses $\mathcal{H}_0$, $\mathcal{H}_1$ et $\mathcal{H}_2$ sont vérifiées pour le ρ_{stk}-calcul. La relation induite par les règles $\delta \cup \mathsf{stk}$ est clairement terminante. On peut également prouver en utilisant une induction facile sur la structure des termes qu'elle est localement confluente et par conséquent confluente. Pour obtenir la confluence de ρ_{stk} comme une conséquence du théorème 20 il suffit de prouver (en utilisant une induction sur la structure des termes) que $\Rightarrow_\rho$ et $(\delta \cup \mathsf{stk})$ satisfont le diagramme de Yokouchi-Hikita.

Proposition 3 *Le ρ_{stk} avec des motifs algébriques linéaires est confluent.*

L'encodage du ρ_d-calcul est similaire à celui de ρ_{stk}. Il faut noter que pour la règle (ρ) du ρ_d-calcul, la fonction $\mathcal{S}ol$ correspondante a des solutions seulement quand l'argument est une valeur stuck. La preuve est donc similaire à celle de ρ_{stk}. La preuve du diagramme de Yokouchi-Hikita est légèrement différente mais l'utilisation des valeurs élimine la plupart des éventuelles paires critiques.

Proposition 4 *Le ρ_d-calcul est confluent.*

6.3.3 Version simplifié du calcul à motifs purs

Dans le λ-calcul, les structures de données telles que les paires ou les listes peuvent être encodées et bien que le λ-calcul permet la définitions de certaines fonctions avec un comportement uniforme par rapport aux structures de données, il n'offre pas le support pour réaliser des opérations qui exploitent des caractéristiques communes à toutes les structures de données comme, par exemple, une fonction de mise à jour qui traverse toutes les structures de données pour mettre à jour ses atomes. Dans le calcul à motifs purs [JK06] où n'importe quel terme peut être un motif, l'accent est mis sur le filtrage dynamique des structures de données.

La syntaxe du calcul à motifs purs dans sa version originale est identique à celle du λ-calcul à motifs dynamiques (sauf que le calcul à motifs purs définit un seul constructeur). Les abstractions sont appliquées en utilisant un algorithme de filtrage particulier. Bien que le papier original utilise une seule règle pour décrire l'application des abstractions (sur les motifs) nous présentons ici la sémantique du calcul en utilisant deux règles. La première règle dans la figure 6.5 est une instance de la règle (ρ) du λ-calcul à motifs dynamiques. La deuxième règle réduit les applications d'abstractions à l'identité (la motivation pour cette seconde règle est donnée dans [JK06]) quand le filtrage ne réussit pas.

ϕ-structures de données et formes ϕ-filtrable

$$
\begin{array}{lcll}
D & ::= & x\,(x \in \phi) \mid c \mid D\,A & (\phi\text{-structures de données}) \\
E & ::= & D \mid A \rightarrowtriangle_\theta B & (\text{formes } \phi\text{-filtrable}) \\
 & & \text{où } A \text{ et } B \text{ sont des termes arbitraires} &
\end{array}
$$

Sémantique du calcul à motifs purs simplifié

$$
\begin{array}{lcll}
(\rho_{\mathsf{pc}}) & (A \rightarrowtriangle_\theta B)C & \rightarrow & B\sigma \\
 & & & \text{si } \sigma = \mathcal{S}ol(A \prec\!\!\!\prec_\theta C) \\
(\rho^{\mathsf{stk}}_{\mathsf{pc}}) & (A \rightarrowtriangle_\theta B)C & \rightarrow & x \rightarrowtriangle x \\
 & & & \text{si } \mathsf{none} = \mathcal{S}ol(A \prec\!\!\!\prec_\theta C)
\end{array}
$$

FIGURE 6.5 – *Version simplifiée du calcul à motifs purs*

L'algorithme de filtrage du calcul à motifs purs est basé sur les notions de ϕ-structures de données (noté D) et les formes ϕ-filtrable (noté E) données dans figure 6.5.

La sémantique opérationnelle du calcul à motifs purs simplifié donnée dans la figure 6.5 est basée sur la fonction partielle $\mathcal{S}ol$ définie par les équations suivantes appliquées en respectant l'ordre ci-dessous

$$
\begin{array}{lcl}
\mathcal{S}ol(x \prec\!\!\!\prec_\theta A) & = & \{A/x\} \quad \text{si } x \in \theta \\
\mathcal{S}ol(c \prec\!\!\!\prec_\theta c) & = & \mathtt{id} \\
\mathcal{S}ol(A_1A_2 \prec\!\!\!\prec_\theta B_1B_2) & = & \mathcal{S}ol(A_1 \prec\!\!\!\prec_\theta B_1) \Cup \mathcal{S}ol(A_2 \prec\!\!\!\prec_\theta B_2) \\
 & & \quad \text{si } A_1A_2 \text{ est une } \theta\text{-structure de données} \\
 & & \quad \text{si } B_1B_2 \text{ est une structure de données} \\
\mathcal{S}ol(A_1 \prec\!\!\!\prec_\theta B_1) & = & \mathsf{none} \\
 & & \quad \text{si } A_1 \text{ est une forme } \theta\text{-filtrable} \\
 & & \quad \text{si } B_1 \text{ est une forme filtrable}
\end{array}
$$

Il faut noter que l'union $\Cup$ est seulement définie pour des substitutions avec les domaines disjoints et que l'union de none et σ est toujours none.

À la première vue, l'algorithme de filtrage peut sembler surprenant parce qu'on décompose syntaxiquement l'application qui est un symbole d'ordre supérieur mais cette décomposition est correcte puisqu'elle est réalisée seulement sur les structures de données, qui sont des formes normales de tête.

Exemple 27 *[JK06] On définit l'éliminateur générique* $\mathtt{elim} \triangleq x \rightarrowtriangle (xy) \rightarrowtriangle_y y$. *Par exemple, étant données deux constantes* `Cons` *et* `Nil` *représentant les constructeurs de liste, on définit la fonction* $\mathtt{singleton} \triangleq x \rightarrowtriangle$

$(\texttt{Cons}\ x\ \texttt{Nil})$ *et on peut vérifier*

$$\begin{array}{lll} \texttt{elim}\ \texttt{singleton} & \equiv & \big(x \twoheadrightarrow (xy) \twoheadrightarrow_y y\big)\big(x \twoheadrightarrow (\texttt{Cons}\ x\ \texttt{Nil})\big) \\ & \mapsto_{\rho_{\mathsf{pc}}} & \big((x \twoheadrightarrow \texttt{Cons}\ x\ \texttt{Nil})y\big) \twoheadrightarrow y \\ & \mapsto_{\rho_{\mathsf{pc}}} & (\texttt{Cons}\, y\, \texttt{Nil}) \twoheadrightarrow y \end{array}$$

Par rapport à la version simplifiée présentée dans cette section, le filtrage ainsi que la relation de réduction du calcul à motifs purs original sont définis par rapport à une information de contexte et plus précisément, dépendent des symboles qui lient (localement ou globalement) les variables des problèmes de filtrage et donc des redexes correspondants. Un telle approche « context sensitive » permet, en particulier, la réduction de termes qui sont irréductibles dans la version simplifiée du calcul présentée dans cette section.

La différence entre les deux approches est clairement illustrée dans [JK09] par le terme $(x \twoheadrightarrow x)\ x \twoheadrightarrow_x x$ qui ne peut pas être réduit dans le calcul à motifs purs défini ici. Ceci vient du fait que x est libre dans $x \twoheadrightarrow x$ et de ce fait attend une substitution avant le déclenchement de la règle. D'un autre côté, la même variable x est liée par l'abstraction externe et donc elle ne pourra jamais être instanciée. Par conséquent, le terme $(x \twoheadrightarrow x)\ x$ est en forme normale.

Ce problème est résolu dans le calcul à motifs purs [JK09] en définissant une relation de réduction qui dépend du contexte donné par les symboles liés (dans l'exemple précédent, l'abstraction externe lie toutes les variables de l'abstraction interne et dans ce cas le filtrage réussit et l'application est réduite à x) ou, d'une manière encore plus élégante, en introduisant une notion de symbole « filtrable » qui permet une définition indépendante du contexte de réduction mais isomorphe à la définition « context sensitive ». Bien que la version simplifiée présentée ici est moins expressive que la version originale et ne peut pas manipuler des exemples pathologiques comme celui présenté ci-dessus, elle préserve néanmoins les fonctionnalités principales du calcul initial.

Nous pouvons montrer la confluence du calcul à motifs purs en utilisant le théorème de confluence énoncé précédemment. On peut tout d'abord remarquer que l'hypothèse $\mathcal{H}_0$ est clairement vérifiée. Les hypothèses $\mathcal{H}_1$ et $\mathcal{H}_2$ suivent, comme on pouvait s'y attendre, des résultats intermédiaires dans [JK06]. En particulier, le lemme 7 [JK06] montre que la fonction $\mathcal{S}ol$ est stable par substitution et le lemme 8 [JK06] montrent que la fonction $\mathcal{S}ol$ est stable par réduction (quand elle renvoie une substitution et quand elle retourne none). Nous utilisons cette propriété pour montrer (en utilisant une induction simple) que la relation induite par la règle $(\rho^{\mathsf{stk}}_{\mathsf{pc}})$ est localement confluente. Elle termine trivialement, et elle est donc confluente. Le fait que les relations $\Rightarrow_{\rho_{\mathsf{pc}}}$ et $\mapsto_{\rho^{\mathsf{stk}}_{\mathsf{pc}}}$ vérifient le diagramme de Yokouchi-Hikita est obtenu encore une fois par induction et en utilisant la stabilité par réduction de la fonction $\mathcal{S}ol$ pour l'application d'une abstraction.

Syntaxe du λ-calculus with constructors

$$
\begin{array}{rcll}
A, B & ::= & x \mid c \mid \maltese \mid A\,B \mid x \twoheadrightarrow A \mid \{\!|\phi|\!\}.\,A & \textit{Termes} \\
\phi & ::= & c_1 \mapsto A_1; \ldots; c_n \mapsto A_n & \textit{Case bindings} \\
 & & (c_i \neq c_j \text{ for } i \neq j) &
\end{array}
$$

Sémantique du λ-calculus with constructors

(AppLam)	$(x \twoheadrightarrow A)B$	$\to$	$A\{x \leftarrow B\}$	
(AppDai)	$\maltese N$	$\to$	$\maltese$	
(CaseCons)	$\{\!|\phi|\!\}.\,c$	$\to$	A	**si** $(c \mapsto A) \in \phi$
(CaseDai)	$\{\!|\phi|\!\}.\,\maltese$	$\to$	$\maltese$	
(CaseApp)	$\{\!|\phi|\!\}.\,(A\,B)$	$\to$	$(\{\!|\phi|\!\}.\,A)\,B$	
(CaseLam)	$\{\!|\phi|\!\}.\,(x \twoheadrightarrow A)$	$\to$	$x \twoheadrightarrow \{\!|\phi|\!\}.\,A$	**si** $x \notin \mathcal{FV}(\phi)$
(CaseCase)	$\{\!|\phi|\!\}.\,\{\!|\psi|\!\}.\,A$	$\to$	$\{\!|\phi \circ \psi|\!\}.\,A$	

FIGURE 6.6 – *Le λ-calcul avec constructeurs*

Proposition 5 *Le calcul à motifs purs simplifié est confluent.*

Il faut préciser que l'approche présentée ici a été généralisée dans [JK09] pour le calcul à motifs purs basé sur la notion de symbole filtrable mentionnée précédemment.

6.3.4 λ-calcul avec constructeurs

Le λ-calcul avec constructeurs introduit dans [AMR06] pour traiter le problème de l'interaction entre des fonctions et des valeurs peut être également encodé dans le λ-calcul à motifs dynamiques [Fau07].

Une version légèrement simplifiée de la syntaxe et de la sémantique opérationnelle du λ-calcul avec constructeurs est donnée dans la figure 6.6 où la composition notée $\circ$ est une opération externe définie par :

$$\phi \circ (c_1 \mapsto A_1; \ldots; c_n \mapsto A_n) \quad \equiv \quad (c_1 \mapsto \{\!|\phi|\!\}.\,A_1; \ldots; c_n \mapsto \{\!|\phi|\!\}.\,A_n)$$

Exemple 28 *[AMR06] Dans le λ-calcul avec constructeurs la fonction prédécesseur est implantée par*

$$\text{pred} \equiv \big(n \twoheadrightarrow \{\!|0 \mapsto 0; s \mapsto (z \twoheadrightarrow z)|\!\}.\,n\big)$$

On peut vérifier que

$$\mathrm{pred}\,0 \to \{\!|0 \mapsto 0; s \mapsto (z \rightarrowtriangle z)|\!\}.\,0 \to 0$$

et

$$\begin{aligned}\mathrm{pred}\,(s\,N) &\to \{\!|0 \mapsto 0; s \mapsto (z \rightarrowtriangle z)|\!\}.\,(s\,N) \to (\{\!|0 \mapsto 0; s \mapsto (z \rightarrowtriangle z)|\!\}.\,s)\,N \\ &\to (z \rightarrowtriangle z)\,N \to N\end{aligned}$$

L'encodage du λ-calcul avec constructeurs dans le λ-calcul à motifs dynamiques est inspiré par l'exemple 25 et utilise la traduction

$$\{\!|c_1 \mapsto A_1; \ldots; c_n \mapsto A_n|\!\}.\,B \triangleq \big((c_1 \hookrightarrow A_1/\ldots/c_n \hookrightarrow A_n) \rightarrowtriangle_x x\big)B$$

avec x une variable fraîche et les symboles $\hookrightarrow$ et $/$ des constantes du λ-calcul à motifs dynamiques de base. Les règles (AppLam) et (CaseCons) sont encodées par une unique règle (ρ) avec une fonction $\mathcal{S}ol$ définie par

$$\begin{array}{lcl}\mathcal{S}ol(x \lessdot_x A) & = & \{A/x\} \\ \mathcal{S}ol\big((c_1 \hookrightarrow A_1/\ldots/c_n \hookrightarrow A_n) \lessdot_x c_{i_0}\big) & = & \{A_{i_0}/x\}\end{array}$$

On peut montrer facilement que la fonction $\mathcal{S}ol$ vérifie $\mathcal{H}_0, \mathcal{H}_1$ et $\mathcal{H}_2$. On prend pour l'ensemble de règles ξ toutes les règles données dans la figure 6.6 à l'exception de (AppLam) et (CaseCons) et on peut montrer que les conditions du théorème 20 sont satisfaites. On obtient ainsi la confluence du calcul.

Proposition 6 ([Fau07]) *Le λ-calcul avec constructeurs est confluent.*

6.4 Conclusion

Nous avons présenté un calcul d'ordre-supérieur avec motifs et nous avons donné les conditions qui assurent la confluence du calcul. La preuve de confluence est modulaire et isole les propriétés principales des algorithmes de filtrage sous-jacents, des éventuelles règles auxiliaires et de l'interaction entre ces composants, qui garantissent la consistance du calcul. Nous avons montré que les principaux calculs à motifs introduits dans la littérature ont des formulations alternatives dans le formalisme général et nous avons obtenu des preuves de confluence alternatives pour tous ces calculs en montrant que les conditions de confluence sont satisfaites dans tous les cas.

La preuve de confluence utilise les techniques standards basées sur la réduction parallèle de Tait et Martin-Löf et le lemme de Yokouchi-Hikita. La méthode proposée par M.Takahashi [Tak95] utilise une notion de développements complets et mène, en général, à des preuves plus élégantes et plus courtes. Nous pourrions évidement reformuler l'hypothèse $\mathcal{H}_2$ et adapter les preuves de confluence correspondantes en conséquence mais puisque les développements complets dépendent de la syntaxe du calcul, les hypothèses

seraient moins génériques et moins modulaires. Par ailleurs, les hypothèses reformulées seraient souvent plus difficile à prouver que dans le formulation originale.

Nous avons montré que la preuve de confluence du ρ-calcul est déduite facilement à partir de notre résultat général dès que le filtrage est unitaire et l'opérateur de structure n'a pas de théorie associée. Il serait certainement intéressant de raisonner modulo des théories non-unitaires et, en particulier, des théories équationnelles (qui sont utilisées couramment dans la pratique) et d'énoncer des résultats de confluence génériques similaires au cas unitaire. Une telle extension n'est pas triviale puisqu'elle nécessite l'existence d'un opérateur (associatif et commutatif) de structure pour grouper les différents résultats (du filtrage) et, comme montré dans [FM09], elle soulève des défis intéressants. Nous pensons néanmoins que ce travail est un bon point de départ pour l'étude des propriétés de confluence des calculs à motifs avec un filtrage non unitaire.

7

Conclusions et perspectives

La réécriture est au cœur de mes travaux de recherche qui ont porté jusqu'à présent sur les fondements théoriques de la réécriture ainsi que sur le développement d'outils de vérification et de preuve basés sur la réécriture. Ce manuscrit présente une partie des mes travaux sur le calcul de réécriture réalisés pendant ces dix dernières années et a également vocation à être utilisé comme support pour un cours introductif sur le calcul de réécriture.

Dans ce chapitre on retrouve un point sur l'état des recherches sur le calcul de réécriture ainsi qu'une présentation de mes autres thèmes de recherche et on discute les perspectives ouvertes par ces travaux. En annexe à ce document on peut trouver un recueil d'articles présentant certains aspects du calcul de réécriture qui n'ont pas été couverts dans ce document ainsi qu'une partie de mes travaux sur les applications possibles de la réécriture.

7.1 Point sur le calcul de réécriture

C'est avec Claude Kirchner, au cours de l'année 1998, que nous avons défini la première version du calcul de réécriture. L'objectif de ce travail a été depuis le début l'intégration dans un même formalisme des propriétés complémentaires de la réécriture de premier ordre et du λ-calcul ainsi que des fonctionnalités permettant d'exprimer le non-déterminisme. Ce calcul s'est avéré suffisamment puissant pour décrire non seulement la réécriture de termes mais également des stratégies classiques de réécriture. En effet, cette première version du calcul [CK98, CK99b] nous a permis de donner une sémantique complète au langage Elan, le langage à base de règles et stratégies développé à la même époque dans l'équipe Protheo.

J'ai continué le développement et l'étude du formalisme après ma thèse et en collaboration avec Claude Kirchner et Luigi Liquori, nous avons proposé une version encore plus simple et plus générique mais avec une expressivité tout aussi puissante [CKL01a]. Nous avons alors relevé un autre défi en

essayant d'aller au-delà de l'encodage des systèmes de réécriture et d'exprimer d'autres formalismes. Les similarités entre les structures du calcul de réécriture et les classes et objets des différents calculs orientés objet nous ont naturellement amené à étudier la relation entre ces formalismes et nous avons montré que plusieurs calculs à objets, à savoir le « Lambda Calculus of Objects » de Fisher, Honsell, et Mitchell et le « Object Calculus » de Abadi et Cardelli peuvent être encodés dans le calcul de réécriture.

Un plus grand nombre de personnes se sont alors impliquées dans l'étude de cette version qui est à la base de plusieurs extensions concernant des aspects dynamiques et statiques du calcul. Tout d'abord, plusieurs systèmes de types ont été proposés aussi bien dans le contexte de la programmation [CLW03, LW05, CKLW06] que dans le contexte des assistants à la preuve [CKL01b, BCKL03]. Les systèmes de la première catégorie vérifient la plupart des propriétés usuelles – la préservation du type, l'unicité et la décidabilité du typage – mais, grâce au filtrage, ces systèmes de types autorisent la définition de points fixes. Ils permettent donc le typage (des encodages) des systèmes de réécriture et des stratégies et nous donne ainsi une garantie supplémentaire sur la sûreté des programmes obtenus. Dans la deuxième catégories on retrouve une autre famille de systèmes de types développés sur la base des « Pure Type Systems » introduit par S. Berardi et J. Terlouw pour le λ-calcul : les « Pure Pattern Type Systems (P^2TS) ». Les types dépendants utilisés dans ce cas permettent de démontrer la normalisation forte des termes acceptés dans certains P^2TS [WH08]. Les P^2TS peuvent donc être utilisés pour définir un Logical Framework avec des capacités de filtrage, ce qui permet une représentation plus concise et plus naturelle des structures manipulées dans le contexte des assistants à la démonstration [Wac05]. Le formalisme est encore généralisé par le « General Logical Framework » [HLL07] avec une abstraction paramétrée par un prédicat qui est instancié par une contrainte de filtrage dans le cas des P^2TS.

L'étude de l'expressivité du calcul a été également continuée et nous avons montré [BCK03, BCK06] que les dérivations des systèmes de réécriture d'ordre supérieur comme les CRS et les HRS peuvent être exprimées en termes de dérivations du calcul de réécriture. Par ailleurs, Clara Bertolissi et Claude Kirchner ont montré [BK07] que la version la plus simple du calcul de réécriture peut également être encodée par un CRS orthogonal.

Nous avons également proposé une extension du calcul avec filtrage et substitutions explicites [CFK04, CFK07], extension qui peut être vue comme un support pour l'implantation de calculs à motifs. Ce calcul satisfait les propriétés classiques de ce type de formalisme, c'est-à-dire la confluence du calcul général ainsi que la terminaison et la confluence de la partie dédiée à la manipulation explicite des contraintes de filtrage. Nous avons montré qu'on peut adopter soit une approche atomique et donner une définition simple des substitutions, soit une approche plus générale et efficace permettant de définir un calcul qui réalise la composition des substitutions. Le calcul

est modulaire et peut être adapté à différentes théories de filtrage pour les constantes définies et pour l'opérateur de structure.

Les termes contraints utilisés dans le calcul avec substitutions explicites ont été étendus pour exprimer un partage explicite et des structures de données cycliques. Ceci a conduit à l'introduction et à l'étude d'un calcul de réécriture de graphes [BBCK06] qui permet l'encodage des formalismes similaires tels que la réécriture de graphes et le λ-calcul cyclique introduit par Z.M. Ariola et J.W. Klop. Ce formalisme permet en particulier l'abstraction sur des motifs algébriques et utilise non seulement des équations récursives mais également des contraintes de filtrage. La différence principale entre le calcul de réécriture de graphes et la réécriture de graphes se situe au niveau de l'application des règles ; dans le premier cas les règles et leur contrôle (la position d'application) sont définis au niveau objet du calcul tandis que dans la réécriture de graphes la stratégie de réduction est implicite.

D'autres extensions du calcul de réécriture général ont été proposées soit pour l'intégration des mécanismes d'exceptions [FK02, CKL02], soit pour l'intégration de traits impératifs [LS04, LS08]. Différentes stratégies d'évaluation ont été étudiées dans le contexte des langages fonctionnels [FMS06, CFF$^+$07] ou du calcul de réécriture de graphes [BBCK07]. Le calcul de réécriture général a été également utilisé comme point de départ pour différents formalismes permettant d'exprimer des modèles de calcul chimiques et biologiques [AK08, AK09].

Nous avons proposé différentes approches pour obtenir la confluence du calcul de réécriture général ainsi que la confluence des différentes versions présentées dans ce manuscrit. Les méthodes de preuves utilisées pour ces calculs ainsi que pour d'autres formalismes similaires sont relativement proches, ce qui montre l'importance d'une étude générique et modulaire de la confluence des calculs à motifs. Le calcul d'ordre-supérieur introduit dans le chapitre 6 représente une première étape dans cette direction puisqu'il généralise les différents calculs à motifs et nous permet, en particulier, d'isoler les principales propriétés garantissant la confluence [CF07]. Nous allons discuter par la suite de l'importance de généraliser ce formalisme pour aller au-delà du filtrage unitaire utilisé dans la version actuelle.

7.2 Fondements et applications de la réécriture

Dans le cadre de mes activité de recherche je me suis intéressé non seulement à la sémantique des langages à base des règles mais également à d'autres aspects de la réécriture stratégique et de ses applications.

Depuis le début de ma thèse, je m'intéresse à l'utilisation de la programmation avec des règles de réécriture et des stratégies pour la preuve et la vérification. Ma première expérience de programmation avec ce paradigme de programmation a été l'implantation en Elan de différents prouveurs et en

particulier du prouveur de prédicats proposé par J.R. Abrial [Abr97]. Je me suis ensuite intéressé à la spécification et l'analyse des protocoles d'authentification et je me suis focalisé dans un premier temps sur le protocole à clés publiques de Needham-Schroeder [NS78]. L'implantation avec des règles de réécriture de ce protocole utilisé pour réaliser une authentification mutuelle entre un initiateur et un répondeur qui communiquent par l'intermédiaire d'un réseau non sécurisé et en particulier, l'utilisation de stratégies, nous a permis de découvrir efficacement des attaques similaires à celles déjà décrites dans la littérature [Cir01]. Nous avons également proposé une implantation alternative en Tom et nous avons montré [CMR04] que l'utilisation des librairies Java pour la manipulation de collections permet une représentation efficace des états du système correspondant et un traitement des espaces de recherche plus importants que dans l'implantation Elan.

Nous avons utilisé des techniques similaires pour étudier des politiques de sécurité et nous nous sommes focalisé sur l'analyse des modèles de politiques d'accès de Bell-LaPadula et de McLean [LB96, McL88]. Nous avons proposé une méthodologie pour la spécification et l'implantation de politiques d'accès en Tom et nous avons montré que des fuites d'informations sont possibles si les niveaux de sécurité dans un modèle de type Bell-LaPadula ne sont pas complètement ordonnés [CMO08].

La notion de stratégie est très importante dans les applications mentionnées précédemment et elle est présente dans la plupart des langages à bases de règles. D'un autre côté, les fondements théoriques des stratégies, que ce soit dans le cadre des langages de programmation ou des assistants à la preuve, ont été relativement peu étudiés. De fait, les notions correspondantes ont été abordées d'une manière générique que récemment [KKK08]. Dans le même esprit, nous avons proposé [BCDK09] une notion de stratégie abstraite définie extensionnellement comme un ensemble de dérivations d'un système de réécriture abstrait. Ce point de vue abstrait est complété par un point de vue plus concret matérialisé par des définitions intentionnelles qui décrivent la façon dont ces ensembles de dérivations sont construits. Nous caractérisons également le lien entre les deux types de définitions et en particulier la classe de stratégies extensionnelles qui acceptent une définition intentionnelle.

Ces différents travaux théoriques contribuent au développement des langages à base des règles et par exemple, certaines constructions du langages Tom, dont les stratégies, sont directement inspirées de l'encodage de stratégies en calcul de réécriture. Je me suis intéressé à l'amélioration de l'expressivité de ce type de langage non seulement au niveau des opérateurs de stratégie mais également au niveau des algorithmes de filtrage utilisés. En particulier, nous avons étudié les implications de l'utilisation dans les algorithmes de filtrage d'une extension de la notion de terme, appelée « anti-terme » ou « anti-pattern », qui permet d'exprimer l'idée de complément [CKKM10].

Nous avons donné la sémantique précise de cette construction ainsi que des algorithmes de filtrage pour les cas syntaxique et associatif avec élément neutre.

7.3 Perspectives de recherche

L'étude du calcul de réécriture a ouvert de nombreuses perspectives concernant, d'une part les propriétés du calcul général et de ses instances, et d'autre part les applications de ce calcul.

La plupart des calculs de réécriture étudiés dans ce document ainsi que les différents calculs à motifs proposés dans la littérature utilisent un filtrage unitaire et par conséquent le résultat de l'application d'une règle de réécriture, si il existe, est unique. Néanmoins, dans des langages de programmation utilisant le paradigme de réécriture la théorie de filtrage est souvent associative ou associative-commutative et donc non-unitaire. Puisque les résultats de l'application des règles sont manipulés au niveau objet du calcul de réécriture, le non-déterminisme intrinsèque aux systèmes utilisant un filtrage non-unitaire peut être facilement encodé dans notre calcul. Nous avons montré que la version du calcul de réécriture permettant cet encodage est confluente si une stratégie d'évaluation de type appel par valeur est utilisée mais ce type de stratégie est relativement « bas-niveau » et n'est pas directement transposable aux autres calculs à motifs.

Il nous parait donc important de généraliser l'approche générique proposée dans le chapitre 6 pour prendre en compte des théories de filtrage non unitaires et d'utiliser cette généralisation pour étudier les extensions possibles de divers calculs à motif avec des algorithmes de filtrage plus puissants. Un point particulièrement intéressant dans cette direction concerne l'étude d'algorithmes de filtrage sur des motifs plus généraux que les termes algébriques et en particulier sur des collections de termes. La possibilité de filtrer sur des structures de résultats permettrait, en particulier, d'exprimer des stratégies plus élaborées que les stratégies de parcours généralement définies et implantées dans les langages à base de règles. On pourrait, par exemple, utiliser le filtrage sur les collections de résultats pour déterminer si différentes applications mènent à des résultats identiques ou pour traiter ces résultats selon un certain ordre. Ce type de stratégies est utilisé, par exemple, dans les analyseurs de politiques et de protocoles qui nécessitent un traitement explicite de l'espace de recherche et donc des collections d'états. Cette étude devrait également conduire à l'introduction de nouveaux opérateurs de stratégie dans le langage Tom qui utilise pour l'instant des constructions du langage hôte et en particulier les collections de Java pour réaliser ce type d'opération.

Cette généralisation avec des motifs plus élaborés et un filtrage potentiellement non-unitaire est envisageable pour toutes les versions du calcul de

réécriture. Nous avons déjà donné quelques pistes dans cette direction pour le calcul avec filtrage explicite du chapitre 4 qui se prête particulièrement bien à ce type d'extension grâce à sa modularité et son adaptabilité à différentes théories de filtrage éventuellement non-unitaires. Dans ce type de théorie, comme l'associativité-commutativité par exemple, il est bien plus efficace de décider si un problème de filtrage a une solution que de résoudre le problème. Il serait donc intéressant d'étudier une version optimisée du calcul explicite qui permette d'étiqueter les équations de filtrage pour indiquer si elles ont une solution et les résoudre seulement si les solutions correspondantes sont effectivement utilisées.

L'efficacité des implantation basées sur le calcul de réécriture peut être également améliorée en utilisant des stratégies d'évaluation appropriées. Dans le cas du calcul explicite, les stratégies d'évaluation doivent privilégier la composition de substitutions et l'élimination des substitutions superflues (c'est-à-dire sans impact sur les termes) par rapport à leur propagation. La situation est encore plus délicate dans le cas du calcul de réécriture de graphes car l'information de partage doit être gardée aussi longtemps que possible et une propagation prématurée peut aller à l'encontre de cet objectif.

La stratégie de partage que nous avons proposée pour le calcul de réécriture de graphes s'est avérée correcte et complète pour des termes normalisant et nous souhaitons tout d'abord étendre ce résultat et montrer la complétude de la stratégie pour des réductions normalisantes (à partir de termes qui ne sont pas nécessairement normalisants). Ensuite, nous avons l'intention d'étudier la question de l'optimalité pour les stratégies d'évaluation, où la notion d'« optimal » doit être formellement définie, par exemple en termes de temps, d'espace ou de préservation du partage. Les travaux sur la réduction optimale pour le λ-calcul constituent évidement un bon point de départ pour ces recherches.

Nous nous intéressons à l'efficacité des implantation basées sur le calcul de réécriture aussi bien qu'à la correction des programmes correspondants. Nous avons proposé des systèmes de types permettant l'analyse statique des programmes et garantissant que les arguments des fonctions ont le type attendu si la théorie de filtrage sous-jacente est en accord avec les types. Nous avons montré que la théorie syntaxique est une théorie de filtrage bien typée, c'est-à-dire garantissant que les substitutions obtenues comme résultat du filtrage sont bien typées, et dans la perspective d'une généralisation à des théories plus élaborées, il faudra étudier les conditions permettant de montrer que cette propriété reste valide pour les algorithmes correspondants.

La règle de typage pour les structures qui reste adaptée même dans le cas d'un filtrage non-unitaire peut être considérée relativement restrictive dans la perspective d'une utilisation pour des encodages typés de systèmes de réécriture généraux dont les règles peuvent porter sur des types différents. En particulier, cette règle impose que tous les membres d'une même structure

aient le même type et nous souhaitons lever cette restriction en utilisant des types conjonctions et une notion de sous-typage appropriée.

Pour résumer les différentes directions de recherche mentionnées jusqu'à maintenant, je souhaite continuer l'étude des calculs à motifs en général et du calcul de réécriture en particulier pour améliorer leur expressivité et leur efficacité en intégrant des théories de filtrage plus puissantes ainsi que leur sûreté en proposant des systèmes de types plus élaborées. Je pense que tous ces travaux permettront non seulement de mieux comprendre le comportement des langages à base de règles et stratégies mais également de les améliorer en inspirant de nouveaux constructions tant au niveau du langage qu'au niveau des types.

En parallèle à ces travaux sur le calcul de réécriture, je souhaite continuer l'étude des stratégies aussi bien d'un point de vue de leur expressivité que d'un point de vue de leurs propriétés. Il faudrait tout d'abord essayer de raffiner la caractérisation des stratégies abstraites et plusieurs pistes sont envisageables. Nous avons déjà commencé l'étude d'une approche logique qui consiste à caractériser les réductions acceptables d'une stratégie définie intentionnellement et qui permet de caractériser une classe plus importante de stratégies abstraites que les stratégies intentionnelles pures. Il faut maintenant caractériser les classes de stratégies abstraites exprimables en fonction de la logique utilisée et étudier l'intégration de ce type de stratégies dans un langage comme Tom. Une autre direction de recherche complémentaire à la généralisation des stratégies intentionnelles concerne l'étude de certaines restrictions de ces stratégies. Nous nous sommes focalisé jusqu'à maintenant soit sur des stratégies intentionnelles sans mémoire soit sur des stratégies intentionnelles arbitraires et un autre cas mérite d'être étudié : les stratégies à mémoire finie. Ces stratégies sont très intéressantes car elles peuvent être calculées par une machine à états finis et ont des applications potentielles dans le domaine de la vérification.

L'étude des propriétés des stratégies, notamment la confluence et la terminaison, est un autre point particulièrement intéressant. Ces propriétés sont très importantes, par exemple, dans le cadre de l'analyse de politiques de sécurité où elles garantissent la consistance des politiques exprimées par des systèmes de réécriture. Ces propriétés ont été relativement peu étudiées et les résultats correspondants ont été généralement obtenus par rapport à des stratégies spécifiques et non dans un cadre générique. Il serait donc intéressant de reformuler ces notions dans le formalisme général que nous avons proposé et d'établir des conditions de confluence et de terminaison similaires à celle de la réécriture classique. Nous étudions également une approche moins générale mais plus directe consistant à traduire des stratégies définies dans un langage donné (très similaire au langage de stratégies de Tom) en un système de réécriture qui n'est pas guidé par une stratégie d'évaluation donnée. Ceci permet l'utilisation de techniques de preuve de terminaison dis-

ponibles pour la réécriture classique et, en particulier, l'utilisation d'outils de terminaison existants (tels que Aprove et TTT2) pour prouver la terminaison de systèmes de réécriture stratégiques.

Nous avons déjà montré que la réécriture et les stratégies peuvent être utilisées pour la spécification et l'analyse de protocoles et de politiques de sécurité. Je souhaite continuer dans cette direction et proposer des formalismes et des outils (basés sur la réécriture avec stratégies) pour la spécification et la validations de politique de sécurité.

Les études de cas que nous avons réalisées jusqu'à maintenant sont relativement modulaires dans le sens où la politique est séparée de l'environnement d'application mais les spécifications correspondantes sont étroitement liées au formalisme de spécification. Nous avons donc commencé le développement et l'étude d'un cadre formel générique qui permet de décrire d'une manière générale la sémantique des politiques de sécurité et d'exprimer différentes propriétés de sécurité. Ce formalisme s'appuie sur des notions classiques comme les systèmes de transition pour spécifier, par exemple, les systèmes à sécuriser mais également sur des notions moins étudiées comme les systèmes de réécriture contraints pour spécifier, par exemple, les politiques. Ce travail ouvre donc plusieurs perspectives concernant les propriétés des différents composants du formalismes ainsi que ses applications.

Tout d'abord, la consistance des politiques exprimées dans notre formalisme dépend des propriétés intrinsèques aux systèmes de réécriture contraints utilisés pour leur spécification et donc, nous pensons qu'un axe de recherche intéressant serait l'étude des conditions garantissant la confluence et la terminaison des systèmes de réécriture contraints. Puisque ces systèmes correspondent dans une certaine mesure aux stratégies (intentionnelles) mentionnées précédemment, cette étude est évidement liée à l'analyse des propriétés de la réécriture sous stratégies.

Nous avons montré comment utiliser notre formalisme pour la spécifications de politiques d'accès et nous avons proposé une technique de transformation permettant la validation de propriétés classiques comme la confidentialité et l'intégrité exprimées d'une façon générique qui ne dépend pas des détails d'implantation de la politique analysée. Nous pensons que cette technique de transformation permettant la séparation explicite de la politique et du système sur lequel elle est appliquée sont particulièrement utiles pour le développement de technique de comparaison et de composition de politiques et de systèmes sécurisés.

Bibliographie

[Abr97] J.-R. Abrial. Le prouveur de prédicat. Technical report, 1997.

[AC96a] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer-Verlag, New York, 1996.

[AC96b] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer Verlag, 1996.

[ACCL91a] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy. Explicit substitutions. *Journal of Functional Programming*, 1(4) :375–416, October 1991.

[ACCL91b] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Levy. Explicit substitutions. *Journal of Functional Programming*, 4(1) :375–416, 1991.

[AK96] Z. M. Ariola and J. W. Klop. Equational term graph rewriting. *Fundam. Inf.*, 26(3-4) :207–240, 1996.

[AK97] Z. M. Ariola and J. W. Klop. Lambda calculus with explicit recursion. *Inf. Comput.*, 139(2) :154–233, 1997.

[AK07] O. Andrei and H. Kirchner. Graph rewriting and strategies for modeling biochemical networks. In V. Negru, T. Jebelean, D. Petcu, and D. Zaharie, editors, *Ninth International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, SYNASC 2007*, pages 407–414, Timisoara, Romania, September 2007. IEEE Computer Society.

[AK08] O. Andrei and H. Kirchner. A rewriting calculus for multigraphs with ports. *Electronic Notes in Theoretical Computer Science*, 219 :67–82, 2008.

[AK09] O. Andrei and H. Kirchner. A higher-order graph calculus for autonomic computing. In M. Lipshteyn, V. E. Levit, and R. M. McConnell, editors, *Graph Theory, Computational Intelligence and Thought*, volume 5420 of *Lecture Notes in Computer Science*, pages 15–26. Springer, 2009.

[All78] J. Allen. *Anatomy of LISP*. McGraw-Hill, Inc., New York, USA, 1978.

[AMR06] A. Arbiser, A. Miquel, and A. Ríos. A lambda-calculus with constructors. In *Term Rewriting and Applications – RTA'06*, volume 4098 of *Lecture Notes in Computer Science*, pages 181–196, Seattle, USA, August 2006. Springer.

[Bar84] H. Barendregt. *The Lambda-Calculus, its syntax and semantics*. Studies in Logic and the Foundation of Mathematics. Elsevier Science Publishers B. V. (North-Holland), Amsterdam, 1984. Second edition.

[Bar92] H. Barendregt. Lambda Calculi with Types. In *Handbook of Logic in Computer Science*, volume II, pages 118–310. Oxford University Press, 1992.

[BBCK05] C. Bertolissi, P. Baldan, H. Cirstea, and C. Kirchner. A rewriting calculus for cyclic higher-order term graphs. In *Proceedings of TERMGRAPH'04, International Workshop on Term Graph Rewriting, Roma, Italy*, volume 127 (5) of *Electronic Notes in Theoretical Computer Science*, pages 21–41. Elsevier Science, 2005.

[BBCK06] P. Baldan, C. Bertolissi, H. Cirstea, and C. Kirchner. A rewriting calculus for cyclic higher-order term graphs. *Mathematical Structures in Computer Science*, 17 :363–406, 2006.

[BBCK07] P. Baldan, C. Bertolissi, H. Cirstea, and C. Kirchner. Towards a sharing strategy for the graph rewriting calculus. In Jürgen Giesl, editor, *7th International Workshop on Reduction Strategies in Rewriting and Programming - WRS'07*, volume 204 of *Electronic Notes in Theoretical Computer Science*, pages 111–127, Paris, France, June 2007. Elsevier.

[BBK+07] E. Balland, P. Brauner, R. Kopetz, P.-E. Moreau, and A. Reilles. Tom : Piggybacking rewriting on java. In *Proceedings of the 18th Conference on Rewriting Techniques and Applications*, volume 4533 of *Lecture Notes in Computer Science*, pages 36–47. Springer-Verlag, 2007.

[BCC+03] O. Bournez, G.-M. Côme, V. Conraud, H. Kirchner, and L. Ibanescu. A Rule-Based Approach for Automated Generation of Kinetic Chemical Mechanisms. In R. Nieuwenhuis, editor, *Rewriting Techniques and Applications (RTA 2003)*, volume 2706 of *Lecture Notes in Computer Science*, pages 30–45. Springer, 2003.

[BCD83] H. Barendregt, M. Coppo, and M. Dezani-Ciancaglini. A filter lambda model and the completeness of type assignment. *The Journal of Symbolic Logic*, 48 :931–940, 1983.

[BCDK09] T. Bourdier, H. Cirstea, D. J. Dougherty, and H. Kirchner. Extensional and Intensional Strategies. In *9th International Workshop on Reduction Strategies in Rewriting and Programming - WRS'09*, volume 15 of *Electronic Proceedings in Theoretical Computer Science*, pages 1–20, Brasilia Brazil, 2009. abs/1001.4427.

[BCK03] C. Bertolissi, H. Cirstea, and C. Kirchner. Translating Combinatory Reduction Systems into the Rewriting Calculus. In J.-L. Giavitto and P.-E. Moreau, editors, *4th International Workshop on Rule-Based Programming - RULE'03*, volume 86(2) of *Electronic Notes in Theoretical Computer Science*, pages 17–32, Valencia, Spain, June 2003. Elsevier.

[BCK06] C. Bertolissi, H. Cirstea, and C. Kirchner. Expressing combinatory reduction systems derivations in the rewriting calculus. *Higher-Order and Symbolic Computation*, 19(4) :345–376, December 2006.

[BCKL03] G. Barthe, H. Cirstea, C. Kirchner, and L. Liquori. Pure Patterns Type Systems. In *30th SIGPLAN-SIGACT Symposium on Principles of Programming Languages - POPL'03*, volume 38(1) of *ACM SIGPLAN Notices*, pages 250–261, New Orleans, USA, January 2003. ACM.

[Ber05] C. Bertolissi. *The graph rewriting calculus : properties and expressive capabilities*. Thèse de doctorat, Institut National Polytechnique de Lorraine - INPL, Oct 2005.

[BK07] C. Bertolissi and C. Kirchner. The rewriting calculus as a combinatory reduction system. In *Foundations of Software Science and Computation Structures - FoSSaCS'07*, Lecture Notes in Computer Science, Braga, Portugal, March 2007. Springer.

[BKK98] P. Borovanský, C. Kirchner, and H. Kirchner. A functional view of rewriting and strategies for a semantics of ELAN. In M. Sato and Y. Toyama, editors, *The Third Fuji International Symposium on Functional and Logic Programming*, pages 143–167, Kyoto, April 1998. World Scientific. Also report LORIA 98-R-165.

[BKKM02] P. Borovansky, C. Kirchner, H. Kirchner, and P.-E. Moreau. ELAN from a rewriting logic point of view. *Theoretical Computer Science*, 2(285) :155–185, July 2002.

[BN98] F. Baader and T. Nipkow. *Term Rewriting and all That*. Cambridge University Press, 1998.

[BT88] V. Breazu-Tannen. Combining algebra and higher-order types. In *Proceedings 3rd IEEE Symposium on Logic in Computer Science, Edinburgh (UK)*, pages 82–90, 1988.

[Bur08] G. Burel. A first-order representation of pure type systems using superdeduction. In F. Pfenning, editor, *LICS*, pages 253–263. IEEE Computer Society, 2008.

[BvEG+87] H. P. Barendregt, M. C. J. D. van Eekelen, J. R. W. Glauert, J. R. Kennaway, M. J. Plasmeijer, and M. R. Sleep. Term graph

rewriting. In *Proceedings of PARLE'87, Parallel Architectures and Languages Europe*, volume 259 of *Lecture Notes in Computer Science*, pages 141–158, Eindhoven, 1987. Springer-Verlag.

[CDE+07] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Reriting Logic*, volume 4350. Springer-Verlag, 2007.

[CELM96] M. Clavel, S. Eker, P. Lincoln, and J. Meseguer. Principles of Maude. In J. Meseguer, editor, *Proceedings of the first international workshop on rewriting logic*, volume 4, Asilomar (California), September 1996. Electronic Notes in Theoretical Computer Science.

[CF07] H. Cirstea and G. Faure. Confluence of pattern-based lambda-calculi. In F. Baader, editor, *18th International Conference on Term Rewriting and Applications - RTA'07*, volume 4533 of *Lecture Notes in Computer Science*, pages 78–92, Paris, France, June 2007. Springer.

[CFF+07] H. Cirstea, G. Faure, M. Fernandez, I. Mackie, and F.-R. Sinot. From functional programs to interaction nets via the Rewriting Calculus. In Sergio Antoy, editor, *6th International Workshop on Reduction Strategies in Rewriting and Programming - WRS'06*, volume 174(10) of *Electronic Notes in Theoretical Computer Science*, pages 39–56, Seattle, Washington, USA, August 2007. Elsevier.

[CFK04] H. Cirstea, G. Faure, and C. Kirchner. A rho-calculus of explicit constraint application. In Narciso Marti Oliet, Manuel Clavel, and Alberto Verdejo, editors, *5th International Workshop on Rewriting Logic and its Applications - WRLA'04*, volume 117 of *Electronic Notes in Theoretical Computer Science*, pages 51–67, Barcelona, Spain, March 2004. Elsevier.

[CFK07] H. Cirstea, G. Faure, and C. Kirchner. A Rho-Calculus of explicit constraint application. *Higher-Order and Symbolic Computation*, 20 :37–72, 2007.

[CG99] A. Corradini and F. Gadducci. Rewriting on cyclic structures : Equivalence of operational and categorical descriptions. *Theoretical Informatics and Applications*, 33 :467–493, 1999.

[CHL96] P.-L. Curien, T. Hardin, and J.-J. Lévy. Confluence properties of weak and strong calculi of explicit substitutions. *Journal of the ACM (JACM)*, 43(2) :362–397, 1996.

[Chu41] A. Church. *The Calculi of Lambda Conversion*. Princeton University Press, Princeton, 1941.

[CHW07] H. Cirstea, C. Houtmann, and B. Wack. Distributive rewriting calculus. In Grit Denker and Carolyn L. Talcott, editors, *6th International Workshop on Rewriting Logic and its Applications - WRLA'06*, volume 176(4) of *Electronic Notes in Theoretical Computer Science*, pages 95–111, Barcelona, Spain, Mars 2007. Elsevier.

[Cir00] H. Cirstea. *Calcul de réécriture : fondements et applications.* Thèse de Doctorat, Université Henri Poincaré - Nancy I, 2000.

[Cir01] H. Cirstea. Specifying authentication protocols using rewriting and strategies. In I. V. Ramakrishnan, editor, *3rd International Symposium on Practical Aspects of Declarative Languages - PADL'01*, volume 1990 of *Lecture Notes in Computer Science*, pages 138–153, Las Vegas, USA, March 2001. Springer.

[CK98] H. Cirstea and C. Kirchner. The rewriting calculus as a semantics of ELAN. In J. Hsiang and A. Ohori, editors, *4th Asian Computing Science Conference - ASIAN'98*, volume 1538 of *Lecture Notes in Computer Science*, pages 84–85, Manila, The Philippines, December 1998. Springer.

[CK99a] S. Cerrito and D. Kesner. Pattern matching as cut elimination. In G. Longo, editor, *Fourteenth Annual IEEE Symposium on Logic in Computer Science*, Trento, Italy, July 1999. IEEE Comp. Soc. Press.

[CK99b] H. Cirstea and C. Kirchner. Combining higher-order and first-order computation using rho-calculus : Towards a semantics of ELAN. In D. Gabbay and M. de Rijke, editors, *Frontiers of Combining Systems 2*, Research Studies, pages 95–120. Wiley, 1999. ISBN 0863802524.

[CK00] H. Cirstea and C. Kirchner. The simply typed rewriting calculus. In K. Futatsugi, editor, *3rd International Workshop on Rewriting Logic and Application - WRLA'00*, volume 36 of *Electronic Notes in Theoretical Computer Science*, pages 21–37, Kanazawa (Japan), September 2000. Elsevier.

[CK01] H. Cirstea and C. Kirchner. The rewriting calculus — Part I *and* II. *Logic Journal of the Interest Group in Pure and Applied Logics*, 9(3) :427–498, May 2001.

[CK04] S. Cerrito and D. Kesner. Pattern matching as cut elimination. *Theoretical Computer Science*, 323 :71–127, 2004.

[CKKM10] H. Cirstea, C. Kirchner, R. Kopetz, and P.-E. Moreau. Anti-patterns for rule-based languages. *Journal of Symbolic Computation*, 45(5) :523 – 550, 2010. Symbolic Computation in Software Science.

[CKL01a] H. Cirstea, C. Kirchner, and L. Liquori. Matching Power. In A. Middeldorp, editor, *12th International Conference on Term Rewriting and Applications - RTA'01*, volume 2051 of *Lecture Notes in Computer Science*, pages 77–92, Utrecht, The Netherlands, May 2001. Springer.

[CKL01b] H. Cirstea, C. Kirchner, and L. Liquori. The Rho Cube. In F. Honsell, editor, *Foundations of Software Science and Computation Structures - FoSSaCS'01*, volume 2030 of *Lecture Notes in Computer Science*, pages 168–183, Genova, Italy, April 2001. Springer.

[CKL02] H. Cirstea, C. Kirchner, and L. Liquori. Rewriting Calculus with(out) Types. In F. Gadducci and U. Montanari, editors, *4th International Workshop on Rewriting Logic and Application - WRLA'02*, volume 71 of *Electronic Notes in Theoretical Computer Science*, pages 3–19, Pisa, Italy, September 2002. Elsevier.

[CKLW03] H. Cirstea, C. Kirchner, L. Liquori, and B. Wack. Rewrite strategies in the rewriting calculus. In B. Gramlich and S. Lucas, editors, *3rd International Workshop on Reduction Strategies in Rewriting and Programming* , volume 86(4) of *Electronic Notes in Theoretical Computer Science*, pages 18–34, Valencia, Spain, June 2003. Elsevier.

[CKLW06] H. Cirstea, C. Kirchner, L. Liquori, and B. Wack. Decidable Type Inference for the Polymorphic Rewriting Calculus. In *17è Journées Francophones des Langages Applicatifs - JFLA 2006*, Pauillac, France, January 2006. Extended version as Research Report INRIA 00000817.

[CLW03] H. Cirstea, L. Liquori, and B. Wack. Rewriting calculus with fixpoints : Untyped and first-order systems. In S. Berardi, M. Coppo, and F. Damiani, editors, *Types for Proofs and Programs - TYPES'03*, volume 3085 of *Lecture Notes in Computer Science*, pages 147–161, Torino, Italy, May 2003. Springer.

[CMO08] H. Cirstea, P.-E. Moreau, and A. S. d. Oliveira. Rewrite based specification of access control policies. In D. Dougherty and S. Escobar, editors, *3rd International Workshop on Security and Rewriting Techniques - SecRet'08*, volume 234 of *Electronic Notes in Theoretical Computer Science*, pages 37–54, Pittsburgh, PA, USA, June 2008. Elsevier.

[CMR04] H. Cirstea, P.-E. Moreau, and A. Reilles. Rule based programming in java for protocol verification. In Narciso Marti Oliet, Manuel Clavel, and Alberto Verdejo, editors, *5th International Workshop on Rewriting Logic and Application - WRLA'04*, volume 117 of *Electronic Notes in Theoretical Computer Science*, pages 209–227, Barcelona, Spain, March 2004. Elsevier.

[Cop85] M. Coppo. A completeness theorem for recursively defined types. In W. Brauer, editor, *Automata, Languages and Programming (ICALP)*, volume 194 of *Lecture Notes in Computer Science*, pages 120–129, Nafplion, Greece, 1985. Springer.

[Cor93] A. Corradini. Term rewriting in CT_Σ. In M. C. Gaudel and J. P. Jouannaud, editors, *Proceedings of TAPSOFT'93, Theory and Practice of Software Developm ent / 4th International Joint Conference CAAP/FASE*, pages 468–484. Springer, Berlin, Heidelberg, 1993.

[dB78] N. G. de Bruijn. Lambda calculus with namefree formulas involving symbols that represent reference transforming mappings. *Indagationes Mathematicae*, 40 :348–356, 1978.

[DG01] R. David and B. Guillaume. A λ-calculus with explicit weakening and explicit substitution. *Mathematical Structures for Computer Science*, 11 :169–206, 2001.

[DHK00] G. Dowek, T. Hardin, and C. Kirchner. Higher order unification via explicit substitutions. *Information and Computation*, 157(1/2) :183–235, 2000.

[DHK02] G. Dowek, T. Hardin, and C. Kirchner. Binding logic : proofs and models. In M. Baaz and A. Voronkov, editors, *Proceedings of the LPAR conference*, volume 2514 of *Lecture Notes in Computer Science*, pages 130–144, Tbilisi, Georgia, October 2002. Springer-Verlag.

[DL67] L. P. Deutsch and B. W. Lampson. An online editor. *Commun. ACM*, 10(12) :793–799, 1967.

[DM82] L. Damas and R. Milner. Principal Type-Schemes for Functional Programs. In *POPL*, pages 207–212. The ACM Press, 1982.

[Eke95] S. Eker. Associative-commutative matching via bipartite graph matching. *Computer Journal*, 38(5) :381–399, 1995.

[Fau07] G. Faure. *Structure et modèles de calcul de réécriture*. Thèse de doctorat, Université Henri Poincaré Nancy, 2007.

[FH83] F. Fages and G. Huet. Unification and matching in equational theories. In *Proceedings Fifth Colloquium on Automata, Algebra and Programming, L'Aquila (Italy)*, volume 159 of *Lecture Notes in Computer Science*, pages 205–220. Springer-Verlag, 1983.

[FHM94] K. Fisher, F. Honsell, and J. C. Mitchell. A Lambda Calculus of Objects and Method Specialization. *Nordic Journal of Computing*, 1(1) :3–37, 1994.

[FK02] G. Faure and C. Kirchner. Exceptions in the rewriting calculus. In S. Tison, editor, *Proceedings of the Rewriting Techniques and Applications - RTA'02*, Lecture Notes in Computer Science, pages 66–82, Copenhagen, July 2002. Springer-Verlag.

[FM09] G. Faure and A. Miquel. A Categorical Semantics for The Parallel Lambda-Calculus. Research Report RR-7063, INRIA, 2009.

[FMS06] M. Fernández, I. Mackie, and F.-R. Sinot. Interaction nets vs. the rewriting calculus : Introducing bigraphical nets. *Electronic Notes in Theoretical Computer Science*, 154(3) :19–32, 2006.

[FN96] K. Futatsugi and A. Nakagawa. An Overview of Cafe Project. In *Proceedings of First CafeOBJ Workshop*, Yokohama (Japan), August 1996.

[Gim73] J. F. Gimpel. A theory of discrete patterns and their implementation in snobol4. *Commun. ACM*, 16(2) :91–100, 1973.

[GKK+87] J. A. Goguen, C. Kirchner, H. Kirchner, A. Mégrelis, J. Meseguer, and T. Winkler. An introduction to OBJ-3. In J.-P. Jouannaud and S. Kaplan, editors, *Proceedings 1st International Workshop on Conditional Term Rewriting Systems, Orsay (France)*, volume 308 of *Lecture Notes in Computer Science*, pages 258–263. Springer-Verlag, July 1987. Also as internal report CRIN : 88-R-001.

[GMW79] M. Gordon, A. Milner, and C. Wadsworth. *Edinburgh LCF : A Mechanized Logic of Computation*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, New York (NY, USA), 1979.

[HF92] P. Hudak and J. H. Fasel. A gentle introduction to Haskell. *SIGPLAN Notices*, 27(5) :1–52, 1992.

[HLL07] F. Honsell, M. Lenisa, and L. Liquori. A framework for defining logical frameworks. *Electronic Notes in Theoretical Computer Science*, 172 :399–436, 2007.

[HO80] G. Huet and D. Oppen. Equations and rewrite rules : A survey. In R. V. Book, editor, *Formal Language Theory : Perspectives and Open Problems*, pages 349–405. Academic Press inc., 1980.

[Hue76] G. Huet. *Résolution d'équations dans les langages d'ordre 1, 2, ..., ω*. Thèse de doctorat d'état, Université Paris, 7, 1976.

[Hue80] G. Huet. Confluent reductions : Abstract properties and applications to term rewriting systems. *Journal of the ACM*, 27(4) :797–821, 1980.

[Jay04] B. Jay. The pattern calculus. *ACM Trans. Program. Lang. Syst.*, 26(6) :911–937, 2004.

[JK86] J.-P. Jouannaud and H. Kirchner. Completion of a set of rules modulo a set of equations. *SIAM Journal of Computing*, 15(4) :1155–1194, 1986.

[JK91] J.-P. Jouannaud and C. Kirchner. Solving equations in abstract algebras : a rule-based survey of unification. In J.-L. Lassez

and G. Plotkin, editors, *Computational Logic. Essays in honor of Alan Robinson*, chapter 8, pages 257–321. The MIT press, Cambridge (MA, USA), 1991.

[JK06] B. Jay and D. Kesner. Pure pattern calculus. In *European Symposium on Programming – ESOP'06*, volume 3924 of *Lecture Notes in Computer Science*, pages 100–114, Vienna, Austria, March 2006. Springer.

[JK09] C. B. Jay and D. Kesner. First-class patterns. *Journal of Functional Programming*, 19(2) :191–225, 2009.

[JO97] J.-P. Jouannaud and M. Okada. Abstract data type systems. *Theoretical Computer Science*, 173(2) :349–391, 28 February 1997.

[Kah03] W. Kahl. Basic pattern matching calculi : Syntax, reduction, confluence, and normalisation. SQRL Report 16, Software Quality Research Laboratory, McMaster Univ., 2003.

[Kam88] S. N. Kamin. Inheritance in smalltalk-80 : A denotational definition. In *POPL*, pages 80–87. ACM Press, 1988.

[Kes00] D. Kesner. Confluence of extensional and non-extensional lambda-calculi with explicit substitutions. *Theoretical Computer Science*, 238(1-2) :183–220, 2000.

[Kes07] D. Kesner. The theory of calculi with explicit substitutions revisited. In J. Duparc and T. A. Henzinger, editors, *Computer Science Logic*, volume 4646 of *Lecture Notes in Computer Science*, pages 238–252. Springer, 2007.

[Kir90] C. Kirchner, editor. *Unification*. Academic Press inc., 1990.

[KK99] C. Kirchner and H. Kirchner. Rewriting, solving, proving. A preliminary version of a book available at `http://www.loria.fr/~ckirchne/=rsp/rsp.pdf`, 1999.

[KKK08] C. Kirchner, F. Kirchner, and H. Kirchner. Strategic computations and deductions. In C. Benzmüller, C. E. Brown, J. Siekmann, and R. Statman, editors, *Reasoning in Simple Type Theory. Festchrift in Honour of Peter B. Andrews on His 70th Birthday*, volume 17 of *Studies in Logic and the Foundations of Mathematics*, pages 339–364. College Publications, 2008.

[KKM07] C. Kirchner, R. Kopetz, and P.-E. Moreau. Anti-pattern matching. In *European Symposium on Programming - ESOP'07*, Lecture Notes in Computer Science, Braga, Portugal, March 2007. Springer.

[KKSdV91] J. R. Kennaway, J. W. Klop, M. R. Sleep, and F. J. de Vries. Transfinite reductions in orthogonal term rewriting systems. In *Proceedings 4th Conference on Rewriting Techniques and Applications, Como (Italy)*, 1991. also Report CS-R9041, CWI, 1990.

[KL05] D. Kesner and S. Lengrand. Extending the explicit substitution paradigm. In *16th International Conference on Term Rewriting and Applications - RTA'05*, volume 3467 of *Lecture Notes in Computer Science*, pages 407–422. Springer, 2005.

[Kli93] P. Klint. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering and Methodology*, 2 :176–201, 1993.

[Klo80] J. W. Klop. *Combinatory Reduction Systems.* Ph.D. thesis, Mathematisch Centrum, Amsterdam, 1980.

[KvOdV08] J. W. Klop, V. van Oostrom, and R. C. de Vrijer. Lambda calculus with patterns. *Theoretical Computer Science*, 398(1-3) :16–31, 2008.

[KvOvR93] J. W. Klop, V. van Oostrom, and F. van Raamsdonk. Combinatory reduction systems : introduction and survey. *Theoretical Computer Science*, 121 :279–308, 1993.

[LB96] L. LaPadula and D. Bell. Secure Computer Systems : A Mathematical Model. *Journal of Computer Security*, 4 :239–263, 1996.

[LDG+04] X. Leroy, D. Doligez, J. Guarrigue, D. Rémy, and J. Vouillon. The Objective Caml system, 2004. `http://caml.inria.fr/pub/docs/manual-ocaml/`.

[Les94] P. Lescanne. From $\lambda\sigma$ to λv a journey through calculi of explicit substitutions. In *Conference on Principles of Programming Languages - POPL'94*, pages 60–69. ACM, January 1994.

[LHL07] L. Liquori, F. Honsell, and M. Lenisa. A framework for defining logical frameworks. In *Electronic Notes in Theoretical Computer Science.*, volume 172, 2007.

[LM03] J. Liu and A. C. Myers. Jmatch : Iterable abstract pattern matching for java. In *PADL '03 : Proceedings of the 5th International Symposium on Practical Aspects of Declarative Languages*, pages 110–127, London, UK, 2003. Springer-Verlag.

[LN04] C. Liang and G. Nadathur. Choices in representation and reduction strategies for lambda terms in intensional contexts. *Journal of Automated Reasoning*, 33(2) :89–132, 2004.

[LS04] L. Liquori and B. Serpette. iRho : an Imperative Rewriting Calculus. In *Proc. of PPDP, ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, pages 167–178. The ACM Press, 2004.

[LS08] L. Liquori and B. P. Serpette. irho : an imperative rewriting calculus. *Mathematical Structures in Computer Science*, 18(3) :467–500, 2008.

[LW05] L. Liquori and B. Wack. The polymorphic rewriting-calculus : Type checking vs. type inference. In N. Marti-Oliet, editor, *Proceedings of the Fifth International Workshop on Rewriting Logic and Its Application*, volume 117 of *entcs*, pages 89 – 111. Elsevier, 2005.

[McB70] F. McBride. *Computer Aided Manipulation of Symbols*. PhD thesis, Queen's University of Belfast, 1970.

[McL88] J. McLean. The algebra of security. In *Proc. IEEE Symposium on Security and Privacy*, pages 2–7. IEEE Computer Society Press, 1988.

[Mel95] P.-A. Melliès. Typed lambda-calculi with explicit substitutions may not terminate. In M. Dezani-Ciancaglini and G. D. Plotkin, editors, *Second International Conference on Typed Lambda Calculi and Applications - TLCA'95*, volume 902 of *Lecture Notes in Computer Science*, pages 328–334. Springer, 1995.

[Mel96] P.-A. Melliès. *Description Abstraite des Systèmes de Réécriture*. PhD thesis, Université Paris 7, 1996.

[Mel02] P.-A. Melliès. Axiomatic rewriting theory VI residual theory revisited. In *Rewriting Techniques and Applications – RTA'02*, volume 2378 of *Lecture Notes in Computer Science*, pages 24–50, Copenhagen, Denmark, July 2002. Springer.

[Men87] N. P. Mendler. *Inductive Definition in Type Theory*. PhD thesis, Cornell University, Ithaca, USA, 1987.

[Mil78] R. Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences*, 17 :348–375, 1978.

[MN98] R. Mayr and T. Nipkow. Higher-order rewrite systems and their confluence. *Theoretical Computer Science*, 192 :3–29, 1998.

[MRV03] P.-E. Moreau, C. Ringeissen, and M. Vittek. A Pattern Matching Compiler for Multiple Target Languages. In G. Hedin, editor, *12th Conference on Compiler Construction, Warsaw (Poland)*, volume 2622 of *LNCS*, pages 61–76. Springer-Verlag, May 2003.

[Muñ97] C. Muñoz. *Un calcul de substitutions pour la représentation de preuves partielles en théorie de types*. Thèse de doctorat, Université Paris 7, 1997. English version available as INRIA research report RR-3309.

[NS78] R. Needham and M. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12) :993–999, 1978.

[Ohl98] E. Ohlebusch. Church-Rosser theorems for abstract reduction modulo an equivalence relation. In *Rewriting Techniques and*

Applications (RTA-98), volume 1379 of *Lecture Notes in Computer Science*, 1998.

[Par92] M. Parigot. $\lambda\mu$-calculus : An algorithmic interpretation of classical natural deduction. In A. Voronkov, editor, *Proc. of Int. Conf. on Logic Programming and Automated Reasoning, LPAR'92, St Petersburg, Russia, 15–20 July 1992*, volume 624 of *Lecture Notes in Artificial Intelligence*, pages 190–201. Springer-Verlag, Berlin, 1992.

[Pea89] G. Peano. *Arithmetices principia, nova methodo exposita*. Fratres Bocca, Turin, 1889.

[Pey87] S. Peyton-Jones. *Implementation of Functional Programming Languages*. Prentice-Hall, 1987.

[Pey03] S. Peyton Jones. Special issue : Haskell 98 language and libraries. *Journal of Functional Programming*, 13, January 2003.

[Pic07] R. Pickering. *Foundations of F#*. Apress, Berkely, CA, USA, 2007.

[PJ87] S. Peyton-Jones. *The implementation of functional programming languages*. Prentice Hall, Inc., 1987.

[Plo75] G. D. Plotkin. Call-by-name, call-by-value, and the λ-calculus. *Theoretical Computer Science*, 1 :125–159, 1975.

[Pot] F. Pottier. Course DEA : Typage et programmation. `http://pauillac.inria.fr/~fpottier/mpri/dea-typage.ps.gz`.

[Ros96] K. H. Rose. *Operational Reduction Models for Functional Programming Languages*. PhD thesis, DIKU, University of Copenhagen, Denmark, February 1996.

[Rou75] P. Roussel. Prolog : Manuel de référence et d'utilisation. Technical report, Université de Marseille (France), 1975.

[SDK+03] A. Stump, A. Deivanayagam, S. Kathol, D. Lingelbach, and D. Schobel. Rogue decision procedures. In *1st International Workshop on Pragmatics of Decision Procedures in Automated Reasoning - PDPAR'03*, 2003.

[SPvE93] M. R. Sleep, M. J. Plasmeijer, and M. C. J. D. van Eekelen, editors. *Term graph rewriting : theory and practice*. Wiley, London, 1993.

[Ste00] M.-O. Stehr. CINNI — A generic calculus of explicit substitutions and its application to λ-, ς- and π-calculi. In K. Futatsugi, editor, *3rd International Workshop on Rewriting Logic and its Applications - WRLA'00*, volume 36 of *Electronic Notes in Theoretical Computer Science*, pages 71–92. Elsevier, 2000.

[Tak95] M. Takahashi. Parallel reductions in λ-calculus. *Information and Compututation*, 118 :120–127, 1995.

[Ter03] Terese. *Term Rewriting Systems*. Cambridge University Press, 2003. M. Bezem, J. W. Klop and R. de Vrijer, eds.

[Tur79] D. A. Turner. A new implementation technique for applicative languages. *Software and Practice and Experience*, 9 :31–49, 1979.

[vD96] A. van Deursen. An Overview of ASF+SDF. In *Language Prototyping*, pages 1–31. World Scientific, 1996. ISBN 981-02-2732-9.

[Vis99] E. Visser. Strategic pattern matching. In P. Narendran and M. Rusinowitch, editors, *Rewriting Techniques and Applications (RTA'99)*, volume 1631 of *Lecture Notes in Computer Science*, pages 30–44, Trento, Italy, July 1999. Springer-Verlag.

[Vis01] E. Visser. Stratego : A language for program transformation based on rewriting strategies. System description of Stratego 0.5. In A. Middeldorp, editor, *Rewriting Techniques and Applications (RTA'01)*, volume 2051 of *Lecture Notes in Computer Science*, pages 357–361. Springer-Verlag, May 2001.

[Vit94] M. Vittek. ELAN *: Un cadre logique pour le prototypage de langages de programmation avec contraintes*. Thèse de Doctorat d'Université, Université Henri Poincaré – Nancy 1, October 1994.

[vO90] V. van Oostrom. Lambda calculus with patterns. Technical report, Vrije Universiteit, Amsterdam, November 1990.

[Wac05] B. Wack. *Typage et déduction dans le calcul de réécriture*. Thèse de doctorat, Université Henri Poincaré Nancy 1, 2005.

[Wel99] J. B. Wells. Typability and Type Checking in System F are Equivalent and Undecidable. *Annals of Pure and Applied Logic*, 98(1–3) :111–156, 1999.

[Wer94] B. Werner. *Une Théorie des Constructions Inductives*. PhD thesis, Université Paris 7, 1994.

[WH08] B. Wack and C. Houtmann. Strong normalisation in two pure pattern type systems. *Mathematical Structures in Computer Science*, 18(3) :431–465, 2008.

[YH90] H. Yokouchi and T. Hikita. A rewriting system for categorical combinators with multiple arguments. *SIAM Journal on Computing*, 19(1) :78–97, February 1990.