Structures et modèles de calculs de réécriture

Germain Faure

Nancy-Université
Université Henri Poincaré

Département de formation doctorale en informatique

École doctorale IAEM Lorraine

Structures et modèles de calculs de réécriture

THÈSE

présentée et soutenue publiquement le 5 juillet 2007

pour l'obtention du

Doctorat de l'Université Henri Poincaré

(spécialité informatique)

par

Germain Faure

Composition du jury

Président :	Dominique Mery	Professeur, Université Henri Poincaré, Nancy, France
Rapporteurs :	Gilles Dowek	Professeur, École Polytechnique, Palaiseau, France
	Femke van Raamdonk	Professeur, Vrije Universiteit, Amsterdam, The Netherlands
Examinateurs :	Horatiu Cirstea (directeur)	Maître de Conférences, Université Nancy II, Nancy, France
	Claude Kirchner (directeur)	Directeur de recherche, INRIA, Nancy, France
	Paul-André Melliès	Chargé de Recherches CNRS, Équipe PPS, Paris, France

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

Loria

Mis en page avec LaTeX

Ever tried. Ever failed. No matter. Try Again. Fail again. Fail better.

By Samuel Beckett in Worstward Ho

The road to wisdom ? Well, it's plain and simple to express :
Err and err and err again, but less and less and less.
By Piet Hein in Knuth's Mathematical Writing

Remerciements

Je suis heureux d'écrire ces remerciements, profitant de l'occasion qui m'est donnée d'exprimer ma gratitude.

Encadrement

Mon travail au sein de l'équipe Protheo a commencé il y a déjà plus de 5 ans. J'ai été accueilli par Claude Kirchner qui m'a fait découvrir le monde de la recherche avec un enthousiasme certain. À chaque étape, sa vision d'ensemble a été fort utile et la confiance qu'il m'a accordée a été souvent précieuse. J'ai toujours été chaleureusement accueilli dans le bureau de Claude et j'en suis toujours sorti étonné de sa grande curiosité.

Pendant mon stage de licence et pendant ma thèse, j'ai lu à plusieurs reprises la thèse d'Horatiu Cirstea, y découvrant à chaque fois de nouvelles idées. Chaque jour d'avantage, j'apprécie la vivacité d'esprit et l'humour d'Horatiu. Mon seul regret est d'avoir trop parlé et de ne pas m'être laissé suffisamment interpeller. Sa présence au quotidien, mélange de pédagogie, de simplicité et de patience, a été extrêmement précieuse. Elle m'a permis de parfaire, par des remarques toujours pertinentes et constructives, mes compétences scientifiques mais aussi la qualité de ma rédaction. Travailler avec lui a toujours été un plaisir.

J'ai choisi d'aborder mon sujet de thèse en utilisant plusieurs approches dont une (sémantique dénotationnelle) est relativement éloignée de la culture de mes encadrants et de l'équipe. Cela fut certes compliqué, mais aussi très instructif, m'obligeant toujours à questionner ma démarche. Ce questionnement a été parfois long et je tiens à remercier chacun de mes encadrants d'avoir permis ces différentes approches (ce qui nécessite une ouverture d'esprit prononcée) et d'avoir respecté ces temps de questionnements.

Je tiens plus généralement à les remercier très chaleureusement pour ce que chacun m'a apporté. Je suis conscient que je leur ai demandé beaucoup et que l'investissement dont ils ont fait preuve est à la fois rare et de qualité.

Jury

C'est à la fois une joie et un honneur d'avoir un jury composé de scientifiques de tout premier plan et issus de communautés relativement différentes. Je remercie chacun d'eux d'avoir accepté d'en faire partie (notamment Dominique Mery d'avoir accepté d'en être le président) et de l'intérêt qu'ils ont porté à mon travail.

À plusieurs reprises, j'ai pu constater la pertinence et la pédagogie de Gilles Dowek et Femke van Raamsdonk. Leurs notes de cours (sur la théorie des types, respectivement la réécriture d'ordre supérieur), leurs articles et leurs exposés m'ont appris beaucoup, aussi

bien par leur contenu que par leur présentation. Les échanges avec Gilles ont toujours été encourageants et plein de discernement et, la thèse de Femke a été pour moi un modèle de clarté et de rigueur.

Je les remercie très fortement d'avoir accepté d'être rapporteurs et d'avoir vu beaucoup plus que ce que j'avais écrit. Je tiens à les remercier, ainsi que Paul-André Melliès, pour l'incroyable qualité de leurs questions qui constituent une mine d'or que je souhaite exploiter au mieux.

Collaborations et cadre de travail

L'ensemble des membres de l'équipe Protheo a largement contribué à cette thèse, notamment par une lecture minutieuse d'une partie de mon manuscrit. Je tiens à les remercier très chaleureusement. Les membres permanents de l'équipe ainsi que Chantal Llorens m'ont offert un cadre de travail matériel très satisfaisant. Je les remercie notamment d'avoir permis mes nombreux déplacements.

Mes journées de travail auront été ponctuées par de nombreuses discussions, aussi bien avec des théoriciens fous (notamment Colin Riba) qu'avec des codeurs fous (notamment Antoine Reilles). J'apprécie toujours leur spontanéité et leur richesse.

Je tiens aussi à remercier les nombreux visiteurs de l'équipe Protheo pour m'avoir fait part de leurs remarques et interrogations. En particulier, Eduardo Bonelli m'a inspiré l'approche décrite dans la première partie de cette thèse. Qu'il en soit sincèrement remercié.

Après avoir accepté d'encadrer mon stage de maîtrise au pied levé, Alexandre Miquel m'a à nouveau accueilli pour m'aider à approcher différemment mon sujet de thèse. Ses qualités scientifiques et pédagogiques ont contribué à améliorer ma compréhension de la sémantique dénotationnelle. Sa capacité tant à proposer une vision globale qu'à passer une journée entière à comprendre pourquoi un diagramme commute m'impressionne encore aujourd'hui.

De nombreuses séances de travail avec Alexandre ont eu lieu au sein du laboratoire PPS. Je tiens à remercier l'ensemble des personnes qui ont rendu ces séjours agréables et instructifs. En particulier, je souhaite remercier très vivement Delia Kesner pour nos échanges extrêmement stimulants.

Je remercie Emmanuel Hainry et Antoine Reilles de m'avoir incité à utiliser Vim et Mutt qui sont devenus, grâce à eux, mes outils préférés. Je les remercie aussi pour leur aide précieuse lors de la mise en page de ce document.

Enseignement

Durant ces quatre années passées à Nancy, j'ai eu la chance de côtoyer des élèves et des enseignants qui m'ont donné envie de m'investir avec le plus de sérieux et de dévouement possibles.

Je tiens tout d'abord à remercier très fortement mes élèves pour avoir accepté de me faire confiance malgré des méthodes de travail parfois originales, dans une période où j'étais encore hésitant sur mon approche de l'enseignement.

Que tous les enseignants avec qui j'ai travaillé et qui, par l'exemple ou le contre-exemple, m'ont aidé à tendre vers une pédagogie qui me ressemble soient remerciés. En particulier, Brigitte Jaray s'est attachée à me transmettre une idée de l'enseignement faite d'un savant mélange de rigueur, de plaisir et de liberté. Qu'elle soit assurée de ma sincère gratitude.

Je souhaiterais remercier les enseignants qui m'ont donné, lorsque j'étais leur étudiant, l'envie de faire ce métier. Merci à Odile Millet-Botta pour m'avoir fait découvrir l'informatique fondamentale et pour m'avoir incité à postuler à l'ENS de Lyon. Merci à Jacques Mazoyer qui a su, à l'occasion de son cours de calculabilité, me transmettre passion et intuitutions. Merci enfin à Jean-Pierre Jouannaud pour son excellent cours sur la réécriture d'ordre supérieur.

Pour finir...

Ce document étant à usage professionnel, je me suis limité aux remerciements directement liés à ce contexte.

Néanmoins, je souhaite remercier toutes les personnes avec qui j'ai partagé un bout de chemin une fois sorti du travail (les citer serait digne d'une énumération à la Prévert). Je trouve rassurant de ne pas savoir ce que je dois réellement à chacun d'eux mais je les remercie très sincèrement pour tout ce que nous avons partagé et partageons. Merci en particulier à ma famille pour son amour. Leur soutient affectif et matériel a toujours été opportun.

Merci enfin à Daniel qui a toujours su m'apporter, en plus du reste, une écoute attentive et réconfortante dans mes nombreux moments d'hésitations.

Table des matières

Extended abstract

In the 1930s, A. Church introduced the λ-calculus as a theoretical framework for describing functions and their evaluation and gave a negative answer to the Entscheidungsproblem [Chu36, Chu41]. Ever since, the λ-calculus greatly influenced functional programming languages and is even very often quoted as *the* paradigm of functional languages. Interestingly, in practice, functional languages strongly rely on pattern-matching. This is the case of O'caml [LDG+04], Haskell [Pey03], Lisp [All78] or F# [Pic07]. A "speed-up" of the λ-calculus with patterns where variable abstraction is replaced by pattern abstraction was then emerging from the practice of functional programming design [Pey87].

On one hand, following S. Peyton Jones the so-called λ-calculus with patterns was introduced [Oos90, KOV07]. Latter on, many different pattern-based λ-calculi arose in the context of the functional programming. For example, we can quote the basic pattern matching calculus [Kah03] and the pure pattern calculus [JK06].

On the other hand, studying rewriting —where pattern-matching is a fundamental operation— and especially rewriting strategies led to the rewriting calculus [CK98b]. The rewriting calculus, a.k.a. the ρ-calculus, was introduced to make explicit all the ingredients of rewriting such as rules, rule applications, strategies and results [CK01, Cir00]. The ρ-calculus is a generalization of the λ-calculus with pattern-matching features and term collections. The work presented here was initially motivated by its study.

There are thus several pattern λ-calculi and each of them has many syntactical variants depending on the intended use: define the relationship between pattern-calculi and first-order term rewriting [CHW06] or graph rewriting [BBCK06, Ber05], encode object λ-calculi [CKL01], study type systems [BCKL03, Wac05], add dynamic patterns [BCKL03, JK06], etc.. This plethora of calculi raises the following questions: what are the basic ingredients on which all these calculi are based, how do these components interact and what is the computational and expressive power of such calculi. To answer these questions there are many possible approaches and we chose four of them in this thesis that are summarized below. A more detailed presentation of each part is given afterwards.

To play with the computational and expressive power of pattern-based λ-calculi, we can study higher-order matching where λ-terms (purely functional programs) are replaced by ρ-terms (functional programs with pattern-matching). The work already done in the typed λ-calculus is not transposable, as the simply typed ρ-calculi is not normalizing [Wac05]. In fact, the normalizing simply typed ρ-calculus uses (weakly) dependent types built using Π-abstractions on patterns. Then for a given matching equation, to give the types of the matching variables almost corresponds to give the solutions. So, it is of strong interest to study higher-order matching in an untyped context. In this thesis, we analyze higher-order matching in the untyped λ-calculus as a preliminary step

to solve matching in the ρ-calculus.

Most of the presentations of the ρ-calculus use meta-level matching and substitution application but the explicit definition of these operations underlines some interactions between both operations. Following the works on explicit substitutions in the λ-calculus, we define and study a ρ-calculus with explicit matching and substitution application. We show that it is a useful theoretical back-end for implementations of pattern-based λ-calculi and especially rewriting-calculi.

To compare the expressiveness and the confluence of pattern-based λ-calculi, we introduce and study a general confluent formalism where the way pattern-abstractions are applied is axiomatized. This is done in the third part of this thesis.

Another way to understand the basic ingredients of pattern-based λ-calculi is to study a denotational semantics for the rewriting calculus. This raises many questions and challenges that are not intrinsically related to the ρ-calculus but that already appear in the parallel λ-calculus. This is why we propose a clear categorical semantics for the parallel λ-calculus that can be seen as a first step towards a denotational semantics for the ρ-calculus.

We now give a more detailed presentation for each of the corresponding contributions.

Higher-order matching modulo superdevelopments We often need to decide β-equivalence. This is the case for the transformation of strict functional programs (programs as λ-terms) and for the proof theory (proofs as λ-terms). Since β-equivalence is undecidable, we usually restrict to *typed* λ-terms in order to recover decidability. But when we transform functional programs with pattern-matching (programs as ρ-terms) and when we consider [Wac05] rich proof terms for the generalized deduction modulo (proofs as ρ-terms), β-equivalence has to be replaced by the equational theory of the ρ-calculus.

The equivalence in the ρ-calculus is also undecidable (as it includes β-equivalence). But unlike in the λ-calculus, the restriction to typed ρ-terms is not suitable in the context of higher-order matching. In fact, the normalizing simply typed ρ-calculus [Wac05] uses (weakly) dependent types built using Π-abstractions on patterns. Then for a given matching equation, to give the types of the matching variables almost corresponds to give the solutions.

Consequently, the work already done concerning higher-order matching in the simply typed λ-calculus is not transposable. This is why we propose a different approach based on a restriction of β-conversion [MS01]. The usual approximation of β-normal forms is given by complete finite developments with the corresponding parallel reduction of Tait and Martin-Löf. An extension of finite developments, called superdevelopments, was introduced in [Raa93] to prove the confluence of a general class of reduction systems containing the λ-calculus and term rewrite systems.

A superdevelopment is a reduction sequence that may reduce the redexes of the term, its residuals and some created redexes. The redexes created by the substitution of a variable in a functional position by a λ-abstraction are not reduced. The approximation given by superdevelopments of a second-order term coincides with its β-normal form. Su-

perdevelopments can be alternatively defined using a suitable parallel reduction [Acz78] that generalises the parallel reduction of Tait and Martin-Löf.

We consider here matching equations built over untyped λ-terms and solve them modulo superdevelopments. The matching problems are of interest particularly because the set of matches modulo superdevelopments contains, but is not restricted to, second-order β-matches. Moreover, the restriction of the β-conversion needed in the case of pattern-matching *à la* Miller [Mil91] is subsumed by superdevelopments and thus matching modulo superdevelopments is complete w.r.t. the matching *à la* Miller.

We propose sound and complete algorithms for the matching in the pure λ-calculus modulo superdevelopments and for the second-order matching modulo β. The algorithms are presented using transformation rules [MM82] based on the parallel reduction of Aczel while the intuitions and the proofs are based on the equivalent notion of superdevelopments. This leads to an intuitive presentation and to simpler proofs.

A ρ-calculus with explicit constraint applications Substitutions and pattern-matching are fundamental operations of the rewriting calculus that are inherited respectively from the λ-calculus and from rewriting. To perform pattern-matching, evaluation rules of the ρ-calculus use helper functions that are indeed implicit computations: all the computations related to the considered matching theory belong to the meta-level. These computations are conceptually and computationally important in all matching theories, from syntactic ones to quite elaborated ones like associative-commutative theories [Eke95]. In concrete implementations, substitutions and pattern-matching should be separated and should interact with one another. In particular, we want matching related computations and applications of substitutions to be explicit.

A first step toward an explicit handling of the matching related computations was the introduction of matching problems as part of the ρ-calculus syntax [CKL02]. More precisely, the *matching constraints* represent constrained terms which are eventually instantiated by the substitution obtained as solution of the corresponding matching problem (if such a solution exists).

We study two versions of the ρ-calculus, one with explicit matching and one with explicit substitutions (following the works on λ-calculi with explicit substitutions [ACCL91, Les94, Ros96]), together with a version that combines the two and considers efficiency issues and more precisely the composition of substitutions. This allows us to isolate the features absolutely necessary in both cases and to analyze the issues related to the two approaches. The result is a full calculus that enjoys the usual good properties of explicit substitutions (conservativity, termination) and which is confluent. We show that the ρ-calculus, and especially explicit ρ-calculi, are suitable as a theoretical back-end for implementations of rewriting-based languages.

This work is implemented in the Tom language [BBK+06] as an interpreter for the ρ-calculus.

Confluence of pattern-based λ-calculi Each of the pattern-based calculi mentioned before differs on the way patterns are defined and on the way pattern abstractions

are applied. Thus, patterns can be simple variables like in the λ-calculus, algebraic terms like in the algebraic rewriting calculus [CLW03], special (static) patterns that satisfy certain (semantic or syntactic) conditions like in the λ-calculus with patterns or dynamic patterns that can be instantiated and possibly reduced like in the pure pattern calculus and some versions of the rewriting calculus. The underlying matching theory strongly depends on the form of the patterns and can be syntactic, equational or more sophisticated [JK06, BCKL03].

Although some of these calculi just extend the λ-calculus by allowing pattern abstractions instead of variable abstractions, the confluence of these formalisms is lost when no restrictions are imposed.

Several approaches are then used to recover confluence. One of these techniques consists in syntacticly restricting the set of patterns and then showing that the reduction relation is confluent for the chosen subset. This is done for example in the λ-calculus with patterns and in the ρ-calculus (with algebraic patterns). The second technique considers a restriction of the initial reduction relation (that is, a strategy) to guarantee that the calculus is confluent on the whole set of terms. This is done for example in the pure pattern calculus where the matching algorithm is a partial function whereas any term is a pattern.

Nevertheless, we can notice that in practice the proof methods share the same structure and that each variation on the way pattern abstractions are applied needs another proof of confluence. There is thus a need for a more abstract and more modular approach. A possible way to have a unified approach for proving the confluence is the application of the general and powerful results on the confluence of higher-order rewrite systems [KOR93, MN98, Ter03]. Although these results have already been applied for some particular pattern-calculi [BK07] the encoding seems to be rather complex for some calculi and in particular for a general setting that encompasses several calculi. Moreover, it would be interesting to have a framework where the expressiveness and (confluence) properties of the different pattern calculi can be compared.

We thus show [CF07] that all the pattern-based calculi can be expressed as a general calculus parameterized by a function that defines the underlying matching algorithm and thus the way pattern abstractions are applied. This function can be instantiated (implemented) by a unitary matching algorithm as in [CK01, JK06] but also by an anti-pattern matching algorithm [KKM07] or it can be even more general [LHL07]. We propose a generic confluence proof where the way pattern abstractions are applied is axiomatized. Intuitively, the sufficient conditions to ensure confluence guarantee that the (matching) function is stable by substitution and by reduction.

We apply our approach to several classical pattern calculi, namely the λ-calculus with patterns, the pure pattern calculus, the rewriting calculus and the λ-calculus with constructors [AMR06]. For all these calculi, we give the encodings in the general framework from which we derive the proofs of confluence. This approach does not provide confluence proofs for free but it establishes a proof methodology and isolates the key points that make the calculi confluent. It can also point out some matching algorithms that although natural at the first sight can lead to non confluent reduction in the corresponding calculus.

A categorical semantics of the parallel λ-calculus The starting point of this work was the semantical study of the ρ-calculus. A Scott semantics revealed the similarity (up to the fact that term collections are not systematically required to be associative, commutative and/or idempotent) between term collections of the ρ-calculus interpreted as the binary join and Boudol's parallel construct of the parallel λ-calculus [Bou94] .

Formally, the parallel λ-calculus is obtained by extending the pure λ-calculus with a binary operator that intuitively represents the parallel execution. The parallel λ-calculus adds to the equational theory of the pure λ-calculus the single equational axiom (δ) expressing the distributivity of function application w.r.t. parallel aggregation. The parallel λ-calculus was initially introduced as a tool to study full-abstraction of the interpretation of λ-terms in Scott domains. In this framework, Boudol extended the interpretation of pure λ-terms to the parallel construction using the join operation.

Scott semantics is well-suited to achieve full-abstraction but it is not sufficient to capture neither the basic equational theory of the calculus nor many interesting extensions of it—typically when dealing with extensionality. In the same way, interpreting the parallel operator as the binary join automatically validates associativity, commutativity and idempotence, although on a purely syntactical level, these equational axioms are clearly independent from the basic equational axiom (β) and (δ). For these reasons, there is a need for a more general and more modular semantics.

We define [FM07] a sound and complete categorical semantics for the parallel λ-calculus, based on a notion of aggregation monad which is modular w.r.t. associativity, commutativity and idempotence. This semantics is complete in a very strong sense: for each extension $\mathcal{T}$ of the basic equational theory of the parallel λ-calculus, it is possible to find a reflexive object such that the equational theory induced by the interpretation of parallel λ-terms in this object is exactly the theory $\mathcal{T}$.

To prove completeness, we introduce a category of partial equivalence relations adapted to parallelism, in which any extension of the basic equational theory of the calculus is induced by some model. We also present abstract methods to construct models of the parallel λ-calculus in categories where particular equations have solutions, such as the category of Scott domains and its variants, and check that G. Boudol's original semantics is a particular case of ours.

The semantical study of the ρ-calculus also pointed out many interesting problems related to the interaction between a mechanism of pattern-matching and the parallel construct. These problems helped us to grasp the importance of linear terms which play a central rôle in the proof of completeness (Boudol's shift towards the λ-calculus with resources [BCL99] seems to be motivated by similar reasons).

We think that the categorical semantics introduced here will contribute to a better understanding of the interaction between pattern-matching and the parallel construct, and thus will constitute a significant step towards a denotational semantics for the ρ-calculus.

Introduction

Plusieurs développements récents montrent que les assistants à la preuve sont suffisamment mûrs pour être utilisés soit pour formaliser des théories mathématiques non triviales [Gon05, ADGR07, GWZ02, Fly07], soit pour spécifier et étudier des programmes de grande taille [MPU07, Fil03, Ler06, BDL06].

Pourtant, la distance entre une preuve papier et une preuve réalisée grâce à un assistant à la preuve est toujours importante. Cette distance est en partie due à la nécessité, dans le cas d'une preuve formelle, d'expliciter tous les raisonnements considérés y compris les plus triviaux et c'est bien pour cela que les mathématiciens ne présentent pas leurs preuves dans des systèmes formels.

> *There are good reasons why Mathematicians do not usually present their proofs in fully formal style. It is because proofs are not only a means to certainty, but also a means to understanding. Behind each substantial formal proof there lies an idea, or perhaps several ideas. The idea, initially perhaps tenuous, explains why the result holds. The idea becomes Mathematics only when it can be formally expressed, but that expression must be so couched as to reveal the idea; it will not do to bury the idea under the formalism.*
> (Saunders MacLane)

> *Proofs really aren't there to convince you something is true – they're there to show you why it is true.*
> (Andrew Gleason)

De plus, l'utilisation des assistants à la preuve révèle qu'un passage à l'échelle n'est possible que si le formalisme sous-jacent possède une capacité calculatoire suffisante. La réécriture est souvent choisie pour cette capacité à exprimer et implémenter simplement le calcul. Par exemple, plusieurs extensions du calcul des constructions, formalisme à la base du projet Coq, combinent le λ-calcul et la réécriture [TG89, Bar91, JO95, Fer93, Bla05].

Les notions de règles et de relations de réécriture jouent donc un rôle fondamental dans ce contexte, comme dans celui de la transformation de programmes [Vis05], de la sécurité des protocoles [Rus06], des politiques de contrôles d'accès [HJM06] , etc. Ces notions sont aussi particulièrement pertinentes pour manipuler des structures de données vérifiant des invariants [Rei06, HJM06], c'est-à-dire pour maintenir la représentation interne de données en forme canonique par rapport à un système de réécriture.

Toutes ces applications de la réécriture utilisent explicitement ou implicitement la notion de stratégies [BKK96, Vis01b, KK04, Kir05, MOMV06] qui s'est imposée universellement dans les langages à base de règles (Tom, Elan, Maude), certains langages

en ayant même fait un véritable paradigme de programmation (Stratego). Il est particulièrement intéressant de remarquer que l'ingrédient fondamental de la réécriture, le filtrage, est présent dans la plupart de langages fonctionnels et intervient donc dans des applications couvrant des domaines variés, comme par exemple le web [Bal06, Fri06] ou la programmation parallèle [CMV+06].

Pourtant, et de manière tout à fait surprenante, le λ-calcul, modèle de calcul particulièrement bien étudié, repose sur une notion triviale de filtrage. Par exemple, S. Peyton-Jones insista sur l'importance d'étudier une généralisation du λ-calcul avec du filtrage, afin de proposer pour les langages fonctionnels un paradigme incluant de manière primitive le filtrage [Pey87].

Cette absence de filtrage dans le λ-calcul nécessite de nombreux encodages tant il est vrai qu'il n'est pas adapté pour décrire de manière simple des mécanismes de calcul. Cette problématique se retrouve par exemple dans le calcul des constructions qui a été à plusieurs reprises enrichi de primitives de calcul [CP88, Bar91], dans la transformation de programmes [MS01, Sit01] où l'on utilise de manière intensive la primitive `fold` pour contourner l'absence de filtrage, ou encore lorsque l'on souhaite donner une sémantique aux langages à base de règles.

C'est pour toutes ces raisons que l'on voit émerger à partir des années 90, de nombreux calculs avec motifs introduits soit pour donner une sémantique aux langages à base de règles comme le calcul de réécriture [CK01, Cir00, CK98a], soit dans le cadre de la programmation fonctionnelle comme le λ-calcul avec motifs [Oos90], le calcul basique de filtrage [Kah03], le calcul pur de motifs [Jay04, JK06] et le λ-calcul avec constructeurs [AMR06].

Introduit par H. Cirstea et C. Kirchner pour expliciter les mécanismes de la réécriture et des stratégies, le calcul de réécriture ou ρ-calcul est *in fine* une généralisation du λ-calcul avec filtrage et agrégation de termes. L'abstraction sur les variables est étendue en une abstraction sur les motifs et dans sa définition la plus générale, le calcul de réécriture permet l'utilisation du filtrage modulo une théorie équationnelle *a priori* arbitraire, utilisant l'agrégation pour collecter les différents résultats possibles. L'agrégation de termes, souvent appelée structure, peut donc être assimilée à des collections.

Dans cette thèse, nous proposons différentes approches pour expliciter les ingrédients de base des calculs avec motifs en général et du ρ-calcul en particulier. Nous cherchons à analyser le filtrage et l'agrégation en présence de mécanismes d'ordre supérieur et à préciser leurs interactions.

Dans une première partie, nous étudions le filtrage d'ordre supérieur dans le λ-calcul. L'approche classique [Hue76, HL78] consistant à se restreindre au sous-ensemble des termes typés est ici remplacée par une approche considérant les équations modulo une restriction de la β-conversion. Cette restriction est suffisamment expressive pour traiter les équations du second-ordre et les équations sur les motifs d'ordre supérieur à la Miller. En plus de ne pas dépendre d'un système de types particulier, les algorithmes que nous

étudions n'introduisent pas de nouvelles variables de filtrage durant le processus de résolution.

Dans une deuxième partie, nous proposons tout d'abord une extension du ρ-calcul où les opérations fondamentales de filtrage et de l'application de substitutions sont rendues explicites, généralisant ainsi les λ-calculs avec substitutions explicites. Nous étudions ensuite la propriété de cohérence (confluence) des calculs avec motifs et nous montrons que cette étude peut être réalisée de manière axiomatique sur les propriétés des algorithmes de filtrage utilisés dans ces calculs. Nous isolons ainsi les propriétés implicitement utilisées dans les différentes preuves de confluence des λ-calculs avec motifs.

Dans la troisième partie de cette thèse, nous introduisons une sémantique catégorique du λ-calcul parallèle dont les premiers modèles donnés par G. Boudol en sont des cas particuliers. Nous montrons que cette sémantique est correcte et complète. Nous donnons plusieurs constructions possibles de tels modèles et notament dans les domaines de Scott. Ce travail montre que les notion d'opérateur pour le parallélisme, de collection de termes et d'opérateur de structure du ρ-calcul peuvent être étudiées simultanément grâce à la notion plus générale de monades d'agrégation.

Guide de lecture Ce document est divisé en trois parties relativement indépendantes. Nous avons pour cela occasionnellement répété certaines définitions. L'intérêt est bien-sûr de permettre une lecture non-linéaire du manuscrit.

Les chapitres 1, 4 et 7 de cette thèse sont introductifs et présentent respectivement la normalisation dans le λ-calcul, le ρ-calcul et les modèles catégoriques du λ-calcul.

Le chapitre 2 définit dans plusieurs contextes (typés ou non) le filtrage d'ordre supérieur dans le λ-calcul et donne plusieurs instances (filtrage modulo β, modulo super-développements, second-ordre, motifs à la Miller etc.). Nous comparons ensuite leur expressivité.
Dans le chapitre 3, nous développons plusieurs algorithmes pour le filtrage d'ordre supérieur et plus spécialement pour le filtrage modulo super-développements dont les propriétés (terminaison, correction et complétude) sont précisément étudiées.

Le chapitre 5 introduit plusieurs extensions du ρ-calcul qui rendent explicite le filtrage puis l'application de substitutions. Nous étudions les propriétés de ces différents calculs (confluence) et montrons différentes interactions possibles entre le filtrage et les substitutions. Nous présentons une implémentation de ces calculs, implémentation simple mais efficace qui offre un interpréteur pour le calcul de réécriture.
Le chapitre 6 présente un calcul où les motifs sont dynamiques, c'est-à-dire qu'ils peuvent être instanciés et réduits. Nous proposons une preuve de confluence générique où la manière dont le filtrage est réalisé est axiomatisée. Nous montrons que cette approche s'applique à différents calculs avec motifs. Nous caractérisons aussi une classe d'algorithmes de filtrage qui conduisent à des calculs non confluents.

Le chapitre 8 est consacré à la définition d'une sémantique catégorique pour le λ-calcul parallèle. Nous introduisons les notions fondamentales de monades d'agrégation

et de termes linéaires. Nous illustrons enfin notre définition sur plusieurs exemples pris dans les domaines de Scott. Nous retrouvons notamment les premiers modèles du λ-calcul parallèle introduits par G. Boudol.

Dans le chapitre 9, nous prouvons la complétude de la sémantique du λ-calcul parallèle introduite dans le chapitre 8. Nous introduisons pour cela une notion de modèles syntaxiques basés sur les « pers ».

Nous concluons notre étude des modèles du λ-calcul parallèle en donnant dans le chapitre 10 deux méthodes de constructions de tels modèles à partir de modèles vérifiant certaines équations. Nous illustrons ces constructions dans la catégorie des domaines de Scott où ces équations ont des solutions intéressantes.

Pour finir nous présentons plusieurs pistes pour prolonger le travail présenté tout au long de ce manuscrit.

Quelques notations ne sont pas uniformes d'une partie à une autre. Nous avons essayé à chaque fois d'utiliser celles qui nous paraissaient les plus claires et les plus couramment utilisées suivant le contexte. Le choix final a été fait en ayant en tête le conseil de Witehead :

> *A good notation sets the mind free to think about really important things.*
> (Alfred North Witehead)

Première partie

Filtrage d'ordre supérieur dans le lambda-calcul

Chapitre 1

Normalisations dans le λ-calcul

Contexte Dans ce chapitre, nous examinons la normalisation du λ-calcul munie de la β-réduction dans un cadre typé puis non typé. Chacune des deux approches conduit, dans le chapitre suivant, à une étude différente du filtrage d'ordre supérieur.

Nous proposons tout d'abord de ne considérer que des termes bien typés. Restreint à ce sous-ensemble de termes, la β-réduction termine. Ensuite, au lieu de restreindre l'ensemble des termes, nous restreignons la β-réduction : nous considérons une première restriction donnée par les développements et une seconde par les super-développements.

Un développement est une suite de réductions qui ne réduit que les résidus des radicaux présents dans le terme initial. Le théorème des développements finis (qui assure que de telles réductions sont toujours finies) apparaît pour la première fois dans la preuve de confluence du λI-calcul donnée par A. Church et J.R. Rosser [CR36]. La preuve générale pour le λ-calcul telle qu'elle est donnée dans [Hin78] illustre l'importance du théorème des développements finis dans toutes les preuves de confluence des λ-calculs.

La notion de super-développements, introduite initialement pour prouver la confluence d'une classe générale de systèmes de réductions contenant le λ-calcul et les systèmes de réécriture [Raa93], généralise la notion de développement en autorisant la réduction additionnelle de tous les radicaux créés et qui ne sont pas obtenus par la substitution d'une variable en position fonctionnelle par une λ-abstraction. Le théorème des développements finis se généralise au théorème des super-développements finis.

Contributions Ce chapitre est introductif. Le lecteur peut consulter [Bar84, Bar92, Kri90, Raa96] pour plus de détails et notamment pour les preuves des résultats cités. La structure de ce chapitre est en partie inspirée du chapitre 2 de [Raa96].

Plan du chapitre Le chapitre est organisé est trois parties qui sont chacune une manière différente d'obtenir la normalisation dans le λ-calcul. Tout d'abord nous étudions dans la section 1.1, le λ-calcul simplement typé pour lequel la β-réduction termine toujours. Les deux sections suivantes se place dans le cadre du λ-calcul non typé, aussi appelé λ-calcul pur. Dans ce contexte, pour obtenir une réduction terminante nous considérons deux sous-ensembles de la β-réduction. Dans la section 1.2 nous considérons les développements et dans la section 1.3 les super-développements.

1.1 Lambda-calcul simplement typé et β-réduction

Nous rappelons ici les définitions de base du λ-calcul simplement typé afin de fixer les notations. Cette section est donc une succession de définitions de base qui se terminent par le théorème de confluence et de forte normalisation du λ-calcul simplement typé.

Définition 1.1 (Types) *Étant donné un ensemble de types de base $\mathfrak{T}_0$, l'ensemble $\mathfrak{T}$ des types est défini inductivement comme le plus petit ensemble*
- *contenant $\mathfrak{T}_0$;*
- *et tel que pour tous $\alpha, \beta \in \mathfrak{T}$ on a $(\alpha \to \beta) \in \mathfrak{T}$.*

Définition 1.2 (Ordre d'un type) *L'ordre d'un type α dénoté $\mathfrak{o}(\alpha)$ est défini par :*
- $\mathfrak{o}(\alpha) = 1$ *si* $\alpha \in \mathfrak{T}_0$ *;*
- $\mathfrak{o}(\alpha \to \beta) = \mathit{max}(\mathfrak{o}(\alpha) + 1, \mathfrak{o}(\beta))$.

Définition 1.3 (Termes bien typés) *Soit $\mathcal{K}$ un ensemble de constantes ayant chacune un type unique. Pour chaque type $\alpha \in \mathfrak{T}$, on suppose donné un ensemble de variables de ce type, dénoté $\mathcal{X}_\alpha$. L'ensemble $\mathcal{X}$ est défini comme l'union des ensembles $\mathcal{X}_\alpha$ supposés distincts, soit $\mathcal{X} = \cup_{\alpha \in \mathfrak{T}} \mathcal{X}_\alpha$.*

Par définition, l'ensemble des termes bien typés noté $\mathcal{T}_t$ est
- *le plus petit ensemble contenant toutes les constantes et toutes les variables ;*
- *et clos par les règles suivantes :*
  - *Si A est un élément de $\mathcal{T}_t$ de type $\alpha \to \beta$ et B est un élément de $\mathcal{T}_t$ de type α, alors (AB) est un élément de $\mathcal{T}_t$ de type β ;*
  - *Si A est un élément de $\mathcal{T}_t$ de type β et x est une variable de $\mathcal{X}_\alpha$, alors $\lambda x.A$ est un élément de $\mathcal{T}_t$ de type $\alpha \to \beta$.*

On utilisera les lettres A, B, C pour désigner des éléments de $\mathcal{T}_t$, les lettres a, b, c, f, g, h pour désigner des constantes et les lettres x, y, z pour désigner des variables.

Les atomes qui représentent soit une variable soit une constante seront dénotés par la lettre ε.

Pour une meilleure lisibilité, le terme $(\ldots((\varepsilon\ A_1)\ A_2)\ldots)\ A_n)$ sera souvent noté $\varepsilon(A_1, A_2, \ldots, A_n)$, où ε est un atome et où $A_1, \ldots, A_n$ sont des termes quelconques.

Étant donné un terme A_1A_2 par définition le terme dit en *position applicative* est par définition le terme A_2 et le terme en *position fonctionnelle* est par définition le terme A_1.

Définition 1.4 (Variables libres et liées) *L'ensemble des variables libres d'un terme A noté $\mathtt{fv}(A)$ et l'ensemble des variables liées noté $\mathtt{bv}(A)$ sont définis par*

$$\begin{array}{rclcrcl} \mathtt{fv}(x) & = & \{x\} & & \mathtt{bv}(x) & = & \emptyset \\ \mathtt{fv}(a) & = & \emptyset & & \mathtt{bv}(a) & = & \emptyset \\ \mathtt{fv}(AB) & = & \mathtt{fv}(A) \cup \mathtt{fv}(B) & & \mathtt{bv}(AB) & = & \mathtt{bv}(A) \cup \mathtt{bv}(B) \\ \mathtt{fv}(\lambda x.A) & = & \mathtt{fv}(A) \backslash \{x\} & & \mathtt{bv}(\lambda x.A) & = & \mathtt{bv}(A) \cup \{x\} \end{array}$$

Une variable x est dite libre dans A si $x \in \mathtt{fv}(A)$. Une variable qui n'est pas libre est dite liée. Un λ-terme qui ne contient pas de variables libres est dit clos.

Nous considérons les termes modulo α-conversion c'est-à-dire modulo renommage des variables liées. De plus nous supposons la condition d'hygiène de Barendregt : les variables libres et liées ont des noms différents.

Définition 1.5 (Positions dans les λ-termes) *L'ensemble des positions dans les λ-termes est l'ensemble $\{0,1\}^*$, c'est-à-dire l'ensemble des mots construits sur l'alphabet $\{0,1\}$. L'opérateur de concaténation sur les mots (et donc sur les positions) est dénoté par la juxtaposition. Le mot vide est dénoté ϵ. On utilise la lettre q (éventuellement indexée) pour désigner une position d'un λ-terme.*

Nous utilisons le symbole q pour les positions ; la lettre p sera utilisée dans la suite pour les étiquettes. Nous utiliserons sans ambiguïté le symbole ϵ pour dénoter le mot vide et le symbole ε pour dénoter un atome.

Définition 1.6 (Ordre sur les positions) *L'ordre sur les positions dans les λ-termes noté $\preceq$ est défini par $q_1 \preceq q_2$ s'il existe une position q' telle que $q_1 q' = q_2$.*

Définition 1.7 (Positions d'un λ-terme) *L'ensemble $\mathcal{P}os(A)$ des positions d'un λ-terme A est défini par induction sur l'ensemble des termes :*
- $\mathcal{P}os(x) = \{\epsilon\}$
- $\mathcal{P}os(c) = \{\epsilon\}$
- $\mathcal{P}os(\lambda x.A_0) = \{\epsilon\} \cup \{0q_0 | q_0 \in \mathcal{P}os(A_0)\}$
- $\mathcal{P}os(A_0A_1) = \{\epsilon\} \cup \{0q_0 | q_0 \in \mathcal{P}os(A_0)\} \cup \{1q_1 | q_1 \in \mathcal{P}os(A_1)\}$

La position ϵ est souvent appelée la position de tête.

Définition 1.8 (Sous-terme) *Soit A un λ-terme et soit q un élément de $\mathcal{P}os(A)$. Le sous-terme de A à la position q, dénotée A_q est défini par :*
- $A|_\epsilon = A$
- $(\lambda x.A_0)|_{0q_0} = A_0|_{q_0}$
- $(A_0A_1)|_{0q_0} = A_0|_{q_0}$
- $(A_0A_1)|_{1q_1} = A_1|_{q_1}$

Définition 1.9 (Substitutions) *Soient A et B deux termes bien typés. La substitution de x par A dans B est dénotée $B[x := A]$ et définie par*
- $x[x := A] = A$
- $y[x := A] = y$ *si* $y \neq x$
- $a[x := A] = a$
- $(\lambda y.B_0)[x := A] = \lambda y.(B_0[x := A])$
- $(B_0B_1)[x := A] = (B_0[x := A])(B_1[x := A])$

Dans la définition précédente, le cas d'une λ-abstraction est correct puisque nous considérons les termes modulo α-conversion : un représentant adéquat est choisi pour éviter d'éventuelles captures de variables.

Définition 1.10 (Relation compatible) *Une relation $\to_{\mathcal{R}}$ sur l'ensemble des termes est compatible si :*

$$A \to_{\mathcal{R}} B \quad \textit{implique} \quad \begin{array}{lcl} A\,C & \to_{\mathcal{R}} & B\,C \\ C\,A & \to_{\mathcal{R}} & C\,B \\ \lambda x.A & \to_{\mathcal{R}} & \lambda x.B \end{array} \quad \textit{pour tout } C$$

Définition 1.11 (Notion de réduction) *Une notion de réduction est une relation binaire sur l'ensemble des termes.*

Définition 1.12 (Relation de réduction) *Une relation de réduction est la fermeture compatible de d'une notion de réduction.*

Définition 1.13 (Notion de β-réduction) *La notion de β-réduction est définie par*

$$(\lambda x.A)B \quad \to \quad A[x := B]$$

et nous notons $\to_\beta$ la relation de réduction associée.

La fermeture réflexive et transitive de $\to_\beta$ est notée $\twoheadrightarrow_\beta$. La fermeture réflexive, symétrique est notée $=_\beta$.

Dans ce manuscrit, nous confondrons sans ambiguïté la notion de réduction et la relation de réduction associée.

Définition 1.14 (Radical – Ordre d'un radical) *Un radical est un terme qui peut s'écrire sous la forme $(\lambda x.A)B$. L'ordre d'un tel radical est l'ordre du type du terme $\lambda x.A$.*

Théorème 1.15 (Normalisation forte de β [HS86]) *La relation de β-réduction termine sur l'ensemble des termes bien typés.*

Ce résultat a été prouvé pour la première fois par Turing (voir [Gan80]). Pour prouver ce théorème, plusieurs approches sont possibles. Par exemple, nous pouvons citer la méthode habituellement appelée méthode des candidats de réductibilité et basée sur les notions introduites par Tait [Tai67].

Théorème 1.16 (Confluence de β [Bar84]) *La β-réduction est confluente sur l'ensemble des termes bien typés.*

Il existe de nombreuses preuves différentes de ce théorème, voir par exemple [Bar84] et [HS86]. Le théorème 1.15 nous assure de l'existence d'une β-forme normale, le théorème 1.16 nous assure de son unicité. On parle ainsi de termes β-normaux et de la forme β-normale d'un terme.

1.2 Lambda-calcul pur et développements

Nous définissons les développements comme un sous-ensemble de la β-réduction qui ne réduit que les radicaux présents dans le terme initial et leurs résidus. Cette notion est formalisée par l'intermédiaire du λ-calcul souligné : on souligne les β-radicaux présents initialement dans un terme et la β-réduction est remplacée par la $\underline{\beta}$-réduction qui ne réduit que les radicaux soulignés. Ainsi, les radicaux créés au fur et à mesure ne sont pas réduits (puisqu'ils ne sont pas soulignés).

1.2.1 Lambda-calcul souligné

Nous définissons l'ensemble des termes soulignés. Nous surchargeons sans ambiguïté les notations du λ-calcul simplement typé pour le λ-calcul pur. Toutes les définitions précédentes n'utilisant pas les informations de type seront utilisées dans le λ-calcul pur sans être répétées ici.

Définition 1.17 (Termes soulignés) *Soient $\mathcal{X}$ un ensemble dénombrable et infini de variables et $\mathcal{K}$ un ensemble de constantes. L'ensemble des termes soulignés $\mathcal{T}_s$ est*

- *le plus petit ensemble contenant toutes les constantes et toutes les variables ;*
- *et clos par les règles suivantes :*
  - *Si* A *et* B *sont des éléments de l'ensemble $\mathcal{T}_s$ alors* (AB) *est un élément de $\mathcal{T}_s$;*
  - *Si* A *est un élément de $\mathcal{T}_s$ et* x *est une variable de $\mathcal{X}$, alors* $\lambda x.A$ *est un élément de $\mathcal{T}_s$;*
  - *Si* A *et* B *sont des éléments de l'ensemble $\mathcal{T}_s$ alors* $(\underline{\lambda} x.A)B$ *appartient à $\mathcal{T}_s$.*

L'ensemble des termes soulignés n'est pas clos par sous-terme : par exemple le terme $\underline{\lambda} x.A$ n'en est pas un élément.

Définition 1.18 (Relation de $\underline{\beta}$-réduction) *La relation de $\underline{\beta}$-réduction est définie sur l'ensemble des termes soulignés $\mathcal{T}_s$ par*

$$(\underline{\lambda} x.A)B \quad \rightarrow_{\underline{\beta}} \quad A[x := B]$$

La $\underline{\beta}$-réduction consomme à chaque étape de réduction un radical souligné, peut en dupliquer mais ne peut pas en créer. C'est ainsi une relation qui termine (théorème 1.21).

Exemple 1.19 (Terminaison) *Il n'existe pas de réduction infinie à partir du terme $(\underline{\lambda} x.xx)(\lambda x.xx)$. Son $\underline{\beta}$-réduit $(\lambda x.xx)(\lambda x.xx)$ est bien en forme $\underline{\beta}$-normale puisqu'il ne contient pas de radical souligné.*

Exemple 1.20 (Duplication et réduction d'un radical) *Un radical souligné peut être dupliqué et ensuite chaque duplicata peut être $\underline{\beta}$-réduit.*

$$\begin{aligned}(\underline{\lambda} x.xx)((\underline{\lambda} y.y)z) &\rightarrow_{\underline{\beta}} ((\underline{\lambda} y.y)z))((\underline{\lambda} y.y)z)) \\ &\rightarrow_{\underline{\beta}} z((\underline{\lambda} y.y)z)) \\ &\rightarrow_{\underline{\beta}} zz\end{aligned}$$

Théorème 1.21 (Normalisation forte de $\underline{\beta}$ [Bar84]) *La relation de $\underline{\beta}$-réduction est terminante sur l'ensemble des termes soulignés.*

Théorème 1.22 (Confluence de $\underline{\beta}$ [Bar84]) *La $\underline{\beta}$-réduction est confluente sur l'ensemble des termes soulignés.*

1.2.2 Développements

Les termes du λ-calcul pur sont définis de la même manière que dans le λ-calcul simplement typé mais en enlevant toutes les contraintes sur les types. Nous donnons néanmoins leur définition.

Définition 1.23 (Termes) *Soient $\mathcal{K}$ un ensemble de constantes et $\mathcal{X}$ un ensemble de variables. L'ensemble des termes du λ-calcul, noté $\mathcal{T}$ est*
- *le plus petit ensemble contenant toutes les constantes et toutes les variables ;*
- *et clos par les règles suivantes :*
  - *Si A et B sont des éléments de l'ensemble $\mathcal{T}$ alors (AB) est un élément de $\mathcal{T}$;*
  - *Si A est un élément de $\mathcal{T}$ et x est une variable de $\mathcal{X}$, alors $\lambda x.A$ est un élément de $\mathcal{T}$.*

Nous pouvons remarquer qu'il existe une injection canonique de l'ensemble des termes bien typés $\mathcal{T}_t$ dans l'ensemble des termes $\mathcal{T}$.

Nous allons dans un premier temps considérer le λ-calcul pur avec un sous-ensemble de la β-réduction, sous-ensemble défini par l'intermédiaire du λ-calcul souligné. Nous définissons tout d'abord un morphisme d'effacement des soulignés.

Définition 1.24 (Morphisme d'effacement) *Le morphisme $\Upsilon : \mathcal{T}_s \to \mathcal{T}$ d'effacement des soulignés est défini par*
- $\Upsilon(x) = x$;
- $\Upsilon(c) = c$;
- $\Upsilon(\lambda x.A) = \lambda x.\Upsilon(A)$;
- $\Upsilon(AB) = \Upsilon(A)\Upsilon(B)$;
- $\Upsilon((\underline{\lambda} x.A)B) = (\lambda x.\Upsilon(A))\Upsilon(B)$.

Nous étendons le morphisme Υ défini sur les termes soulignés à toute suite de $\underline{\beta}$-réductions, ce qui nous permet de définir la notion de développements.

Définition 1.25 (Développements) *Une suite de β-réductions ζ dans le λ-calcul pur est un développement (aussi appelé développement complet) s'il existe une suite de $\underline{\beta}$-réductions dans le λ-calcul souligné σ qui termine sur un terme en forme $\underline{\beta}$-normale et telle que $\Upsilon(\sigma) = \zeta$.*

Nous donnons tout d'abord un exemple de β-réduction qui est un développement.

Exemple 1.26 (Développement) *La suite de β-réductions suivante est un développement dont la suite de β-réductions correspondante a été donnée dans l'exemple 1.20.*

$$\begin{array}{rll}(\lambda x.xx)((\lambda y.y)z) & \to_\beta & ((\lambda y.y)z))((\lambda y.y)z)) \\ & \to_\beta & z((\lambda y.y)z)) \\ & \to_\beta & zz\end{array}$$

Comme conséquence de la terminaison de la β-réduction (théorème 1.21), on obtient que tous les développements terminent. De plus, ils terminent tous sur le même terme.

Le théorème suivant a été prouvé pour la première fois pour le λ-calcul pur dans [Sch65] (une première preuve avait été donnée par Church et Rosser pour le λI-calcul).

Théorème 1.27 (Développements finis [Bar84])
- *Tous les développements sont finis.*
- *Si ζ_1 et ζ_2 sont deux développements complets avec le même terme initial, alors leurs termes finals sont égaux.*

1.2.3 Réduction parallèle

Nous introduisons dans cette section une définition à grand pas [Des98] des développements : la notion de réduction parallèle. Cette définition est initialement due à Tait et Martin-Löf.

Nous retrouverons par la suite plusieurs variantes de la réduction parallèle, toutes construites sur même schéma (une définition générale pour les systèmes d'ordre supérieur linéaire gauche peut être donnée, voir par exemple [Ter03]). On part d'une relation de réduction, ici la β-réduction, que l'on étend en une version parallèle qui autorise à réduire simultanément tous les radicaux d'un terme. En particulier, on peut réduire aucun radical et donc la relation obtenue est réflexive.

Ceci est obtenu par une règle de réflexivité pour les atomes, ici la règle $(\mathtt{Red}-\varepsilon)$, par des règles de congruence, ici les règles $(\mathtt{Red}-\lambda)$ et $(\mathtt{Red}-@)$, et par la version parallèle de la réduction initiale, ici la règle $(\mathtt{Red}-\beta)$.

Notons qu'il est équivalent de considérer la règle de réflexivité pour tous les termes (et non pas uniquement sur les atomes).

Définition 1.28 (Réduction parallèle) *La relation de réduction parallèle définie sur l'ensemble des termes et notée $\Longrightarrow_\beta$ est définie inductivement dans la figure 1.1.*

Théorème 1.29 (Développements et réduction parallèle [Bar84]) *Les notions de développements et de réduction parallèle coïncident :*

Pour tous termes $A, B \in \mathcal{T}$

$$\textit{il existe un développement } A \twoheadrightarrow_\beta B \textit{ ssi } A \Longrightarrow_\beta B$$

$$\frac{}{\varepsilon \Longrightarrow_\beta \varepsilon}\ (\mathsf{Red}-\varepsilon) \qquad \frac{A_1 \Longrightarrow_\beta A_2}{\lambda x.A_1 \Longrightarrow_\beta \lambda x.A_2}\ (\mathsf{Red}-\lambda)$$

$$\frac{A_1 \Longrightarrow_\beta A_2 \quad B_1 \Longrightarrow_\beta B_2}{A_1B_1 \Longrightarrow_\beta A_2B_2}\ (\mathsf{Red}-@) \qquad \frac{\lambda x.A_1 \Longrightarrow_\beta \lambda x.A_2 \quad B_1 \Longrightarrow_\beta B_2}{(\lambda x.A_1)B_1 \Longrightarrow_\beta A_2[x := B_2]}\ (\mathsf{Red}-\beta)$$

FIG. 1.1: Réduction parallèle dans le λ-calcul

1)	$((\lambda x.(\lambda y.A))B)C$	$\to_\beta$	$(\lambda y.A[x := B])C$
2)	$((\lambda x.x)(\lambda y.A))B$	$\to_\beta$	$(\lambda y.A)B$
3)	$(\lambda x.A)(\lambda y.B)$	$\to_\beta$	$A[x := (\lambda y.B)]$ s'il existe une position q tel que $A\|_q = xA_0$

FIG. 1.2: Création de radicaux dans le λ-calcul

1.3 Lambda-calcul pur et super-développements

Nous avons vu précédemment que les développements dans le λ-calcul pur ne réduisent que les radicaux présents initialement dans le terme ainsi que leurs résidus. La notion de développement complet donne une première approximation de la β-forme normale.

Dans la suite, nous étudions une approximation plus fine dont la définition est liée à la classification [Lév78] des créations de radicaux dans le λ-calcul, proposée par J.J. Lévy.

Nous allons voir que les super-développements généralisent la notion de développements en réduisant en plus les radicaux créés qui n'ont pas été obtenus par la substitution d'une variable en position fonctionnelle par une λ-abstraction.

1.3.1 Créations de radicaux dans le λ-calcul

Dans le λ-calcul, on distingue trois manières de créer des radicaux par β-réduction que nous donnons dans la figure 1.2 et informellement dans la figure 1.3. Les numéros indiqués dans la figure 1.3 sont utilisés pour l'instant informellement mais correspondent à des termes étiquetés que nous définirons plus tard. On peut remarquer que dans chaque cas, c'est la réduction de $(\lambda^1.\ )@^1$ qui crée un radical étiqueté $(\lambda^2.\ )@^2$.

Une suite de β-réductions qui ne réduit que les résidus des radicaux présents dans le terme initial et les radicaux créés de la première ou deuxième manière est un super-

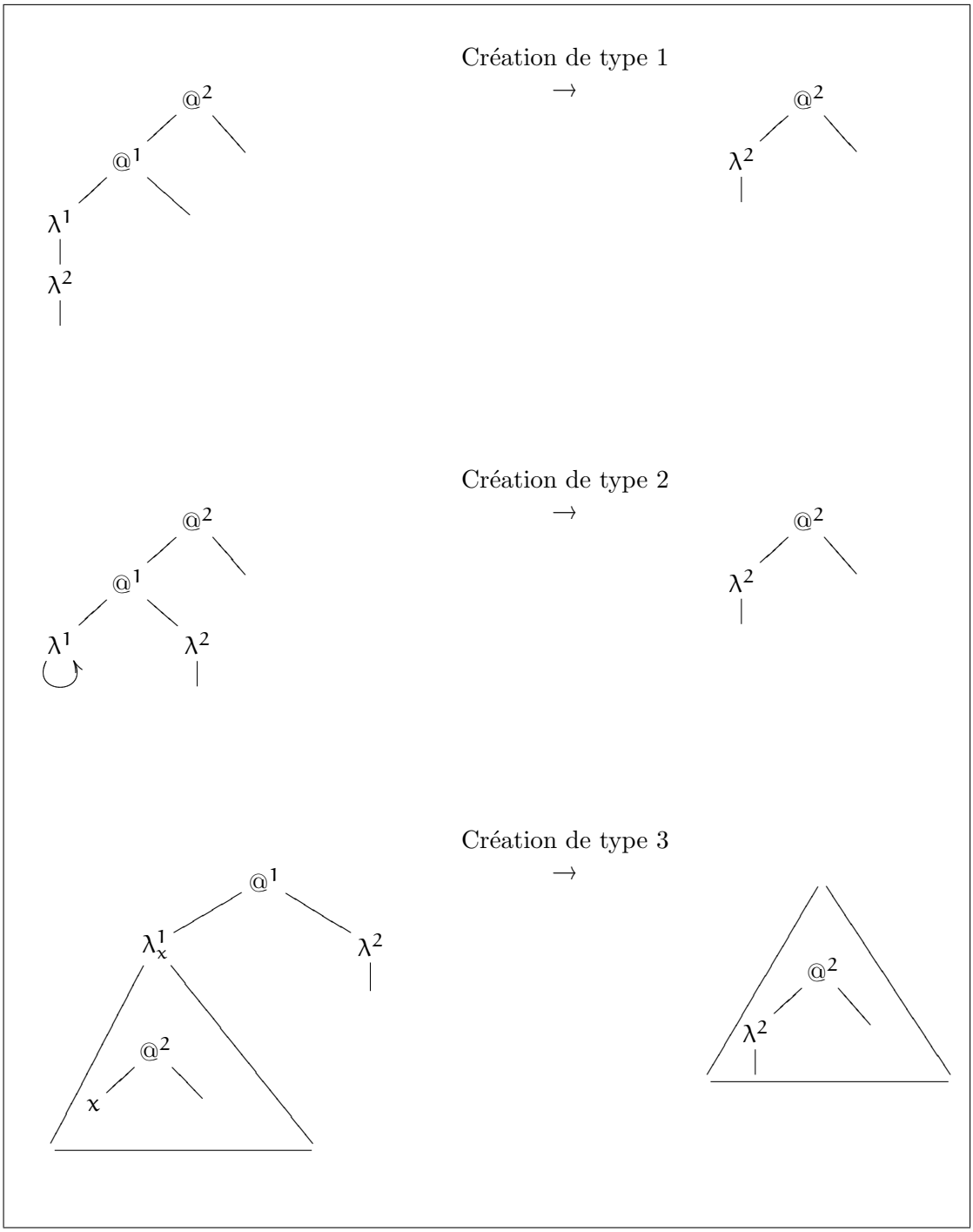


FIG. 1.3: Création de radicaux dans le λ-calcul

développement. Tous les super-développements, comme les développements, sont finis.

De plus, les créations de type 1 et 2 se font « par le haut » alors que la création de type 3 se fait « par le bas ». C'est précisément cette remarque qui justifie intuitivement la restriction de termes bien étiquetés définie ci-dessous.

Avant de définir le λ-calcul étiqueté, nous insistons sur la différence entre la création de radicaux de type 1 ou 2 et de type 3. Cette remarque sera utile par la suite pour définir la notion adéquate de réduction parallèle qui correspond aux super-développements.

Remarque 1.30 (Création de type 1&2 vs de type 3) *Les créations de radicaux de type 1 ou 2 et de type 3 se distinguent par la remarque suivante : dans le premier cas c'est la* réduction *du terme en position fonctionnelle de l'application étiquetée 2 qui crée le radical. Dans la création de type 3, c'est la* substitution *d'une variable en position fonctionnelle par une λ-abstraction qui crée le radical.*

1.3.2 Lambda-calcul étiqueté

On généralise l'ensemble des termes soulignés en un ensemble de termes étiquetés. Les étiquettes sont des entiers naturels, c'est-à-dire des éléments de $\mathbb{N}$.

Définition 1.31 (Termes étiquetés) *Soient $\mathcal{K}$ un ensemble de constantes et $\mathcal{X}$ un ensemble infini dénombrable de variables. L'ensemble des termes étiquetés, noté $\mathcal{T}_e$ est*

- *le plus petit ensemble contenant toutes les constantes et toutes les variables*
- *et clos par les règles suivantes :*
  - *si A est un élément de l'ensemble $\mathcal{T}_e$ et p est un élément de $\mathbb{N}$ alors $\lambda_p x.A$ est un élément de $\mathcal{T}_e$.*
  - *si A et B sont des éléments de $\mathcal{T}_e$ et p un élément de $\mathbb{N}$, alors $(AB)^p$ est un élément de $\mathcal{T}_e$.*

Définition 1.32 (Relation de β_e-réduction) *La relation de β_e-réduction est définie sur l'ensemble des termes étiquetés par*

$$((\lambda_p A.)B)^p \quad \to_{\beta_e} \quad A[x := B]$$

Lorsque l'étiquette d'une application n'intervient pas dans les réductions, nous pouvons sans ambiguïté ne pas la donner explicitement.

Pour définir la notion de super-développements nous nous restreignons aux termes bien étiquetés, qui limite la réduction des radicaux créés « par le haut », comme remarqué précédemment.

Définition 1.33 (Termes bien étiquetés) *Un terme étiqueté A de l'ensemble $\mathcal{T}_e$ est dit bien étiqueté si pour toutes positions q_1 et q_2 telles que $A|_{q_1} = (B_0 B_1)^p$ et $A|_{q_2} = \lambda_p x.C$ alors nécessairement $q_1 \preceq q_2$.*

Il n'est pas difficile de remarquer que l'ensemble des termes bien étiquetés est clos par β_e-réduction. Dans la suite, on supposera que tous les termes étiquetés sont bien étiquetés.

Définition 1.34 (Termes initialement bien étiquetés) *Un terme est dit initialement bien étiqueté :*
- *si c'est un terme bien étiqueté ;*
- *si pour toutes positions $\mathfrak{q}_1$ et $\mathfrak{q}_2$ telles que $A|_{\mathfrak{q}_1} = \lambda_p x.C$ et $A|_{\mathfrak{q}_2} = \lambda_p x'.C'$ alors nécessairement $\mathfrak{q}_1 = \mathfrak{q}_2$.*

Théorème 1.35 (Normalisation forte de β_e [Raa93]) *La β_e-réduction est terminante sur l'ensemble des termes étiquetés.*

Théorème 1.36 (Confluence de β_e [Raa93]) *La β_e-réduction est confluente sur l'ensemble des termes étiquetés.*

1.3.3 Super-développements

Nous définissons tout d'abord une correspondance entre le λ-calcul avec étiquette et le λ-calcul.

Définition 1.37 (Morphisme d'effacement) *Le morphisme $\Upsilon : \mathcal{T}_e \to \mathcal{T}$ d'effacement des étiquettes est défini comme suit*

1. $\Upsilon(x) = x$;
2. $\Upsilon(c) = c$;
3. $\Upsilon(AB)^p = \Upsilon(A)\Upsilon(B)$;
4. $\Upsilon(\lambda_p x.A) = \lambda x.\Upsilon(A)$.

On peut étendre le morphisme Υ à toute suite de β_e-réductions, ce qui nous permet de définir la notion de super-développements.

Définition 1.38 (Super-développement) *Une suite de β-réductions ζ du λ-calcul pur est un super-développement (aussi appelé super-développement complet) s'il existe une suite de β_e-réductions σ du λ-calcul étiqueté qui termine sur une β_e-forme normale et telle que $\Upsilon(\sigma) = \zeta$.*

Exemple 1.39 (Super-développement) *La suite de β-réductions suivante :*

$$(\lambda x.\lambda y.xy)zz' \quad \to_\beta \quad (\lambda y.zy)z' \quad \to_\beta \quad zz'$$

est un super-développement puisqu'elle correspond à la suite de β_e-réductions suivante :

$$(((\lambda_1 x.\lambda_2 y.xy)z)^1)z')^2 \quad \to_{\beta_e} \quad ((\lambda_2 y.zy)z')^2 \quad \to_{\beta_e} \quad zz' \ .$$

Exemple 1.40 (Super-développement) *La suite de β-réductions suivante :*

$$((\lambda x.x)(\lambda y.y))z \quad \to_\beta \quad (\lambda y.y)z \quad \to_\beta \quad z$$

est un super-développement puisqu'elle correspond à la suite de β_e-réductions suivante :

$$(((\lambda_1 x.x)(\lambda_2 y.y))^1 z)^2 \quad \to_{\beta_e} \quad ((\lambda_2 y.y)z)^2 \quad \to_\beta \quad z$$

Exemple 1.41 (Super-développement) *La suite de β-réductions suivante :*

$$(\lambda x.xx)(\lambda x.xx) \quad \to_\beta \quad (\lambda x.xx)(\lambda x.xx) \quad \to_\beta \quad \dots$$

n'est pas un super-développement.

Essayons en effet « d'étiqueter » le terme $(\lambda x.xx)(\lambda x.xx)$ *et de trouver la suite de* β_e*-réductions adéquate. Nous allons voir étape par étape que nous ne pouvons pas obtenir un terme bien étiqueté satisfaisant.*

Commençons par étiqueter les λ-abstractions. Comme on cherche un terme initialement bien étiqueté, chaque λ-abstraction doit avoir une étiquette différente. Sans perte de généralité on peut supposer que l'on a les étiquettes suivantes :

$$(\lambda_1 x.xx)(\lambda_2 x.xx).$$

Pour que le terme contienne un β_e*-radical nécessairement on doit avoir*

$$((\lambda_1 x.xx)(\lambda_2 x.xx))^1.$$

Pour conclure l'étiquetage de $(\lambda x.xx)(\lambda x.xx)$ *il nous reste à déterminer les étiquettes* p_1 *et* p_2 *telles que le terme*

$$((\lambda_1 x.(xx)^{p_1})(\lambda_2 x.(xx)^{p_2}))^1$$

soit bien étiqueté. Or si on souhaite obtenir un terme réductible après une étape de β-réduction, alors p_1 *est nécessairement égal à* 2 *; ce qui est impossible puisque l'on souhaite un terme bien étiqueté.*

Étant donné un λ-terme, on peut « l'étiqueter » d'une certaine manière (et donc obtenir un terme étiqueté) pour β_e-réduire les radicaux créés de la première et deuxième manière mais pas de la troisième. C'est exactement pour cela que l'on s'est restreint aux termes bien étiquetés.

À partir de maintenant, la suite de β_e-réductions associée à un super-développement ne sera pas directement explicitée. Nous pouvons remarquer de plus que comme β_e est fortement normalisant et confluent alors nous pouvons parler de la β_e-forme normale.

Exemple 1.42 (Radicaux et (super-)développements) *Nous donnons 4 exemples de suite de β-réductions qui sont des super-développements en explicitant les créations de radicaux mise en jeu.*

1. *Les résidus des radicaux présents dans le terme initial peuvent être contractés. Cette suite de β-réductions est aussi un développement.*

$$\begin{array}{ll} & (\lambda x.f(x,x))\ ((\lambda y.y)\ a) \\ \to_\beta & f((\lambda y.y)\ a, (\lambda y.y)\ a) \\ \to_\beta & f(a, (\lambda y.y)\ a) \\ \to_\beta & f(a,a) \end{array}$$

2. *La première réduction de la séquence suivante crée un radical (création de type 1). Ce radical est ensuite réduit dans le super-développement suivant :*

$$\begin{array}{ll}((\lambda x.\lambda y.f(x,y))a)b \\ \rightarrow_\beta & (\lambda y.f(a,y))b \\ \rightarrow_\beta & f(a,b)\end{array}$$

3. *Comme dans la réduction précédente, un radical est créé par réduction mais d'une manière différente (création de type 2).*

$$\begin{array}{ll}((\lambda x.x)(\lambda y.y))a \\ \rightarrow_\beta & (\lambda y.y)a \\ \rightarrow_\beta & a\end{array}$$

4. *Il n'existe pas de super-développement du terme* $(\lambda x.xa)(\lambda y.y)$ *vers le terme* a*. Le seul super-développement envisageable serait le suivant.*

$$\begin{array}{ll}(\lambda x.xa)(\lambda y.y) \\ \rightarrow_\beta & (\lambda y.y)a\end{array}$$

La création de radical est de type 3. Le radical créé ne peut pas être réduit par super-développement.

Théorème 1.43 (Super-développements finis [Raa96])
- *Tous les super-développements sont finis.*
- *Si* ζ_1 *et* ζ_2 *sont deux super-développements complets avec le même terme initial, alors leurs termes finals sont égaux.*

1.3.4 Réduction parallèle forte

On peut généraliser la notion de réduction parallèle (qui coïncide avec la notion de développements) pour obtenir de la même manière une correspondance avec la notion de super-développements. Historiquement, la notion de réduction parallèle forte introduite dans [Acz78] apparaît antérieurement à la notion de super-développements.

Définition 1.44 (Réduction parallèle forte) *La réduction parallèle forte définie sur l'ensemble des termes* $\mathcal{T}$ *et notée* $\Longrightarrow_{\beta_f}$ *est la plus petite relation close par les règles donnée dans la figure 1.4.*

Nous disons simplement que le terme A se β_{sd}-réduit sur le terme B s'il existe un super-développement entre A et B. Dans la définition de la réduction parallèle forte, la règle $(\mathsf{Red}\text{-}\beta_f)$ est venue remplacer la règle $(\mathsf{Red}\text{-}\beta)$ de la réduction parallèle, différence significative qui fait passer des développements aux super-développements.

La règle $(\mathsf{Red}\text{-}\beta_f)$ réduit le radical formé à partir de la λ-abstraction donnée par le *réduit* du terme A_1. Cette λ-abstraction a donc bien été obtenue *après* réduction de A_1 (par opposition à la règle $(\mathsf{Red}\text{-}\beta)$ qui réduit le radical présent dans le terme initial). Le radical ainsi réduit par la règle n'était donc pas nécessairement présent dans le terme initial A_1A_2. Il a donc été éventuellement créé par réduction.

$$\frac{}{\varepsilon \Longrightarrow_{\beta_f} \varepsilon}\ (\mathsf{Red}-\varepsilon) \qquad \frac{A_1 \Longrightarrow_{\beta_f} A_2}{\lambda x.A_1 \Longrightarrow_{\beta_f} \lambda x.A_2}\ (\mathsf{Red}-\lambda)$$

$$\frac{A_1 \Longrightarrow_{\beta_f} A_2 \quad B_1 \Longrightarrow_{\beta_f} B_2}{A_1B_1 \Longrightarrow_{\beta_f} A_2B_2}\ (\mathsf{Red}-@) \qquad \frac{A_1 \Longrightarrow_{\beta_f} \lambda x.A_2 \quad B_1 \Longrightarrow_{\beta_f} B_2}{A_1B_1 \Longrightarrow_{\beta_f} A_2[x := B_2]}\ (\mathsf{Red}-\beta_f)$$

FIG. 1.4: Réduction parallèle forte dans le λ-calcul

Définition 1.45 (β_{sd}-forme normale) *La β_{sd}-forme normale d'un terme A est le terme B tel que*

1. $A \Longrightarrow_{\beta_f} B$
2. *il n'existe pas de terme $B' \neq B$ tel que $A \Longrightarrow_{\beta_f} B'$ et $B \Longrightarrow_{\beta_f} B'$.*

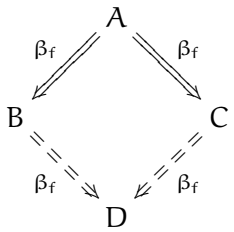
La β_{sd}-forme normale est obtenue en appliquant prioritairement la règle $(\mathsf{Red}\text{-}\beta_f)$. La remarque 1.30 sur la différence entre les différentes créations de radicaux conduit au théorème suivant.

Théorème 1.46 (Super-développements et réduction parallèle forte [Raa96]) *Pour tous termes A et B de l'ensemble $\mathcal{T}$*

$$\textit{il existe un super-développement } A \twoheadrightarrow_\beta B \textit{ ssi } A \Longrightarrow_{\beta_f} B$$

Du théorème précédent et de la confluence de β_e on en déduit que la relation $\Longrightarrow_{\beta_f}$ vérifie la propriété du diamant.

Propriété 1.47 (Propriété du diamant pour $\Longrightarrow_{\beta_f}$) *Soient A, B, C des termes de $\mathcal{T}$. Si $A \Longrightarrow_{\beta_f} B$ et $A \Longrightarrow_{\beta_f} C$ alors il existe un terme D tel que $B \Longrightarrow_{\beta_f} D$ et $C \Longrightarrow_{\beta_f} D$.*



Conclusion

Le matériel de base introduit dans ce chapitre sera utilisé tout au long de la première partie de cette thèse. Nous allons voir que les super-développements conduisent à une nouvelle approche pour le filtrage d'ordre supérieur.

Chapitre 2

Filtrage d'ordre supérieur

Contexte L'égalité de deux termes modulo β est un problème indécidable dans le λ-calcul, comme l'a montré Church. L'unification et le filtrage dans le λ-calcul pur ne peuvent donc pas être étudier directement puisqu'ils nécessitent de décider de l'égalité de deux termes. Néanmoins, en pratique on n'a pas besoin de toute la puissance du λ-calcul pur et dans le cadre de la déduction automatique par exemple on étudie l'unification dans le cadre du λ-calcul simplement typé (isomorphisme de Curry-Howard-de Bruijn). L'unification reste dans ce contexte toujours indécidable [Hue76] mais comme l'a montré D. Miller dans [Mil91] les termes que l'on écrit en pratique vérifient souvent des contraintes qui font que l'unification devient décidable et peut même être résolue en temps linéaire [Qia96].

Le travail présenté ici concerne le filtrage d'ordre supérieur qui requiert une étude particulière bien qu'étant un cas particulier de l'unification (voir par exemple [HL78]). Il est utilisé principalement pour la réécriture d'ordre supérieur [KOR93, MN98, Ter03, BCK06] et pour la transformation de programmes [MS01, Hag90, HM88, HL78]. Néanmoins, les algorithmes utilisés sont souvent une spécialisation de l'algorithme général d'unification tel qu'il a été introduit dans [Hue76] et sont complexes à comprendre et mettre en œuvre.

Contributions Nous proposons dans ce chapitre et le chapitre suivant une approche nouvelle au filtrage d'ordre supérieur. Au lieu de décider de l'égalité modulo β dans le λ-calcul typé, nous proposons de considérer une restriction décidable de la β-équivalence dans un cadre non typé. Cette restriction est donnée par les super-développements qui ont été introduits au chapitre 1.

Elle peut paraître arbitraire et restrictive mais remarquablement, l'approximation donnée par les super-développements coïncide avec la β-forme normale pour les termes du second-ordre. Autrement dit, l'ensemble des filtres modulo super-développements contient les filtres du second-ordre. Ceci est particulièrement important puisqu'en pratique, les problèmes que l'on considère sont très souvent du second ordre. Nous montrons aussi que les super-développements sont suffisamment expressifs pour traiter les problèmes de filtrage sur des motifs à la Miller.

Le filtrage d'ordre supérieur dans un cadre non typé a été étudié pour la première fois dans [Sit01, MS01]. Les équations de filtrage sont résolus modulo une réduction atomique (une étape) qui ne coïncident pas avec la notion de réduction parallèle de Tait and Martin-Löf. Elle termine néanmoins et fournit une approximation à l'opération

de β-normalisation. La réduction atomique considérée est comme l'indique les auteurs du papier original, difficile à comprendre. La théorie des super-développements donne des intuitions claires et un cadre formel solide permettant une comparaison simple avec les autres approches du filtrage d'ordre supérieur. Les preuves par rapport aux travaux initiaux [Sit01, MS01] sont considérablement simplifiées.

Nous verrons dans le chapitre suivant que l'algorithme et les concepts pour le filtrage modulo super-développements étant clarifiés, il est maintenant simple de déduire plusieurs propriétés et variantes de cet algorithme. En particulier, nous obtenons un algorithme original pour le filtrage du second-ordre et des propriétés de minimalité dans le cadre des motifs à la Miller.

Les principales idées de ce chapitre et du suivant ont été présentées dans [Fau06]. Une version détaillée proche de la présentation donnée ici est disponible dans [Fau07].

Plan du chapitre Dans une section préliminaire, nous introduisons les variables de filtrage qui seront les inconnues des équations que nous allons résoudre. Nous définissons ensuite le filtrage d'ordre supérieur dans le λ-calcul simplement typé (modulo β section 2.1.1 et modulo βη section 2.1.3) et le filtrage modulo super-développements dans le λ-calcul pur (section 2.2.1 et section 2.2.5 avec η). Nous comparons ce dernier avec différentes instances du filtrage dans le λ-calcul simplement typé : filtrage du second ordre (section 2.2.2), du troisième ordre (section 2.2.3) et des motifs à la Miller (section 2.2.4).

Variables de filtrage

En général, une équation de filtrage est définie par la donnée d'un couple de termes. Un des deux termes est clos. Les variables libres de l'autre terme sont les inconnues du problème de filtrage, c'est-à-dire les variables que l'on cherche à instancier afin de rendre les deux termes égaux modulo une équivalence donnée (traditionnellement, pour le λ-calcul simplement typé, la βη-équivalence).

Nous proposons une présentation légèrement différente, où les inconnues appartiennent à un ensemble séparé. Autrement dit, dans toute la suite du chapitre, aussi bien dans le cadre typé que non typé, nous supposerons donnés deux ensembles disjoints de variables : l'ensemble $\mathcal{X}$ défini comme dans le chapitre 1 et l'ensemble $\mathcal{V}$, appelé ensemble des variables de filtrage et qui représentent les inconnues.

Les variables de filtrage seront dénotées par les lettres $\mathsf{X}, \mathsf{Y}, \mathsf{Z} \ldots$. Elles ne peuvent pas être liées par une λ-abstraction. Elles seront substituées durant la résolution d'un problème de filtrage. Dans le cadre typé, chaque variable de filtrage a un type donné et nous supposerons que $\mathcal{V}$ peut être partitionné en des ensembles disjoints de variables de filtrage d'un type donné.

Ainsi l'ensemble des termes (non typés) est défini par

$$
\begin{array}{rcll}
A, B & ::= & x & \text{(Variable)} \\
& | & X & \text{(Variable de filtrage)} \\
& | & c & \text{(Constante)} \\
& | & \lambda x.A & \text{(Abstraction)} \\
& | & A\ B & \text{(Application)}
\end{array}
$$

Afin de lever toute ambiguïté, nous rappelons la définition des variables libres pour souligner qu'elle ne concerne pas les variables de filtrage.

Définition 2.1 (Variables libres et liées) *L'ensemble des variables libres d'un terme* A *noté* $\mathtt{fv}(A)$ *et l'ensemble des variables liées noté* $\mathtt{bv}(A)$ *sont définis par*

$$
\begin{array}{rclcrcl}
\mathtt{fv}(x) & = & \{x\} & \qquad & \mathtt{bv}(x) & = & \emptyset \\
\mathtt{fv}(X) & = & \emptyset & & \mathtt{bv}(X) & = & \emptyset \\
\mathtt{fv}(a) & = & \emptyset & & \mathtt{bv}(a) & = & \emptyset \\
\mathtt{fv}(AB) & = & \mathtt{fv}(A) \cup \mathtt{fv}(B) & & \mathtt{bv}(AB) & = & \mathtt{bv}(A) \cup \mathtt{bv}(B) \\
\mathtt{fv}(\lambda x.A) & = & \mathtt{fv}(A) \backslash \{x\} & & \mathtt{bv}(\lambda x.A) & = & \mathtt{bv}(A) \cup \{x\}
\end{array}
$$

Définition 2.2 (Terme $\mathcal{X}$-clos et terme $\mathcal{V}$-clos) *Un terme* A *est dit* $\mathcal{X}$*-clos s'il ne contient pas de variable libre, c'est-à-dire si* $\mathtt{fv}(A) = \emptyset$.

Un terme est dit $\mathcal{V}$*-clos s'il ne contient pas de variable de filtrage.*

Définition 2.3 (Substitution) *Une substitution de variables de filtrage ou plus simplement une substitution est une fonction de l'ensemble des variables de filtrage dans l'ensemble des termes* $\mathcal{V}$*-clos (typés ou non suivant le contexte). La substitution de la variable de filtrage* X *par* A *dans* B *est dénotée par* $B\{A/X\}$.

Si l'on avait utilisé la terminologie des CRS [Klo80, KOR93], on aurait appelé « méta-variables » les variables de filtrage, « méta-termes » les termes contenant éventuellement des variables de filtrage et « termes » nos termes $\mathcal{V}$-clos. Les substitutions des variables de filtrage pour des termes $\mathcal{V}$-clos correspondent bien aux « assignement » des CRS.

Dans ce chapitre, nous ne considérons que des *substitutions normales*, c'est-à-dire des substitutions de termes normaux. Nous notons par $\mathtt{id}$ la substitution identité (aussi appelée substitution vide).

Comme nous considérons les termes modulo α-conversion, lorsque l'on applique une substitution de variable de filtrage, un représentant adéquat est choisi pour éviter d'éventuelles captures de variables.

De la même manière qu'il existe une injection canonique de l'ensemble des termes typés dans l'ensemble des termes, il existe une injection canonique des substitutions définies sur les termes typés $\mathcal{T}_t$ dans l'ensemble des substitutions définies sur l'ensemble des termes non typés $\mathcal{T}$. Cette injection ne sera pas explicitement indiquée.

Définition 2.4 (Domaine, co-domaine) *Si* $\phi = \{A_1/X_1, \ldots, A_n/X_n\}$ *est une substitution alors son domaine est l'ensemble* $\{X_i\}_{i=1}^n$ *et son co-domaine est l'ensemble* $\{A_i\}_{i=1}^n$.

Nous définissons l'union de deux substitutions, union bien définie uniquement pour des substitutions qui coïncident sur l'intersection de leur domaine respectif.

Définition 2.5 (Union) *Deux substitutions sont compatibles si leur image coïncide sur l'intersection de leur domaine. On définit ainsi l'union de deux substitutions qui coïncident par :*

$$X(\sigma \cup \tau) = \begin{cases} X\sigma & \text{si } X \in \mathcal{D}om(\sigma) \\ X\tau & \text{si } X \in \mathcal{D}om(\phi) \end{cases}$$

Par exemple, les substitutions $\{A/X\}$ et $\{B/Y\}$ sont compatibles puisque leurs domaines sont disjoints.

2.1 Filtrage modulo β

Dans cette section, nous considérons des termes du λ-calcul simplement typé. Dans ce contexte typé, les substitutions de variables de filtrage sont toujours bien typées c'est-à-dire qu'à chaque variable de type α est forcément associée un terme de même type.

2.1.1 Définition

Nous définissons tout d'abord la nature des équations que nous allons résoudre.

Définition 2.6 (β-équation et β-système) *Une β-équation est un couple de termes bien typés en β-forme normale et de même type, notée $A \leqslant_\beta B$ et telle que B soit $\mathcal{V}$-clos. Un β-système $\mathbb{S}$, ou plus simplement système, est un multi-ensemble de β-équations.*

Nous considérons les notations classiques pour les multi-ensembles et nous notons par $\cup$ l'opération de multi-union. Si E_1 et E_2 sont deux équations de filtrage, nous notons simplement $(E_1) \cup (E_2)$ l'union des deux multi-ensembles singletons.

Considérons par exemple un type de base ι, une constante a de type ι et deux variables de filtrage X et Y de type respectif $\iota \to \iota$ et ι. Alors $XY \leqslant_\beta a$ est une β-équation.

Nous définissons ce qu'est une solution d'une β-équation.

Définition 2.7 (β-filtre) *Une substitution ϕ est un β-filtre pour l'équation $A \leqslant_\beta B$ si on a $A\phi =_\beta B$. Une substitution est un β-filtre pour un système $\mathbb{S}$ si elle est un β-filtre pour chacune des équations de $\mathbb{S}$.*

La substitution $\{\lambda x.x/X, a/Y\}$ est un β-filtre pour la β-équation $XY \leqslant_\beta a$.

2.1.2 Filtrage d'ordre n

En pratique, nous ne cherchons souvent à résoudre qu'un sous-ensemble de l'ensemble des β-équations en restreignant l'ordre des variables et des constantes considérées. En effet, dans le cas général (c'est-à-dire sans restriction d'ordre), le filtrage dans le λ-calcul simplement typé modulo β est indécidable [Loa03]. Nous verrons dans les sections suivantes comment se placer dans un cadre décidable.

Définition 2.8 (Ordre d'un système) *Une* β*-équation est dite d'ordre au plus* n *si toutes ses inconnues sont d'ordre au plus* n *et si toutes ses constantes sont d'ordre au plus* $n+1$*. Un système est dit d'ordre au plus* n *s'il est composé d'équations d'ordre au plus* n*.*

Par exemple, la β-équation $XY \leqslant_\beta a$ est d'ordre 2.

Définition 2.9 (Filtrage d'ordre n modulo β) *Nous désignons par filtrage d'ordre* n *modulo* β *la recherche des* β*-filtres de* β*-systèmes d'ordre au plus* n*.*

2.1.3 Filtrage modulo βη

En pratique l'égalité qui est intéressante est non pas l'égalité modulo β mais l'égalité modulo βη c'est-à-dire que l'on rajoute dans la théorie équationnelle l'équation

$$(\eta) \qquad \lambda x.(Ax) \quad = \quad A \quad \textbf{si } x \notin \mathtt{fv}(A)$$

La complexité du filtrage modulo βη a été abondamment étudiée. Sa décidabilité est restée un problème ouvert pendant plus de trente ans (conjecturée dans [Hue76]). La décidabilité à l'ordre 2 est établi dans [HL78], à l'ordre 3 dans [Dow94] et à l'ordre 4 dans [Pad00]. La preuve de décidabilité générale (sans restriction d'ordre) n'apparaît que dans les récents travaux de [Sti06] utilisant la théorie des jeux.

Nous reformulons les définitions données précédemment modulo βη.

Définition 2.10 (βη-équation et βη-système) *Une* βη*-équation est un couple de* λ*-termes bien typés en forme normale* η*-longue et de même type, notée* $A \leqslant_{\beta\eta} B$ *et telle que* B *soit* $\mathcal{V}$*-clos. Un* βη*-système* $\mathbb{S}$ *est un multi-ensemble de* βη*-équations.*

Définition 2.11 (βη-filtre) *Une substitution* ϕ *est un* βη*-filtre pour l'équation* $A \leqslant_{\beta\eta} B$ *si on a* $A\phi =_{\beta\eta} B$*. Une substitution est un* βη*-filtre pour un système* $\mathbb{S}$ *si elle est un* βη*-filtre pour chacune des équations de* $\mathbb{S}$*.*

2.1.4 Filtrage de motifs à la Miller

Nous définissons tout d'abord l'ensemble de motifs à la Miller.

Définition 2.12 (Motifs à la Miller) *Un* λ*-terme typé* A *en* β*-forme normale est un motif à la Miller si chaque occurrence d'une variable de filtrage est dans un sous-terme* $X(A_1, \ldots, A_n)$ *de* A *tel que* $(A_1, \ldots, A_n)$ *soit* η*-équivalent à une liste de variables liées distinctes.*

Par opposition à l'unification d'ordre supérieur qui est indécidable en générale [Hue73, Hue75, Gol81], l'unification de motifs à la Miller est décidable. De plus, quand un problème a une solution, il en existe une plus générale. La recherche de cette solution la plus générale peut être faite en un temps linéaire [Qia96]. Nous montrerons dans la section 3.2.2 que l'algorithme proposé pour le filtrage modulo super-développements et η donne exactement (et uniquement) le filtre le plus général pour toute équation ayant une solution.

2.2 Filtrage modulo super-développements

2.2.1 Définition

Nous considérons dans cette section des termes du λ-calcul pur, c'est-à-dire des éléments de $\mathcal{T}$. Nous allons considérer la résolution d'équations modulo la restriction de la β-réduction donnée par les super-développements qui a été introduite au chapitre 1.

Nous définissons comme dans la section précédente, tout d'abord la notion d'équation dans ce contexte. Ensuite, nous définissons ce que ce sont les solutions de telles équations.

Définition 2.13 (β_{sd}-équation et β_{sd}-système) *Une β_{sd}-équation est un couple de termes dénotée $A \leqslant_{\beta_{sd}} B$,telle que B est $\mathcal{V}$-clos et en β-forme normale. Un β_{sd}-système $\mathbb{S}$, ou plus simplement système, est un multi-ensemble (éventuellement vide) de β_{sd}-équations.*

On dit que la variable de filtrage X appartient à un système $\mathbb{S}$ et on note $X \in \mathbb{S}$ si cette variable apparaît dans une des équations du système $\mathbb{S}$.

Exemple 2.14 (β_{sd}-équations) *Par exemple, le couple $(XY, \lambda x.x)$ ainsi que le couple $((\lambda x.x)X, a)$ sont des β_{sd}-équations de filtrage alors que le couple $(XY, (\lambda x.x)a)$ n'en est pas une puisque le terme $(\lambda x.x)a$ n'est pas en forme normale.*

Comme l'indique l'exemple précédent, contrairement aux β-équations, le terme A d'une β_{sd}-équation $A \leqslant_{\beta_{sd}} B$ n'est pas supposé en forme normale, ce que nous justifierons dans la section 3.1 (paragraphe sur les règles (ε)).

Définition 2.15 (β_{sd}-filtre) *Une substitution ϕ est un β_{sd}-filtre pour $A \leqslant_{\beta_{sd}} B$ s'il existe un super-développement entre $A\phi$ et B (soit $A\phi \Longrightarrow_{\beta_f} B$). Une substitution est un β_{sd}-filtre pour un système $\mathbb{S}$ si elle est un β_{sd}-filtre pour chacune des équations qui constituent $\mathbb{S}$. L'ensemble des filtres d'un système $\mathbb{S}$ est noté $\mathbb{M}(\mathbb{S})$.*

De la même manière que l'on peut associer à tout terme typé un terme non typé, on peut associer à chaque β-équation une β_{sd}-équation. Par abus de notation, nous noterons simplement $A \leqslant_{\beta_{sd}} B$ la β_{sd}-équation associée à la β-équation $A \leqslant_{\beta} B$.

Rappelons que nous ne considérons que des substitutions normales et closes c'est-à-dire des substitutions de termes clos et normaux. En particulier un β_{sd}-filtre est donc bien une substitution normale et close.

Exemple 2.16 (β_{sd}-filtres) *Considérons l'équation*

$$XY \leqslant_{\beta_{sd}} \lambda x.x.$$

– *Les substitutions*

$$\sigma_1 = \{\lambda x.\lambda y.y/X\} \qquad et \qquad \sigma_2 = \{\lambda y.y/X, \lambda x.x/Y\}$$

sont des β_{sd}-filtres pour cette équation puisque

$$(XY)\sigma_1 = (\lambda x.\lambda y.y)Y \Longrightarrow_{\beta_f} \lambda x.x \qquad et \qquad (XY)\sigma_2 = (\lambda y.y)(\lambda x.x) \Longrightarrow_{\beta_f} \lambda x.x.$$

– *La substitution*

$$\{\lambda z.z(\lambda x.x)/X, \lambda y.y/Y\}$$

n'est pas un β_{sd}*-filtre puisque*

$$(\lambda z.z(\lambda x.x))(\lambda y.y) \not\Longrightarrow_{\beta_f} \lambda x.x$$

bien que ces deux-termes soient β*-convertibles.*

– *La substitution*

$$\sigma_3 = \{(\lambda y.y)(\lambda z.z)/X, \lambda x.x/Y\}$$

n'est pas un β_{sd}*-filtre puisqu'elle n'est pas normale.*

Nous allons voir qu'une équation peut avoir un nombre infini de β_{sd}-filtres, qu'il n'existe pas de filtre plus général mais qu'il existe toujours une partie génératrice (proposition 3.14).

Exemple 2.17 (Nombre infini de β_{sd}-filtres) *Nous continuons l'exemple 2.16. Nous pouvons remarquer que de la même manière que*

$$\sigma_1 = \{\lambda x.\lambda y.y/X\}$$

est un β_{sd}*-filtre pour l'équation* $XY \leqslant_{\beta_{sd}} \lambda x.x$*, toutes les substitutions qui sont compatibles avec* σ_1 *sont aussi solutions, quelle que soit la valeur associée à* Y *(il y en a donc une infinité). Nous pouvons citer par exemple les substitutions*

$$\sigma_2 = \{\lambda x.\lambda y.y/X, \lambda x.x/Y\} \qquad \textit{et} \qquad \sigma_3 = \{\lambda x.\lambda y.y/X, \lambda x.\lambda y.y/Y\}.$$

Ces deux substitutions seront « représentées » par la substitution σ_1 *dans le sens où :*

$$\sigma_1 \leq \sigma_2 \qquad \textit{et} \qquad \sigma_1 \leq \sigma_3$$

Par contre, il n'existe pas de filtre plus général comme le montre l'exemple suivant.

Exemple 2.18 (β_{sd}-filtres incomparables) *Nous continuons encore l'exemple 2.16. Une autre solution de l'équation considérée est donnée par*

$$\sigma_4 = \{\lambda x.\lambda y.x/X, \lambda x.x/Y\}.$$

Les deux substitutions σ_1 *et* σ_4 *sont incomparables.*

Nous cherchons à résoudre des β_{sd}-équations. Pour cela, nous allons transformer par étapes successives [MM82, Kir84] un système jusqu'à obtenir un système à partir duquel on peut extraire facilement un β_{sd}-filtre (s'il existe). Un tel système est un système dit en forme résolue dans le sens de la définition suivante.

Définition 2.19 (Forme résolue) *Une équation* $X \leqslant_{\beta_{sd}} A$ *est en forme résolue si le terme* A *est* $\mathcal{X}$*-clos. La substitution associée à une telle équation est définie par* $\{A/X\}$.

Un système est en forme résolue si chacune des équations qui le constituent sont en forme résolue et si les membres gauches des équations étant deux à deux disjoints. La substitution associée à un système résolu $\mathbb{S}$ *est l'union des substitutions associées à chacune des équations. Elle est dénotée* $\sigma_{\mathbb{S}}$.

Si le système est le multi-ensemble vide alors la substitution associée est la substitution identité.

La substitution associée à un système résolu est bien définie puisque c'est l'union de substitutions qui sont compatibles deux à deux (elles sont compatibles puisque leurs domaines sont disjoints, les membres gauches des équations étant deux à deux disjoints).

Nous allons donner dans la section 3.1 un algorithme pour la recherche des β_{sd}-filtres. Or, il peut exister un nombre infini de β_{sd}-filtres (exemple 2.17) et il n'existe pas de plus général filtre (exemple 2.18). C'est pour cela que nous introduisons la notion d'ensemble complet de filtres. Nous prouverons dans la proposition 3.14 que dans le cas du filtrage modulo super-développements dans le λ-calcul pur, si une équation a une solution alors il existe toujours un ensemble complet fini de filtres.

Définition 2.20 (Ensemble complet de filtres) *Soit* $\mathbb{S}$ *un ensemble d'équations. Un ensemble complet de filtre pour* $\mathbb{S}$ *est un ensemble de substitutions* $\mathbb{M}$ *de domaine inclus dans les variables de filtrage de* $\mathbb{S}$ *et tel que*

1. **Correction** *Pour toutes les substitutions* $\phi \in \mathbb{M}$, ϕ *est un* β_{sd}*-filtre de* $\mathbb{S}$ *;*
2. **Complétude** *Pour toutes les substitutions* ϕ *telle que* ϕ *est un* β_{sd}*-filtre pour* $\mathbb{S}$*, il existe* $\psi \in \mathbb{M}$ *telle que* $\psi \leq \phi$ *c'est-à-dire qu'il existe une substitution* ψ *telle que* $\phi = \xi \circ \psi$ *où* $\circ$ *dénote la composition de substitution.*

Lemme 2.21 (Correction des formes résolues) *Si un système* $\mathbb{S}$ *est en forme résolue alors* $\{\sigma_{\mathbb{S}}\}$ *est un ensemble complet de filtre pour* $\mathbb{S}$.

2.2.2 Comparaison avec le filtrage du second ordre

Nous avons vu qu'à toute β-équation correspond une β_{sd}-équation. En nous restreignant à des β-équations d'ordre au plus 2, nous allons montrer que tout β-filtre d'une telle équation est un β_{sd}-filtre pour la β_{sd}-équation correspondante.

Nous avons décrit dans le chapitre 1, les trois manières de créer un radical dans le λ-calcul. La troisième manière induit l'existence dans le terme initial d'un radical d'ordre au moins 3. Nous rappelons tout d'abord que les super-développements réduisent tous les radicaux résidus de radicaux présents dans le terme initial et tous les radicaux créés qui ne l'ont pas été par la troisième manière. Or, si l'on considère une β-équation du second ordre, il n'y aura jamais de radical d'ordre au moins 3 et donc jamais de création de radical de type 3. Autrement dit, dans ce contexte, la β-forme normale et la β_{sd}-forme normale coïncident et donc tout β-filtre est un β_{sd}-filtre. C'est ce que nous allons formaliser dans la suite de cette section.

Nous commençons par un lemme technique sur la création de radicaux.

Lemme 2.22 (Ordre d'un radical et super-développements) *Soient* $A_1, \dots A_{n-1}$ *et* A_n *des termes typés. Supposons de plus que* A_n *contienne un radical d'ordre au moins 3 et qu'il existe un super-développement* $A_1 \to_\beta \dots \to_\beta A_n$. *Alors* A_1 *contient aussi un radical d'ordre au moins 3.*

Preuve. Nous prouvons le résultat par induction sur n. Le résultat est vrai dans le cas de base ($n = 1$). Supposons maintenant que le résultat est vrai pour n. On veut le prouver pour $n + 1$. Par hypothèse d'induction, nous savons que A_2 contient un radical d'ordre au moins 3 que nous dénoterons par la suite par $R = (\lambda x.C)\, D$. Nous raisonnons par cas suivant suivant que R est un résidu ou a été créé durant la réduction de A_1 en A_2 en analysant les trois manières dont il a pu être créé.

- Si R est le résidu d'un radical de A_1 alors le résultat est vrai.
- Sinon, c'est-à-dire si le radical R est créé durant la réduction de A_1 en A_2 de la première manière alors le terme A_1 contient nécessairement un sous-terme de la forme $(((\lambda z.\lambda x.C')\, E)\, D$ avec $C = C'[z := E]$. L'ordre du radical $(\lambda z.\lambda x.C')\, E$ est au moins celui de R. Ce qui conclut le cas.
- Sinon, c'est-à-dire si R est créé durant la réduction de A_1 en A_2 de la deuxième manière alors A_1 doit contenir un sous-terme de la forme $(\lambda y.y)\, (\lambda x.C)\, D$. L'ordre du radical $(\lambda y.y)\, (\lambda x.C)$ est strictement supérieur à l'ordre de R. Ce qui conclut le cas.
- Sinon, c'est-à-dire si R est créé durant la réduction de A_1 en A_2 de la troisième manière alors A_1 doit contenir un sous-terme de la forme $R' = (\lambda z.E)(\lambda x.C)$ et il existe une position q telle que $E|_q = zD'$ avec $D = D'[z := \lambda x.C]$. L'ordre du radical R' est strictement supérieur à l'ordre de R. Ce qui conclut le cas.

□

Propriété 2.23 (Super-développements et second ordre) *Si* ϕ *est un* β*-filtre pour une* β*-équation du second ordre alors* ϕ *est un* β_{sd}*-filtre pour la* β_{sd}*-équation correspondante.*

Preuve. Raisonnons par l'absurde. Supposons qu'il existe une substitution ϕ qui soit un β-filtre de l'équation du second ordre $A \leqslant_\beta B$ et qui ne soit pas un β_{sd}-filtre de la β_{sd}-équation correspondante. En d'autres termes, A ne contient aucun radical, ϕ ne contient aucun terme d'ordre supérieur à deux et $A\phi =_\beta B$ et $A\phi \not\Longrightarrow_{\beta_f} B$. Il existe donc une suite $(A_i)_i$ telle que $A\phi \to_\beta A_1 \to_\beta \dots \to_\beta A_n$ et A_n contient un radical $(\lambda x.C)D$ qui ne peut pas être réduit dans le super-développement ci-dessus. C'est donc un résidu d'un radical créé en réduisant A_{i_0} (i_0 entre 1 et n). Comme il n'existe pas de β-séquence qui soit un super-développement et qui permette de réduire ce radical, la création est de type 3 et induit donc un radical d'ordre au moins 3. Le lemme 2.22 implique que $A\phi$ contient un radical d'ordre au moins 3. Mais comme A et ϕ appartiennent à l'ensemble des formes β-normales, il existe une position q de A et un terme E telle que $A|_q = XE$ où X est une variable de filtrage envoyée par ϕ sur une λ-abstraction d'ordre au moins 3, ce qui contredit l'hypothèse sur l'ordre de l'équation initiale. □

La proposition précédente se généralise aux systèmes.

Propriété 2.24 (Super-développements et second ordre) *Nous supposons donné un β-système du second ordre. Si ϕ est un β-filtre pour ce β-système alors ϕ est un β_{sd}-filtre pour le β_{sd}-système correspondant.*

Preuve. Considèronns le β-système du second ordre

$$(A_1 \leqslant_\beta B_1) \cup \ldots \cup (A_n \leqslant_\beta B_n).$$

Dire que ϕ est un β-filtre pour ce système est équivalent à dire que ϕ est un β-filtre pour la β-équation

$$\lambda x.x(A_1, \ldots, A_n) \leqslant_\beta \lambda x.x(B_1, \ldots, B_n)$$

où la variable x est du type adéquat et $x(A_1, \ldots, A_n)$ dénote le terme $(\ldots(x\,A_1)\ldots)$. En appliquant la proposition précédente, on obtient que ϕ est un β_{sd}-filtre pour la β_{sd}-équation

$$\lambda x.x(A_1, \ldots, A_n) \leqslant_{\beta_{sd}} \lambda x.x(B_1, \ldots, B_n)$$

ce qui est équivalent à dire que ϕ est un β_{sd}-filtre pour le β_{sd}-système

$$(A_1 \leqslant_{\beta_{sd}} B_1) \cup \ldots \cup (A_n \leqslant_{\beta_{sd}} B_n).$$

□

2.2.3 Comparaison avec le filtrage du troisième ordre

Dès que l'on considère des problèmes de filtrage du troisième ordre, l'ensemble des solutions minimales est potentiellement infini. Comme le filtrage modulo les super-développements ne génère qu'un nombre fini de solutions minimales (résultat démontré dans la proposition 3.14), on en déduit que le filtrage modulo les super-développements ne peut pas être complet par rapport au filtrage du troisième ordre.

Nous exhibons donc un exemple de β-équation qui contient des inconnues d'ordre 3 et qui admet une substitution qui est un β-filtre mais qui n'est pas un β_{sd}-filtre. Pour trouver un tel exemple il suffit de trouver deux termes bien typés dont la preuve de β-égalité nécessite de réduire un radical créé de la troisième manière. De ces deux termes, nous extrayons ensuite une équation.

Nous allons voir que l'équation considérée est du troisième ordre et admet [Dow01] un nombre infini de solutions minimales de type $\iota \to (\iota \to \iota) \to \iota$ qui sont exactement les nombres de Church $\lambda x.\lambda f.(f \ldots (f\,x) \ldots)$.

Exemple 2.25 (Super-développement et troisième ordre) *Considérons la suite de β-réductions suivante :*

$$\begin{array}{lll} \lambda z.((\lambda x.\lambda f.fx)\ z\ (\lambda y.y)) & \to_\beta & \lambda z.((\lambda f.fz)\lambda y.y) \\ & \to_\beta & \lambda z.((\lambda y.y)z) \\ & \to_\beta & \lambda z.z \end{array}$$

On peut remarquer que la dernière étape de réduction réduit un radical créé de la troisième manière. Donc ce radical ne peut pas être réduit par β_{sd}-réduction. On en déduit que la substitution $\{\lambda x.\lambda f.fx/X\}$ est un β-filtre mais n'est pas un β_{sd}-filtre pour l'équation $\lambda z.(X\ z\ (\lambda y.y)) \ll \lambda z.z$.

2.2.4 Comparaison avec le filtrage des motifs à la Miller

Nous allons prouver dans cette section, que la restriction de la β-réduction donnée par les super-développements est suffisante pour le filtrage de motifs à la Miller.

D. Miller souligne que pour les motifs à la Miller, il suffit de considérer une restriction de la β-réduction [Mil91], appelée β_0-réduction, et définie par

$$(\lambda x.M)x \qquad \to_{\beta_0} \qquad M$$

que l'on peut exprimer encore

$$(\lambda y.M)x \qquad \to_{\beta_0} \qquad M[y := x]$$

Nous avons remarqué dans le chapitre 1 qu'une création de radical de type 2 ou 3 nécessite la réduction d'un radical dont le terme en position applicative est une λ-abstraction. Comme la β_0-réduction ne permet de réduire que les radicaux dont le terme en position applicative est une variable, les seules créations de radicaux par β_0-réduction sont de type 1. Les radicaux créés de la première manière étant réduit par super-développements, on a le résultat suivant :

Propriété 2.26 (Motifs à la Miller et super-développements) *Si P un motif à la Miller et si ϕ un β-filtre pour l'équation $P \leqslant_\beta A$ alors il existe un super-développement $P\phi \twoheadrightarrow_\beta A$. La substitution ϕ est donc un β_{sd}-filtre pour cette équation.*

2.2.5 Filtrage modulo super-développements et η

Dans le λ-calcul simplement typé, la différence technique entre le filtrage d'ordre supérieur modulo β et le filtrage d'ordre supérieur modulo $\beta\eta$ est principalement expliquée par l'utilisation fondamentale de la forme normale η-longue. Cette différence est cruciale puisque l'utilisation de la règle η dans le filtrage permet de passer d'un problème indécidable [Loa03] à un problème décidable [Sti06].

Dans le contexte du filtrage d'ordre supérieur modulo les super-développements, l'utilisation de η n'influence ni fondamentalement ni techniquement le filtrage comme nous le montrons dans la section 3.2 en proposant un algorithme directement inspiré de l'algorithme modulo super-développements (sans η).

Dans le cadre typé, nous considérons des termes en formes normale η-longue. Dans un contexte non typé, nous allons au contraire considérer des termes en forme η-normale c'est-à-dire, des termes en forme normale par rapport à la η-réduction définie par

$$\lambda x.(A\,x) \quad \rightarrow_\eta \quad A \quad \text{si } x \notin \mathtt{fv}(A)$$

Définition 2.27 ($\beta_{sd}\eta$-équation et $\beta_{sd}\eta$-système) *Une $\beta_{sd}\eta$-équation de filtrage est définie comme un couple (A, B) de termes telle que B est $\beta\eta$-normal et $\mathcal{V}$-clos. On dénote une telle équation par $A \leqslant_{\beta_{sd}\eta} B$. Un $\beta_{sd}\eta$-système est un multi-ensemble de $\beta_{sd}\eta$-équations.*

Définition 2.28 ($\beta_{sd}\eta$-filtre) *On dit qu'une substitution ϕ est un $\beta_{sd}\eta$-filtre pour la $\beta_{sd}\eta$-équation $A \leqslant_{\beta_{sd}\eta} B$ s'il existe un terme C tel que $A\phi \Longrightarrow_{\beta_f} C \twoheadrightarrow_\eta B$.*

Ces deux notions seront illustrées dans la section 3.2.

Conclusion

Nous avons défini dans ce chapitre le filtrage modulo super-développements et nous l'avons comparé avec différentes instances du filtrage dans le λ-calcul simplement typé. Dans le chapitre suivant, nous allons donner différents algorithmes implémentant ces deux approches et et montrer leur correction et complétude.

Chapitre 3

Algorithmes de filtrage d'ordre supérieur

Contributions Après avoir défini dans le chapitre précédent le filtrage modulo $\beta(\eta)$ et modulo super-développements (η), nous donnons ici plusieurs algorithmes.

Nous présentons et étudions tout d'abord un algorithme pour le filtrage modulo super-développements. Cet algorithme présenté par des règles de transformations [MM82, Kir84, SG89, JK91] est déduit de la définition de la réduction parallèle forte (qui coïncident avec les super-développements). Nous étudions en détail les propriétés de cet algorithme : terminaison, correction et complétude.

Ensuite, nous proposons plusieurs variantes de cet algorithme qui aboutissent respectivement à un algorithme original pour le filtrage du second-ordre (approche différente de celle de Huet et Lang) et à un algorithme pour le filtrage modulo super-développements et η qui, dans le cadre du filtrage de motifs à la Miller, donne (s'il existe) le filtre principal.

Plan du chapitre Dans la première section qui est le cœur de ce chapitre, nous donnons et étudions un algorithme pour le filtrage modulo super-développements. Dans la deuxième section, nous donnons deux algorithmes pour le filtrage d'ordre 2. Le premier est dû à Huet et Lang et le deuxième est directement inspiré de l'algorithme pour le filtrage modulo super-développements de la première section. La troisième section propose un algorithme pour le filtrage modulo super-développements et η.

3.1 Filtrage modulo super-développements

Nous présentons dans cette section un algorithme pour le filtrage modulo super-développements dans le λ-calcul pur. Nous l'illustrons sur des exemples. Ensuite nous étudions ses différentes propriétés.

3.1.1 Régles

Les règles pour le filtrage modulo super-développements est donné dans la figure 3.1 en utilisant des règles de transformations [MM82, Kir84, SG89, JK91]. Nous entendons par règles de transformations des règles de réécriture qui ne sont appliquées qu'en tête. Ainsi,

– un système est transformé par étapes successives jusqu'à obtenir une forme normale (l'ensemble des règles est terminant) ; cette forme normale peut être une forme

$(A_1 B_1 \leqslant_{\beta_{sd}} A_2 B_2) \cup \mathbb{S}$	$\rightarrow_{@_@}$	$(A_1 \leqslant_{\beta_{sd}} A_2) \cup (B_1 \leqslant_{\beta_{sd}} B_2) \cup \mathbb{S}$
$(A_1 B_1 \leqslant_{\beta_{sd}} C) \cup \mathbb{S}$	$\rightarrow_{@_\pi}$	$(A_1 \leqslant_{\beta_{sd}} \lambda x.C) \cup \mathbb{S}$ **où** x **variable fraîche**
$(A_1 B_1 \leqslant_{\beta_{sd}} C) \cup \mathbb{S}$	$\rightarrow_{@_\beta}$	$(A_1 \leqslant_{\beta_{sd}} \lambda x.A_2) \cup (B_1 \leqslant_{\beta_{sd}} B_2) \cup \mathbb{S}$ **où** $A_2[x := B_2] = C$ **et** x **variable fraîche**, $x \in \mathtt{fv}(A_2)$ **et** A_2, B_2 **β-normaux**
$(\lambda x.A \leqslant_{\beta_{sd}} \lambda x.B) \cup \mathbb{S}$	$\rightarrow_{\lambda_\lambda}$	$(A \leqslant_{\beta_{sd}} B) \cup \mathbb{S}$
$(x \leqslant_{\beta_{sd}} x) \cup \mathbb{S}$	$\rightarrow_{\varepsilon_v}$	$\mathbb{S}$
$(a \leqslant_{\beta_{sd}} a) \cup \mathbb{S}$	$\rightarrow_{\varepsilon_c}$	$\mathbb{S}$
$(X \leqslant_{\beta_{sd}} A) \cup \mathbb{S}$	$\rightarrow_{\varepsilon_X}$	$(X \leqslant_{\beta_{sd}} A) \cup \mathbb{S}\{A/X\}$ **si** $\mathtt{fv}(A) = \emptyset$ **et** $X \in \mathbb{S}$

FIG. 3.1: Règles pour le filtrage d'ordre supérieur modulo super-développements

résolue qui donne une solution au problème de filtrage considéré (l'algorithme est correct) ;
– en explorant toutes les réductions (l'application des règles est *non-déterministe* dans le sens où deux réductions d'un même système peuvent donner deux formes normales distinctes) et en collectant toutes les formes normales résolues, on obtient un ensemble complet de filtres (l'algorithme est correct et complet).

Nous notons $\mathbb{S} \to \mathbb{S}'$ s'il existe une des règles de transformation de la figure 3.1 qui s'applique et qui permet de transformer $\mathbb{S}$ en $\mathbb{S}'$. Nous notons également $\mathbb{S} \twoheadrightarrow \mathbb{S}'$ s'il existe $n \geqslant 0$ systèmes $\mathbb{S}_1, \ldots, \mathbb{S}_n$ tels que $\mathbb{S} = \mathbb{S}_1 \to \ldots \to \mathbb{S}_n = \mathbb{S}'$. L'algorithme de filtrage modulo super-développements est basé sur l'équivalence entre super-développements et réduction parallèle forte. Nous analysons chaque règle de transformation de la figure 3.1 corrélativement aux règles de la réduction parallèle forte données dans la définition 1.44.

Les règles ε prennent en compte les atomes. Les règles (ε_c) et (ε_v) sont des règles éliminant les équations trivialement résolues. Quand la règle (ε_c) est appliquée au multi-ensemble singleton $a \leqslant_{\beta_{sd}} a$ alors le système obtenu est le multi-ensemble vide.

La règle (ε_X) substitue une variable de filtrage par sa valeur correspondante. Nous ne substituons pas des termes non $\mathcal{X}$-clos et lors de l'application de la substitution, les termes ainsi obtenus ne sont pas normalisés (contrairement aux algorithmes de filtrage modulo $\beta(\eta)$ dans le λ-calcul simplement typé). Sinon, la règle (ε_X) ne serait pas correcte. Considérons en effet la règle suivante :

$$(X \leqslant_{\beta_{sd}} A) \cup \mathbb{S} \quad \to_{\varepsilon_X^{\downarrow}} \quad (X \leqslant_{\beta_{sd}} A) \cup (\mathbb{S}\{A/X\}) \downarrow$$

où $(\mathbb{S}\{A/X\}) \downarrow$ dénote la β_{sd}-forme normale de $\mathbb{S}\{A/X\}$. Soit $A \leqslant_{\beta_{sd}} B$ une β_{sd}-équation de filtrage. Nous cherchons une substitution ϕ telle que $A\phi$ et B sont égaux modulo *une* étape de réduction parallèle forte. Considérons par exemple l'équation suivante :

$$f(XYZ, X, Y, Z) \leqslant_{\beta_{sd}} f(1, \lambda x.\lambda y.xy, \lambda z.z, 1)$$

On peut réduire le problème en

$$(XYZ \leqslant_{\beta_{sd}} 1) \cup (X \leqslant_{\beta_{sd}} \lambda x.\lambda y.xy) \cup (Y \leqslant_{\beta_{sd}} \lambda z.z) \cup (Z \leqslant_{\beta_{sd}} 1)$$

en appliquant ensuite la règle $(\varepsilon_X^{\downarrow})$ trois fois et la règle (ε_c) une fois, nous obtenons :

$$(X \leqslant_{\beta_{sd}} \lambda x.\lambda y.xy) \cup (Y \leqslant_{\beta_{sd}} \lambda z.z) \cup (Z \leqslant_{\beta_{sd}} 1)$$

qui est en forme résolue bien que la substitution correspondant ne soit pas un β_{sd}-filtre (mais c'est bien-sûr un β-filtre).

Notons enfin à propos de la règle (ε_X) que la condition d'application $X \in \mathbb{S}$ est nécessaire pour assurer la terminaison des règles comme nous le verrons dans le section 3.1.2.

La règle λ prend en compte les λ-abstractions en imitant la règle (Red-λ). Notons que comme nous considérons que des substitutions closes, il est tout à fait correct d'éliminer ainsi les lieurs. Dans beaucoup d'algorithmes pour le filtrage d'ordre supérieur, les λ-abstractions sont conservés en préfixe. Les deux choix sont pertinents. L'algorithme pour le filtrage modulo super-développements et η sera présenté en utilisant la deuxième possibilité (voir section 3.2).

On retrouve une règle similaire dans les travaux sur l'unification d'ordre supérieur dans le λ-calcul avec indices de de Bruijn et substitutions explicites [DHK00].

Les règles @ prennent en compte les applications. La règle $(@_@)$ est en correspondance directe avec la règle (Red-@) et ne nécessite donc pas plus de commentaires. Les règles $(@_\pi)$ et $(@_\beta)$ sont toutes deux liées à la règle (Red-β_f). On cherche à exprimer le membre droit C de l'équation comme le résultat d'une β-réduction disons $A_2[x := B_2]$. Selon l'appartenance de x dans A_2, nous obtenons la règle $(@_\pi)$ ou $(@_\beta)$. Si x n'appartient pas à A_2 alors nous obtenons la règle $(@_\pi)$: le membre gauche de l'équation est associé à une λ-abstraction qui ignore son argument et retourne le membre droit de l'équation initiale. Sinon, si x appartient à A_2, nous obtenons la règle $(@_\beta)$ en imitant la règle (Red-β_f) pour tous les termes tels que $C = A_2[x := B_2]$ où x appartient à A_2 et A_2 et B_2 sont β-normaux. Pour trouver de tels termes A_2 et B_2, on peut remarquer dans un premier temps que B_2 est nécessairement un sous-terme de C (puisque x appartient à A_2). Considérons un de ces sous-termes. Choisissons ensuite un sous-ensemble de l'ensemble des positions de C telles que le sous-terme de C à ces positions soit exactement B_2. On obtient ensuite A_2 à partir de C en mettant x à chacune des positions du sous-ensemble considéré. Il n'existe qu'un nombre fini de couples (A_2, B_2) qui satisfont ces conditions.

L'approche proposée ici pour le filtrage d'ordre supérieur se distingue fortement des approches comme celles de [HL78] puisqu'on n'introduit pas de nouvelles variables de filtrage durant le calcul des solutions. Les solutions d'un système $\mathbb{S}$ calculées par notre algorithme ont donc un domaine inclut dans les variables de fitrage de $\mathbb{S}$. Lorsqu'on compare une telle solution avec un solution quelconque de $\mathbb{S}$, il n'est donc pas techniquement nécessaire de limiter cette comparaison aux variables de filtrage de $\mathbb{S}$; comme cela est souvent nécessaire (voir par exemple [Bür90]).

Exemple 3.1 (Calcul des solutions d'une β_{sd}-équation) *Considérons* $XY \leqslant_{\beta_{sd}} ab$ *une* β_{sd}*-équation . Comme les membres gauche et droit de l'équation sont des applications, on peut appliquer les règles* $(@_@)$*,* $(@_\pi)$ *ou* $(@_\beta)$*.*

1. *Règle* $(@_@)$ *:*
$$(XY \leqslant_{\beta_{sd}} ab) \to (X \leqslant_{\beta_{sd}} a) \cup (Y \leqslant_{\beta_{sd}} b).$$
2. *Règle* $(@_\pi)$ *:*
$$(XY \leqslant_{\beta_{sd}} ab) \to (X \leqslant_{\beta_{sd}} \lambda x.ab).$$
3. *Règle* $(@_\beta)$ *: pour trouver* A_1 *et* A_2 *tels que* $A_1[x := A_2] = ab$*, tout d'abord on choisit* A_2 *parmi les sous-termes de « ab » :* a*,* b *et* ab*. L'ensemble des positions de* ab *pour lesquelles* A_2 *est le sous-terme de* C *correspondant est un singleton*

puisque chaque sous-terme de ab *n'apparaît qu'une seule fois dans* ab*. On obtient donc trois manières d'appliquer la règle* $(@_\beta)$ *correspondant aux trois sous-termes du membre droit de l'équation* $XY \leqslant_{\beta_{sd}} ab$ *:*

a) $(XY \leqslant_{\beta_{sd}} ab) \to (X \leqslant_{\beta_{sd}} \lambda x.xb) \cup Y \leqslant_{\beta_{sd}} a)$.

b) $(XY \leqslant_{\beta_{sd}} ab) \to (X \leqslant_{\beta_{sd}} \lambda x.ax) \cup (Y \leqslant_{\beta_{sd}} b)$.

c) $(XY \leqslant_{\beta_{sd}} ab) \to (X \leqslant_{\beta_{sd}} \lambda x.x) \cup (Y \leqslant_{\beta_{sd}} ab)$.

Exemple 3.2 (Calcul des solutions d'une β_{sd}-équation) *Considérons l'équation* $X(YX) \leqslant_{\beta_{sd}} a$ *une* β_{sd}*-équation . On peut appliquer les règles* $(@_\pi)$ *ou* $(@_\beta)$*.*

1. *Règle* $(@_\pi)$ *:*

$$(X(YX) \leqslant_{\beta_{sd}} a) \to (X \leqslant_{\beta_{sd}} \lambda x.a).$$

2. *Règle* $(@_\beta)$ *:*

$$(X(YX) \leqslant_{\beta_{sd}} a) \to (X \leqslant_{\beta_{sd}} \lambda x.x) \cup (YX \leqslant_{\beta_{sd}} a).$$

Pour simplifier $YX \leqslant_{\beta_{sd}} a$ *on peut appliquer la règle* $(@_\pi)$ *ou la règle* $(@_\beta)$*.*

a) Règle $(@_\pi)$ *:*

$$(X \leqslant_{\beta_{sd}} \lambda x.x) \cup (YX \leqslant_{\beta_{sd}} a) \to (X \leqslant_{\beta_{sd}} \lambda x.x) \cup (Y \leqslant_{\beta_{sd}} \lambda x.a).$$

b) Règle $(@_\beta)$ *:*

$$\begin{array}{ll} & (X \leqslant_{\beta_{sd}} \lambda x.x) \cup (YX \leqslant_{\beta_{sd}} a) \\ \to & (X \leqslant_{\beta_{sd}} \lambda x.x) \cup (Y \leqslant_{\beta_{sd}} \lambda x.x) \cup (X \leqslant_{\beta_{sd}} a) \\ \to & (X \leqslant_{\beta_{sd}} \lambda x.x) \cup (Y \leqslant_{\beta_{sd}} \lambda x.x) \cup (\lambda x.x \leqslant_{\beta_{sd}} a). \end{array}$$

Dans le dernier cas, le système n'est pas en forme résolue (bien qu'il soit en forme normale) et ne correspond donc à aucune substitution solution. Le problème de filtrage initial n'a donc que deux solutions.

Remarque 3.3 *L'application de la règle* $(@_\beta)$ *dépend fortement du choix d'un terme* B_2 *qui filtre le terme* B_1*, ce qui est une condition forte. Dans une implémentation de l'algorithme, à partir de plusieurs heuristiques simples on peut largement limiter l'application de la règle* $(@_\beta)$*. Par exemple, si* B_1 *est une constante ou une variable alors nécessairement* $B_2 = B_1$*. Ce cas se produit toujours lors du filtrage de motifs à la Miller puisqu'une variable de filtrage ne peut être appliquée qu'à des variables liées.*

3.1.2 Propriétés de l'algorithme

Terminaison

Nous définissons tout d'abord la taille d'un terme (resp. d'une équation de filtrage, resp. d'un système) :

Définition 3.4 (Taille) *La taille d'un terme* A *dénotée* $\mathfrak{S}(A)$ *est définie par induction*

$$\begin{array}{lcll} \mathfrak{S}(\varepsilon) & = & 1 & \text{pour tous les atomes } \varepsilon \\ \mathfrak{S}(\lambda x.B) & = & \mathfrak{S}(B)+1 & \\ \mathfrak{S}(BC) & = & \mathfrak{S}(B)+\mathfrak{S}(C)+1 & \end{array}$$

La taille d'une équation de filtrage $A \leqslant_{\beta_{sd}} B$ *est déterminée uniquement par la taille de* A*. La taille d'un système est la somme des tailles de chacune des équations qui composent ce système. Nous utilisons la même notation pour la taille d'un terme, d'une équation et d'un système.*

Pour tout système $\mathbb{S}$, nous dénotons par $\mathfrak{U}(\mathbb{S})$ le nombre de variables non résolues de $\mathbb{S}$ suivant la définition suivante.

Définition 3.5 (Variable résolue) *Une variable de filtrage d'une équation* $X \leqslant_{\beta_{sd}} A$ *appartenant à un système* $\mathbb{S}$ *est une variable résolue si* X *n'apparaît nulle part ailleurs dans* $\mathbb{S}$*.*

Lemme 3.6 (Variables non résolues) *Pour tous systèmes* $\mathbb{S}$ *et* $\mathbb{S}'$ *tels que* $\mathbb{S} \to \mathbb{S}'$ *on a l'inégalité suivante :*

$$\mathfrak{U}(\mathbb{S}) \geqslant \mathfrak{U}(\mathbb{S}')$$

L'inégalité est stricte si la réduction est faite en utilisant la règle (ε_X)*.*

Lemme 3.7 (Décroissance de la taille) *Pour tous systèmes* $\mathbb{S}$ *et* $\mathbb{S}'$ *tels que* $\mathbb{S} \to \mathbb{S}'$ *en utilisant n'importe laquelle des règles de réduction à l'exception de la règles*(ε_X) *on a l'inégalité suivante :*

$$\mathfrak{S}(\mathbb{S}) > \mathfrak{S}(\mathbb{S}')$$

Preuve. La preuve se fait en examinant l'inégalité pour chacune des règles comme indiqué dans la Figure 3.2. Rappelons que la taille d'une équation ne dépend pas du membre droit : l'inégalité est donc vraie en particulier pour la règle $(@_\beta)$.

□

Propriété 3.8 (Terminaison de l'ensemble des règles) *L'ensemble des règles de transformation donné dans la figure 3.1 est terminant.*

Preuve. Pour tous les systèmes $\mathbb{S}$ et $\mathbb{S}'$ tels que $\mathbb{S} \to \mathbb{S}'$, chaque règle fait décroître le produit lexicographique composé du nombre de variables non résolues $\mathfrak{U}(\_)$ et de la taille du système $\mathfrak{S}(\_)$:

ε_v $\quad \mathfrak{S}((x \leqslant_{\beta_{sd}} x) \cup \mathbb{S})$
$= \mathfrak{S}(x \leqslant_{\beta_{sd}} x) + \mathfrak{S}(\mathbb{S})$
$> \mathfrak{S}(\mathbb{S})$

ε_c $\quad \mathfrak{S}((c \leqslant_{\beta_{sd}} c) \cup \mathbb{S})$
$= \mathfrak{S}(c \leqslant_{\beta_{sd}} c) + \mathfrak{S}(\mathbb{S})$
$> \mathfrak{S}(\mathbb{S})$

λ_λ $\quad \mathfrak{S}((\lambda x.A \leqslant_{\beta_{sd}} \lambda x.B) \cup \mathbb{S})$
$= \mathfrak{S}(\lambda x.A \leqslant_{\beta_{sd}} \lambda x.B) + \mathfrak{S}(\mathbb{S})$
$= 1 + \mathfrak{S}(A) + \mathfrak{S}(\mathbb{S})$
$> \mathfrak{S}(A \leqslant_{\beta_{sd}} B) + \mathfrak{S}(\mathbb{S})$

$@_@$ $\quad \mathfrak{S}((A_1B_1 \leqslant_{\beta_{sd}} A_2B_2) \cup \mathbb{S})$
$= \mathfrak{S}(A_1B_1) + \mathfrak{S}(\mathbb{S})$
$= 1 + \mathfrak{S}(A_1) + \mathfrak{S}(B_1) + \mathfrak{S}(\mathbb{S})$
$> \mathfrak{S}(A_1 \leqslant_{\beta_{sd}} A_2) + \mathfrak{S}(B_1 \leqslant_{\beta_{sd}} B_2) + \mathfrak{S}(\mathbb{S})$

$@_\pi$ $\quad \mathfrak{S}((A_1B_1 \leqslant_{\beta_{sd}} A_2) \cup \mathbb{S})$
$= \mathfrak{S}(A_1B_1) + \mathfrak{S}(\mathbb{S})$
$= 1 + \mathfrak{S}(A_1) + \mathfrak{S}(B_1) + \mathfrak{S}(\mathbb{S})$
$> \mathfrak{S}(A_1 \leqslant_{\beta_{sd}} \lambda x.A_2) + \mathfrak{S}(\mathbb{S})$

$@_\beta$ $\quad \mathfrak{S}((A_1B_1 \leqslant_{\beta_{sd}} C) \cup \mathbb{S})$
$= \mathfrak{S}(A_1B_1) + \mathfrak{S}(\mathbb{S})$
$= 1 + \mathfrak{S}(A_1) + \mathfrak{S}(B_1) + \mathfrak{S}(\mathbb{S})$
$> \mathfrak{S}(A_1 \leqslant_{\beta_{sd}} \lambda x.A_2) + \mathfrak{S}(B_1 \leqslant_{\beta_{sd}} B_2) + \mathfrak{S}(\mathbb{S})$

pour tous termes A_2 et B_2 normaux

FIG. 3.2: Décroissance de la taille

$$\frac{}{\emptyset \in \beta_f} \qquad \frac{\overline{(A_1, B_1) \in \beta_f} \quad E \in \beta_f}{(A_1, B_1) \cup E \in \beta_f}$$

$$\frac{\dfrac{(A'_1, B'_1) \in \beta_f \quad (A''_1, B''_1) \in \beta_f}{(A_1, B_1) \in \beta_f} \qquad (A'_1, B'_1) \cup (A''_1, B''_1) \cup E \in \beta_f}{(A_1, B_1) \cup E \in \beta_f}$$

$$\frac{\dfrac{(A'_1, B'_1) \in \beta_f}{(A_1, B_1) \in \beta_f} \quad (A'_1, B'_1) \cup E \in \beta_f}{(A_1, B_1) \cup E \in \beta_f}$$

FIG. 3.3: Extension de $\Longrightarrow_{\beta_f}$ aux multi-ensembles

	$\mathfrak{U}(\_)$	$\mathfrak{S}(\_)$
ε_v	$=$	$>$
ε_c	$=$	$>$
λ_λ	$\geqslant$	$>$
ε_X	$>$	
$@_@$	$\geqslant$	$>$
$@_\pi$	$\geqslant$	$>$
$@_\beta$	$\geqslant$	$>$

Notons que la règle (ε_X) fait toujours décroître le nombre de variables résolues grâce à la condition d'application de la règle : $X \in \mathbb{S}$. □

Complétude

Nous définissons une extension de la relation $\Longrightarrow_{\beta_f}$ aux multi-ensembles. Nous notons dans cette section $(A_1, B_1) \in \beta_f$ à la place de $A_1 \Longrightarrow_{\beta_f} B_1$.

Définition 3.9 (Extension de $\Longrightarrow_{\beta_f}$ aux multi-ensembles) *Nous notons $\emptyset$ le multi-ensemble vide. Nous définissons la relation sur les multi-ensembles par les règles d'inférence (nous surchargeons la notation utilisée pour la relation du même nom définie sur les termes) définies dans la Figure 3.3.*

Propriété 3.10 (Complétude) *Pour tout système $\mathbb{S}$, si $\phi \in \mathbb{M}(\mathbb{S})$ alors il existe une suite de transformations*

$$\mathbb{S} = \mathbb{S}_0 \to \mathbb{S}_1 \to \ldots \to \mathbb{S}_n$$

où $\mathbb{S}_n$ *est en forme résolue et* $\sigma_{\mathbb{S}_n} \leq \phi$.

Preuve. Nous supposons donné un système $\mathbb{S}_0$ tel que $\phi \in \mathbb{M}(\mathbb{S}_0)$. On veut montrer qu'il existe une dérivation tel que

$$\mathbb{S} = \mathbb{S}_0 \to \mathbb{S}_1 \to \ldots \to \mathbb{S}_n$$

où $\mathbb{S}_n$ est en forme résolue et $\sigma_{\mathbb{S}_n} \leq \phi$.

Posons $\mathbb{S}_0 = (A_1 \leqslant_{\beta_{sd}} B_1) \cup \ldots \ (A_p \leqslant_{\beta_{sd}} B_p)$. Soit E_0 le multi-ensemble défini par

$$E_0 \quad = \quad (A_1\phi, B_1) \cup \ldots \cup (A_p\phi, B_p)$$

Dire que $\phi \in \mathbb{M}(\mathbb{S}_0)$ est équivalent à dire que $E_0 \in \beta_f$. Nous montrons la proposition par induction sur la preuve de $E_0 \in \beta_f$.

1. Si $E_0 = \emptyset$ alors $\mathbb{S}_0 = \emptyset$ et $\sigma_{\mathbb{S}_0}$ est la substitution identité `id`. Le résultat est immédiat puisque pour toute substitution ϕ on a, $\mathtt{id} \leqslant \phi$.
2. Si $E_0 = (A_1\phi, B_1) \cup E \in \beta_f$ avec $(A_1\phi, B_1) \in \beta_f$ est prouvable sans aucune hypothèse et $E \in \beta_f$. Par hypothèse d'induction il existe une dérivation (D)

   $$\begin{array}{l} \mathbb{S}'_0 = (A_2 \leqslant_{\beta_{sd}} B_2) \cup \ldots \cup \ (A_p \leqslant_{\beta_{sd}} B_p) \\ \to \ldots \\ \to \mathbb{S}'_n \end{array}$$

   telle que $\sigma_{\mathbb{S}'_n} \leqslant \phi$. Ainsi si $X \in \mathcal{D}om(\sigma'_n)$ alors $X\sigma'_n = B_1$.

   Supposons tout d'abord que A_1 soit une constante ou une variable. Alors la dérivation

   $$\begin{array}{lll} \mathbb{S}_0 & \to_{\substack{\varepsilon_c \\ \varepsilon_v}} (A_2 \leqslant_{\beta_{sd}} B_2) \cup \ldots \cup \ (A_p \leqslant_{\beta_{sd}} B_p) \\ & \to \ldots \\ & \to \mathbb{S}'_n \end{array}$$

   est la dérivation recherchée.

   Supposons maintenant que A_1 soit une variable de filtrage alors s'il n'existe pas d'étape de réduction de (D) où X est instanciée (application de la règle (ε_X) à la variable X) alors la dérivation

   $$\begin{array}{lll} & (X \leqslant_{\beta_{sd}} B_1) \cup (A_2 \leqslant_{\beta_{sd}} B_2) \cup \ \ldots \ \cup (A_p \leqslant_{\beta_{sd}} B_p) \\ \to & \ldots \\ \to & (X \leqslant_{\beta_{sd}} B_1) \cup \mathbb{S}'_n \end{array}$$

   est la dérivation recherchée.

   Sinon (c'est-à-dire s'il existe une étape de dérivation i_0 de D durant laquelle X est instanciée) alors X est forcément instanciée par B_1 et la dérivation suivante est la

dérivation recherchée :

$$\begin{array}{ll} & (X \leqslant_{\beta_{sd}} B_1) \cup (A_2 \leqslant_{\beta_{sd}} B_2) \cup \ \ldots \ \cup (A_p \leqslant_{\beta_{sd}} B_p) \\ \rightarrow & \ldots \\ \rightarrow_{i_0} & (B_1 \leqslant_{\beta_{sd}} B_1) \cup \mathbb{S}'_{i_0+1} \\ \twoheadrightarrow & \mathbb{S}'_{i_0+1} \\ \rightarrow & \ldots \\ \rightarrow & \mathbb{S}'_n \end{array}$$

3. Si $E_0 = (A_1\phi, B_1) \cup E \in \beta_f$ avec $(A_1\phi, B_1) \in \beta_f$ est prouvable en utilisant deux hypothèses. On raisonne par cas sur la dernière règle utilisée dans la preuve de $(A_1\phi, B_1) \in \beta_f$.
   – **Règle ($\mathsf{Red} - @$)** Alors on a $E_0 = (A_1^1\phi A_1^2\phi, B_1B_2) \cup E \in \beta_f$ avec

$$\frac{(A_1^1\phi, B_1^1) \in \beta_f \quad (A_1^2\phi, B_1^2) \in \beta_f}{(A_1^1\phi A_1^2\phi, B_1^1B_1^2) \in \beta_f}$$

   et

$$(A_1^1\phi, B_1^1) \cup (A_1^2\phi, B_1^2) \cup E \in \beta_f$$

   Par hypothèse d'induction, il existe une dérivation telle que

$$\begin{array}{ll} & (A_1^1 \leqslant_{\beta_{sd}} B_1^1) \cup (A_1^2 \leqslant_{\beta_{sd}} B_1^2) \cup (A_2 \leqslant_{\beta_{sd}} B_2) \cup \ldots \cup (A_p \leqslant_{\beta_{sd}} B_p) \\ \rightarrow & \ldots \\ \rightarrow & \mathbb{S}_n \end{array}$$

   avec $\sigma_{\mathbb{S}_n} \leqslant \phi$.
   Si $A_1 = X$ alors on a aussi (puisqu'alors $A_1^1 = B_1^1$ et $A_1^2 = B_1^2$ et donc que les deux équations correspondantes sont $\mathcal{V}$-closes et n'influencent pas la dérivation)

$$\begin{array}{l} (A_2 \leqslant_{\beta_{sd}} B_2) \cup \ldots \cup (A_p \leqslant_{\beta_{sd}} B_p) \\ \rightarrow \ldots \\ \rightarrow \mathbb{S}'_n \end{array}$$

   telle que $\sigma_{\mathbb{S}'_n} \leqslant \phi$. On conclut comme dans la cas précédent.
   Sinon, la dérivation

$$\begin{array}{ll} & (A_1 \leqslant_{\beta_{sd}} B_1) \cup \ \ldots \ \cup (A_p \leqslant_{\beta_{sd}} B_p) \\ = & (A_1^1A_1^2 \leqslant_{\beta_{sd}} B_1^1B_1^2) \cup (A_2 \leqslant_{\beta_{sd}} B_2) \cup \ \ldots \ \cup (A_p \leqslant_{\beta_{sd}} B_p) \\ \rightarrow & (A_1^1 \leqslant_{\beta_{sd}} B_1^1) \cup \ (A_1^2 \leqslant_{\beta_{sd}} B_1^2) \cup \ldots \cup (A_p \leqslant_{\beta_{sd}} B_p) \\ \rightarrow & \ldots \\ \rightarrow & \mathbb{S}_n \end{array}$$

   est la dérivation recherchée.
   – **Règle ($\mathsf{Red} - \beta_f$)** Alors on a $E_0 = (A_1^1\phi A_1^2\phi, B_1^1[x := B_1^2]) \cup E$ avec

$$\frac{(A_1^1\phi, \lambda x.B_1^1) \in \beta_f \quad (A_1^2\phi, B_1^2) \in \beta_f}{(A_1^1\phi A_1^2\phi, B_1^1[x := B_1^2]) \in \beta_f}$$

et

$$(A_1^1\phi, \lambda x.B_1^1) \cup (A_1^2\phi, B_1^2) \cup E \in \beta_f$$

Par hypothèse d'induction, il existe une dérivation telle que

$$\begin{array}{lll} & (A_1^1 \leqslant_{\beta_{sd}} \lambda x.B_1^1) \cup (A_1^2 \leqslant_{\beta_{sd}} B_1^2) \cup \ldots \cup (A_p \leqslant_{\beta_{sd}} B_p) \\ \rightarrow & \ldots \\ \rightarrow & \mathbb{S}'_n \end{array}$$

avec $\sigma'_{\mathbb{S}_n} \leqslant \phi$. La dérivation

$$\begin{array}{lll} & (A_1^1 A_1^2 \leqslant_{\beta_{sd}} B_1^1[x := B_1^2]) \cup (A_1^1 \leqslant_{\beta_{sd}} \lambda x.B_1^1) \cup (A_1^2 \leqslant_{\beta_{sd}} B_1^2) \cup \ldots \cup (A_p \leqslant_{\beta_{sd}} B_p) \\ \rightarrow & \ldots \\ \rightarrow & \mathbb{S}_n \end{array}$$

est la dérivation recherchée si $x \in \mathtt{fv}(B_1^1)$. Sinon, la dérivation

$$\begin{array}{lll} & (A_1^1 A_1^2 \leqslant_{\beta_{sd}} B_1^1[x := B_1^2]) \cup (A_1^1 \leqslant_{\beta_{sd}} \lambda x.B_1^1) \cup \ldots \cup (A_p \leqslant_{\beta_{sd}} B_p) \\ \rightarrow & \ldots \\ \rightarrow & \mathbb{S}_n \end{array}$$

est la dérivation recherchée.

4. Ce cas est similaire au précédent.

□

Correction

Lemme 3.11 *Pour tous systèmes $\mathbb{S}$ et $\mathbb{S}'$ tels que $\mathbb{S} \to \mathbb{S}'$ en utilisant les règles (ε_c), (ε_v), (λ_λ) ou (ε_x), on a $\mathbb{M}(\mathbb{S}') = \mathbb{M}(\mathbb{S})$.*

Preuve. Le seul cas non trivial concerne la règle (ε_x). Soient X une variable de filtrage, $\mathbb{S}$ un système et A un terme tel que $\mathtt{fv}(A) = \emptyset$ et $X \in \mathbb{S}$.

$$\begin{array}{lll} \phi \in \mathbb{M}((X \leqslant_{\beta_{sd}} A) \cup \mathbb{S}) & \Leftrightarrow & X\phi \Longrightarrow_{\beta_f} A \text{ et } \phi \in \mathbb{M}(\mathbb{S}) \\ & \overset{\phi \text{ normale}}{\Leftrightarrow} & X\phi = A \text{ et } \phi \circ \{A/X\} \in \mathbb{M}(\mathbb{S}) \\ & \Leftrightarrow & X\phi \Longrightarrow_{\beta_f} A \text{ et } \phi \in \mathbb{M}(\mathbb{S}\{A/X\}) \\ & \Leftrightarrow & \phi \in \mathbb{M}((X \leqslant_{\beta_{sd}} A) \cup \mathbb{S}\{A/X\}) \end{array}$$

□

Lemme 3.12 *Pour tous systèmes $\mathbb{S}$ and $\mathbb{S}'$ tels que $\mathbb{S} \to \mathbb{S}'$ en utilisant les règles $(@_@)$, $(@_\pi)$ ou $(@_\beta)$ on a $\mathbb{M}(\mathbb{S}') \subseteq \mathbb{M}(\mathbb{S})$.*

Preuve. On prouve le résultat pour chaque règle.

$$
\begin{array}{ll}
@_{@} & \phi \in \mathbb{M}((A_1 \leqslant_{\beta_{sd}} A_2) \cup (B_1 \leqslant_{\beta_{sd}} B_2) \cup \mathbb{S}) \\
& \Leftrightarrow \phi \in \mathbb{M}(A_1 \leqslant_{\beta_{sd}} A_2) \text{ et } \phi \in \mathbb{M}(B_1 \leqslant_{\beta_{sd}} B_2) \text{ et } \phi \in \mathbb{M}(\mathbb{S}) \\
& \Leftrightarrow A_1\phi \Longrightarrow_{\beta_f} A_2 \text{ et } B_1\phi \Longrightarrow_{\beta_f} B_2 \text{ et } \phi \in \mathbb{M}(\mathbb{S}) \\
& \Rightarrow (A_1B_1)\phi \Longrightarrow_{\beta_f} A_2B_2 \text{ et } \phi \in \mathbb{M}(\mathbb{S}) \\
& \Leftrightarrow \phi \in \mathbb{M}((A_1B_2 \leqslant_{\beta_{sd}} A_2B_2) \cup \mathbb{S}) \\
\\
@_{\pi} & \phi \in \mathbb{M}((A_1 \leqslant_{\beta_{sd}} \lambda x.A_2) \cup \mathbb{S}) \\
& \Leftrightarrow A_1\phi \Longrightarrow_{\beta_f} \lambda x.A_2 \text{ et } \phi \in \mathbb{M}(S) \\
& \Rightarrow (A_1B_1)\phi \Longrightarrow_{\beta_f} A_2[x := (B_1\phi)] \text{ et } \phi \in \mathbb{M}(S) \\
& \Leftrightarrow (A_1B_1)\phi \Longrightarrow_{\beta_f} A_2 \text{ et } \phi \in \mathbb{M}(S) \\
& \qquad\qquad \text{comme } x \text{ est fraîche pour } A_1, B_1, A_2, \mathbb{S} \\
& \Leftrightarrow \phi \in \mathbb{M}((A_1B_1 \leqslant_{\beta_{sd}} A_2) \cup \mathbb{S}) \\
\\
@_{\beta} & \phi \in \mathbb{M}((A_1 \leqslant_{\beta_{sd}} \lambda x.A_2) \cup (B_1 \leqslant_{\beta_{sd}} B_2) \cup \mathbb{S}) \\
& \Leftrightarrow A_1\phi \Longrightarrow_{\beta_f} \lambda x.A_2 \text{ et } B_1\phi \Longrightarrow_{\beta_f} B_2 \text{ et } \phi \in \mathbb{S} \\
& \Rightarrow (A_1B_1)\phi \Longrightarrow_{\beta_f} A_2[x := B_1] \text{ et } \phi \in \mathbb{S} \\
& \Leftrightarrow \phi \in \mathbb{M}((A_1B_1 \leqslant_{\beta_{sd}} A_2[x := B_1]) \cup \mathbb{S})
\end{array}
$$

□

Propriété 3.13 (Correction) *Pour tous systèmes $\mathbb{S}$ et $\mathbb{S}'$ tels que $\mathbb{S} \twoheadrightarrow \mathbb{S}'$ et $\mathbb{S}'$ est une forme résolue on a $\sigma_{\mathbb{S}'} \in \mathbb{M}(\mathbb{S})$.*

Preuve. Par une induction simple sur la longueur de la suite de transformations et en utilisant les lemmes précédents pour l'étape d'induction. □

Finitude de l'ensemble complet de filtres

Les propriétés de correction et complétude de l'algorithme nous donnent qu'il existe un ensemble complet de filtre. Il est obtenu en explorant toutes les réductions possibles et en construisant l'ensemble des substitutions obtenu à partir de toutes les formes résolues. On sait que ce processus est fini puisque l'algorithme termine et donc que l'on obtient un ensemble complet de filtre qui est fini.

Propriété 3.14 (Finitude de l'ensemble complet de filtres) *Étant donné un système $\mathbb{S}$, il existe un ensemble complet de filtres $\mathbb{M}$ qui est fini. Il est donné par*

$$\mathbb{M} = \{\sigma'_{\mathbb{S}} \mid \mathbb{S} \twoheadrightarrow \mathbb{S}' \text{ et } \mathbb{S}' \text{ est en forme résolue }\}.$$

3.2 Filtrage modulo η et super-développements

3.2.1 Règles

L'algorithme pour le filtrage modulo super-développements donné dans la figure 3.1 doit être ajusté en deux points pour tenir compte de l'η-conversion. Tout d'abord, l'η-expansion est faite à la demande en ajoutant la règle suivante :

$$(\lambda x.A \leqslant_{\beta_{sd}\eta} B) \cup \mathbb{S} \quad \rightarrow \quad (A \leqslant_{\beta_{sd}\eta} Bx) \cup \mathbb{S}$$
si B n'est pas une λ-abstraction
et x est une variable fraîche

Cette règle remplace le membre droit B par $\lambda x.Bx$ et effectue (comme dans la règle (λ_λ) de la figure 3.1) une élimination de la λ-abstraction en tête.

Ensuite, nous devons ajouter une condition à la règle $(@_\beta)$ de manière à ce que $\lambda x.A_2$ et A_1 soient en forme βη-normale (et non plus seulement en forme β-normale).

En effectuant ces deux changements on obtient ainsi un algorithme pour le filtrage modulo super-développements et η. Nous le redonnons explicitement dans la figure 3.4 mais en choisissant de ne pas éliminer les λ-abstractions mais plutôt de les conserver en préfixe (et donc la règle (λ_λ) disparaît).

On peut finalement remarquer que l'algorithme ainsi obtenu dispose des mêmes propriétés (terminaison, correction et complétude). La preuve de terminaison ainsi que la preuve de correction sont sans aucune difficulté. La preuve de complétude est légèrement plus technique mais fondamentalement identique à celle donnée dans la section 3.1.2.

Exemple 3.15 ($\beta_{sd}\eta$-filtres) *Considérons l'équation donnée dans l'exemple 3.1. Si on résout cette équation modulo $\beta_{sd}\eta$ on obtient seulement 4 solutions. En effet, les deux solutions*

$$(X \leqslant_{\beta_{sd}} a) \cup (Y \leqslant_{\beta_{sd}} b) \qquad \textit{et} \qquad (X \leqslant_{\beta_{sd}} \lambda x.ax) \cup (Y \leqslant_{\beta_{sd}} b)$$

sont η-équivalentes.

Exemple 3.16 (Solutions d'une $\beta_{sd}\eta$-équation) *Considérons le couple de termes $(\lambda x.X(Yx), a)$. Ce couple est une β_{sd}-équation qui n'a pas de β_{sd}-filtre solution.*

Par contre, cette équation a deux $\beta_{sd}\eta$-filtres donnés par

$$\{a/X, \lambda z.z/Y\} \qquad \textit{et} \qquad \{\lambda z.z/X, a/Y\}.$$

$(\lambda\overline{x_n}.(A_1B_1) \leqslant_{\beta_{sd}} \lambda\overline{x_n}.(A_2B_2)) \cup \mathbb{S}$	$\rightarrow_{@_@}$	$(\lambda\overline{x_n}.A_1 \leqslant_{\beta_{sd}} \lambda\overline{x_n}.A_2) \cup$ $(\lambda\overline{x_n}.B_1 \leqslant_{\beta_{sd}} \lambda\overline{x_n}.B_2) \cup \mathbb{S}$
$(\lambda\overline{x_n}.(A_1B_1) \leqslant_{\beta_{sd}} \lambda\overline{x_n}.C) \cup \mathbb{S}$	$\rightarrow_{@_\pi}$	$(\lambda\overline{x_n}.A_1 \leqslant_{\beta_{sd}} \lambda\overline{x_n}.(\lambda y.C)) \cup \mathbb{S}$ **où** y **variable fraîche**
$(\lambda\overline{x_n}.(A_1B_1) \leqslant_{\beta_{sd}} \lambda\overline{x_n}.C) \cup \mathbb{S}$	$\rightarrow_{@_\beta}$	$(\lambda\overline{x_n}.A_1 \leqslant_{\beta_{sd}} \lambda\overline{x_n}.(\lambda x.A_2)) \cup$ $(\lambda\overline{x_n}.B_1 \leqslant_{\beta_{sd}} \lambda\overline{x_n}.B_2) \cup \mathbb{S}$ **où** $A_2[x := B_2] = C$ **et** x **variable fraîche**, $x \in \mathtt{fv}(A_2)$ **et** $\lambda x.A_2$, B_2 $\beta\eta$**-normaux**
$(\lambda\overline{x_n}.A \leqslant_{\beta_{sd}} \lambda\overline{x_{n-k}}.B) \cup \mathbb{S}$	$\rightarrow_{\lambda\_}$	$(\lambda\overline{x_n}.A \leqslant_{\beta_{sd}} \lambda\overline{x_{n-k+1}}.Bx_{n-k+1}) \cup \mathbb{S}$
$(\lambda\overline{x_n}.x \leqslant_{\beta_{sd}} \lambda\overline{x_n}.x) \cup \mathbb{S}$	$\rightarrow_{\varepsilon_v}$	$\mathbb{S}$
$(\lambda\overline{x_n}.a \leqslant_{\beta_{sd}} \lambda\overline{x_n}.a) \cup \mathbb{S}$	$\rightarrow_{\varepsilon_c}$	$\mathbb{S}$
$(\lambda\overline{x_n}.X \leqslant_{\beta_{sd}} \lambda\overline{x_n}.A) \cup \mathbb{S}$	$\rightarrow_{\varepsilon_X}$	$(\lambda\overline{x_n}.X \leqslant_{\beta_{sd}} \lambda\overline{x_n}.A) \cup \mathbb{S}\{A/X\}$ **si** $\mathtt{fv}(A) = \emptyset$ **et** $X \in \mathbb{S}$

FIG. 3.4: Règles pour le filtrage modulo η et super-développements

En effet,

$$\begin{aligned}
(\lambda x.(X(Yx) \leqslant_{\beta_{sd}\eta} a) &\rightarrow (\lambda x.X(Yx) \leqslant_{\beta_{sd}\eta} \lambda x.ax)\\
&\rightarrow (\lambda x.X \leqslant_{\beta_{sd}\eta} \lambda x.a) \cup (\lambda x.Yx \leqslant_{\beta_{sd}\eta} \lambda x.x)\\
&\rightarrow (\lambda x.X \leqslant_{\beta_{sd}\eta} \lambda x.a) \cup (\lambda x.Y \leqslant_{\beta_{sd}\eta} \lambda x.\lambda z.z)
\end{aligned}$$

$$\begin{aligned}
(\lambda x.X(Yx) \leqslant_{\beta_{sd}\eta} a) &\rightarrow (\lambda x.X(Yx) \leqslant_{\beta_{sd}\eta} \lambda x.ax)\\
&\rightarrow (\lambda x.X \leqslant_{\beta_{sd}\eta} \lambda x.\lambda z.z) \cup (\lambda x.Yx \leqslant_{\beta_{sd}\eta} \lambda x.ax)\\
&\rightarrow (\lambda x.X \leqslant_{\beta_{sd}\eta} \lambda x.\lambda z.z) \cup (\lambda x.Y \leqslant_{\beta_{sd}\eta} \lambda x.a)
\end{aligned}$$

3.2.2 Minimalité pour les motifs à la Miller

Nous poursuivons la comparaison, commencée dans la section 2.2.4, entre le filtrage modulo super-développements et le filtrage de motifs à la Miller. Dans la suite de cette section, nous appellerons équation de motifs à la Miller une β_{sd}-équation dont le premier terme est un motif à la Miller.

L'algorithme donné pour le filtrage modulo η et super-développements appliqué à un motif à la Miller donne au plus une solution (proposition suivante). Comme l'algorithme est correct et complet, cette solution est le filtre le plus général.

On peut tout d'abord remarquer que tout réduit d'un système d'équations à la Miller est un système d'équations à la Miller.

Propriété 3.17 *Soit* $P \leqslant_{\beta_{sd}\eta} B$ *une* β_{sd}*-équation où* P *est un motif à la Miller qui a une solution et* A *un terme typé. Alors l'ensemble des formes résolues données par les règles de la figure 3.4 est un singleton.*

Preuve. Nous pouvons tout d'abord remarquer que si la règle $(\lambda\_)$ s'applique alors aucune autre règle ne peut s'appliquer et donc cette situation n'introduit pas de choix dans l'application des règles. Nous l'exclurons donc dans le reste de la preuve.

Nous prouvons le résultat par induction sur la taille des motifs. On distingue trois cas suivant le symbole de tête du motif.

1. Si $P = \lambda\overline{x_n}.f(A_1,\ldots,A_n)$ alors, pour que l'équation est une solution alors nécessairement B doit être de la forme $B = \lambda\overline{x_n}.f(B_1,\ldots,B_p)$. Toutes les dérivations qui mènent à une solution doivent réduire $P \leqslant_{\beta_{sd}} B$ en

$$(\lambda\overline{x_n}.A_1 \leqslant_{\beta_{sd}} \lambda\overline{x_n}.B_1) \cup \ldots \cup (\lambda\overline{x_n}.A_p \leqslant_{\beta_{sd}} \lambda\overline{x_n}.B_p)$$

(modulo l'ordre d'application des règles).

Or $\lambda\overline{x_n}.A_1, \ldots \lambda\overline{x_n}.A_p$ sont des motifs à la Miller donc on peut appliquer l'hypothèse d'induction et conclure que

$$\begin{array}{ll} \lambda\overline{x_n}.A_1 \leqslant_{\beta_{sd}} \lambda\overline{x_n}.B_1 & \text{a pour unique solution } \sigma_1 \\ \ldots & \\ \lambda\overline{x_n}.A_p \leqslant_{\beta_{sd}} \lambda\overline{x_n}.B_p & \text{a pour unique solution } \sigma_p \end{array}$$

donc le problème initial a pour unique solution $\sigma_1 \cup \ldots \cup \sigma_p$.

2. Le cas $P = \lambda\overline{x_n}.x(A_1, \ldots, A_n)$ est similaire au précédent.
3. Si $P = \lambda\overline{x_n}.X(x_{i_1}, \ldots, x_{i_p})$ où les variables $x_{i_1}, \ldots, x_{i_p}$ sont deux à deux différentes. Alors trois règles peuvent s'appliquer $(@_\pi)$, $(@_@)$ ou $(@_\beta)$. Nous allons montrer que dans tous les cas un seul de ces choix mène à une solution.
   Dans la suite de cette preuve, nous utiliserons constamment le fait que si l'application de la règle $(@_@)$ mène à une solution alors nécessairement $B = B_1 x_{i_p}$. De même si l'application de la règle $(@_\beta)$ mène à une solution alors

$$\begin{array}{rcl} (\lambda\overline{x_n}.X(x_{i_1}, \ldots, x_{i_p}) \leqslant_{\beta_{sd}} \lambda\overline{x_n}.B) & \rightarrow & (\lambda\overline{x_n}.X(x_{i_1}, \ldots, x_{i_{p-1}}) \leqslant_{\beta_{sd}} \lambda\overline{x_n}.\lambda z.C_1 \cup) \\ & & (\lambda\overline{x_n}.x_{i_p} \leqslant_{\beta_{sd}} \lambda\overline{x_n}.x_{i_p}) \\ & & \text{avec } B = C_1[z := x_{i_p}] \text{ et } z \in \mathtt{fv}(C_1) \end{array}$$

   et donc $x_{i_p} \in \mathtt{fv}(B)$.
   Supposons tout d'abord que l'application de la règle $(@_\pi)$ mène à une solution. Montrons alors que ni l'application de la règle $(@_@)$ ni l'application de la règle $(@_\beta)$ aboutissent à une solution. On a

$$\begin{array}{l} (\lambda\overline{x_n}.X(x_{i_1}, \ldots, x_{i_p}) \leqslant_{\beta_{sd}} \lambda\overline{x_n}.B) \\ \rightarrow \ (\lambda\overline{x_n}.X(x_{i_1}, \ldots, x_{i_{p-1}}) \leqslant_{\beta_{sd}} \lambda\overline{x_n}.\lambda y.B) \end{array}$$

   Forcément, le terme B ne contient pas la variable x_{i_p} (car aucune des variables $x_{i_1}, \ldots, x_{i_{p-1}}$ n'est égale à x_{i_p}). Le résultat est donc immédiat.
   Supposons maintenant que l'application de la règle $(@_@)$ mène à une solution. Montrons alors que ni l'application de la règle $(@_\pi)$ ni l'application de la règle $(@_\beta)$ aboutissent une solution. On a

$$\begin{array}{l} (\lambda\overline{x_n}.X(x_{i_1}, \ldots, x_{i_p}) \leqslant_{\beta_{sd}} \lambda\overline{x_n}.B_1 x_{i_p}) \\ \rightarrow \ (\lambda\overline{x_n}.X(x_{i_1}, \ldots, x_{i_{p-1}}) \leqslant_{\beta_{sd}} \lambda\overline{x_n}.B_1) \cup (\lambda\overline{x_n}.x_{i_p} \leqslant_{\beta_{sd}} \lambda\overline{x_n}.x_{i_p} \end{array}$$

   De la deuxième équation, on déduit que $x_{i_p} \in \mathtt{fv}(B)$ et donc (voir ci-dessus) que la règle $(@_\pi)$ ne mène pas à une solution. De la première équation on déduit que $x_{i_p} \notin \mathtt{fv}(B_1)$.
   Essayons d'appliquer la règle $(@_\beta)$. On cherche C_1 tel que $B_1 x_n = C_1[z := x_n]$. Mais comme $x_{i_p} \notin \mathtt{fv}(B_1)$, nécessairement $C_1 = B_1 z$ avec $z \notin \mathtt{fv}(B_1)$ mais le terme $\lambda z.B_1 z$ n'est pas $\beta\eta$-normal donc la règle ne s'applique pas. Ce qui conclut la preuve.

□

3.3 Filtrage du second ordre

Dans cette section, nous considérons le filtrage d'ordre deux modulo $\beta\eta$. Nous allons montrer que l'algorithme pour le filtrage modulo les super-développements appliqué dans un cadre typé donne un algorithme pour le filtrage d'ordre deux. Ceci repose sur les propriétés de l'algorithme de filtrage modulo les super-développements que nous étudions dans la section 3.3.2. Nous rappelons tout d'abord l'algorithme de filtrage du second-ordre proposé dans [HL78]. Un algorithme plus efficace a été proposé dans [CQS96].

3.3.1 Algorithme de Huet et Lang

Nous rappelons dans cette section l'algorithme pour le filtrage du second ordre modulo $\beta\eta$ donné dans [HL78]. Cet algorithme est une particularisation de l'algorithme d'unification proposé dans [Hue76] et présenté sous forme de règles de transformations dans [SG89]. Nous adaptons cette dernière présentation au cas du second ordre comme cela a été fait dans [CQS96]. Rappelons que nous ne considérons pour cet algorithme que tous les termes sont en forme normale β-longue. L'application de substitutions est en particulier normalisante.

Les règles de transformation de l'algorithme de la figure 3.5 transforment un couple formé d'un multi-ensemble d'équations de filtrage et d'une substitution. Étant donné un couple $\langle \mathbb{S}, \mathtt{id} \rangle$ l'algorithme réussit s'il existe une suite de transformations terminant sur $\langle \emptyset, \sigma \rangle$. La substitution σ est dans ce cas (correction de l'algorithme) un filtre pour $\mathbb{S}$.

La première règle (Décomposition) simplifie une équation dont les symboles de tête sont identiques. Les règles (Projection) et (Imitation) traitent le cas des équations dont la tête du membre gauche est une variable de filtrage. Dans le premier cas, la variable est instanciée par une fonction de projection. C'est donc l'argument projeté qu'il va falloir rendre égal au membre droit. Dans le second cas, on réalise une instanciation partielle de la variable de filtrage dans le sens suivant : le symbole de tête du terme associé à cette variable est fixé à partir de celui du membre droit ; on introduit de nouvelles variables de filtrage pour effectuer les choix restants[1].

Nous aurions pu présenter les algorithmes donnés précédemment en construisant l'éventuelle substitution solution au fur et à mesure. Nous ne l'avons pas fait pour alléger la présentation des règles de transformation (alors que dans le cas présent c'est indispensable).

Nous illustrons l'algorithme donné dans la figure 3.5 sur un exemple.

Exemple 3.18 *Considérons l'équation* $(\lambda x.X(x, a) \leqslant_{\beta\eta} \lambda x.f(a, x, a))$ *où* a *et* f *sont des constantes de types appropriés. Clairement, la règle (Décomposition) ne peut pas s'appliquer. Appliquer la règle (Projection) mène à un blocage puisque les deux arguments de* X *(qui sont* x *et* a*) ne sont pas filtrables avec le membre droit. Par contre, la règle*

[1] L'instanciation partielle implique notamment que l'on peut substituer des variables de filtrage par des termes non $\mathcal{V}$-clos. Nous sortons donc temporairement dans cette section du cadre défini jusqu'ici afin d'exprimer l'algorithme de Huet et Lang dans sa présentation standard.

(Décomposition)

$$\langle (\lambda\overline{x}.\varepsilon(A_1,\ldots,A_n) \leqslant_{\beta\eta} \lambda\overline{x}.\varepsilon(B_1,\ldots,B_n)) \cup \mathbb{S}, \sigma\rangle$$
$$\downarrow$$
$$\langle (\lambda\overline{x}.A_1 \leqslant_{\beta\eta} \lambda\overline{x}.B_1) \cup \ldots \cup (\lambda\overline{x}.A_n \leqslant_{\beta\eta} \lambda\overline{x}.B_n) \cup \mathbb{S}, \sigma\rangle$$

(Projection)

$$\langle (\lambda\overline{x}.X(A_1,\ldots,A_n) \leqslant_{\beta\eta} \lambda\overline{x}.B) \cup \mathbb{S}, \sigma\rangle$$
$$\downarrow$$
$$\langle (\lambda\overline{x}.(A_i\sigma') \leqslant_{\beta\eta} \lambda\overline{x}.B) \cup (\mathbb{S}\sigma'), \sigma' \circ \sigma\rangle$$
$$\textbf{où } \sigma' = \{\lambda\overline{y_n}.y_i/X\}$$
$$\textbf{pour } 1 \leq i \leq n$$

(Imitation)

$$\langle (\lambda\overline{x}.X(A_1,\ldots,A_n) \leqslant_{\beta\eta} \lambda\overline{x}.\varepsilon(B_1,\ldots,B_m) \cup \mathbb{S}, \sigma\rangle$$
$$\downarrow$$
$$\langle (\lambda\overline{x}.H_1(A_1\sigma',\ldots,A'_n\sigma') \leqslant_{\beta\eta} \lambda\overline{x}.B_1)$$
$$\cup \ldots \cup (\lambda\overline{x}.H_m(A_1\sigma',\ldots,A'_n\sigma') \leqslant_{\beta\eta} \lambda\overline{x}.B_m)$$
$$\cup \mathbb{S}\sigma' \quad , \ \sigma' \circ \sigma\rangle$$

$$\textbf{où } \sigma' = \{\lambda\overline{y_n}.\varepsilon(H_1(y_1,\ldots,y_n),\ldots H_m(y_1,\ldots,y_n))/X\}$$
$$\textbf{et } H_1,\ldots,H_n \textbf{ sont des variables de filtrage fraîches et de types adéquats}$$

FIG. 3.5: Règles pour le filtrage du second-ordre de Huet et Lang

(Imitation) s'applique et donne

$$\begin{array}{l}\langle (\lambda x.H_1(x,a) \leqslant_{\beta\eta} \lambda x.a) \\ \quad \cup (\lambda x.H_2(x,a) \leqslant_{\beta\eta} \lambda x.x) \\ \quad \cup (\lambda x.H_3(x,a) \leqslant_{\beta\eta} \lambda x.a)\end{array} \quad , \{\lambda y_1 y_2.f(H(y_1,y_2),H_2(y_1,y_2),H_3(y_1,y_2)/X\}\rangle$$

La résolution des équations faisant intervenir H_1, H_2 *et* H_3 *donnent pour chacune de ces trois équations les solutions suivantes*

$$\{\lambda z_1 z_2.a/H_1\} \quad ou \quad \{\lambda z_1 z_2.z_2/H_1\}$$
$$\{\lambda z_1 z_2.z_1/H_2\}$$
$$\{\lambda z_1 z_2.a/H_3\} \quad ou \quad \{\lambda z_1 z_2.z_2/H_3\}$$

Il existe donc quatre solutions à l'équation initiale qui sont données par

$$\{\lambda y_1 y_2.f(a,y_1,a)/X\}$$
$$\{\lambda y_1 y_2.f(a,y_1,y_2)/X\}$$
$$\{\lambda y_1 y_2.f(y_2,y_1,a)/X\}$$
$$\{\lambda y_1 y_2.f(y_2,y_1,y_2)/X\}$$

3.3.2 Filtrage modulo super-développements et types

Nous pouvons tout d'abord remarquer que si un système est constitué uniquement de termes bien typés alors tous ses réduits sont constitués de termes bien typés. L'algorithme donné dans la figure 3.1 se comporte bien vis à vis des types dans le sens suivant : une suite de transformations qui aboutit à un système bien typé (c'est-à-dire un système tel que les deux membres de chacune des équations sont des termes bien typés et de même type) ne peut pas mettre en œuvre de systèmes mal typés.

Propriété 3.19 *Pour tous systèmes* $\mathbb{S}$ *et* $\mathbb{S}'$ *tels que* $\mathbb{S} \twoheadrightarrow \mathbb{S}'$*, si le système* $\mathbb{S}'$ *est bien typé alors le système* $\mathbb{S}$ *est aussi bien typé.*

Cette proposition est une conséquence directe du lemme suivant :

Lemme 3.20 *Pour tout système* $\mathbb{S}$ *tel que* $\mathbb{S}$ *ne soit pas bien typé et tout système* $\mathbb{S}'$ *tel que* $\mathbb{S} \to \mathbb{S}'$*, le système* $\mathbb{S}'$ *n'est pas bien typé.*

Preuve. Supposons donné un système $\mathbb{S}$ qui ne soit pas bien typé. On montre que tous les réduits de $\mathbb{S}$ ne sont pas bien typés en analysant règle par règle

- *Règles* (ε_v) *et* (ε_c) *:* immédiat.
- *Règle* (ε_X) *:* Les systèmes $\mathbb{S}$ et $\mathbb{S}'$ sont respectivement définis par $\mathbb{S} = (X \leqslant_{\beta_{sd}} A) \cup \mathbb{S}_0$ et par
  $\mathbb{S}' = (X \leqslant_{\beta_{sd}} A) \cup \mathbb{S}_0\{A/X\}$. Si l'équation $(X \leqslant_{\beta_{sd}} A)$ n'est pas bien typée alors le résultat est immédiat. Sinon (si $X \leqslant_{\beta_{sd}} A$ est bien typée c'est-à-dire si X et A sont deux termes bien typés de même type) le système $\mathbb{S}_0$ n'est pas bien typé. Mais comme X et A sont de même type, alors $\mathbb{S}_0\{A/X\}$ n'est donc pas bien typé.

- *Règle* $(@_\lambda)$ *:* immédiat.
- *Règle* $(@_\beta)$ *:* Si $\mathbb{S} = (A\,B \leqslant_{\beta_{sd}} C) \cup \mathbb{S}_0$ et $\mathbb{S}_0$ n'est pas bien typé alors le résultat est immédiat. Sinon, supposons que l'équation $A\,B \leqslant_{\beta_{sd}} C$ ne soit pas bien typée. Alors posons σ le type de $A\,B$ et τ celui de C avec $\tau \neq \sigma$. Il existe υ_0 et υ_1 éventuellement égaux tels que le terme A soit de type $\upsilon_0 \to \sigma$ et le terme $\lambda x.C$ de type $\upsilon_1 \to \tau$. L'équation $A \leqslant_{\beta_{sd}} \lambda x.C$ n'est pas donc bien typée, ce qui conclut le cas.
- *Règle* $(@_\pi)$ *et* $(@_@)$ *:* Même raisonnement que pour le cas précédent.

□

Nous pouvons remarquer néanmoins qu'étant donné un système bien typé, ces réduits ne sont pas forcément bien typés comme l'indique l'exemple suivant.

Exemple 3.21 (Réduits d'un système bien typé) *Soit l'équation* $XY \leqslant_{\beta_{sd}} fa$*. Si on suppose que* X *est de type* $s \to \iota$*, que* Y *est de type* s*, que* f *est de type* $r \to \iota$ *et* a *est de type* r *où* ι*,* r *et* s *sont trois types de bases différents, alors cette équation forme un système bien typé. Pourtant, en appliquant la règle* $(@_@)$ *on obtient le système* $(X \leqslant_{\beta_{sd}} f) \cup (Y \leqslant_{\beta_{sd}} a)$ *qui lui n'est pas bien typé. Néanmoins, cette équation a aussi pour solution* $\{\{\lambda x.x/X\}, \{fa/Y\}\}$ *qui est bien typée.*

La question qui suit naturellement la remarque précédente est de savoir s'il existe une β_{sd}-équation bien typée et du second ordre qui n'admet que des solutions mal typées. Dans le cas d'une réponse négative, on pourrait ainsi prouver la NP-complétude du filtrage modulo les super-développements à partir du résultat de NP-complétude du filtrage du second ordre [Bax77] (l'existence d'une solution (typée) modulo β serait alors équivalente à l'existence d'une solution modulo super-développements, ce qui prouverait que le filtrage du second ordre est NP-difficile, l'appartenance à la classe NP étant immédiate). Mais, la réponse est positive comme le montre l'exemple suivant.

Exemple 3.22 (β_{sd}-équation bien typée n'ayant que des solutions mal typées) *Soient* a *et* b *deux constantes de type* r*. Soit* f *une constante de type* $r \to \iota$ *et soit* g *une constante de type* $\iota \to \iota \to \iota$*. Soient* X *une variable de filtrage de type* $s \to \iota$*. Soient* Y *et* Z *deux variables de filtrage de type* s*.*

Considérons l'équation

$$g(XY, XZ) \leqslant_{\beta_{sd}} g(fa, fb)$$

qui est une équation bien typée et du second ordre.

L'équation $XY \leqslant_{\beta_{sd}} fa$ *possède cinq solutions données par*

$$\begin{array}{lll} X \leqslant_{\beta_{sd}} f & \cup & Y \leqslant_{\beta_{sd}} a \\ X \leqslant_{\beta_{sd}} \lambda x.fa & & \\ X \leqslant_{\beta_{sd}} \lambda x.fx & \cup & Y \leqslant_{\beta_{sd}} a \\ X \leqslant_{\beta_{sd}} \lambda x.xa & \cup & Y \leqslant_{\beta_{sd}} f \\ X \leqslant_{\beta_{sd}} \lambda x.x & \cup & Y \leqslant_{\beta_{sd}} fa \end{array}$$

De même l'équation $XZ \leqslant_{\beta_{sd}} fb$ *possède cinq solutions données par :*

$$\begin{array}{lll} X \leqslant_{\beta_{sd}} f & \cup & Z \leqslant_{\beta_{sd}} b \\ X \leqslant_{\beta_{sd}} \lambda x.fb & & \\ X \leqslant_{\beta_{sd}} \lambda x.fx & \cup & Z \leqslant_{\beta_{sd}} b \\ X \leqslant_{\beta_{sd}} \lambda x.xa & \cup & Z \leqslant_{\beta_{sd}} f \\ X \leqslant_{\beta_{sd}} \lambda x.x & \cup & Z \leqslant_{\beta_{sd}} fb \end{array}$$

Dans chaque cas, une seule des solutions est bien typée, respectivement $X \leqslant_{\beta_{sd}} \lambda x.fa$ *et* $X \leqslant_{\beta_{sd}} \lambda x.fb$. *Mais ces solutions sont bien-sûr incompatibles.*

La seule solution de l'équation initiale est

$$(X \leqslant_{\beta_{sd}} f) \cup (Y \leqslant_{\beta_{sd}} a) \cup (Z \leqslant_{\beta_{sd}} b)$$

qui est mal typée. On a donc exhibée une β*-équation du second ordre qui n'a pas de solutions mais dont la* β_{sd}*-équation correspondante en a une.*

3.3.3 Algorithme basé sur les super-développements

Dans cette section, nous nous plaçons dans le cadre du λ-calcul typé. On cherche à déterminer les β-filtres (qui sont des substitutions bien typées) d'équations d'ordre au plus 2 modulo β.

Nous avons vu que la règle η n'influence pas fondamentalement notre algorithme. Ainsi, nous aurions pu aussi bien écrire cette section pour le filtrage d'ordre 2 modulo $\beta\eta$ mais cela n'apporterait rien de plus y compris dans la comparaison avec l'algorithme de Huet et Lang.

On peut tout d'abord remarquer que l'on peut se restreindre à des équations bien typées (c'est-à-dire à des équations dont les deux termes sont de même type). Ceci est justifié par la proposition 3.19. Ainsi les règles de la figure 3.1 ne seront appliquées que si le système obtenu après réduction vérifie ces conditions. Cette contrainte sur l'application des règles est imposée dans tous les algorithmes de filtrage d'ordre supérieur dans un cadre typé (particulièrement mise en évidence dans le cas de l'application de la règle (Projection)).

De plus, on a montré (proposition 2.24) que tout β-filtre d'une équation du second-ordre était un β_{sd}-filtre pour la β_{sd}-équation correspondante. On en déduit donc qu'en appliquant l'algorithme donné figure 3.1 dans un cadre bien typé (termes toujours bien typés et systèmes toujours bien typés) alors on obtient un algorithme pour le filtrage du second ordre.

Théorème 3.23 (Algorithme pour le filtrage du second ordre) *Les règles de la figure 3.1 appliquées dans un cadre typé donnent un algorithme correct et complet pour le filtrage du second ordre.*

Nous illustrons l'algorithme sur un exemple.

Exemple 3.24 *Nous considérons l'équation* $\lambda x.X(x,a) \leqslant_{\beta\eta} \lambda x.f(a,x,a)$ *donnée dans l'exemple 3.18. Nous appliquons tout d'abord la règle* (λ_λ) *pour éliminer les* λ*-abstractions*

de tête. Examinons ensuite quelles sont les possibilités d'application de la règle $(@_\beta)$*, autrement dit quelles sont les instanciations de* B_1 *et* B_2 *(en suivant les notations de la figure 3.1). Puisque l'on doit avoir* $a \leqslant_{\beta_{sd}} B_2$*, nécessairement* $B_2 = a$*. Il y a donc trois choix pour* B_1 *:*

1. $B_1 = \lambda y_2.f(y_2, x, y_2)$ *;*
2. $B_1 = \lambda y_2.f(y_2, x, a)$ *;*
3. $B_1 = \lambda y_2.f(a, x, y_2)$.

Chacun de ses choix conduit aux solutions

$$\{\lambda y_1 y_2.f(y_2, y_1, y_2)/X\}$$
$$\{\lambda y_1 y_2.f(y_2, y_1, a)/X\}$$
$$\{\lambda y_1 y_2.f(a, y_1, y_2)/X\}$$

De plus, en appliquant la règle $(@_\pi)$ *on retrouve la quatrième solution à savoir*

$$\{\lambda y_1 y_2.f(a, y_1, a)/X\}.$$

Remarque 3.25 (Comparaisons des algorithmes pour le filtrage du 2nd ordre) *Les algorithmes pour lne filtrage d'ordre 2 de Huet et Lang et celui basé sur les super-développements sont relativement différents. En pensant à une représentation des termes sous forme d'arbre, l'algorithme de Huet et Lang compare les termes par le bas (symbole de tête) alors que l'algorithme basé sur les super-développements compare les termes par le haut (comparaison des arguments, voir remarque 3.3).*

Conclusion

Nous avons vu que le filtrage modulo super-développements est un outil simple mais puissant qui a l'avantage de ne pas être lié à un système de type. Nous verrons dans le chapitre suivant, après avoir introduit les définitions nécessaires, que cette approche est particulièrement prometteuse pour l'étude du filtrage d'ordre supérieur dans les calculs avec motifs et plus particulièrement le ρ-calcul.

D'autre part, la compréhension que nous avons apportée du filtrage modulo super-développements permet un regard nouveau sur le travail d'explicitation [BCK06] du filtrage dans les CRS. Il nous apparaît maintenant claire que ce filtrage est le filtrage modulo développements, ce qui n'est pas surprenant a posteriori puisque le méta-langage des CRS est le λ-calcul avec développements.

En mettant en avant les propriétés du λ-calcul avec super-développements et plus particulièrement celles du filtrage associé, nous soulignons que le λ-calcul avec super-développements est un méta-langage (ou « substitution calculus » au sens de van Raamdonk et van Oostrom) bien adapté à la réécriture d'ordre supérieur. On définit ainsi un nouveau formalisme d'ordre supérieur qu'il convient d'appeler les super-CRS.

Part II

Structures of the rewriting calculus

Chapter 4

The rewriting calculus

Context We present here the background theory that originally lead to the different works presented in this thesis. The rewriting calculus, also called the ρ-calculus, was introduced to give a semantics to rewrite-based languages such as Elan and in particular to rewriting strategies [BKK98a, BKK96]. To have a control on the rewriting relation, the ρ-calculus makes all the ingredients of rewriting explicit objects, in particular the notion of *rule*, *application* and *result*.

The ρ-calculus is a higher-order formalism that inherits from the λ-calculus its higher-order capabilities: the explicit treatment of functions and their applications. To express rewrite rules, the λ-abstraction is generalized to an abstraction on patterns. This means that the evaluation is based on *pattern-matching* that can be performed syntactically or modulo an equational theory. In the latter case, it may give several results that are all represented, in the ρ-calculus, with *term collections*, allowing a natural handling of the non-determinism of rewriting.

The first version of the ρ-calculus appears in [CK01, Cir00, CK98a] and focusses on the encoding of rewriting and rewriting strategies. A simplified version [CKL01] focuses on the encoding of λ-calculi of objects. Several extensions of the plain ρ-calculus have been proposed in order to handle different problems: to deal with exceptions [FK02], to handle explicitly matching and substitutions (Chapter 5), to deal with imperative features [LS04], to handle term-graphs instead of terms [BBCK04, Ber05] etc.

A polymorphic type system can be found in [CKL02] and type checking and type inference issues have been studied in [LW04]. The approach is extended to Pure Patterns Type Systems [BCKL03], a generalization of Pure Type Systems for the ρ-calculus. It has been shown that the simply-typed pure pattern type system ensures strong normalization [Wac05]. The ρ-calculus is useful in logic, more particularly in relation with deduction modulo [Wac05].

The ρ-calculus is *in fine* an extension of the λ-calculus with pattern-matching and term-collections. The combination of the λ-calculus with first-order term rewriting has been already handled by enriching first-order rewriting with higher-order capabilities, like for example in the Combinatory Reduction Systems (CRS) [Klo80, KOR93]. The link between the two formalisms has already been studied for the simplest version of the ρ-calculus (first-order algebraic patterns and term collections as tuples). In fact, CRS can be encoded in the ρ-calculus [BCK06, Ber05] and conversely, the simplest version of the ρ-calculus can also be encoded by an orthogonal CRS [BK07]. As a consequence, important properties like confluence, finite developments and standardisation are deduced for the

corresponding version of the rewriting calculus.

Contributions In this chapter, we summarize basic definitions and properties of some ρ-calculi. In the following chapter, we will extend the basic ρ-calculus to deal explicitly with the matching and the application of substitutions. The different extensions of the ρ-calculus will be used to give instances of the general calculus defined in Chapter 6.

Following the work presented in Part I on higher-order matching in the λ-calculus, we give some hints on higher-order matching in the ρ-calculus.

Outline of the chapter In Section 4.1 and Section 4.2 we introduce the syntax and the operational semantics of the ρ-calculus. In Section 4.3 we give some already known results on the confluence of the ρ-calculus. In Section 4.4, we give some results on the expressiveness of the ρ-calculus. In Section 4.5, we present two extensions of the ρ-calculus. Section 4.6 discusses higher-order matching in the ρ-calculus.

4.1 Syntax

Given a set of constants (denoted $c, d, e, f, \ldots$) anda set of variables denoted $x, y, z, \ldots$) and the set of ρ-terms is defined in Figure 4.1.

In the ρ-calculus, the usual λ-abstraction $\lambda x.A$ is generalized by a *pattern-abstraction* (also called rule abstraction) $P \twoheadrightarrow A$ where P can be a more general pattern than a single variable x. The left-hand side of an abstraction defines the variables we abstract on and some context information. An *application* is implicitly denoted by concatenation. The terms can be grouped together into *structures* that can be seen as collections of terms. In an abstraction of the form $P \twoheadrightarrow A$ we call the term P the pattern and the term A the body.

We assume that the application operator associates to the left, while the other operators associate to the right. The priority of the application operator is higher than that of $\_ \twoheadrightarrow \_$ which is, in turn, of higher priority than the $\_ \wr \_$.

Definition 4.1 (Algebraic terms) *We call* algebraic *the terms of the form*

$$(\ldots((f\,A_1)\,A_2)\ldots)\,A_n$$

where f *is a constant. We usually denote them by* $f(A_1, A_2, \ldots, A_n)$.

Definition 4.2 (Free and bound variables) *The set of free variables of a term* A *and the set of bound variables, denoted* $\mathtt{fv}(A)$ *and* $\mathtt{bv}(A)$*, are defined inductively by:*

$$\begin{array}{lcl@{\qquad}lcl}
\mathtt{fv}(c) & \triangleq & \emptyset & \mathtt{fv}(x) & \triangleq & \{x\} \\
\mathtt{bv}(c) & \triangleq & \emptyset & \mathtt{bv}(x) & \triangleq & \emptyset \\
\mathtt{fv}(A\,B) & \triangleq & \mathtt{fv}(A \wr B) \triangleq \mathtt{fv}(A) \cup \mathtt{fv}(B) & \mathtt{fv}(P \twoheadrightarrow A) & \triangleq & \mathtt{fv}(A) \backslash \mathtt{fv}(P) \\
\mathtt{bv}(A\,B) & \triangleq & \mathtt{fv}(A \wr B) \triangleq \mathtt{bv}(A) \cup \mathtt{bv}(B) & \mathtt{bv}(P \twoheadrightarrow A) & \triangleq & \mathtt{bv}(A) \cup \mathtt{bv}(P) \cup \mathtt{fv}(P)
\end{array}$$

A, B, P	$::=$	x	(Variable)
	$\mid$	c	(Constant)
	$\mid$	$P \twoheadrightarrow A$	(Abstraction)
	$\mid$	$A\ B$	(Application)
	$\mid$	$A \wr B$	(Structure)

Figure 4.1: Syntax of the ρ-calculus

When the set of free variables of a term is empty we say that the term is closed. Otherwise, it is open.

In what follows we work modulo α-conversion, that is two terms that are α-convertible are not distinguishable. Equality modulo α-conversion is denoted here by $\equiv$. We adopt Barendregt's *hygiene-convention* [Bar84], *i.e.* for a given term, the set of free and bound variables are disjoint.

A *substitution* is a partial function from variables to terms (we use post-fix notation for substitution application). We denote by $\sigma = \{x_1 \leftarrow A_1, \ldots, x_n \leftarrow A_n\}$ the substitution that maps each variable x_i to a term A_i. The set $\{x_1, \ldots, x_n\}$ is called the domain of σ and is denoted $\mathcal{D}om(\sigma)$.

Definition 4.3 (Substitution) *The application of a substitution* σ *to a term* A *is inductively defined by*

$x\sigma$	$\triangleq$	A	*if* $\sigma = \{\ldots, x \leftarrow A, \ldots\}$
$y\sigma$	$\triangleq$	y	*if* $y \notin \mathcal{D}om(\sigma)$
$c\sigma$	$\triangleq$	c	
$(P \twoheadrightarrow B)\sigma$	$\triangleq$	$P \twoheadrightarrow (B\sigma)$	
$(A_1A_2)\sigma$	$\triangleq$	$(A_1\sigma)(A_2\sigma)$	
$(A_1 \wr A_2)\sigma$	$\triangleq$	$(A_1\sigma) \wr (A_2\sigma)$	

In the abstraction case, we take the usual precautions to avoid variable captures.

The range of a substitution σ, denoted $\mathcal{R}an(\sigma)$, is the union of the sets $\mathtt{fv}(x\sigma)$ where $x \in \mathcal{D}om(\sigma)$. The set of the variables of a substitution σ is denoted $\mathcal{V}ar(\sigma)$ and is defined by $\mathcal{D}om(\sigma) \cup \mathcal{R}an(\sigma)$.

The composition of two substitutions σ and τ is denoted $\sigma \circ \tau$ and defined as usually, that is $x(\sigma \circ \tau) = (x\tau)\sigma$. We denote by $\mathtt{id}$ the empty substitution.

The restriction of (the domain of) a substitution σ to a set of variables θ is denoted $\sigma_{|\theta}$.

One can easily remark that the set of ρ-terms strictly contains the set of λ-terms.

Remark 4.4 (Translation of λ-terms) *One can encode the* λ*-calculus into the* ρ*-calculus : given the set of* λ*-terms the translation function* $[\![\ ]\!]$ *from* λ*-terms to* ρ*-terms is defined by*

$$\begin{array}{lcl} [\![x]\!] & = & x \\ [\![\lambda x.A]\!] & = & x \twoheadrightarrow [\![A]\!] \\ [\![A\ B]\!] & = & [\![A]\!]\ [\![B]\!] \end{array}$$

Thus, when translating λ*-terms into* ρ*-terms the binder* λ *is replaced by the (rule) abstraction operator* $\twoheadrightarrow$ *like for the following terms:*

λ-*calculus*	ρ-*calculus*
$\lambda x.x$	$x \twoheadrightarrow x$
$\lambda x.\lambda y.x$	$x \twoheadrightarrow y \twoheadrightarrow x$
$\lambda x.(x\,x)$	$x \twoheadrightarrow x\,x$

As we will see in the next section, this translation is consistent with the reduction in the two formalisms: for each β*-reduction of a* λ*-term there exists a corresponding reduction of the translated term in the* ρ*-calculus.*

Example 4.5 (Encoding of propositional formulae) *Using the constants* t, f, not, and, or, xor *(denoting respectively the boolean values true and false, the negation, the conjunctionn the disjunction and the exclusive disjunction) we can define the following propositional formulae:* $\mathsf{and}(x,\mathsf{t})$ *and* $\mathsf{or}(\mathsf{not}(x),\mathsf{not}(y))$.

Example 4.6 (Rewrite rules) *Some rules to compute in the Boolean algebra:*

- $\mathsf{and}(x,\mathsf{t}) \twoheadrightarrow x$*; the occurence of the variable* x *in the body is bound by the pattern* $\mathsf{and}(x,\mathsf{t})$.

  in this abstraction the variable x *in the body is bound by the variable in the pattern.*

- $\mathsf{not}(\mathsf{and}(x,y)) \twoheadrightarrow \mathsf{or}(\mathsf{not}(x),\mathsf{not}(y))$*; this rule bounds the variables* x *and* y.

- $\mathsf{xor}(x,x) \twoheadrightarrow \mathsf{f}$*; a non-linear rule.*

The application of the second rewrite rule to the term $\mathsf{not}(\mathsf{and}(\mathsf{t},\mathsf{f}))$ *is represented by the term*

$$\big(\mathsf{not}(\mathsf{and}(x,y)) \twoheadrightarrow \mathsf{or}(\mathsf{not}(x),\mathsf{not}(y))\big)\ \mathsf{not}(\mathsf{and}(\mathsf{t},\mathsf{f}))$$

and, as we will see in the next section, this term reduces to $\mathsf{or}(\mathsf{not}(\mathsf{t}),\mathsf{not}(\mathsf{f}))$.

4.2 Operational semantics

The evaluation mechanism of the ρ-calculus relies of the fundamental operation of *matching*. When applying a pattern-abstraction, this operation allows to bind variables to their corresponding values. The general framework of the ρ-calculus allows pattern-matching to be performed syntactically or modulo an equational theory. The set of patterns and the congruence on terms modulo which the matching is performed are thus parameters of the calculus. The third parameter is the underlying theory for the $\wr$ operator (typically, a combination of associativity, commutativity and idempotence). We will see that in case of non-unitary matching the third parameter can not be arbitrarily chosen.

In this section, we give some examples of matching algorithms that can be used in the evaluation of the ρ-calculus. We then define the operational semantics of the ρ-calculus.

4.2.1 Matching

We first define the operation of matching and then give some examples of matching algorithms for different sets of patterns.

Definition 4.7 (Matching) *Given a theory $\mathbb{T}$, i.e. a set of axioms defining a congruence relation $\overset{\mathbb{T}}{=}$, we define*

1. *A matching equation (also called a matching problem) is a pair denoted $P \prec\!\!<_{\mathbb{T}} A$ with P a pattern and A a term.*

2. *A substitution σ is a solution of the matching equation $P \prec\!\!<_{\mathbb{T}} A$ if $P\sigma \overset{\mathbb{T}}{=} A$.*

The set of solutions of $P \prec\!\!<_{\mathbb{T}} A$ is denoted by $\mathcal{S}\mathtt{ol}(P \prec\!\!<_{\mathbb{T}} A)$.

We often consider $\mathcal{S}\mathtt{ol}$ as a function that given a pattern and a term returns the set of substitutions (possibly empty) of the matching problem. When it is clear from the context, we do not always explicitly mention the congruence used to performed matching.

As we will see in the next sections, when studying confluent instances of the ρ-calculus, we consider particular sets of patterns.

Definition 4.8 ((Structure) algebraic patterns) *The set of algebraic and structure algebraic patterns are defined as follows:*

$$\begin{array}{lrcl} \textbf{Algebraic patterns} & P, Q & ::= & x \\ & & | & c \\ & & | & f(P_1, P_2, \ldots, P_n) \end{array}$$

$$\begin{array}{lrcl} \textbf{Structure algebraic patterns} & P, Q & ::= & x \\ & & | & c \\ & & | & f(P_1, P_2, \ldots, P_n) \\ & & | & P \wr Q \end{array}$$

Definition 4.9 (Syntactic matching) *We consider that the matching constraints are of the form $A \ll_\emptyset B$ and we consider possibly empty conjunctions of such problems built with the associative commutative operator $\wedge$. The following set of terminating and confluent rules can be used to solve the (non-linear) syntactic matching:*

$$\begin{array}{lcl} A \ll_\emptyset A \wedge M & \to & M \\ (A_1 A_2 \ll_\emptyset B_1 B_2) \wedge M & \to & (A_1 \ll_\emptyset B_1) \wedge (A_2 \ll_\emptyset B_2) \wedge M \end{array}$$

The set of solutions of a matching problem $P \prec\!\!<_\emptyset A$ is obtained by normalizing the matching constraint $P \ll_\emptyset A$ w.r.t. the above rewrite rules and according to the obtained result we have:

- $\mathcal{S}ol(P \lessdot A) = \{x_i \leftarrow A_i\}_{i\in I}$ *if the result is of the form* $\bigwedge_{i\in I\neq\emptyset}(x_i \ll_\emptyset A_i)$ *with* $A_i = A_j$ *if* $x_i = x_j$*;*
- $\mathcal{S}ol(P \lessdot A) = \mathtt{id}$ *if the result is the empty conjunction;*
- $\mathcal{S}ol(P \lessdot A) = \emptyset$ *otherwise.*

We will see that syntactical matching is often used for a particular set of patterns. We thus give particular cases of this algorithm.

Definition 4.10 (Syntactic matching of algebraic patterns) *When dealing with algebraic patterns one could instead replace the decomposition rule for application by the following rule:*

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

Definition 4.11 (Syntactical matching of the structure operator) *When no theory is used for the structure operator then the structure operator can be considered as a constant and the previous matching algorithms should be enriched with the following rule:*

$$(P \wr Q \ll_\emptyset A_1 \wr A_2) \wedge M \qquad \rightarrow \qquad (P \ll_\emptyset A_1) \wedge (Q \ll_\emptyset A_2) \wedge M$$

4.2.2 Evaluation rules

The operational semantics of the ρ-calculus is given in Figure 4.2 where $\mathbb{T}$ denotes the congruence modulo which matching problems are solved. It consists of two rules.

The (ρ) rule deals with the application of pattern-abstractions : when applying a rule $P \twoheadrightarrow A$ to a term B, we compute the potential solutions of the matching between P and B modulo the congruence $\mathbb{T}$. When such solutions exist, the corresponding substitutions are applied to A and the results are then aggregated using the structure operator.

The (δ) rule deals with the application of structures. When applying a structure $A_1 \wr A_2$ to a term B, we simply distribute the application of the argument to the elements of the structure. Intuitively, when A_1 and A_2 are rules, this could model the application of several rules of a rewrite system to B.

Following the notation introduced in Part I, we denote the compatible closure of a relation $\mathcal{R}$ either $\rightarrow_\mathcal{R}$ or by a litte abuse of notation simply $\mathcal{R}$. Its transitive and reflexive closure is denoted either by $\twoheadrightarrow_\mathcal{R}$ or by $\mathcal{R}^*$. For example, $\twoheadrightarrow_{\rho\delta}$ denotes the reflexive and transitive closure of the compatible relation induced by the rules (ρ) and (δ).

$$
\begin{array}{lll}
(P \twoheadrightarrow A)\, B & \to_{\rho} & A\sigma_1 \wr \ldots \wr A\sigma_n \\
 & & \textbf{where } \{\sigma_1, \ldots, \sigma_n\} = \mathcal{S}ol(P \lessdot_{\mathbb{T}} C) \\
 & & \textbf{and } n > 0 \\
(A_1 \wr A_2)\, B & \to_{\delta} & A_1\, B \wr A_2\, B
\end{array}
$$

Figure 4.2: Operational semantics of the ρ-calculus

Example 4.12 (Reductions)

1. *When considering syntactic matching ($\mathbb{T} = \emptyset$), we have*

$$
\begin{array}{rll}
(x \twoheadrightarrow A)\, B & \to_{\rho} & A\{x \leftarrow B\} \\
(a \twoheadrightarrow A)\, a & \to_{\rho} & A \\
(a \twoheadrightarrow a)\, b & & \textit{is in normal form since } a \textit{ and } b \textit{ do not match} \\
(f(x) \twoheadrightarrow g(x))\, f(a) & \to_{\rho} & g(a) \\
(a \twoheadrightarrow b \wr a \twoheadrightarrow c)\, a & \to_{\delta} & (a \twoheadrightarrow b)\, a \wr (a \twoheadrightarrow c)\, a \\
 & \to_{\rho} & b \wr (a \twoheadrightarrow c)\, a \\
 & \to_{\rho} & b \wr c
\end{array}
$$

2. *In an equational theory where $f\,x\,y =_{\mathbb{T}} f\,y\,x$, we have*

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

Note that in case of non-unitary matching, the underlying theory given to the structure operator must include associativity and commutativity if we want to consider confluent reductions. In fact, the function $\mathcal{S}ol$ associated to a non-unitary matching theory returns a (multi)set of substitutions and to the knowledge of the author there is no order for substitutions that is adequate to higher-order contexts (the subsumption order is not).

Example 4.13 (Application of a rewrite system) *In a first approximation, the application of a structure of rewrite rules can be seen as the application of a rewrite system. The term*

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

represents the application of the rewrite system

$$
\begin{cases}
\mathsf{and}(x, \mathsf{or}(y, z)) & \to & \mathsf{or}(\mathsf{and}(x, y), \mathsf{and}(x, z)) \\
\mathsf{and}(\mathsf{or}(x, y), z) & \to & \mathsf{or}(\mathsf{and}(x, z), \mathsf{and}(y, z))
\end{cases}
$$

to the term $\mathsf{and}(\mathsf{or}(\mathsf{t},\mathsf{f}),\mathsf{or}(\mathsf{f},\mathsf{f}))$. *We use the* ($\delta$) *rule to distribute the two rewrite rules and we obtain:*

$$\begin{aligned}\rightarrow_{\delta}\quad & \big(\mathsf{and}(x,\mathsf{or}(y,z)) \twoheadrightarrow \mathsf{or}(\mathsf{and}(x,y),\mathsf{and}(x,z))\big)\ \mathsf{and}(\mathsf{or}(\mathsf{t},\mathsf{f}),\mathsf{or}(\mathsf{f},\mathsf{f}))\ \wr\\ & \big(\mathsf{and}(\mathsf{or}(x,y),z) \twoheadrightarrow \mathsf{or}(\mathsf{and}(x,z),\mathsf{and}(y,z))\big)\ \mathsf{and}(\mathsf{or}(\mathsf{t},\mathsf{f}),\mathsf{or}(\mathsf{f},\mathsf{f}))\end{aligned}$$

and we perform two reductions to obtain:

$$\mathsf{or}(\mathsf{and}(\mathsf{or}(\mathsf{t},\mathsf{f}),\mathsf{f}),\mathsf{and}(\mathsf{or}(\mathsf{t},\mathsf{f}),\mathsf{f}))\ \wr\ \mathsf{or}(\mathsf{and}(\mathsf{t},\mathsf{or}(\mathsf{f},\mathsf{f})),\mathsf{and}(\mathsf{f},\mathsf{or}(\mathsf{f},\mathsf{f})))$$

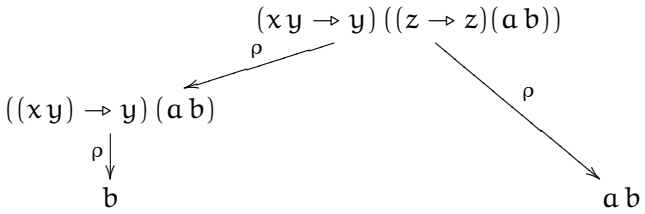
The application of a rewrite system is actually never as simple as presented above. Here, we encode only one (meta) rewriting step but, in general, the encoding is more complicated because one needs to encode not only the application of a rewrite rule at the top position of a term but also the reduction strategy guiding the application of the rules. The problem can be solved, for example, by using (typed) fix-points to apply the rewrite system recursively (see [CKLW03] for a full presentation).

4.3 Confluence

This section recalls basic results for confluence of the ρ-calculus. A more detailed study of confluence of pattern-based calculi will be done in Chapter 6.

4.3.1 Syntactical restrictions

Using syntactic matching can lead to non-confluent reductions if we do not restrict the shape of patterns. The reductions



illustrate that when using higher-order patterns (like xy), first-order matching has to be used with care (this is not surprising).

More surprisingly first-order non-linear patterns can lead to non-confluent reductions. To ensure confluence, we usually use the Rigid Pattern Condition which is due to V. van Oostrom [Oos90, KOV07]. We present in the following definition a characterization of this condition and we will be discussed in the general definition in Chapter 6.

Definition 4.14 (Rigid Pattern Condition) *A pattern* P *satisfies the rigid pattern condition if it is*

— *linear (each free variable occurs at most once),*
— *in* $\rho\delta$*-normal form,*
— *with no active variables (i.e., no sub-terms of the form* xA *where* x *is free).*

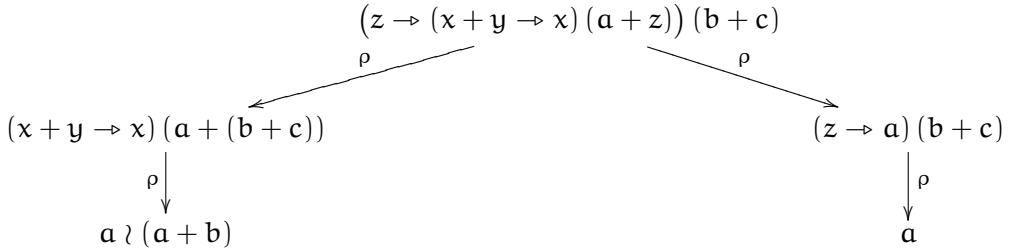
Theorem 4.15 (Confluence by RPC) *The relation $\rho\delta$ is confluent if each pattern verifies the Rigid Pattern Condition.*

Unfortunately, this result does not generalize to other matching theories even for very natural and simple ones.

Example 4.16 (Confluence with non-unitary matching) *Consider an associative symbol "+"*

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

We have two reductions



When the inner redex is reduced, since the variable z is not yet instantiated, it is not possible to apply the associativity to "rearrange" the term $a+z$ – like for the term $a+(b+c)$ in the alternative reduction – and thus some solutions are lost.

To recover confluence, we can use reduction strategies.

4.3.2 Reduction strategies

In the previous section, we give a syntactical restriction to guarantee confluence of the ρ-calculus (Theorem 4.15). We show on an example that these restrictions are difficult to adapt in the case of non-unitary pattern-matching. We propose in this section a call by value calculus, inspired by [Plo75], which leads to confluent instances of the ρ-calculus for any matching theory.

Definition 4.17 (Reduction strategy for the ρ-calculus) *A reduction strategy for the ρ-calculus is given by a predicate $\mathcal{C}$ on the set of ρ-terms such that the (ρ) evaluation rule is now defined by*

$$(P \rightarrowtail A)\, B \quad \to_{\rho} \quad A\sigma_1 \wr \ldots \wr A\sigma_n \quad \text{if } \mathcal{S}ol(P \lessdot_{\mathbb{T}} B) = \{\sigma_1, \ldots, \sigma_n\} \text{ with } n > 0 \text{ and } B \text{ satisfies } \mathcal{C}$$

The condition $\mathcal{C}$ is often defined by (the membership to) a set of terms.

We define two evaluation strategies in the ρ-calculus given by two sets of terms.

Definition 4.18 (Values in the ρ-calculus) *The set of values is defined by*

$$V \quad ::= \quad c \mid P \twoheadrightarrow T \mid f(V_1, \ldots, V_n) \mid V \wr V$$

The call by value strategy is the evaluation strategy defined using the given set of values.

The following theorem states that the ρ-calculus is confluent when using the call by value strategy and the syntactic matching (syntactical decomposition of the application as in Definition 6.2) on arbitrary patterns.

Theorem 4.19 (Confluence by values [CK01, Cir00]) *The relation ρδ is confluent if the ρ-rule is applied with the call by value strategy and the syntactical matching.*

It is not difficult to remark that if the (ρ) rule is applied when the arguments of pattern-abstractions cannot evolve anymore then the calculus is confluent since there are only trivial critical pairs. Example 4.16 of non-confluent reductions in presence of an associative symbol is now solved because the evaluation which leads to a is no longer possible.

Definition 4.20 (Rigid values) *A ρ-term A is called a rigid value if it is closed and it is in normal form for ρδ.*

The call by rigid value strategy is the evaluation strategy defined using the set of rigid values.

Theorem 4.21 (Confluence by rigid values [CHW06, Cir00]) *The relation ρδ is confluent for any matching equational theory defined on a set of algebraic patterns if the ρ-rule is applied with the call by rigid value strategy.*

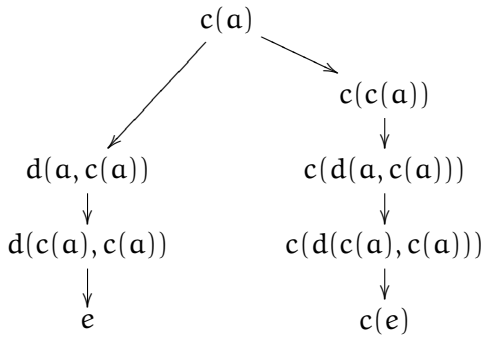
4.3.3 Encoding Klop's counter example in the ρ-calculus

Confluence of higher-order rewriting systems dealing with non-linear matching is a difficult task since we usually obtain non-joinable critical pairs as those coined for the first time in [Klo80].

We first recall that Klop's counter example in classical rewriting is based on a non-linear rewrite rule. We can adapt it in the λ-calculus enriched with a rewriting rule and finally in the ρ-calculus. Let us begin by describing Klop's counter example in classical rewriting.

Example 4.22 (Classical rewriting) *Let us consider the first-order rewrite system consisting of the rewrite rules* $\{d(x,x) \to e, c(x) \to d(x, c(x)), a \to c(a)\}$. *The following*

reductions are obtained when reducing the term $\mathsf{c}(\mathsf{a})$*:*



We cannot close the diagram since e *is in normal form and the smallest reduction from* $\mathsf{c}(\mathsf{e})$ *to* e *would reduce* $\mathsf{d}(\mathsf{e},\mathsf{c}(\mathsf{e}))$ *which can only be reduced by a reduction from* $\mathsf{c}(\mathsf{e})$ *to* e*, which contradicts the minimality of the reduction.*

We can go a step further in the encoding by considering λ-terms and higher-order rewriting.

Example 4.23 (λ-calculus and rewriting) *The usual fixpoint combinator* Y *is defined by*

$$\mathsf{Y} \triangleq \Big(\lambda y.\lambda x.(x\,(y\,y\,x))\Big)\,\Big(\lambda y.\lambda x.(x\,(y\,y\,x))\Big)$$

We introduce the following λ*-terms and the corresponding reductions:*

$$\begin{array}{lcl} \mathsf{C} & \equiv & \mathsf{Y}\big(\lambda y.\lambda x.\mathsf{d}(x,y\,x)\big) \\ \mathsf{C} & \twoheadrightarrow_\beta & \big(\lambda y.\lambda x.\mathsf{d}(x,y\,x)\big)\;\mathsf{C} \\ & \to_\beta & \lambda x.\mathsf{d}(x,\mathsf{C}\,x) \end{array} \qquad \begin{array}{lcl} \mathsf{A} & \equiv & \mathsf{Y}\mathsf{C} \\ \mathsf{A} & \twoheadrightarrow_\beta & \mathsf{C}\,\mathsf{A} \end{array}$$

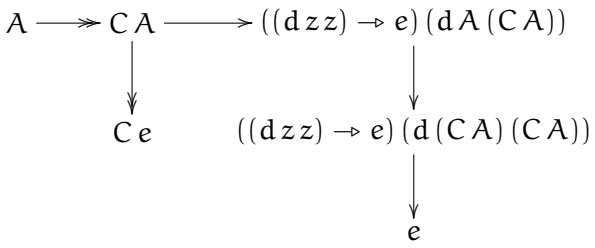
The λ*-term* C *simulates the behavior of the constant* c *and of the second rewrite rule, whereas* A *simulates the behavior of the constant* a *and of the third rewrite rule. We can thus omit these two rules and consider a variation of Klop's example. If we add in the* λ*-calculus the non-linear rewrite rule* $\mathcal{R} = \{\mathsf{d}(x,x) \to \mathsf{e}\}$ *to the* β*-reduction then we obtain a non confluent calculus. In fact, the* λ*-term* A *reduces by* $\to_{\beta\cup\mathcal{R}}$ *on the one hand to* e *and on the other hand to* $(\mathsf{C}\,\mathsf{e})$*. These two terms do not share a common reduct.*

We are now ready to give the encoding in the ρ-calculus.

Example 4.24 (ρ-calculus) *The usual fixpoint combinator* Y *is defined as in the* λ*-calculus and for any* ρ*-term* A *we have* $\mathsf{Y}\,\mathsf{A} \twoheadrightarrow_\rho \mathsf{A}\,(\mathsf{Y}\,\mathsf{A})$*. We now define the following terms as in the previous example except that the rule* $\mathsf{d}(z,z) \to \mathsf{e}$ *is directly encoded in the term* C*.*

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

We have the following reductions:



The constant e *is in normal form and in the smallest reduction from* $(\mathsf{C}\,\mathsf{e})$ *to* e *the head redex must be reduced. After several reductions, we obtain* $((\mathsf{d}\,\mathsf{z}\,\mathsf{z}) \nrightarrow \mathsf{e})\,(\mathsf{d}\,\mathsf{e}\,(\mathsf{C}\,\mathsf{e}))$ *which can be head reduced only after a reduction from* $(\mathsf{C}\,\mathsf{e})$ *to* e*. Since the reduction is supposed to be minimal, we conclude by contradiction that* $(\mathsf{C}\,\mathsf{e})$ *cannot be reduced to* e *and thus that the two reductions cannot be joined.*

4.4 Expressiveness

An easy but fundamental result is that when using syntactic pattern-matching, the λ-calculus can be simulated in the ρ-calculus. We refer to Section 4.1 for the (trivial) encoding of λ-terms in ρ-terms that we omit in the following theorem.

Theorem 4.25 (Simulating λ-calculus in ρ-calculus) *If* A *and* B *are two* λ*-terms viewed as* ρ*-terms then*

$$\mathsf{A} \to_\beta \mathsf{B} \qquad \textit{if and only if} \qquad \mathsf{A} \to_\rho \mathsf{B}.$$

Several object oriented calculi, namely the Lambda Calculus of Objects of Fisher, Honsell, and Mitchell [MHF94] and the Object Calculus of Abadi and Cardelli [AC96] can also be encoded in the ρ-calculus [CKL01].

As far as it concerns rewriting, the comparison is a bit more elaborated since the evaluation in the ρ-calculus and in rewriting is quite different:

- in the ρ-calculus rules are consumed once they have been used ;
- the positions at which a rewrite rule is applied must be explicitly by using congruence rules ;
- the order of applications of the rules (the strategy) must be also explicitly given.

The first result states that rewriting derivations can be encoded in the ρ-calculus. Algebraic terms of rewriting are directly translated into the ρ-calculus by algebraic ρ-terms. We underline this trivial encoding using the function $[\![\ ]\!]$.

Theorem 4.26 (Representing a rewriting path [CK01]) *Let* $\mathcal{R}$ *be a rewrite system and* t, t' *be two algebraic terms. If* $\mathsf{t} \twoheadrightarrow_{\mathcal{R}} \mathsf{t}'$*, then there exists a* ρ*-term* A *such that*

$$\mathsf{A}\,[\![\mathsf{t}]\!] \twoheadrightarrow_{\rho\delta} [\![\mathsf{t}']\!].$$

Syntax of the ρ_{stk}-calculus

Patterns	P	$::=$	$x \mid \mathsf{stk} \mid f(P, \ldots, P)$ *(variables occur only once in any P)*
Terms	A, B	$::=$	$x \mid c \mid \mathsf{stk} \mid P \twoheadrightarrow A \mid A\,B \mid A \wr B$

Semantics of the ρ_{stk}-calculus

(ρ_{stk})	$(P \twoheadrightarrow A)\,B$	$\to$	$A\sigma$	with $\mathcal{S}ol(P \lessdot B) = \sigma$
(δ)	$(A_1 \wr A_2)\,B$	$\to$	$A_1\,B \wr A_2\,B$	
(stk)	$(P \twoheadrightarrow A)\,B$	$\to$	stk	if $P \not\sqsubseteq A$
(stk)	$\mathsf{stk} \wr A$	$\to$	A	
(stk)	$A \wr \mathsf{stk}$	$\to$	A	
(stk)	$\mathsf{stk}\,A$	$\to$	stk	

Figure 4.3: The ρ_{stk}-calculus

We can go a step further and show that the behavior of a certain class of rewrite systems can be simulated.

Theorem 4.27 (Representing a rewriting system [CLW03]) *Let $\mathcal{R}$ be a rewrite system. There exists a ρ-term A such that for all algebraic terms t, t'*

1. *If $A\,[\![t]\!] \twoheadrightarrow_{\rho\delta} [\![t']\!]$, then $t \twoheadrightarrow_{\mathcal{R}} t'$.*
2. *If $\mathcal{R}$ is confluent and terminating and if $t \twoheadrightarrow_{\mathcal{R}} t'$, then $A\,[\![t]\!] \twoheadrightarrow_{\rho\delta} [\![t']\!]$.*

The last step is to show that the behavior of a rewriting system under some strategies can also be simulated. This is done for example in [CLW03, CKLW03].

4.5 Extensions of the ρ-calculus

4.5.1 The ρ_{stk}-calculus

We have seen in the previous section that when matching does not succeed then no reduction is performed. For example, the term $(a \twoheadrightarrow b)\,c$ is in normal form. We focus here on a version of the ρ-calculus that uses a special constant stk to represent matching failures. It is called the ρ_{stk}-calculus in [CLW03] and was used to encode rewrite systems but also to give semantics to functional programming languages. The syntax of the ρ_{stk}-calculus is given in Figure 4.3. The patterns are algebraic patterns built using the special constant stk. We have to decide when a matching problem associated to the application of pattern-abstraction has and will have no solution in order to reduce

the term to stk. This is the case when neither an instantiation of the argument nor a reduction can make the problem solvable. In other words,

$$\frac{\forall \sigma_1, \sigma_2,\ \forall B',\ B\theta_1 \twoheadrightarrow_{\rho\delta} B' \ \Rightarrow\ P\sigma_2 \not\equiv B'}{(P \rightarrowtriangle A)\, B \to_{\mathsf{stk}} \mathsf{stk}}$$

Nevertheless, this evaluation rule is not decidable. This is why we introduce a superposition relation $\not\trianglelefteq$ between (patterns and) terms whose aim is to characterize a broad class of matching equations that have not and will never have a solution (*i.e.* independently of subsequent instantiations and reductions).

Definition 4.28 (Definitive failures) *The relation $\not\trianglelefteq$ is defined as follows*

stk	$\not\trianglelefteq$	$g(B_1, \ldots, B_n)$	*if* $g \not\equiv \mathit{stk}$
stk	$\not\trianglelefteq$	$Q \rightarrowtriangle B$	
$f(P_1, \ldots, P_m)$	$\not\trianglelefteq$	$g(B_1, \ldots, B_n)$	*if* $f \not\equiv g$ *or* $n \neq m$ *or* $\exists i,\ P_i \not\trianglelefteq B_i$
$f(P_1, \ldots, P_m)$	$\not\trianglelefteq$	stk	
$f(P_1, \ldots, P_m)$	$\not\trianglelefteq$	$Q \rightarrowtriangle B$	
$f(P_1, \ldots, P_m)$	$\not\trianglelefteq$	$(Q \rightarrowtriangle A)B$	*if* $Q \not\trianglelefteq B$ *or* $f(P_1, \ldots, P_m) \not\trianglelefteq A$

The ρ_{stk}-calculus handles uniformly matching failures and eliminates them when they are not significant for the computation. The semantics of the ρ_{stk}-calculus is given in Figure 4.3.

We show on an example how the ρ_{stk}-calculus can be used to encode term rewriting systems and to give semantics to functional programming languages. Further details can be found in [CLW03].

Example 4.29 (Encoding of the Peano addition) *We suppose given the constants* $0, S, \mathtt{add}$ *and* $\mathtt{rec}$. *We define the following* ρ*-term*

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

The variable z *will contain a copy of* $\mathtt{plus}$ *to allow "recursive calls". We show in Figure 4.4 that this term actually computes the addition over Peano integers.*

The notations $\overline{m}$, $\overline{m+n}$ *and* $\overline{m-n}$ *have to be understood as the terms* $S(\ldots(S\,0)\ldots)$ *with the right number of* S *symbols.*

The term $\mathtt{plus}(\mathtt{rec}\,\mathtt{plus})\ (\mathtt{add}\,\overline{n}\,\overline{m})$ *can be infinitely reduced but it is weakly normalising: the reduction* $\to_{\mathit{stk}}$ *removes the* $n-1$ *first attempt to apply the rule* $(\mathtt{add}\,0\,y) \rightarrowtriangle y$ *since* $0 \not\trianglelefteq \overline{n}$ *as soon as* $n > 0$. *At the end of the reduction, the step* $\to_{\mathit{stk}}$ *removes the non-terminating term since* $S\,x \not\trianglelefteq 0$. *The normal form* $\overline{m+n}$ *(which is actually unique since the* ρ_{stk}*-calculus is confluent) is sound.*

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

Figure 4.4: Reduction of a ρ-term encoding the addition

4.5.2 The ρ_d-calculus

Term rewriting systems and classical guiding strategies have been encoded in the original rewriting calculus [CK01] and in the ρ_{stk}-calculus [CLW03]. The former encoding is rather elaborated while the latter is simpler but restricted to convergent (*i.e.* confluent and strongly normalizing) term rewriting systems. To extend this encoding to arbitrary term rewrite systems, a new evaluation rule that enriches the semantics of the structure operator is added and an evaluation strategy is enforced by imposing a certain discipline on the application of the evaluation rules. This strategy is defined syntactically using an appropriate notion of value and is used in order to recover confluence of the calculus that is lost in the general case.

The obtained calculus is the distributive rewriting calculus, called the ρ_d-calculus and introduced in [CHW06]. This formalism shares the syntax of the ρ_{stk}-calculus and uses the notion of *values* that intuitively represent the terms that we do not need to evaluate. These values can be extended to the so-called *structure values* and *stuck values* that will be used to restrict the applications of the evaluation rules.

The syntax and the operational semantics of the ρ_d-calculus is given in Figure 4.5. The main difference w.r.t. the ρ_{stk}-calculus is the γ rule that although seems very natural and not dangerous leads to simple non confluent reductions if not restricted to structure values [CHW06].

Thanks to the (γ) rule, which defines the right-distributivity of the application over the structure, we can encode potentially non-confluent TRS in the ρ_d-calculus (this contrasts with the ρ_{stk}-calculus). More precisely, given a TRS $\mathcal{R}$ we can build the terms $\Omega^1_{\mathcal{R}}$ and $\Omega_{\mathcal{R}}$ such that

- $\Omega^1_{\mathcal{R}}\ m$ represents (*i.e.* reduces to) the one-step reducts of m *w.r.t.* $\mathcal{R}$,
- $\Omega_{\mathcal{R}}\ m$ represents the normal forms of m *w.r.t.* $\mathcal{R}$ (if they exist).

4.6 Higher-order matching in the ρ-calculus

In Part I, we studied higher-order matching in the untyped λ-calculus modulo a restriction of the β-conversion. Now that the ρ-calculus has been introduced, we give a smooth and very short introduction to higher-order matching in the ρ-calculus. We mainly emphasize some of the problems that are interesting for a future study.

In this section, we use the notations of Chapter 2 and in particular capital letters denote matching variables. We call matching in the ρ-calculus the process of finding a substitution σ of matching variables into ρ-terms such that an equation $A \ll B$ has a solution if and only if $A\sigma =_{\rho\delta} B$. In this section, we focus on the role played by the ρ-rule and thus we solve equations only modulo this rule.

4.6.1 Matching in the λ-calculus vs matching in the ρ-calculus

We begin with a very simple example: we consider the matching problem

$$X\,a \ll g(a, a)$$

Syntax of the ρ_d-calculus

Patterns $P ::= x \mid \mathsf{stk} \mid f(P, \ldots, P)$ *(variables occur only once in any P)*

Terms $A, B ::= x \mid c \mid \mathsf{stk} \mid P \twoheadrightarrow A \mid A\,B \mid A \wr B$

Values of the ρ_d-calculus

$V ::= x \mid c \mid f(V, \ldots, V) \mid P \twoheadrightarrow T$ (Values)

$V^\gamma ::= V \mid V^\gamma \wr V^\gamma$ (Structure Values)

$V^{\rho\delta} ::= V \mid \mathsf{stk}$ (Stuck Values)

Semantics of the ρ_d-calculus

(β_{ρ_d})	$(P \twoheadrightarrow A)\,V^{\rho\delta}$	$\to$	$A\sigma$	if $\mathcal{S}ol(P \lessdot V^{\rho\delta}) = \sigma$
(δ)	$(A_1 \wr A_2)\,V^{\rho\delta}$	$\to$	$A_1\,V^{\rho\delta} \wr A_2\,V^{\rho\delta}$	
(γ)	$A\,(V_1^\gamma \wr V_2^\gamma)$	$\to$	$A\,V_1^\gamma \wr A\,V_2^\gamma$	
(stk)	$(P \twoheadrightarrow A)\,B$	$\to$	stk	if $P \not\sqsubseteq A$
(stk)	$\mathsf{stk} \wr A$	$\to$	A	
(stk)	$A \wr \mathsf{stk}$	$\to$	A	
(stk)	$\mathsf{stk}\,A$	$\to$	stk	

Figure 4.5: The ρ_d-calculus

In the λ-calculus, this problem has 4 independent solutions given by

$$\{\lambda x.g(a,a)/X\}, \quad \{\lambda x.g(a,x)/X\}, \quad \{\lambda x.g(x,a)/X\}, \quad \{\lambda x.g(x,x)/X\}.$$

This problem has an extra solution in the ρ-calculus because of its matching capabilities:

$$\{a \twoheadrightarrow g(a,a)/X\}.$$

In this example, the matching capabilities of the ρ-calculus give an extra solution. Moreover, some equations have solutions in the ρ-calculus but not in the λ-calculus. Typically, such an equation should instantiate a matching variable by a pattern-abstraction that "decomposes" its argument. This is the case in the following equation

$$\big(x \twoheadrightarrow (X\,g(x))\big) \ll (x \twoheadrightarrow x)$$

which has no solution in the λ-calculus whereas the substitution

$$\{(g(y) \twoheadrightarrow y)/X\}$$

is a solution in the ρ-calculus.

4.6.2 Matching in the ρ-calculus needs unification

We conjecture that is is not possible to study higher-order matching in the ρ-calculus independently from unification.

We show on a simple example that there exist matching problems that are equivalent to unification problems.

Consider for example the following matching problem in the ρ-calculus

$$(f(g(x), y) \twoheadrightarrow y)\, f(Z\, g(a), a) \ll a$$

In the body of the pattern-abstraction, the variable x is not used. The problem has thus as many solutions as there are instantiations of Z such that the match between the reducts of $Z\, g(a)$ and $g(x)$ succeeds. In other words, we should equivalently solve the unification equation

$$Z\, g(a) =_\rho g(X)$$

where X is a fresh variable.

4.6.3 Extensionality in the ρ-calculus

To conclude this section, we give a short discussion on the extensionality of terms (or programs) defined with pattern-matching that is not yet (to the knowledge of the author) well-understood. We claim that higher-order matching in the ρ-calculus should be defined as the process that solves of equation modulo $\rho(\delta)$. But it is not so clear that, in practice, this is the right equality one should consider. For example, the two following terms

$$g(g(x)) \twoheadrightarrow x \qquad\qquad g(x) \twoheadrightarrow (g(y) \twoheadrightarrow y)\, x$$

should be equal in the same way the two following functional programs

```
let f z = match z with
  g(g(X)) -> X
```

and

```
let f z =
 match z with
 | g(X) -> match X with
           | g(Y) -> Y
```

are. But in fact, in the plain ρ-calculus these two terms are distinct normal forms. At first sight, this problem seems easy to solve by adding the equality

$$c(P_1, \ldots, P_n) \twoheadrightarrow M \quad = \quad c(x_1, \ldots, x_n) \twoheadrightarrow (P_1 \twoheadrightarrow \ldots (P_n \twoheadrightarrow M)\, x_n) \ldots)\, x_1 \quad \text{if } \exists i, P_i \text{ is not a variable}$$

where the variables x_i are fresh. Nevertheless, this equality does not make equal the terms

$$g(x') \twoheadrightarrow f\big((g(x) \twoheadrightarrow x)\, x',\ (g(x) \twoheadrightarrow x)\, x'\big) \qquad \text{and} \qquad g(g(x)) \twoheadrightarrow f(x, x)$$

and this seems to be rather difficult to achieve.

Conclusion

We first give a presentation of the ρ-calculus that makes clear that matching and substitutions are the two fundamental operations of the calculus. We will analyze them in more detail in the following chapter.

We also show that confluence of the ρ-calculus should be studied with care. In Chapter 6, we will study this property and compare the ρ-calculus with other pattern-based calculi.

Chapter 5

The explicit ρ-calculus

Context The substitution application (inherited from the λ-calculus) and the pattern-matching (inherited from rewriting) are the fundamental operations of the rewriting calculus.

To perform pattern-matching, evaluation rules of the ρ-calculus use helper functions that are indeed implicit computations: all the computations related to the considered matching theory belong to the meta-level. These computations are conceptually and computationally important in all matching theories, from syntactic ones to quite elaborated ones like associative-commutative theories [Eke95]. In concrete implementations, substitutions and pattern-matching should be separated and interact with one another. In particular, we want matching related computations and applications of substitutions to be explicit.

Explicit substitution λ-calculi [ACCL91, Les94, Mel95, Mel96, Ros96, DG01, Kes07] have been widely studied. They play a central role in some implementations of ML [LN04] and provide nice tools to deal with higher-order unification [DHK00] or to represent incomplete proofs in type theory [Muñ97].

In this work, we need to make explicit both substitutions and matching computations.

Contributions A first step toward an explicit handling of the matching related computations was the introduction of matching problems as part of the ρ-calculus syntax [CKL02]. More precisely, the *matching constraints* represent constrained terms which are eventually instantiated by the substitution obtained as solution of the corresponding matching problem (if such a solution exists).

The matching constraints are solved and the resulting substitutions are applied in one step. In concrete implementations these operations the matching constraint solving and the substitution application) should be separated and should interact with other computations. In particular, we want computations on constraints and applications of constraints (substitutions) to be explicit.

The evaluation rules solving the syntactic matching problems eventually transform a term constrained by a matching problem to the same term but constrained by the solved problem, that is, by a substitution. The application of the resulting substitutions follows the approaches used in λ-calculi with explicit substitutions.

In all the explicit substitution calculi, substitutions can be delayed thanks to the β rule that transforms a β-redex $(\lambda x.A)\,B$ into the *explicit* application on A of the substitution that replaces x by B. In the explicit ρ-calculus, the application of substitutions is delayed

to the moment where the original matching constraint is solved. Thus, the role of the β rule is taken by the ρ rule which transforms the application of a rewrite rule into the application of a matching constraint and by the evaluation rules solving the obtained matching problem. This led to a calculus *à la* λ_x-calculus [Ros96] simple and without substitution compositions.

This calculus [CFK04] handles at the object level not only the matching problems but also the application of the resulted substitutions. Nevertheless, these two computations are handled differently and considered one at a time.

We present two versions of the ρ-calculus, one with explicit substitutions and one with explicit matching, together with a version that combines the two and considers efficiency issues and more precisely the composition of substitutions. This allows us to isolate the features absolutely necessary in both cases and to analyze the issues related to the two approaches. The result is a full calculus that enjoys the usual good properties of explicit substitutions (conservativity, termination) and which is confluent. We show that the ρ-calculus, and especially explicit ρ-calculi, are suitable as a useful theoretical back-end for implementations of rewriting-based languages. This work has been implemented using the Tom language [BBK+06] and gives an interpreter for the ρ-calculus.

A first version of this work was published in [CFK04]. A long version close to this chapter was published in [CFK07].

Outline of the chapter Sections 5.1 and 5.2 consider respectively the extensions of the plain ρ-calculus for explicit substitution ($\rho_{\mathtt{s}}$) and explicit matching ($\rho_{\mathtt{m}}$). Section 5.3 presents the combination (ρ°_{x}) of the two previous calculi enriched with a rule for combining term traversal (composition rule for substitutions). Properties such as confluence of (ρ°_{x}) and the termination of the explicit part are then established in Section 5.4. We finally give some hints on a direct implementation of explicit ρ-calculi in Section 5.5 and on a ρ-calculus with de Bruijn indices and explicit substitutions in Section 5.6.

5.1 Explicit substitution $\rho_{\mathtt{s}}$

We introduce in this section a generalization of the λ_x-calculus [Ros96] that we call ρ-calculus with explicit substitutions and that considers abstractions not only on single variables but also on patterns potentially containing several variables. On the other hand, the obtained ρ-calculus with explicit substitutions, denoted shortly $\rho_{\mathtt{s}}$, can be seen as an extension of the plain ρ-calculus with explicit substitutions.

In the plain ρ-calculus, when reducing the application of a rule to a term, the matching between the left-hand side of the rule and the term is solved and the resulting substitution is applied to the right-hand side of the rule at the meta-level of the calculus. This means that, in one step, we compute the substitution solving the corresponding matching problem and apply it.

This reduction can obviously be decomposed into two steps, one computing the substitution and the other one describing the application of this substitution. This decomposition does not mean that the matching related computations and the application of

Terms	A, B, P	$::=$	x	(Variable)
		$\mid$	c	(Constant)
		$\mid$	$P \twoheadrightarrow A$	(Abstraction)
		$\mid$	$A\ B$	(Functional application)
		$\mid$	$A \wr B$	(Structure)
		$\mid$	$B\,[\phi]$	(Substitution application)
Substitutions	ϕ	$::=$	$\mathtt{id}$	(Identity)
		$\mid$	$\bigwedge_{i=1\ldots n} x_i = A_i$	(Conjunction)

where $\bigwedge$ is associative

Figure 5.1: Syntax of ρ_s

substitutions are explicit but just that they are clearly separated. In this section we go a step further toward an explicit version of the ρ-calculus by proposing a version of the calculus where the substitution application is performed explicitly while the matching problems are still solved at the meta-level.

5.1.1 Syntax of ρ_s

The syntax of the plain ρ-calculus given in the previous chapter is extended with an explicit substitution operator and is given in Figure 5.1. Also, we follow the conventions mentioned in the previous chapter.

The *substitution application* operator is a generalization of the similar one from λ_x-calculus. In the λ-calculus, a λ-abstraction binds only one variable and thus an explicit substitution consists in a single variable binding. In the ρ-calculus, the pattern-abstraction can bind an arbitrary number of variables and thus the definition of a substitution is extended to support multiple bindings. The $\mathtt{id}$ symbol represents the identity substitution. We should point out that in this context the symbol "=" is not symmetric.

The symbols $\phi, \psi, \ldots$ range over the set Φ of substitutions. A term is called *pure* if it does not contain any explicit substitution application.

To simplify the reading, we adopt the following notation for explicit substitution application

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

In the same way, we adopt the following notation for (meta) substitution application

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

The *domain* of a substitution $\phi = \bigwedge_{i=1\ldots n} x_i = A_i$, denoted $\mathcal{D}om(\phi)$, is the set $\{x_i\}_{i=1}^{n}$.

For the purpose of this chapter, we restrict to structure patterns as defined in Section 4.2.1. The formal definition for the set of free variables for the fore-coming ρ_x° calculus is given in Section 5.3 and the restriction to the ρ-calculus with explicit substitutions is straightforward.

5.1.2 Operational semantics of ρ_s

The operational semantics of the ρ-calculus with explicit substitutions is given in Figure 5.2. We follow the conventions in the previous chapters in particular, we consider terms modulo α-conversion. The reduction rules given in Figure 5.2 are split into two categories:

- Rules describing the application of structures and abstractions on ρ-terms.
- Rules defining the application of substitutions.

The (ρ) rule is used to reduce the application of an abstraction to a term by matching the left-hand side of the abstraction against the term (using the matching algorithm given in Section 4.2.1) and triggering the application of the obtained substitution to the right-hand side of the abstraction. As a particular case, when the function $\mathcal{S}ol$ returns the identity substitution (represented by $\{Q_i/x_i\}_{i=1}^{0}$) then the result is the explicit application of `id` (represented by $A[x_i = Q_i]_{i=1}^{0}$). If no match exists, the rule is not applied. The rule (δ) is inherited from the plain ρ-calculus (see Chapter 4).

The rules handling the substitution application distribute it over the different operators until a variable or a constant is reached. If the variable is in the domain of the substitution then the corresponding term replaces it; a substitution applied to a variable that is not in its domain or to a constant is ignored. Since we consider classes of terms modulo α-conversion, the appropriate representatives are always chosen in order to avoid potential variable captures (introduced by the rule (Abs)).

We define the relations σ and ρ_s induced by the rules dealing with substitutions (*i.e.* (Identity), (Replace), (Var), (Const), (Abs), (App), (Struct)) and by the set of all rules in Figure 5.2, respectively. We should point out that in ρ_s and in the other calculi introduced in the next sections all the evaluation steps are performed modulo the underlying theory (associativity or associativity with neutral elements) for the conjunction (see Definition 5.9 for a precise definition of rewriting modulo).

One can notice that when replacing the pattern P by a variable in rule (ρ), we recover the (Beta) rule of λ-calculi with explicit substitution.

$$
\begin{array}{lll}
(P \multimap A)\, B & \to_{\rho} & A\,[x_i = C_i]_{i=1}^{n} \\
 & & \quad \text{where } \mathcal{S}ol(P \lessdot B) = \{C_i/x_i\}_{i=1}^{n} \\
(A_1 \wr A_2)\, B & \to_{\delta} & A_1\, B \wr A_2\, B \\
 & & \\
A\,[\mathtt{id}] & \to_{\text{Identity}} & A \\
x\,[\phi \wedge (x = A) \wedge \psi] & \to_{\text{Replace}} & A \\
y\,[\phi] & \to_{\text{Var}} & y \qquad \text{if } y \notin \mathcal{D}om(\phi) \\
c\,[\phi] & \to_{\text{Const}} & c \\
(P \multimap A)\,[\phi] & \to_{\text{Abs}} & P\,[\phi] \multimap A\,[\phi] \\
(A\ B)\,[\phi] & \to_{\text{App}} & A\,[\phi]\ B\,[\phi] \\
(A \wr B)\,[\phi] & \to_{\text{Struct}} & A\,[\phi] \wr B\,[\phi]
\end{array}
$$

Figure 5.2: Operational semantics of ρ_s

Example 5.1 (Application of a rewrite rule) *In order to compute the disjunctive normal form of a propositional formula, we use the rewrite rule*

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

The application of this rewrite rule to the term $\mathsf{not}(\mathsf{and}(t, f))$ *is described in* ρ_s *by the following reduction*

$$
\begin{array}{ll}
 & \big(\mathsf{not}(\mathsf{and}(x, y)) \multimap \mathsf{or}(\mathsf{not}(x), \mathsf{not}(y))\big)\ \mathsf{not}(\mathsf{and}(t, f)) \\
\to_{\rho} & \mathsf{or}(\mathsf{not}(x), \mathsf{not}(y))\,[x = t \wedge y = f] \\
\twoheadrightarrow_{\text{App}} & (\mathsf{or}\,[x = t \wedge y = f])(\mathsf{not}(x\,[x = t \wedge y = f]), \mathsf{not}(y\,[x = t \wedge y = f])) \\
\to_{\text{Const}} & \mathsf{or}(\mathsf{not}(x\,[x = t \wedge y = f]), \mathsf{not}(y\,[x = t \wedge y = f])) \\
\twoheadrightarrow_{\text{Replace}} & \mathsf{or}(\mathsf{not}(t), \mathsf{not}(f))
\end{array}
$$

Example 5.2 (Multiple applications) *Since substitutions cannot be composed, the application of each substitution is done independently and can involve a lot of evaluation steps needed in order to access to the leaves of the term (seen as a tree).*

$$
\begin{array}{ll}
 & (g(x) \multimap ((f(y) \multimap h(x, y))\ f(a))\ g(b) \\
\to_{\mathfrak{p}} & ((f(y) \multimap h(x, y))\ f(a))\,[x = b] \\
\to_{\mathfrak{p}} & h(x, y)\,[y = a]\,[x = b] \\
\twoheadrightarrow_{\sigma} & h(x, y\,[y = a])\,[x = b] \\
\twoheadrightarrow_{\sigma} & h(x, a)\,[x = b] \\
\twoheadrightarrow_{\sigma} & h(x\,[x = b], a) \\
\twoheadrightarrow_{\sigma} & h(b, a)
\end{array}
$$

As we will see later on, if the substitutions are composed, then the term traversal related steps can be factorized (see Example 5.8).

Terms	A, B, P	$::=$	x	(Variable)
		$\mid$	c	(Constant)
		$\mid$	$P \twoheadrightarrow A$	(Abstraction)
		$\mid$	$A\ B$	(Functional application)
		$\mid$	$A \wr B$	(Structure)
		$\mid$	$B\,[\mathcal{C}]$	(Constraint application)
Constraints	$\mathcal{C}, \mathcal{D}$	$::=$	$\mathtt{id}$	(Identity)
		$\mid$	$P \ll M$	(Match-equation)
		$\mid$	$\mathcal{C} \wedge \mathcal{D}$	(Conjunction of constraints)
		where $\wedge$ is associative and $\mathtt{id}$ is a neutral element		

Figure 5.3: Syntax of $\rho_{\mathtt{m}}$

5.2 Explicit matching $\rho_{\mathtt{m}}$

In this section we concentrate on the matching problems intrinsic to the ρ-calculus and, more precisely, we want to make explicit the matching computations performed during the ρ-evaluation. We propose here the ρ-calculus with explicit matching, denoted also $\rho_{\mathtt{m}}$, that extends the plain ρ-calculus with evaluation rules dealing with the (syntactic) matching. In this calculus the obtained matching problems are solved explicitly while the substitution application is done at the meta-level.

5.2.1 Syntax of $\rho_{\mathtt{m}}$

The syntax of the ρ-calculus with explicit matching is given in Figure 5.3. The explicit substitutions of $\rho_{\mathtt{s}}$ that can be considered as match equations in solved form are replaced by match constraints that will be solved in the evaluation process.

The symbols $\mathcal{C}, \mathcal{D}, \mathcal{E} \ldots$ range over the set of (possibly empty) constraints. The $\mathtt{id}$ symbol represents here the identity constraint while in the $\rho_{\mathtt{s}}$ the same symbol was used to represent the empty substitution. We consider that an empty conjunction and $\mathtt{id}$ represent the same object.

The domain of a constraint $\mathcal{C}$, denoted $\mathcal{D}\mathtt{om}(\mathcal{C})$, is intuitively the same as the domain of the substitution that solves all the corresponding matching problems and is computed by taking the union of the sets of free variables of the patterns of all the match-equations in $\mathcal{C}$. The formal definition is given in Section 5.3.1.

5.2.2 Operational semantics of $\rho_{\mathtt{m}}$

The operational semantics of the ρ-calculus with explicit matching consists of two parts as shown in Figure 5.4.

As for ρ_s, the first part describes the application of structures and abstractions on ρ-terms. This time the (ρ) rule always applies and reduces the application of an abstraction to a term constraint by the corresponding matching problem.

The matching problems are handled by the second part of the evaluation rules that are clearly inspired by the ones presented in Section 4.2.1. A matching constraint is simplified using the decomposition rules which are strongly related to the considered matching theory. As we have already mentioned, we consider only structures of algebraic terms as patterns and we restrict to a decidable and unitary matching theory and, more precisely, to syntactic matching. Therefore, we do not handle the higher-order symbols (*e.g.* "$\rightarrowtriangle$", "$\ll$") and we only decompose the restricted patterns. The two decomposition rules given in Figure 5.4 are thus performed *w.r.t.* to an empty matching theory for the structure operator and for the constant symbols. Since an empty conjunction and $\mathtt{id}$ represent the same object and thus, the rule ($\mathsf{Decompose}_{\mathcal{F}}$) can be used for constants (*i.e.* $n = 0$) in which case the result is the identity.

When a part of the constraint is solved and independent of the rest of the constraint, the corresponding substitution can be applied at the meta-level. We consider that the meta-application of the substitution $\{A/x\}$ to the term B, denoted $B\{A/x\}$, is higher-order and thus performs α-conversion in order to avoid the possible variable captures. The condition in rule ($\mathsf{ToSubst}$) guarantees that a matching problem is solved and thus (part of) the corresponding substitution can be applied only if all its variables are assigned the same term. Non-linear matching problems can lead to matching constraints which assign different terms to the same variable and which represent, intuitively, a failure (see Example 5.5). The doubletons in a matching constraint are eliminated with the rule (Idem).

Notice that in the rule ($\mathsf{ToSubst}$), because of the hygiene-convention, the intersection between the set of free variables of A and $\mathcal{D}om(\mathcal{C} \wedge \mathcal{D})$ is empty and thus, the variables in A cannot be captured in the right-hand side of the rule.

The ρ-calculus (and in particular ρ_m) is well-suited to deal with (matching) errors, represented by constraints without solution, that is, constraints that do not represent substitutions. Depending on the intended use of the calculus we may want or not to propagate such (constraint) failures. If the failures are propagated, the error's location is lost and the final result would be a term with constraints with no solution applied on each leaf of the term (considered as a tree). The information contained in such a term seems useless when one wants to analyze the error and, for debugging reasons, we do not want to lose the error's location. This is why the failures are not propagated as they are but only the corresponding substitution (if one exists) is propagated.

Example 5.3 (Application of a rewrite rule) *The term presented in Example 5.1*

$$
\begin{array}{lll}
(P \twoheadrightarrow A)\, A & \to_{\rho} & A[P \ll B] \\
(A_1 \wr A_2)\, B & \to_{\delta} & A_1\, B \wr A_2\, B \\
\\
A_1 \wr A_2 \ll B_1 \wr B_2 & \to_{\text{Decompose}_\wr} & A_1 \ll B_1 \wedge A_2 \ll B_2 \\
f(A_1, \ldots, A_n) \ll f(B_1, \ldots, B_n) & \to_{\text{Decompose}_{\mathcal{F}}} & \bigwedge_{i=1}^{n} (A_i \ll B_i) \\
\mathcal{C} \wedge (x \ll A) \wedge \mathcal{D} \wedge (x \ll A) \wedge \mathcal{E} & \to_{\text{Idem}} & \mathcal{C} \wedge (x \ll A) \wedge \mathcal{D} \wedge \mathcal{E} \\
\\
A\,[\mathtt{id}] & \to_{\text{Identity}} & A \\
B\,[\mathcal{C} \wedge (x \ll A) \wedge \mathcal{D}] & \to_{\text{ToSubst}} & (B\{A/x\})\,[\mathcal{C} \wedge \mathcal{D}] \\
 & & \text{if } x \notin \mathcal{D}\text{om}(\mathcal{C} \wedge \mathcal{D})
\end{array}
$$

Figure 5.4: Small-step operational semantics of $\rho_{\mathtt{m}}$

reduces in the ρ-calculus with explicit matching as follows

$$
\begin{array}{ll}
 & \big(\mathsf{not}(\mathsf{and}(x,y)) \twoheadrightarrow \mathsf{or}(\mathsf{not}(x), \mathsf{not}(y))\big)\ \mathsf{not}(\mathsf{and}(t,f)) \\
\to_{\rho} & \mathsf{or}(\mathsf{not}(x), \mathsf{not}(y))\,[\mathsf{not}(\mathsf{and}(x,y)) \ll \mathsf{not}(\mathsf{and}(t,f))] \\
\to\!\!\!\!\to_{\text{Decompose}_{\mathcal{F}}} & \mathsf{or}(\mathsf{not}(x), \mathsf{not}(y))\,[x \ll t \wedge y \ll f] \\
\to\!\!\!\!\to_{\text{ToSubst}} & \mathsf{or}(\mathsf{not}(t), \mathsf{not}(f))\,[id] \\
\to_{\text{Identity}} & \mathsf{or}(\mathsf{not}(t), \mathsf{not}(f))
\end{array}
$$

Example 5.4 (Application of a non-linear rewrite rule) *When non-linear patterns are used, the rule (*Idem*) can merge the solved matching problems that are identical:*

$$
\begin{array}{ll}
 & (\mathsf{xor}(x,x) \twoheadrightarrow f)\ \mathsf{xor}(t,t) \\
\to_{\rho} & f\,[\mathsf{xor}(x,x) \ll \mathsf{xor}(t,t)] \\
\to_{\text{Decompose}_{\mathcal{F}}} & f\,[x \ll t \wedge x \ll t] \\
\to_{\text{Idem}} & f\,[x \ll t] \\
\to\!\!\!\!\to_{\text{ToSubst}} & f\,[id] \\
\to_{\text{Identity}} & f
\end{array}
$$

Example 5.5 (Application of a non-linear rewrite rule with failure) *Of course, the application of a non-linear rewrite rule may lead to failures due to merging clashes. Merging clashes are not reduced but kept as a constraint application failure.*

$$
\begin{array}{ll}
 & (\mathsf{xor}(x,x) \twoheadrightarrow f)\ \mathsf{xor}(t,f) \\
\to_{\rho} & f\,[\mathsf{xor}(x,x) \ll \mathsf{xor}(t,f)] \\
\to_{\text{Decompose}_{\mathcal{F}}} & f\,[x \ll t \wedge x \ll f]
\end{array}
$$

The next example illustrates the usefulness of explicit matching when we want to track the source (cause) of the failure.

Example 5.6 (Run-time error: matching failure) *Let us consider the following rule that checks if two persons are brothers, i.e., if they have the same father:*

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

*When checking if two concrete persons (*Alice *and* Bob*) are brothers by applying this rule to the corresponding term:*

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

we obtain as result the term

$$\texttt{t}\,[\texttt{z} \ll \texttt{John} \wedge \texttt{z} \ll \texttt{Jim}]$$

indicating that the variable z *corresponding to the father cannot be instantiated correctly, i.e., that the father of the two persons is not the same.*

This is in contrast with the ρ*-calculus with explicit substitutions where the fact that the application of the rule to the term is in normal form indicates that the matching has no solution but gives no information on the source of this failure.*

5.3 Explicit substitution and explicit matching ρ_x°

The combination of the two previously introduced calculi, $\rho_{\mathtt{s}}$ and $\rho_{\mathtt{m}}$, leads to a version of the calculus that handles explicitly the matching constraints resolution as well as the application of the substitutions. This calculus, called ρ_x and introduced in [CFK04] does not handle the composition of substitutions, a key issue when one wants to obtain efficient implementations.

In what follows we add this feature and define ρ_x°. The properties of this calculus are then studied.

5.3.1 Syntax of ρ_x°

The syntax presented in Figure 5.5 merges the ones of $\rho_{\mathtt{s}}$ and $\rho_{\mathtt{m}}$ and defines ρ_x°-terms. In what follows we refer to the three categories of ρ_x°-terms by simply calling them *terms*, *substitutions* and *constraints* respectively.

One can notice that the conjunction operator $\wedge$ is overloaded and it is used to build substitutions as well as constraints. Since the two types of conjunctions are disjoint, in what follows, we will generally denote the corresponding identities $\mathtt{id}_{\mathtt{s}}$ and $\mathtt{id}_{\mathtt{m}}$ by the same symbol id.

We assume that the functional, substitution and constraint application operators associate to the left, while the other operators associate to the right. The priority of the substitution application is higher than that of the constraint application which is higher than that of the functional application. The application has a higher priority than

Terms	A, B, P	::=	x	(Variables)
		|	c	(Constants)
		|	$P \multimap A$	(Abstraction)
		|	$A\ B$	(Functional application)
		|	$A \wr B$	(Structure)
		|	$B\,[\phi]$	(Substitution application)
		|	$B\,[\mathcal{C}]$	(Constraint application)
Substitutions	ϕ, ψ	::=	$\mathtt{id}_s$	(Identity)
		|	$x = A$	(Equation)
		|	$\phi \wedge \psi$	(Conjunction of equations)
Constraints	$\mathcal{C}, \mathcal{D}$	::=	$\mathtt{id}_m$	(Identity)
		|	$P \ll A$	(Match-equation)
		|	$\mathcal{C} \wedge \mathcal{D}$	(Conjunction of constraints)

where $\wedge$ is associative
and $\mathtt{id}_s$ and $\mathtt{id}_m$ are neutral elements

Figure 5.5: Syntax of ρ_x°

"$\_ \twoheadrightarrow \_$" which is, in turn, of higher priority than the "$\_ \wr \_$". The equation operators are of higher priority than the conjunction operators. The equation and conjuntion operators have a lower priority than the other ones.

Definition 5.7 (Free variables and constraint domains) *The set of free variables and the domain of a constraint (resp. substitution) are defined by:*

$$\begin{array}{lcllcl}
\mathsf{fv}(x) & = & \{x\} & & & \\
\mathsf{fv}(c) & = & \emptyset & \mathsf{fv}(P \twoheadrightarrow A) & = & \mathsf{fv}(A)\backslash\mathsf{fv}(P) \\
\mathsf{fv}(A\ B) & = & \mathsf{fv}(A) \cup \mathsf{fv}(B) & \mathsf{fv}(A \wr B) & = & \mathsf{fv}(A) \cup \mathsf{fv}(B)
\end{array}$$

$$\begin{array}{c}
\mathsf{fv}(B\,[\mathcal{C}]) = \mathsf{fv}(\mathcal{C}) \cup \big(\mathsf{fv}(B)\backslash\mathcal{D}\mathrm{om}(\mathcal{C})\big) \\
\mathsf{fv}(B\,[\phi]) = \mathsf{fv}(\phi) \cup \big(\mathsf{fv}(B)\backslash\mathcal{D}\mathrm{om}(\phi)\big)
\end{array}$$

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

$$\begin{array}{lcllcl}
\mathcal{D}\mathrm{om}(P \ll A) & = & \mathsf{fv}(P) & \mathcal{D}\mathrm{om}(\mathcal{C} \wedge \mathcal{D}) & = & \mathcal{D}\mathrm{om}(\mathcal{C}) \cup \mathcal{D}\mathrm{om}(\mathcal{D}) \\
\mathcal{D}\mathrm{om}(\mathit{id}) & = & \emptyset & & &
\end{array}$$

$$\begin{array}{lcllcl}
\mathcal{D}\mathrm{om}(x = A) & = & \{x\} & \mathcal{D}\mathrm{om}(\phi \wedge \psi) & = & \mathcal{D}\mathrm{om}(\phi) \wedge \mathcal{D}\mathrm{om}(\psi)
\end{array}$$

5.3.2 Operational semantics of ρ°_x

The evaluation rules of ρ°_x are presented in Figure 5.6 and consist of those used for ρ_s and ρ_m together with a composition rule.

The application of rule abstractions and structures as well as the matching constraints decomposition are inherited from ρ_m. As in ρ_m, when part of the constraint is solved and independent of the rest of the constraint, the corresponding substitution should be applied. In ρ°_x the application of the substitution is just triggered (as in ρ_s) in the rule (ToSubst) and the rules inherited from ρ_s perform its application. The (Identity) rule represents in fact two rules, one for the identity substitution and a second one for the identity constraint. When not clear from the context, the former is called ($\mathtt{Identity}_s$) while the latter is called ($\mathtt{Identity}_c$).

The newly introduced rule (Constraint) distributes the substitutions in the constraints. The rule (Compose) defines the composition of substitutions. The side condition for these two rules says that the constraint and respectively the substitution cannot be identities. For simplicity, we used an abuse of notation in the rule (Compose) where $B[\phi \wedge x_i = A_i\,[\phi]]_{i=1}^n$ denotes the term $B\,[\phi \wedge x_1 = A_1\,[\phi] \wedge \ldots \wedge x_n = A_n\,[\phi]]$.

Example 5.8 (Multiple applications) *The composition of substitutions leads to more efficient evaluations. The following evaluation is obtained for the term considered in Ex-*

$$
\begin{array}{lll}
(P \twoheadrightarrow A)\ B & \to_{\rho} & A[P \ll B] \\
(A_1 \wr A_2)\ B & \to_{\delta} & A_1\ B \wr A_2\ B \\
\\
A_1 \wr A_2 \ll B_1 \wr B_2 & \to_{\text{Decompose}_{\wr}} & A_1 \ll B_1 \wedge A_2 \ll B_2 \\
f(A_1, \ldots, A_n) \ll f(B_1, \ldots, B_n) & \to_{\text{Decompose}_{\mathcal{F}}} & \bigwedge_{i=1}^{n} (A_i \ll B_i) \\
\mathcal{C} \wedge (x \ll A) \wedge \mathcal{D} \wedge (x \ll A) \wedge \mathcal{E} & \to_{\text{Idem}} & \mathcal{C} \wedge (x \ll A) \wedge \mathcal{D} \wedge \mathcal{E} \\
B\,[\mathcal{C} \wedge (x \ll A) \wedge \mathcal{D}] & \to_{\text{ToSubst}} & (B\,[x = A])\,[\mathcal{C} \wedge \mathcal{D}] \\
 & & \textbf{if } x \notin \mathcal{D}om(\mathcal{C} \wedge \mathcal{D}) \\
\\
A\,[\mathtt{id}] & \to_{\text{Id}} & A \\
x\,[\phi \wedge (x = A) \wedge \psi] & \to_{\text{Replace}} & A \\
y\,[\phi] & \to_{\text{Var}} & y \qquad \textbf{if } y \notin \mathcal{D}om(\phi) \\
c\,[\phi] & \to_{\text{Const}} & c \\
(P \twoheadrightarrow A)\,[\phi] & \to_{\text{Abs}} & P\,[\phi] \twoheadrightarrow A\,[\phi] \\
(A\ B)\,[\phi] & \to_{\text{App}} & A\,[\phi]\ B\,[\phi] \\
(A \wr B)\,[\phi] & \to_{\text{Struct}} & A\,[\phi] \wr B\,[\phi] \\
(A[P_i \ll B_i]_{i=1}^{n})\,[\phi] & \to_{\text{Constraint}} & (A\,[\phi])[P_i\,[\phi] \ll B_i\,[\phi]]_{i=1}^{n} \\
 & & \textbf{if } n > 0 \\
(B[x_i = A_i]_{i=1}^{n})\,[\phi] & \to_{\text{Compose}} & B[\phi \wedge x_i = A_i\,[\phi]]_{i=1}^{n} \\
 & & \textbf{if } n > 0
\end{array}
$$

Figure 5.6: Small-step operational semantics of ρ_x°

ample 5.2.

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

All the evaluation steps dealing with the traversal of the term $h(x,y)$ are done only once this time since only one substitution should be propagated.

The non-efficient route given in Example 5.2 can also be taken but while this was the only alternative for ρ_s, this problem disappears for ρ°_x if we apply the evaluation rules with a strategy [BKK96] that gives the highest priority to the composition rule, a canonical way to limit term traversal.

As mentioned at the beginning of this section, a ρ-calculus handling explicitly the constraint solving and the substitution application was coined for the first time in [CFK04]. In ρ_x there was no mechanism for the composition of substitutions (*i.e.* no (Compose) rule) and consequently, no substitution conjunctions. Therefore, we can consider ρ_x as a restriction of ρ°_x where all substitutions are simple equations and the (Compose) rule is not available.

5.4 Properties of ρ°_x

5.4.1 Proof scheme

As mentioned before, all the evaluation steps are performed modulo the theory of the conjunction (in ρ°_x modulo the associativity and the neutral elements) and thus, we are performing rewriting modulo a set of axioms. We give first a formal definition for this evaluation and then we introduce a more operational evaluation that is usually used in proofs and implementations. For a detailed exposition about rewriting modulo, we refer to [Hue80, JK86, KK99, Ohl98].

Definition 5.9 ((R/A)) *Given a rewrite system R and a set of axioms A, the term t (R/A)-rewrites to t', denoted $t \rightarrow_{R/A} t'$, if there exists a rule $l \rightarrow r \in R$, a term u, an occurrence q in u and a substitution σ such that $t \overset{*}{\longleftrightarrow}_A u[q \hookleftarrow \sigma(l)]$ and $t' \overset{*}{\longleftrightarrow}_A u[q \hookleftarrow \sigma(r)]$ where $u[q \hookleftarrow v]$ denotes the term u with the sub-term at the position q replaced by v.*

Definition 5.10 ((R, A)) *Given a rewrite system R and a set of axioms A , the term t (R, A)-rewrites to a term t', which is denoted by $t \rightarrow_{R,A} t'$ if there exists a rule $l \rightarrow r \in R$, a position q in t, and a substitution σ such that $t_{|q} \overset{*}{\longleftrightarrow}_A \sigma(l)$ (i.e. the sub-term of t at position q is equivalent to $\sigma(l)$), and $t' = t[q \hookleftarrow \sigma(r)]$ (i.e. t' is equal to t where the sub-term at the position q is replaced by $\sigma(r)$).*

In the following, we will denote by $\mathtt{A1}$ the set of axioms defining the associativity and the neutral elements $\mathtt{id}_m$ and $\mathtt{id}_s$ for the conjunction and by $\sim_{\mathtt{A1}}$ the equivalence relation induced by these axioms. In order to simplify the reading, we denote by $\to_{\mathcal{C}}$ the relation $\to_{k,\mathtt{A1}}$ where k is the rewrite relation induced by all the evaluation rules except (ρ) and (δ). Similarly, we denote by $\to_{\sigma}$ the relation induced by the rules dealing with the application of substitutions (the $(\mathtt{Identity})$ rule for substitutions and the rules from $(\mathtt{Replace})$ to $(\mathtt{Compose})$). We denote ρ°_{x} the rewrite system given in Figure 5.6, and thus we obtain the relations $\to_{\rho^{\circ}_{x},\mathtt{A1}}$ and $\to_{\rho^{\circ}_{x}/\mathtt{A1}}$.

As mentioned in the previous chapter, non-linear patterns lead to non-confluent reductions. This justifies the following definition.

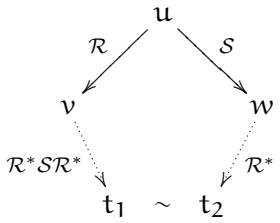
Definition 5.11 (Linear ρ°_{x}) *A pattern is* linear *if it does not contain two occurrences of the same variable. We say that a substitution $(x_i = A_i)_{i=1}^{n}, n > 0$ is* linear *if all the variables x_i are different. We say that a matching constraint $(P_i \ll A_i)_{i=1}^{n}, n > 0$ is* linear *if $\bigcap_{i=1}^{n} fv(P_i) = \emptyset$. $\mathtt{id}_s$ and $\mathtt{id}_m$ are both linear.*

The linear ρ°_{x} *is the ρ°_{x} where all the patterns, substitutions and constraints are linear.*

The general scheme of the proof of confluence of the linear ρ°_{x} follows the proof of confluence of the full theory of the $\lambda\sigma_{\Uparrow}$-calculus presented in [CHL96] and uses a variant of Yokouchi-Hikita's lemma [YH90]:

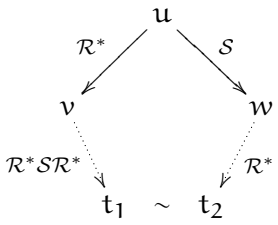
Lemma 5.12 (Yokouchi-Hikita modulo) *Let $\mathcal{R}$ and $\mathcal{S}$ be two relations defined on the same set $\mathcal{T}$ and $\sim$ an equivalence relation such that*

- *$\mathcal{R}$ is strongly normalizing.*
- *$\mathcal{R}$ is confluent modulo $\sim$, i.e., for all u, v, w in $\mathcal{T}$ such that $u\ \mathcal{R}^*\ v$ and $u\ \mathcal{R}^*\ w$ there exist t_1, t_2 in $\mathcal{T}$ such that $v\ \mathcal{R}^*\ t_1$, $w\ \mathcal{R}^*\ t_2$ and $t_1 \sim t_2$*
- *$\mathcal{S}$ has the diamond property, i.e., for all u, v, w in $\mathcal{T}$ such that $u\mathcal{S}v$ and $u\mathcal{S}w$ there exists an element t in $\mathcal{T}$ such that $v\mathcal{S}t$ and $w\mathcal{S}t$*
- *$\mathcal{R}$ and $\mathcal{S}$ are coherent modulo $\sim$, i.e., for all u, v, w such that $u \sim v$, $u\mathcal{R}w$ (resp. $u\mathcal{S}w$) there exists a t such that $v\mathcal{R}t$ (resp. $v\mathcal{S}t$) and $w \sim t$.*
- *the following diagram (often mentioned as Yokouchi-Hikita's diagram modulo) holds:*

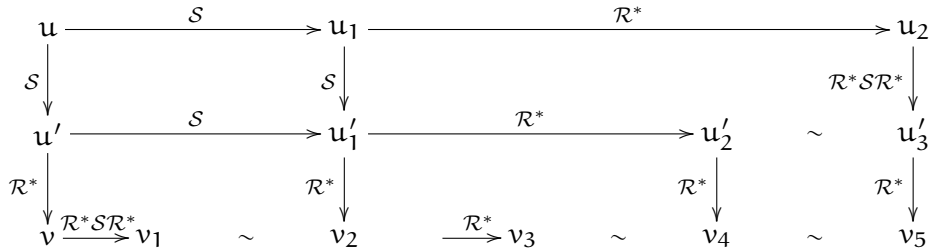


Then the relation $\mathcal{R}^*\mathcal{S}\mathcal{R}^*$ is confluent modulo $\sim$.

Proof. Since the coherence properties are strong, to go from the classical lemma given in [CHL96] to this generalization is just a straightforward verification. We only have to use these coherence properties as much as needed to "factorize" the coherence relation at the bottom of all commutative diagrams given in [CHL96]. In fact, we first prove by induction on the $\mathcal{R}$-depth that:



Then we conclude by induction on the $\mathcal{R}$-depth of u, and where we distinguish between the depth zero and nonzero. The following diagram holds for the case zero



and confluence follows since $\mathcal{R}$ is coherent with $\sim$ and thus $v_1 \sim v_2\ \mathcal{R}^*\ v_3$ can be replaced by $v_1\ \mathcal{R}^*\ v'_3 \sim v_3$.

The diagram for the induction case is more complex but is handled similarly. □

We thus split the evaluation rules of ρ°_x into two relations corresponding to the relations $\mathcal{R}$ and $\mathcal{S}$ of the previous lemma. A natural choice for the relation $\mathcal{R}$ is $\to_{\mathscr{C}}$. In fact, the rules dealing with constraints and explicit substitutions should be strongly normalizing and confluent (in explicit substitution calculi we always ask these properties for the explicit part). The relation $\mathcal{S}$ cannot be the rewrite relation induced by the (ρ) and (δ) rules since this relation verifies neither the diamond property (the (δ) rule duplicates redexes) nor the Yokouchi-Hikita's diagram modulo (the relation $\to_{\mathscr{C}}$ duplicates also redexes). This is why we use a parallelization of the (ρ), (δ) rules.

We first prove in Section 5.4.3 the termination of $\to_{\mathscr{C}}$. Then we prove that $\to_{\mathscr{C}}$ is confluent, taking into account the fact that the rules can be applied modulo the associativity and the neutral elements ($\mathtt{id}_s$, $\mathtt{id}_m$) for the conjunctions. Then we formally define the parallelization of (ρ) and (σ) and show that it verifies the diamond property (Section 5.4.5). Finally, we prove the Yokouchi-Hikita's diagram modulo and conclude the proof of confluence of the linear ρ°_x by noticing that the parallelization and the original relation have the same transitive closure.

First of all, we show the soundness of the explicit substitution reductions, a mandatory and useful lemma in any calculus involving explicit substitution.

5.4.2 Soundness of explicit substitutions

First, we show that $\to_\sigma$ is confluent and strongly normalizing and thus defines a function σ on terms. Secondly, we show that explicit substitutions soundly relate to meta substitutions. The corollary is that the function σ associates to every term a pure term (where all the substitutions have been applied).

Lemma 5.13 (Convergence of $\to_\sigma$) *In the linear ρ°_x, the relation $\to_\sigma$ is confluent and strongly normalizing.*

Proof. The relation $\to_\sigma$ is strongly normalizing by Lemma 5.22 which proves a more general result that is, the termination of $\to_{\mathcal{C}}$. The local confluence is easily proved by observing that all critical pairs modulo $\mathtt{A1}$ are convergent and as a consequence of the above properties, confluence is obtained. We must notice that the linearity condition is mandatory since for example the non-linear substitution application $x\,[x = a \wedge x = b]$ leads either to a or to b. □

Lemma 5.14 (Soundness of explicit substitutions) *For all n, for all terms A_i and B, there exists a term B' such that*

$$B\,[x_i = A_i]_{i=1}^n \quad \twoheadrightarrow_{\mathcal{C}} \quad B'\{A_i/x_i\}_{i=1}^n \qquad \text{and} \qquad B \twoheadrightarrow_{\mathcal{C}} B'$$

Proof. If $n = 0$ then the result follows by the application of the (Identity) rule. Otherwise, we proceed by induction on B.

- If B is a variable, then if this variable is one of the x_i for i between 1 and n then apply the (Replace) rule. If not, then apply the (Var) rule.
- If B is an application, then $B_1\,B_2\,[x_i = A_i]_{i=1}^n$ can be reduced using the (App) rule to $B_1\,[x_i = A_i]_{i=1}^n\,B_2\,[x_i = A_i]_{i=1}^n$. Then, the result follows by induction.
- If B is a substitution application, then

$$\begin{array}{lll} B'[x_i = A_i]_{i=1}^n\,[\phi] & \to_{\text{Compose}} & B'[\phi \wedge x_i = A_i\,[\phi]]_{i=1}^n \\ & \equiv_{(1)} & B'\{\phi, A_i\{\phi\}/x_i\}_{i=1}^n \\ & \equiv_{(2)} & B'\{\phi\}\{A_i\{\phi\}/x_i\}_{i=1}^n \\ & \equiv_{(3)} & B'\{A_i/x_i\}_{i=1}^n\{\phi\} \end{array}$$

  The first step (1) is the application of the induction hypothesis twice. The second step is valid since by α-conversion we can suppose that all the x_i do not belong to the free variables of A_i. The third step is exactly the substitution lemma of the plain ρ-calculus ($x_i \notin \mathsf{fv}(A_i)$).
- The other cases are similar to the application one.

□

5.4.3 Termination of the constraint handling rules

First of all, we will show that $\rightarrow_{\mathcal{C}}$ is strongly normalizing and for this we use the lexicographic product of two orders. The first order is a measure (a size) on terms such that the size of the right-hand side is smaller than that of the left-hand side for the rules (ToSubst), (Idem), (Replace), and the decomposition rules and equal for all the other rules. The second order is based on a polynomial interpretation which decreases on all the other rules.

The measure on terms defined below represents (an upper bound of) the size of the corresponding pure term where all the pending substitutions were applied. The size of a term $B\,[x = A]$ is thus the size of B plus the size of A multiplied by the number of occurrences of x is B (called here the multiplicity of x in B), taking thus into account the possible instantiation(s) of x by A in B. As far as it concerns the matching constraints, we have to take into account that they can (possibly) become substitutions. We make thus an approximation and consider that for the terms of the form $B\,[P \ll A]$, each variable of P is (potentially) instantiated by a sub-term of A that is approximated by A.

This measure is preserved by the duplicative rules (*e.g.* the term $(x \wr x)\,[\phi]$ and its (Struct) reduct $x\,[\phi] \wr x\,[\phi]$ have the same size independently of ϕ). Notice also that in the right-hand side of the rule (ToSubst), A is not affected by the variables of the domain of $\mathcal{C} \wedge \mathcal{D}$ (by α-conversion) and thus, the size decreases (see Lemma 5.19).

Definition 5.15 (Multiplicity) *The multiplicity of the variable x in the term A, denoted $\mathfrak{A}_x(A)$, is defined inductively in Figure 5.7.*

Definition 5.16 (Size) *The size of a term A, denoted $\mathfrak{S}(A)$, is defined in Figure 5.8.*

We should point out that the notions of multiplicity and size are compatible *w.r.t.* to neutrality of the conjunction operator since id has no impact on the two notions. It is also compatible *w.r.t.* the associativity of the conjunction and *w.r.t.* α-conversion.

Lemma 5.17 (Soundness of multiplicity) *For any term A and variable x such that $x \notin \mathsf{fv}(A)$ we have $\mathfrak{A}_x(A) = 0$.*

Proof. By induction on the structure of A.

- $A = y$ with $x \neq y$. By definition.
- $A = c$. By definition.
- $A = P \twoheadrightarrow A_1$. We have $\mathfrak{A}_x(P \twoheadrightarrow A_1) = \mathfrak{A}_x(A_1)$ and the result holds by applying the induction hypothesis.
- $A = A_1\,A_2$. Then $\mathfrak{A}_x(A_1\ A_2) = \mathfrak{A}_x(A_1) + \mathfrak{A}_x(A_2)$ and the result holds by applying twice the induction hypothesis since $x \notin \mathsf{fv}(A_1A_2)$ implies $x \notin \mathsf{fv}(A_1)$, $x \notin \mathsf{fv}(A_2)$.
- $A = A_1 \wr A_2$. Similar to the previous case.

$$
\begin{array}{lcl}
\mathfrak{A}_x(x) & = & 1 \\
\mathfrak{A}_x(y) & = & 0 \qquad \text{if } x \neq y \\
\mathfrak{A}_x(c) & = & 0 \\
\\
\mathfrak{A}_x(P \multimap A) & = & \mathfrak{A}_x(P) + \mathfrak{A}_x(A) \\
\\
\mathfrak{A}_x(A\ B) & = & \mathfrak{A}_x(A) + \mathfrak{A}_x(B) \\
\mathfrak{A}_x(A \wr B) & = & \mathfrak{A}_x(A) + \mathfrak{A}_x(B) \\
\\
\mathfrak{A}_x(B\,[\mathcal{C}]) & = & \mathfrak{A}^c_x(B\,[\mathcal{C}]) + \mathfrak{A}_x(B) \\
\mathfrak{A}_x(B\,[\phi]) & = & \mathfrak{A}^c_x(B\,[\phi]) + \mathfrak{A}_x(B)
\end{array}
$$

$$
\begin{array}{lcl}
\mathfrak{A}^c_x(B\,[\mathcal{C} \wedge \mathcal{D}]) & = & \mathfrak{A}^c_x(B\,[\mathcal{C}]) + \mathfrak{A}^c_x(B\,[\mathcal{D}]) \\
\mathfrak{A}^c_x(B\,[\phi \wedge \psi]) & = & \mathfrak{A}^c_x(B\,[\phi]) + \mathfrak{A}^c_x(B\,[\psi]) \\
\\
\mathfrak{A}^c_x(A\,[y = B]) & = & \mathfrak{A}_x(B) \times \mathfrak{A}_y(A) \\
\\
\mathfrak{A}^c_x(B\,[\mathtt{id}]) & = & 0 \\
\\
\mathfrak{A}^c_x(A\,[P \ll B]) & = & \displaystyle\sum_{y_j \in \mathsf{fv}(P)} \mathfrak{A}_x(B)\left(\mathfrak{A}_{y_j}(A) + 1\right) \\
 & & \text{where } x \neq y_j \text{ and } x \notin \mathsf{fv}(P)
\end{array}
$$

Figure 5.7: Multiplicity of a variable

$$
\begin{array}{lcl}
\mathfrak{S}(x) & = & 1 \\
\mathfrak{S}(c) & = & 1 \\
\\
\mathfrak{S}(P \multimap A) & = & \mathfrak{S}(P) + \mathfrak{S}(A) \\
\mathfrak{S}(A\ B) & = & \mathfrak{S}(A) + \mathfrak{S}(B) \\
\mathfrak{S}(A \wr B) & = & \mathfrak{S}(A) + \mathfrak{S}(B) \\
\\
\mathfrak{S}(B[\mathcal{C}]) & = & \mathfrak{S}^{c}(B[\mathcal{C}]) + \mathfrak{S}(B) \\
\mathfrak{S}(B[\phi]) & = & \mathfrak{S}^{c}(B[\phi]) + \mathfrak{S}(B)
\end{array}
$$

$$
\begin{array}{lcl}
\mathfrak{S}^{c}(B[\mathcal{C} \wedge \mathcal{D}]) & = & \mathfrak{S}^{c}(B[\mathcal{C}]) + \mathfrak{S}^{c}(B[\mathcal{D}]) \\
\mathfrak{S}^{c}(B[\phi \wedge \psi]) & = & \mathfrak{S}^{c}(B[\phi]) + \mathfrak{S}^{c}(B[\psi]) \\
\mathfrak{S}^{c}(A[y = B]) & = & \mathfrak{S}(B) \times \mathfrak{A}_{y}(A) \\
\\
\mathfrak{S}^{c}(B[\mathtt{id}]) & = & 0 \\
\\
\mathfrak{S}^{c}(A[P \ll B]) & = & \displaystyle\sum_{y_j \in \mathsf{fv}(P)} \mathfrak{S}(B) \times \left(\mathfrak{A}_{y_j}(A) + 1\right)
\end{array}
$$

Figure 5.8: Size of a term

- $A = A_1[\mathcal{C}]$. We can apply the induction hypothesis to A_1 and get $\mathfrak{A}_x(A_1) = 0$. Then, we show that for all constraint $\mathcal{C}$, we have $\mathfrak{A}^c_x(A_1[\mathcal{C}]) = 0$, by structural induction on $\mathcal{C}$.
  - $\mathcal{C} = (P \ll B)$. Apply the induction hypothesis on B (induction on the set of terms).
  - $\mathcal{C} = (\mathcal{D}_1 \wedge \mathcal{D}_2)$ then the result follows by applying the induction hypothesis to $\mathcal{D}_1$ and $\mathcal{D}_2$ (induction on the set of constraint).
  - $\mathcal{C} = \mathtt{id}$. The result holds by definition.
- $A = A_1[\phi]$. We proceed as before and we use an induction on ϕ
  - $\phi = (y = B)$. Apply the induction hypothesis to B.
  - $\phi = \psi_1 \wedge \psi_2$. The result follows by applying the induction hypothesis to ψ_1 and ψ_2 (induction on the set of substitutions).

□

Lemma 5.18 (Preservation of multiplicity) *For any variable* x *and any terms* A *and* B *such that* $A \to_{\mathcal{C}} B$*, the following inequality holds:*

$$\mathfrak{A}_x(A) \geq \mathfrak{A}_x(B)$$

Proof. We prove that we have

$$\mathfrak{A}^c_x(B[y = A][\mathcal{C}]) = \mathfrak{A}^c_x(B[\mathcal{C}]) \tag{5.1}$$

for all constraints $\mathcal{C}$, for all terms A and B such that $\mathcal{D}om(\mathcal{C}) \cap \mathsf{fv}(A) = \emptyset$. The proof is done by induction on $\mathcal{C}$. We only show the basic case, *i.e.*, if $\mathcal{C} = P \ll A'$.

$$\begin{aligned}
&\mathfrak{A}^c_x(B[y = A][P \ll A']) \\
= &\sum_{z_j \in \mathsf{fv}(P)} \mathfrak{A}_x(A')\left(\mathfrak{A}_{z_j}(B[y = A]) + 1\right) \\
= &\sum_{z_j \in \mathsf{fv}(P)} \mathfrak{A}_x(A')\left(\mathfrak{A}_{z_j}(A)\,\mathfrak{A}_y(B) + \mathfrak{A}_{z_j}(B) + 1\right) \\
= &\sum_{z_j \in \mathsf{fv}(P)} \mathfrak{A}_x(A')\left(\mathfrak{A}_{z_j}(A)\,\mathfrak{A}_y(B) + \mathfrak{A}_{z_j}(B) + 1\right) \\
= &\sum_{z_j \in \mathsf{fv}(P)} \mathfrak{A}_x(A')\left(\mathfrak{A}_{z_j}(B) + 1\right) && \text{By hypothesis and Lemma 5.17} \\
= &\ \mathfrak{A}^c_x(B[P \ll A'])
\end{aligned}$$

One can notice that the condition $x \notin \mathsf{fv}(A)$ is not needed since the multiplicities in A only depend on the variables in the domain of $\mathcal{C}$ (and not on x).

We can also prove by an easy induction on n that if x does not belong to $\cup_{i=1}^{n}\mathsf{fv}(P_i)$ and if for all i, $x \neq y_i$ then

$$\mathfrak{A}_x\left(B\left[\bigwedge_{i=1..n} P_i \ll M_i\right]\right) = \mathfrak{A}_x(B) + \sum_{i=1}^{n}\sum_{x_j \in \mathsf{fv}(P_i)} \mathfrak{A}_x(A_i) \times \left(\mathfrak{A}_{x_j}(B) + 1\right) \tag{5.2}$$

$$\mathfrak{A}_x\left(B\left[\bigwedge_{i=1..n} y_i = A_i\right]\right) = \mathfrak{A}_x(B) + \sum_{i=1}^{n} \mathfrak{A}_x(A_i) \times \mathfrak{A}_{x_i}(B) \tag{5.3}$$

Since the multiplicity is defined by a monotonic induction it is sufficient to prove the result when the reduction takes place at the root of A. We give all the computations in Figure 5.9. □

Lemma 5.19 (Preservation of size) *For any terms A and B such that $A \to_{\mathcal{C}} B$, the following inequality holds:*

$$\mathfrak{S}(A) \geq \mathfrak{S}(B)$$

*The inequality is strict if the reduction is done using either the (*ToSubst*) rule, or the (*Replace*) rule, or the (*Idem*) rule, or the decomposition rules.*

Proof. The proof of the lemma is similar to that of Lemma 5.18. We only give some cases.

$$\begin{array}{lll}
\text{(Replace)} & \mathfrak{S}(x[\phi \wedge x = A \wedge \psi]) \\
& = & \mathfrak{S}(x) + \mathfrak{S}^{c}(x[\phi]) + \mathfrak{S}^{c}(x[x = A]) + \mathfrak{S}^{c}(x[\psi]) \\
& = & 1 + \mathfrak{S}^{c}(x[\phi]) + \mathfrak{S}(A) \times \mathfrak{A}_x(x) + \mathfrak{S}^{c}(x[\psi]) \\
& > & \mathfrak{S}(A)
\end{array}$$

$$\begin{array}{lll}
\text{(ToSubst)} & \mathfrak{S}(x[\mathcal{C} \wedge (x \ll A) \wedge \mathcal{D}]) \\
& = & \mathfrak{S}(x) + \mathfrak{S}^{c}(x[\mathcal{C} \wedge (x \ll A) \wedge \mathcal{D}]) \\
& = & \mathfrak{S}(x) + \mathfrak{S}^{c}(x[\mathcal{C}]) + \mathfrak{S}^{c}(x[x \ll A]) + \mathfrak{S}^{c}(x[\mathcal{D}]) \\
& > & 1 + \mathfrak{S}(A) \times \mathfrak{A}_x(x) \\
& > & \mathfrak{S}(A)
\end{array}$$

$$\begin{array}{lll}
\text{(Idem)} & \mathfrak{S}(B[\mathcal{C} \wedge (x \ll A) \wedge \mathcal{D} \wedge (x \ll A) \wedge \mathcal{E}]) \\
& = & \mathfrak{S}(B) + \mathfrak{S}^{c}(B[\mathcal{C}]) + 2\mathfrak{S}^{c}(B[x \ll A]) + \mathfrak{S}^{c}(B[\mathcal{D}]) + \mathfrak{S}^{c}(B[\mathcal{E}]) \\
& > & \mathfrak{S}(B) + \mathfrak{S}^{c}(B[\mathcal{C}]) + \mathfrak{S}^{c}(B[x \ll A]) + \mathfrak{S}^{c}(B[\mathcal{D}]) + \mathfrak{S}^{c}(B[\mathcal{E}]) \\
& = & \mathfrak{S}(B[\mathcal{C} \wedge (x \ll A) \wedge \tau \wedge \mathcal{D}])
\end{array}$$

□

$$
\begin{array}{lll}
\text{(Identity)} & & \mathfrak{A}_x\,(M\,[\mathtt{id}]) \\
& = & \mathfrak{A}^c_x\,(A\,[\mathtt{id}]) + \mathfrak{A}_x\,(A) \\
& = & \mathfrak{A}_x\,(A) \\
\\
\text{(Replace)} & & \mathfrak{A}_x\,(y\,[\phi \wedge y = A \wedge \psi]) \\
& = & \mathfrak{A}_x\,(y) + \mathfrak{A}^c_x\,(y\,[\phi \wedge y = A \wedge \psi]) \\
& = & \mathfrak{A}_x\,(y) + \mathfrak{A}^c_x\,(y\,[\phi]) + \mathfrak{A}^c_x\,(y\,[y = A]) + \mathfrak{A}^c_x\,(y\,[\psi]) \\
& \geq & \mathfrak{A}^c_x\,(y\,[y = A]) \\
& = & \mathfrak{A}^c_x\,(A) \times \mathfrak{A}_y\,(y) \\
& = & \mathfrak{A}^c_x\,(A) \\
\\
\text{(Const)} & & \mathfrak{A}_x\,(c\,[\phi]) \\
& = & \mathfrak{A}_x\,(c) + \mathfrak{A}^c_x\,(c\,[\phi]) \\
& \geq & \mathfrak{A}_x\,(c) \\
\\
\text{(Abs)} & & \mathfrak{A}_x\,((A \multimap B)\,[\phi]) \\
& = & \mathfrak{A}^c_x\,((A\ B)\,[\phi]) + \mathfrak{A}_x\,(A \multimap B) \\
& = & \mathfrak{A}^c_x\,(A\,[\phi]) + \mathfrak{A}_x\,(B) + \mathfrak{A}^c_x\,(A\,[\phi]) + \mathfrak{A}_x\,(B) \\
& = & \mathfrak{A}_x\,(A\,[\phi] \multimap B\,[\phi]) \\
\\
\text{(App)} & & \mathfrak{A}_x\,((A\ B)\,[\phi]) \\
& = & \mathfrak{A}^c_x\,((A\ B)\,[\phi]) + \mathfrak{A}_x\,(A\ B) \\
& = & \mathfrak{A}^c_x\,(A\,[\phi]) + \mathfrak{A}_x\,(B) + \mathfrak{A}^c_x\,(A\,[\phi]) + \mathfrak{A}_x\,(B) \\
& = & \mathfrak{A}_x\,(A\,[\phi]\ B\,[\phi]) \\
\\
\text{(Constraint)} & & \mathfrak{A}_x\left(A[P_i \ll B_i]_{i=1}^n\,[x_j = A_j]_{i=1}^m\right) \\
& = & \mathfrak{A}_x\,(A[P_i \ll B_i]_{i=1}^n) + \sum_j \mathfrak{A}_x\,(A_j)\,\mathfrak{A}_{x_j}\,(A[P_i \ll B_i]_{i=1}^n) \\
& \overset{\text{Eq. 5.2}}{=} & \mathfrak{A}_x\,(A) + \sum_i \sum_{y_j \in \mathrm{fv}(P_i)} \mathfrak{A}_x\,(B_i) \times \mathfrak{A}_{y_j}\,(A) + \sum_i \mathfrak{A}^c_x\,((A\,[P_i \ll B_i])\,[\phi]) \\
& = & \sum_i \sum_{y_j \in \mathrm{fv}(P_i)} \mathfrak{A}_x\,(B_i) \times \mathfrak{A}_{y_j}\,(A) + \sum_i \mathfrak{A}^c_x\,((A\,[P_i \ll B_i])\,[\phi]) \\
& = & \sum_i \sum_{y_j \in \mathrm{fv}(P_i)} (\mathfrak{A}_x\,(B_i) + \mathfrak{A}^c_x\,(B_i\,[\phi])) \times (\mathfrak{A}_{y_j}\,(A) + \mathfrak{A}^c_{y_j}\,(A\,[\phi])) \\
& = & \sum_i \sum_{y_j \in \mathrm{fv}(P_i)} \mathfrak{A}_x\,(B_i\,[\phi]) \times \mathfrak{A}_{y_j}\,(A\,[\phi]) \\
& = & \mathfrak{A}_x\,(A\,[\phi][P_i \ll B_i\,[\phi]]_{i=1}^n) \\
\\
\text{(Compose)} & & \mathfrak{A}_x\left(B[x_j = A_j]_{j=1}^m[x_i = A_i]_{i=1}^n\right) \\
& \overset{\text{Eq. 5.3}}{=} & \mathfrak{A}_x\left(B[x_j = A_j]_{j=1}^m\right) + \sum_i \mathfrak{A}_x\,(A_i)\,\mathfrak{A}_{x_i}\left(B[x_j = A_j]_{j=1}^m\right) \\
& \overset{\text{Eq. 5.3}}{=} & \mathfrak{A}_x\,(B) + \sum_j \mathfrak{A}_x\,(A_j)\,\mathfrak{A}_{x_j}\,(B) + \sum_i \mathfrak{A}_x\,(A_i)\left(\mathfrak{A}_{x_i}\,(B) + \sum_j \mathfrak{A}_{x_i}\,(A_j)\,\mathfrak{A}_{x_j}\,(B)\right) \\
& = & \mathfrak{A}_x\,(B) + \sum_i \mathfrak{A}_x\,(A_i)\,\mathfrak{A}_{x_i}\,(B) + \sum_j \mathfrak{A}_{x_j}\,(B)\left(\mathfrak{A}_x\,(A_j) + \sum_i \mathfrak{A}_x\,(A_j)\,\mathfrak{A}_{x_i}\,(A_j)\right) \\
& = & \mathfrak{A}_x\,(B) + \sum_i \mathfrak{A}_x\,(A_i)\,\mathfrak{A}_{x_i}\,(B) + \sum_j \mathfrak{A}_x\,(A_j[x_i = A_i]_{i=1}^n) \\
& \overset{\text{Eq. 5.3}}{=} & \mathfrak{A}_x\left(\left[x_i = A_i \wedge x_j = A_j\,[x_i = A_i]_{i=1}^n\right]_{i,j}(B)\right) \\
\\
\text{(ToSubst)} & & \mathfrak{A}_x\,(B\,[\mathcal{C} \wedge (y \ll A) \wedge \mathcal{D}]) \\
& = & \mathfrak{A}_x\,(B) + \mathfrak{A}^c_x\,(B\,[\mathcal{C} \wedge (y \ll A) \wedge \mathcal{D}]) \\
& = & \mathfrak{A}_x\,(B) + \mathfrak{A}^c_x\,(B\,[\mathcal{C}]) + \mathfrak{A}^c_x\,(B\,[y \ll A]) + \mathfrak{A}^c_x\,(B\,[\mathcal{D}]) \\
& = & \mathfrak{A}_x\,(B) + \mathfrak{A}^c_x\,(B\,[\mathcal{C}]) + \mathfrak{A}_x\,(A) \times (\mathfrak{A}_y\,(B) + 1) + \mathfrak{A}^c_x\,(B\,[\mathcal{D}]) \\
& > & \mathfrak{A}_x\,(B) + \mathfrak{A}^c_x\,(B\,[\mathcal{C}]) + \mathfrak{A}_x\,(A) \times \mathfrak{A}_y\,(B) + \mathfrak{A}^c_x\,(B\,[\mathcal{D}]) \\
& \overset{\text{Eq. 5.1}}{=} & \mathfrak{A}_x\,(B) + \mathfrak{A}^c_x\,(B\,[y = A]\,[\mathcal{C}]) + \mathfrak{A}_x\,(A) \times \mathfrak{A}_y\,(B) + \mathfrak{A}^c_x\,(B\,[y = A]\,[\mathcal{D}]) \\
& = & \mathfrak{A}_x\,(B\,[y = A]) + \mathfrak{A}^c_x\,(B\,[y = A]\,[\mathcal{C} \wedge \mathcal{D}]) \\
& = & \mathfrak{A}_x\,(B\,[y = A]\,[\mathcal{C} \wedge \mathcal{D}])
\end{array}
$$

Figure 5.9: Preservation of multiplicity

Definition 5.20 (Polynomial interpretation) *We use the standard order on natural numbers in order to define the following polynomial interpretation:*

$$\begin{array}{lcllcl}
\mathfrak{I}(k) & = & 3 & \mathfrak{I}(P \twoheadrightarrow B) & = & \mathfrak{I}(P) + \mathfrak{I}(B) + 1 \\
\mathfrak{I}(A \wr B) & = & \mathfrak{I}(A) + \mathfrak{I}(B) + 1 & \mathfrak{I}(A\ B) & = & \mathfrak{I}(A) + \mathfrak{I}(B) + 1 \\
\mathfrak{I}(B\,[\mathcal{C}]) & = & \mathfrak{I}(\mathcal{C}) + \mathfrak{I}(B) + 1 & \mathfrak{I}(A\,[\phi]) & = & (\mathfrak{I}(\phi) + 2) \times \mathfrak{I}(A) \\
 & & & & & \\
\mathfrak{I}(\mathcal{C} \wedge \mathcal{D}) & = & \mathfrak{I}(\mathcal{C}) + \mathfrak{I}(\mathcal{D}) & \mathfrak{I}(\phi \wedge \psi) & = & \mathfrak{I}(\phi) + \mathfrak{I}(\psi) \\
\mathfrak{I}(\mathit{id}) & = & 0 & \mathfrak{I}(x = A) & = & \mathfrak{I}(A) \\
\mathfrak{I}(P \ll A) & = & \mathfrak{I}(P) + \mathfrak{I}(A) & & &
\end{array}$$

We should point out that the polynomial interpretation is compatible *w.r.t.* to neutrality of the conjunction operator since `id` has no impact on the two notions. It is also compatible *w.r.t.* the associativity of the conjunction and *w.r.t.* α-conversion. Moreover, since the addition and the multiplication are increasing on naturals, the monotonicity condition $a > b$ implies $\mathfrak{I}(a) > \mathfrak{I}(b)$ is clearly satisfied. We show that for any terms A and B such that $A \to_{\mathcal{C}} B$ the image of A is greater than that of B for any replacement of the interpretation of the variables of A and B by naturals bigger than 2.

Lemma 5.21 *For any terms A and B such that $A \to_{\mathcal{C}} B$ using either (*Compose*), or (*Constraint*), or (*Abs*), or (*Const*), or (*Var*), or (*Identity*), or (*App*), or (*Struct*) then*

$$\mathfrak{I}(A) > \mathfrak{I}(B)$$

Proof. For any natural number n, for any terms B, A_i for any patterns P_i and for any variables y_i the following equalities hold:

$$\mathfrak{I}(B\left[\bigwedge_{i=1..n} P_i \ll A_i\right]) = \mathfrak{I}(B) + 1 + \sum_i (\mathfrak{I}(P_i) + \mathfrak{I}(A_i)) \tag{5.4}$$

$$\mathfrak{I}(B\left[\bigwedge_{i=1..n} y_i = A_i\right]) = \mathfrak{I}(B) \times \left(2 + \sum_i \mathfrak{I}(A_i)\right) \tag{5.5}$$

We can first remark that for any constraint $\mathcal{C}$ (resp. substitution ϕ) we have $\mathfrak{I}(\mathcal{C}) \geq 0$ (resp. $\mathfrak{I}(\phi) \geq 0$) and for any term A we have $\mathfrak{I}(A) > 2$.
We check the inequality for each rule mentioned in the Lemma in Figure 5.10. □

Lemma 5.22 (SN of $\to_{\mathcal{C}}$) *The relation $\to_{\mathcal{C}}$ is strongly normalizing.*

Proof. Every pair of terms in $\to_{\mathcal{C}}$ (strictly) decreases the lexicographic product $(\mathfrak{S}(\,), \mathfrak{I}(\,))$.

$$
\begin{array}{lll}
(\mathtt{Identity_s}) & \mathfrak{I}(A\,[\mathtt{id_s}]) = (\mathfrak{I}\,(\mathtt{id_s}) + 2) \times \mathfrak{I}(A) = 2 \times \mathfrak{I}(A) > \mathfrak{I}(A) \\
\\
(\mathtt{Identity_c}) & \mathfrak{I}(A\,[\mathtt{id_c}]) = \mathfrak{I}\,(\mathtt{id_c}) + \mathfrak{I}(A) + 1 = \mathfrak{I}(A) + 1 > \mathfrak{I}(A) \\
\\
(\mathrm{Var}) & \mathfrak{I}(y\,[\phi]) = (\mathfrak{I}\,(\phi) + 2) \times \mathfrak{I}(y) > \mathfrak{I}(y) \\
\\
(\mathrm{Const}) & \mathfrak{I}(c\,[\phi]) = (\mathfrak{I}\,(\phi) + 2) \times \mathfrak{I}(c) > \mathfrak{I}(c) \\
\\
(\mathrm{Abs}) & \mathfrak{I}((P \twoheadrightarrow A)\,[\phi]) = (\mathfrak{I}\,(\phi) + 2) \times \mathfrak{I}(P \twoheadrightarrow A) \\
= & (\mathfrak{I}\,(\phi) + 2) \times (\mathfrak{I}(P) + \mathfrak{I}(A) + 1) \\
> & (\mathfrak{I}\,(\phi) + 2) \times \mathfrak{I}(P) + (\mathfrak{I}\,(\phi) + 2) \times \mathfrak{I}(A) + 1 \\
= & \mathfrak{I}(P\,[\phi]) + \mathfrak{I}(A\,[\phi]) + 1 \\
= & \mathfrak{I}(P\,[\phi] \twoheadrightarrow A\,[\phi]) \\
\\
(\mathrm{App}) & \mathfrak{I}((A\,N)\,[\phi]) = (\mathfrak{I}\,(\phi) + 2) \times \mathfrak{I}(A\,N) \\
= & (\mathfrak{I}\,(\phi) + 2) \times (\mathfrak{I}(A) + \mathfrak{I}(N) + 1) \\
> & (\mathfrak{I}\,(\phi) + 2) \times \mathfrak{I}(A) + (\mathfrak{I}\,(\phi) + 2) \times \mathfrak{I}(N) + 1 \\
= & \mathfrak{I}(A\,[\phi]) + \mathfrak{I}(N\,[\phi]) + 1 \\
= & \mathfrak{I}(A\,[\phi]\;N\,[\phi]) \\
\\
(\mathrm{Struct}) & \mathfrak{I}((A \wr N)\,[\phi]) = (\mathfrak{I}\,(\phi) + 2) \times \mathfrak{I}(A \wr N) + 1 \\
= & (\mathfrak{I}\,(\phi) + 2) \times (\mathfrak{I}(A) + \mathfrak{I}(N) + 1) + 1 \\
> & (\mathfrak{I}\,(\phi) + 2) \times \mathfrak{I}(A) + 1 + (\mathfrak{I}\,(\phi) + 2) \times \mathfrak{I}(N) + 1 \\
= & \mathfrak{I}(A\,[\phi]) + \mathfrak{I}(N\,[\phi]) + 1 \\
= & \mathfrak{I}(A\,[\phi] \wr N\,[\phi]) \\
\\
(\mathrm{Constraint}) & \mathfrak{I}(A[P_i \ll N_i]_{i=1}^n\,[\phi]) \\
\overset{\text{Eq. 5.4}}{=} & (\mathfrak{I}\,(\phi) + 2) \times (\mathfrak{I}(A) + \textstyle\sum_i \mathfrak{I}(P_i) + \sum_i \mathfrak{I}(N_i) + 1) \\
> & \textstyle\sum_i((\mathfrak{I}(\phi) + 2) \times \mathfrak{I}(P_i)) + \sum_i((\mathfrak{I}(\phi) + 2) \times \mathfrak{I}(N_i)) + \mathfrak{I}(\phi) + \mathfrak{I}(A) + 1 \\
= & \textstyle\sum_i(\mathfrak{I}(P_i\,[\phi]) + \mathfrak{I}(N_i\,[\phi])) + \mathfrak{I}(\phi) + \mathfrak{I}(A) + 1 \\
= & \textstyle\sum_i \mathfrak{I}(P_i\,[\phi] \ll N_i\,[\phi]) + \mathfrak{I}(A\,[\phi]) + 1 \\
= & \mathfrak{I}(A\,[\phi][P_i\,[\phi] \ll N_i\,[\phi]]_{i=1}^n) \\
\\
(\mathrm{Compose}) & \mathfrak{I}(N\,[x_i = A_i]_{i=1}^n\,[\phi]) \\
= & (\mathfrak{I}(\phi) + 2) \times \mathfrak{I}(N\,[x_i = A_i]_{i=1}^n) \\
\overset{\text{Eq. 5.5}}{=} & (\mathfrak{I}(\phi) + 2) \times (\textstyle\sum_i \mathfrak{I}(A_i) + 2) \times \mathfrak{I}(N) \\
= & \mathfrak{I}(N) \times (2 \times (\mathfrak{I}(\phi) + 2) + (\mathfrak{I}(\phi) + 2) \times \textstyle\sum_i \mathfrak{I}(A_i)) \\
> & \mathfrak{I}(N) \times (\mathfrak{I}(\phi) + 2 + (\mathfrak{I}(\phi) + 2) \times \textstyle\sum_i \mathfrak{I}(A_i)) \\
= & \mathfrak{I}(N) \times (\mathfrak{I}(\phi) + 2 + \textstyle\sum_i \mathfrak{I}(A_i\,[\phi])) \\
= & \mathfrak{I}(N)\,(\mathfrak{I}(\phi) + 2 + \textstyle\sum_i \mathfrak{I}(x_i = A_i\,[\phi])) \\
= & \left(\mathfrak{I}(\phi \wedge \bigwedge_i (x_i = A_i\,[\phi])) + 2\right) \times \mathfrak{I}(N) \\
= & \mathfrak{I}(N\,[\phi \wedge x_i = A_i\,[\phi]]_{i=1}^n)
\end{array}
$$

Figure 5.10: Strict decrease of polynomial interpretation

	$\mathfrak{S}(\ )$	$\mathfrak{I}(\ )$
(Decompose$_\iota$)	$>$	
(Decompose$_\mathcal{F}$)	$>$	
(Idem)	$>$	
(ToSubst)	$>$	
(Identity)	$\geq$	$>$
(Replace)	$>$	
(Var)	$\geq$	$>$
(Const)	$\geq$	$>$
(Abs)	$\geq$	$>$
(App)	$\geq$	$>$
(Struct)	$\geq$	$>$
(Constraint)	$\geq$	$>$
(Compose)	$\geq$	$>$

□

5.4.4 Confluence of the sub-relations

First, we show that $\twoheadrightarrow_{\mathcal{C}}$ is coherent modulo `A1`, *i.e.*, two equivalent (modulo `A1`) terms can be reduced to two equivalent ones.

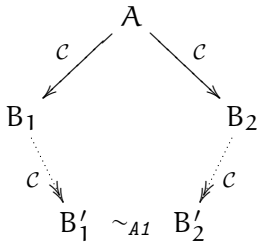
Lemma 5.23 (Coherence modulo `A1`) *For any terms* A, B, A' *such that* $\mathsf{A} \twoheadrightarrow_{\mathcal{C}} \mathsf{B}$ *and* $\mathsf{A} \sim_{A1} \mathsf{A}'$, *there exists* B' *such that* $\mathsf{B} \sim_{A1} \mathsf{B}'$ *and* $\mathsf{A}' \twoheadrightarrow_{\mathcal{C}} \mathsf{B}'$.

$$\begin{array}{ccc} \mathsf{A} & \sim_{A1} & \mathsf{A}' \\ {\scriptstyle \mathcal{C}}\downarrow & & \vdots{\scriptstyle \mathcal{C}} \\ \mathsf{B} & \sim_{A1} & \mathsf{B}' \end{array}$$

Proof. The proof is done by case analysis on the rule used to reduced A into B. The diagram is easily closed for all rules except for the (`Idem`) rule, for which we may observe that, thanks to the extension variables (the variables called $\mathcal{C}$ and $\mathcal{E}$ in the (`Idem`) rule), all the computations from A to A' can be reproduced when performing the $\twoheadrightarrow_{\mathcal{C}}$ reductions from A' to B'. □

Lemma 5.24 (Local confluence modulo `A1`) *For any terms* A, B_1, B_2 *such that* $\mathsf{A} \twoheadrightarrow_{\mathcal{C}} \mathsf{B}_1$ *and that* $\mathsf{A} \twoheadrightarrow_{\mathcal{C}} \mathsf{B}_2$, *there exist two terms* B'_1 *and* B'_2 *such that* $\mathsf{B}_1 \twoheadrightarrow_{\mathcal{C}} \mathsf{B}'_1$ *and*

$B_2 \twoheadrightarrow_{\mathcal{C}} B_2'$ *with* $B_1' \sim_{A1} B_2'$*:*



Proof. We proceed by induction on A. We suppose that the redexes are not disjoint (otherwise the result is easy to prove). In what follows, we call "the first reduction" the reduction from A to B_1 and "the second reduction" the reduction from A to B_23.

- If $A = A_1\, A_2$ then the two reductions take place in the same A_i. The result follows by applying the induction hypothesis.
- If $A = A_1\,[\mathcal{C}]$ then we proceed by induction on $\mathcal{C}$. If the two reductions take place in A_1 then the result follows by induction.
  - If $\mathcal{C} = \mathtt{id}_m$ then the result is obvious.
  - If $\mathcal{C} = P \ll A_2$ then, if the two reductions take place in A_2 then the result follows by induction. If the first reduction is a decomposition rule, lets say $(\mathsf{Decompose}_\wr)$ then

$$\begin{array}{ccc} & A_1\,[P_1 \wr P_2 \ll C_1 \wr C_2] & \\ \swarrow & & \searrow \\ A_1\,[P_1 \ll C_1 \wedge P_2 \ll C_2] & & A_1\,[P_1 \wr P_2 \ll C_1 \wr C_2'] \\ \searrow & & \swarrow \\ & A_1\,[P_1 \ll C_1 \wedge P_2 \ll C_2'] & \end{array}$$

    If the first reduction is $(\mathsf{ToSubst})$ then just swap the two reduction steps.
  - If $\mathcal{C} = \mathcal{C}_1 \wedge \mathcal{C}_2$ then, if the two reductions take place in $\mathcal{C}_1$ or in $\mathcal{C}_2$ then the result follows by induction. If the first reduction is $(\mathsf{ToSubst})$ then just swap the two reduction steps. Otherwise the first reduction reduces $\mathcal{C}$ at its top position and thus $\mathcal{C}$ must be equal modulo $\mathtt{A1}$ to $\mathcal{D}_1 \wedge (x \ll B) \wedge \mathcal{D}_2 \wedge (x \ll B) \wedge \mathcal{D}_3$. If the second reduction takes place in one of the $\mathcal{D}_i$ then the result follows easily. If the second reduction occurs in one occurrence of B by reducing it to B' then close the diagram on the left by reducing the remaining occurrence of B into B' and on the right by first reducing the other occurrence of B into B' and then apply the (Idem) rule. If the two reductions use the rule (Idem) then there are two cases (any permutation of constraints in the first case is similar) :
    We can have

$$\mathcal{C} \sim_{\mathtt{A1}} \Big(\mathcal{C}_1 \wedge (x \ll A) \wedge \mathcal{C}_2 \wedge (y \ll B) \wedge \mathcal{C}_3 \wedge (x \ll A) \wedge \mathcal{C}_4 \wedge (y \ll B) \wedge \mathcal{C}_5\Big)$$

and the first reduction eliminates the doubletons related to x and the second reduction eliminates the doubletons related to y. Then, to close the diagram simply swap the two reductions (this is possible thanks to the extension variables in the (Idem) rule).

Otherwise

$$\mathcal{C} \sim_{A1} \left(\mathcal{C}_1 \wedge (x \ll A) \wedge \mathcal{C}_2 \wedge (x \ll A) \wedge \mathcal{C}_3 \wedge (x \ll A) \wedge \mathcal{C}_4\right)$$

and conclude the case as in the previous case.

- If $A = A_1[\phi]$ then we proceed by induction on ϕ and by case analysis on the rule used for the first reduction.

  The interesting case is when the first reduction uses the (Constraint) rule and the second reduction is done using the (ToSubst) rule. In this case, the result follows using the (Compose) rule and Lemma 5.14. Let us denote $\wedge_i(P_i \ll B_i)$ by $\mathcal{C}$ and $\wedge_j(P_j \ll B_j)$ by $\mathcal{D}$. In the following we simply write $\mathcal{C}[x = B]$ for $\wedge_i(P_i[x = B] \ll B_i[x = B])$. Let us suppose moreover that $A_1 = A_3[\mathcal{C} \wedge (y \ll A_2) \wedge \mathcal{D}]$ in which case the first reduction gives:

$$\begin{array}{ll} & (A_3[\mathcal{C} \wedge (y \ll A_2) \wedge \mathcal{D}])[x = B] \\ \to_{\text{Constraint}} & A_3[x = B][\mathcal{C}[x = B] \wedge (y[x = B] \ll A_2[x = B]) \wedge \mathcal{D}[x = B]] \\ \to_{\text{Var}} & A_3[x = B][\mathcal{C}[x = B] \wedge (y \ll A_2[x = B]) \wedge \mathcal{D}[x = B]] \\ \to_{\text{ToSubst}} & A_3[x = B][y = A_2[x = B]][\mathcal{C}[x = B] \wedge \mathcal{D}[x = B]] \\ \to_{\text{Compose}} & A_3[y = A_2[x = B] \wedge x = B[y = A_2[x = B]]][\mathcal{C}[x = B] \wedge \mathcal{D}[x = B]] \end{array}$$

  By α-conversion, $y \notin \mathsf{fv}(B)$ and so, by applying Lemma 5.14, we get

$$\twoheadrightarrow_{\mathcal{C}} \quad A_3[y = A_2[x = B] \wedge x = B'][\mathcal{C}[x = B] \wedge \mathcal{D}[x = B]] \qquad (1)$$

  with $B \twoheadrightarrow_{\mathcal{C}} B'$. The second reduction gives:

$$\begin{array}{ll} & (A_3[\mathcal{C} \wedge (y \ll A_2) \wedge \mathcal{D}])[x = B] \\ \to_{\text{ToSubst}} & (A_3[y = A_2][\mathcal{C} \wedge \mathcal{D}])[x = B] \\ \to_{\text{Constraint}} & A_3[y = A_2][x = B][\mathcal{C}[x = B] \wedge \mathcal{D}[x = B]] \\ \to_{\text{Compose}} & A_3[x = B \wedge y = A_2[x = B]][\mathcal{C}[x = B] \wedge \mathcal{D}[x = B]] \\ \\ \twoheadrightarrow_{\mathcal{C}} & A_3[x = B' \wedge y = A_2[x = B]][\mathcal{C}[x = B] \wedge \mathcal{D}[x = B]] \qquad (2) \end{array}$$

  Lemma 5.14 concludes the case showing that the terms (1) and (2) have a common reduct.

□

Lemma 5.25 *The relation $\rightarrow_{\mathcal{C}}$ is confluent modulo* `A1`*.*

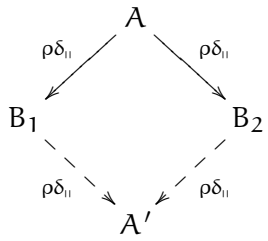
Proof. We actually prove a stronger property, that is that the relation is Church-Rosser modulo `A1`, which is obtained according to [Ohl98] from the strong normalization (Lemma 5.22), the coherence (Lemma 5.23) and the local confluence (Lemma 5.24) modulo `A1` of $\rightarrow_{\mathcal{C}}$. □

5.4.5 Parallel version of the ρδ

The parallelization of the ρ and δ rules is an easy extension of the parallel reduction of Tait and Martin-Löf given in Part I. The definition is here a bit longer because of the many congruence rules.

Definition 5.26 (Parallelization of ρδ) *The parallelization of the relation induced by the rules (ρ) and (δ), denoted $\Rightarrow_{\rho\delta}$, is defined inductively in Figure 5.11.*

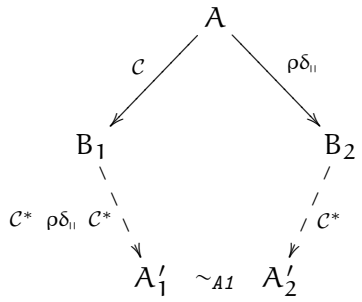
Lemma 5.27 (Diamond Property of $\Rightarrow_{\rho\delta}$) *For any terms A, B_1, B_2 there exists a term A' such that the following diagram holds:*



Proof. The relation $\Rightarrow_{\rho\delta}$ is the parallelization of an orthogonal system. □

5.4.6 Yokouchi-Hikita's diagram modulo and confluence of ρ°_{x}

Lemma 5.28 (Yokouchi-Hikita's diagram modulo) *For any terms A, B_1 and B_2 in the* linear ρ°_{x} *there exists the terms A'_1, A'_2 such that the following diagram holds:*



$$\frac{A \Rightarrow_{\rho\delta} A' \quad B \Rightarrow_{\rho\delta} B'}{(P \multimap A)\, B \Rightarrow_{\rho\delta} A'[P \ll B']}$$

$$\frac{A'_1 \Rightarrow_{\rho\delta} A_1 \quad A_2 \Rightarrow_{\rho\delta} A'_2 \quad B \Rightarrow_{\rho\delta} B'}{(A_1 \wr A_2)\, B \Rightarrow_{\rho\delta} A_1\, B' \wr A'_2\, B'}$$

$$\frac{}{A \Rightarrow_{\rho\delta} A} \qquad \frac{A \Rightarrow_{\rho\delta} A'}{P \multimap A \Rightarrow_{\rho\delta} P \multimap A'}$$

$$\frac{A \Rightarrow_{\rho\delta} A' \quad B \Rightarrow_{\rho\delta} B'}{A\, B \Rightarrow_{\rho\delta} A'\, B'} \qquad \frac{A \Rightarrow_{\rho\delta} A' \quad B \Rightarrow_{\rho\delta} B'}{A \wr B \Rightarrow_{\rho\delta} A' \wr B'}$$

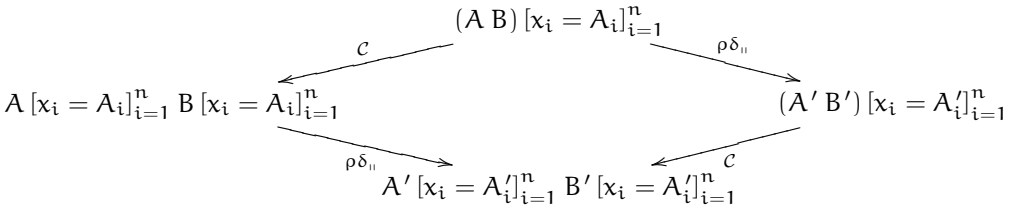
$$\frac{\mathcal{C} \Rightarrow_{\rho\delta} \mathcal{C}' \quad A \Rightarrow_{\rho\delta} A'}{A\,[\mathcal{C}] \Rightarrow_{\rho\delta} A'\,[\mathcal{C}']} \qquad \frac{\phi \Rightarrow_{\rho\delta} \phi' \quad A \Rightarrow_{\rho\delta} A'}{A\,[\phi] \Rightarrow_{\rho\delta} A'\,[\phi']}$$

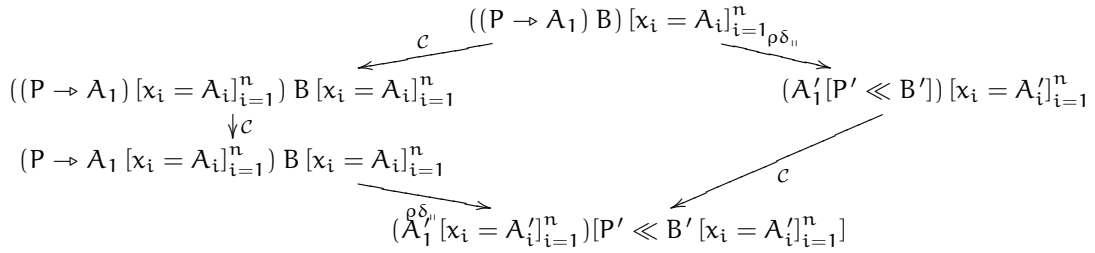
$$\frac{B \Rightarrow_{\rho\delta} B'}{(P \ll B) \Rightarrow_{\rho\delta} (P \ll B')} \qquad \frac{B \Rightarrow_{\rho\delta} B'}{(x = B) \Rightarrow_{\rho\delta} (x = B')}$$

$$\frac{\mathcal{C} \Rightarrow_{\rho\delta} \mathcal{C}' \quad \mathcal{D} \Rightarrow_{\rho\delta} \mathcal{D}'}{\mathcal{C} \wedge \mathcal{D} \Rightarrow_{\rho\delta} \mathcal{C}' \wedge \mathcal{D}'} \qquad \frac{\phi \Rightarrow_{\rho\delta} \phi' \quad \psi \Rightarrow_{\rho\delta} \psi'}{\phi \wedge \psi \Rightarrow_{\rho\delta} \phi' \wedge \psi'}$$

Figure 5.11: Parallel version of the ρδ-rules

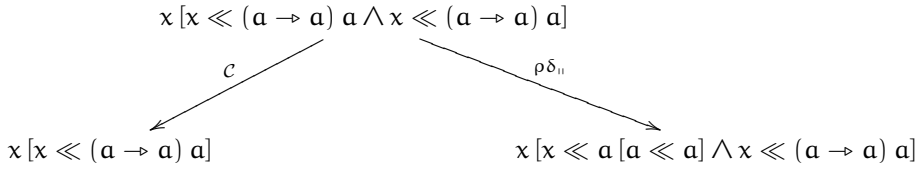
Proof. When the two steps from A to B_1 and from A to B_2 do not overlap, the proof is easy. So we have to inspect every critical pair[1]. Since a strict subexpression of a $\rho\delta_{\shortparallel}$ redex can never overlap with a $\mathcal{C}$ redex, it is sufficient to work by cases on the derivation from A to B_1. We only consider the (App) rule. The other cases are simpler and similar.





The case where the $\rho\delta_{\shortparallel}$ reduction from A to B_2 concerns a δ redex is similar to the previous one. □

Notice that the linearity condition is essential here since it ensures that the rule (Idem) is never used. If this condition is not enforced and non-linear terms are allowed, the following (non-joinable) diagram gives a counterexample of Yokouchi-Hikita's diagram modulo.



Lemma 5.29 *The relation $\twoheadrightarrow_{\mathsf{K}} \Rightarrow_{\rho\delta} \twoheadrightarrow_{\mathsf{K}}$ is confluent modulo* `A1`.

Proof. All the hypotheses of the Yokouchi-Hikita's lemma modulo (Lemma 6.10) are proved in the previous lemmas:

- The relation $\rightarrow_{\mathsf{K}}$ is strongly normalizing by Lemma 5.22.
- The relation $\rightarrow_{\mathsf{K}}$ is confluent modulo `A1` by Lemma 5.25.

[1] As in the $\lambda\sigma_{\Uparrow}$-calculus a critical pair has a slightly different meaning than the standard one because of the parallel reduction.

- The relation $\Rightarrow_{\rho\delta}$ has the diamond property by Lemma 5.27.
- The relations $\rightarrow_{\mathsf{K}}$ and $\Rightarrow_{\rho\delta}$ are coherent modulo $\mathtt{A1}$: the coherence of $\Rightarrow_{\rho\delta}$ modulo $\mathtt{A1}$ is obvious and the coherence of $\rightarrow_{\mathsf{K}}$ is obtained by Lemma 5.23.
- Yokouchi-Hikita's diagram modulo holds by Lemma 5.28.

□

Lemma 5.30 *The relation $\rightarrow_{\rho^\circ_{x},A1}$ is confluent modulo $A1$.*

Proof. We have $\rightarrow_{\rho\delta} \subseteq \Rightarrow_{\rho\delta} \subseteq \twoheadrightarrow_{\rho\delta}$ thus $\rightarrow_{\rho^\circ_{x},\mathtt{A1}} \subseteq \twoheadrightarrow_{\mathsf{K}} \Rightarrow_{\rho\delta} \twoheadrightarrow_{\mathsf{K}} \subseteq \twoheadrightarrow_{\rho^\circ_{x},\mathtt{A1}}$ and the reflexive and transitive closure of $(\twoheadrightarrow_{\mathsf{K}} \Rightarrow_{\rho\delta} \twoheadrightarrow_{\mathsf{K}})$ is equal to $\twoheadrightarrow_{\rho^\circ_{x},\mathtt{A1}}$. Then confluence modulo $\mathtt{A1}$ of $\rightarrow_{\rho^\circ_{x},\mathtt{A1}}$ is equivalent to confluence modulo $\mathtt{A1}$ of $\twoheadrightarrow_{\mathsf{K}} \Rightarrow_{\rho\delta} \twoheadrightarrow_{\mathsf{K}}$. Lemma 5.29 concludes the proof. □

Theorem 5.31 *The linear ρ°_x is confluent (the relation $\rightarrow_{\rho^\circ_{x}/A1}$ is confluent).*

Proof. The property follows from the previous lemma and the coherence of $\rightarrow_{\rho^\circ_{x},\mathtt{A1}}$ modulo $\sim_{\mathtt{A1}}$ [Ohl98]. □

5.5 Implementation in the Tom language

Explicit substitution calculi have been studied from different points of view. Different calculi have been proposed so as to obtain meta properties like confluence and preservation of strong normalization [Les94, Ros96, CHL96, DG01]. They have been used to perform higher-order unification [DHK00] just like to represent incomplete proofs in type theory [Muñ97] or to prove correctness of compiler optimizations [Ler02]. Moreover, explicit substitution calculi have been used in two significant practical systems [Nad02]: the Standard ML of New Jersey compiler and the Teyjus implementation of Lambda Prolog. In particular the use of de Bruijn indices and the ability to merge together structure traversals (*i.e.* the composition of substitutions) have a strong positive impact on the system performances [LN04].

The study of explicit ρ-calculi is thus also important for future implementations of languages based on the ρ-calculus. Actually, the ρ-calculus provides the basis for a language combining functional languages and rewrite based languages (Elan, Maude ...) features and thus besides provides a toy interpreter allowing us to experiment with the ρ-calculus. Our implementation of explicit ρ-calculi is a first step toward such a language. Moreover, the ρ-calculus provides proof terms for the proof assistants based on superdeduction [BHK07, Wac05] such as Lemuridae.

The rest of the section is devoted to the brief presentation of our implementation and of the support language, Tom. The gap between the different calculi defined in the

previous sections and the actual implementation is rather small and so, we mainly focus here on the key features of Tom that led to a natural implementation of the explicit ρ-calculi.

Following the experience of Elan, a strategic rewriting based language [BKK+98b, KK04], the Tom language extends Java with the purpose of providing high level constructs inspired by the rewriting community such as powerful matching capabilities along with a rich strategy language inspired by Elan and Stratego [Vis01a]. A full presentation of Tom[2] is out of the scope of this chapter but we refer to [BBK+06] for a detailed presentation.

First, Tom performs associative and unitary matching (unlike Elan that performs associative and commutative matching) also known as list matching. Constraints and substitutions are represented by lists of atomic matching equations (built using the operator "$\ll$") and respectively equations (built using the operator " $=$ ").

Secondly, the overall evaluation process of a term with respect to a rewrite system and a given strategy can be implemented straightforwardly in Tom since one can define separately the rewrite system and the strategy to evaluate it. This is a good way to obtain a modular and easily maintainable implementation.

Thirdly, thanks to an intensive use of the Tom component called Gom [Rei06] that automatically generates tree-like data-structures (following [BMV05] and [BJKO00]), we obtain for free the corresponding maximal sharing representation of terms for a given signature. Maximal sharing has strong impact on performances. For example, we can test the equality of terms in constant time (check the equality of memory addresses).

Lastly, an elaborated parser was easily obtained by the combination of Tom and a second generation parser generator such as Antlr : the parser generator is used for parsing, Tom is used for term construction.

The implementation of the operational semantics of the ρ°_{x} follows the overall evaluation process given earlier. All previously given examples can be simulated by the implementation. Our interpreter is available on the ρ-calculus web page[3].

The approach followed here is to give a simple but trustable interpreter for the ρ-calculus. There are other implementations of the ρ-calculus but that are quite different.

iRho It is an interpreter an imperative version of the ρ-calculus [LS04]. The approach is quite different from the one presented in this chapter since the final objective is to make an agile programming language. The interesting point is that the core of this implementation (evaluator) is certified using the proof assistant Coq [Coq07].

Rogue It is an imperative rewriting language that extends the untyped ρ-calculus with mutable expression attributes. The approach is quite different since the main objective is to implement efficient decision procedures [SDK+03]. This leads to the development of a calculus of explicit binding [SS04] in which the arrow operator of the ρ-calculus

[2] `http://tom.loria.fr`
[3] `http://rho.loria.fr/implementations.html`

is analyzed into two operators: a scoping operator alpha, which binds a variable in an expression; and a weaker form of arrow showing how to transform expressions matching a pattern. This is a particular case of dynamic patterns described in Chapter 6.

5.6 Explicit substitutions with generalized de Bruijn indices in the ρ-calculus

Until now, we do not deal explicitly with α-conversion. There are many works that should be considered: deBruijn [Bru78] indices, the ʎ-calculus with explicit scoping [HO03], director strings [Sin05] etc. These works should be adapted in the ρ-calculus since the arrow operator may bind more than one variable.

In this section, we set the main ideas for a ρ-calculus with de Bruijn indices and explicit substitutions. We first recall the framework of explicit substitutions with de Bruijn indices in the λ-calculus.

5.6.1 Explicit substitution with de Bruijn indices in the λ-calculus

In the λ-calculus, the substitution application is performed modulo α-conversion in order to avoid captures of variables when substitutions go through λ-abstractions. This means that each β-step requires an explicit treatment in order to rename variables. This operation highly influences the performance of the implementation and should thus be studied with care.

The λ-calculus with de Bruijn indices is a formalism that replaces each variable of the λ-calculus by an integer representing the number of λ-abstractions from this variable to its binder. Examples are given in the following table.

λ-terms	λ-terms with de Bruijn indices
$\lambda x.\, x$	$\lambda 1$
$\lambda x.\, \lambda y.\, x$	$\lambda\lambda 2$
$(\lambda x.\, \lambda y.\, x)\, (\lambda x.\, x)$	$(\lambda\lambda 2)\, (\lambda 1)$

The notion of substitution and substitution application should be adapted in the context of de Bruijn indices. When reducing a β-redex $(\lambda A)\, B$, the free indices (variables) of A should be increased by one to take into account that one λ-abstraction has been removed. This is for example the case in

$$\begin{array}{ll} & \lambda x.\, (\lambda y.\, x)\, x \\ \to & \lambda x.\, x \end{array} \qquad\qquad \begin{array}{ll} & \lambda(\lambda 2)\, 1 \\ \to & \lambda 1 \end{array}$$

Moreover, when applying a substitution $\{x \leftarrow B\}$ to an abstraction, the indices of B must be increased by one to take into account the extra λ-abstraction that is crossed by the substitution. This is for example the case in the following reduction

$$\begin{array}{rl} & \lambda z_1.\,\lambda z_2.\,(\lambda x_1.\,\lambda x_2.\,x_1)\,z_1 \\ \to & \lambda z_1.\,\lambda z_2.\,\lambda x_2.\,z_1 \end{array} \qquad \begin{array}{rl} & \lambda\lambda(\lambda\lambda 2)\,2 \\ \to & \lambda\lambda\lambda 3 \end{array}$$

The definition of substitution is thus based on the fundamental *shift* operation, denoted ↑, that increases by one all the indices.

To make explicit the substitution application in a λ-calculus with de Bruijn indices, the λσ-calculus was introduced [ACCL91, CHL96]. The substitutions are no longer composed of pairs of indices (variable) and terms but are equivalently represented by a list of terms where the position in the list corresponds to the indices. The symbol ∘ is used to represent substitution composition and the symbol `id` to represent the identity which is nothing but the sequence of de Bruijn indices $(n)_{n\geq 1}$. The shift substitution is interpreted as the sequence of de Bruijn indices $(n+1)_{n\geq 1}$. For example, in this framework we represent the substitution $\{A/1, B/2\}$ by the substitution $[A.B.\mathtt{id}]$. The (`Beta`) rule which reduces a redex into an explicit substitutions application is described as follows

$$(\mathtt{Beta}) \qquad (\lambda A)\,B \;\to\; A[B.\mathtt{id}]$$

For example, the following β-reduction sequence

$$\begin{array}{rl} & (\lambda x.\,\lambda y.\,x)\,(\lambda x.\,x) \\ \to & \lambda y.\,\lambda z.\,z \end{array}$$

is represented in the λσ-calculus by

$$\begin{array}{rl} & (\lambda\lambda 2)\,(\lambda 1) \\ \twoheadrightarrow & (\lambda 2)\,[\lambda 1.\mathtt{id}] \\ \twoheadrightarrow & \lambda(2[1.(\lambda 1.\mathtt{id})\circ \uparrow] \\ \twoheadrightarrow & \lambda(2[1.(\lambda 1).\uparrow] \\ \twoheadrightarrow & \lambda\lambda 1 \end{array}$$

where the first step reduces the redex into the explicit substitution application, the second one describes how a substitution is updated when traversing a λ-abstraction (often named the *lift* operation), the third one simplifies the composition of the two substitutions and finally the last step replaces the indice 2 by its corresponding term.

The precise rules of the λσ-calculus can be found, for example, in [CHL96] and a general survey on explicit substitution calculi can be found in [Kes07].

5.6.2 Explicit substitutions with generalized de Bruijn indices in the ρ-calculus

H. Cirstea proposed to adapt [Cir00] these notions to the ρ-calculus. The main difficulty is that the binders can bind more than one variable. If the notion of substitution remains

unchanged w.r.t. the λ-calculus, then when traversing a pattern-abstraction with a pattern which contains n variables, we should lift the substitution n times.

We sketch here an approach that adapts the notion of substitutions to the binder of the ρ-calculus : substitutions become lists of lists of terms. The position in this list indicates the number associated to the binder (as in the λ-calculus) and the inner lists are used to indicate the substitution for each of the indices bound by this abstraction. Then a bound indice in the body of an abstraction must mention first a position to indicate the binder and another position to indicate the corresponding variables of the binder. Note that this is a generalization of the notion of substitution of the λ-calculus (substitutions as lists of terms) since if the abstraction binds only one variable then all the substitutions are list of singletons and thus, they are in a one-to-one correspondence with list of terms.

For example, consider the ρ-rewrite sequence

$$\begin{array}{ll} & \big(f(x,y) \twoheadrightarrow (g(z,t) \twoheadrightarrow h(x,y,z,t))\, g(x,a)\big)\, f(b,c) \\ \to & (g(z,t) \twoheadrightarrow h(b,c,z,t))\, g(b,a) \\ \to & h(b,c,b,a) \end{array}$$

The initial term can be encoded using generalized de Bruijn indices by

$$\big(f(1,2) \twoheadrightarrow (g(1,2) \twoheadrightarrow h(2_1,2_2,1_1,1_2))\, g(1_1,a)\big)\, f(b,c)$$

where 1_2 indicates that we refer to the second variable of the outer binder. This should be evaluated as follows

$$\begin{array}{ll} & \big(f(1,2) \twoheadrightarrow (g(1,2) \twoheadrightarrow h(2_1,2_2,1_1,1_2))\, g(1_1,a)\big)\, f(b,c) \\ \overset{(1)}{\to\!\!\!\!\to} & (g(1,2) \twoheadrightarrow h(2_1,2_2,1_1,1_2))\, g(1_1,a)\big)[[b,c].\mathtt{id}] \\ \overset{(2)}{\to\!\!\!\!\to} & (g(1,2) \twoheadrightarrow h(2_1,2_2,1_1,1_2))[[b,c].\mathtt{id}]\, g(1_1,a)[[b,c].\mathtt{id}]\big) \\ \overset{(3)}{\to\!\!\!\!\to} & (g(1,2) \twoheadrightarrow h(2_1,2_2,1_1,1_2)[[\mathtt{id}].[[b,c].\mathtt{id}]\circ\uparrow)]\, g(1_1,a)[[b,c].\mathtt{id}]\big) \\ \overset{(4)}{\to\!\!\!\!\to} & (g(1,2) \twoheadrightarrow h(2_1,2_2,1_1,1_2)[[\mathtt{id}].[[b,c].\mathtt{id}]\circ\uparrow)]\, g(b,a)\big) \\ \overset{(5)}{\to\!\!\!\!\to} & (g(1,2) \twoheadrightarrow h(2_1,2_2,1_1,1_2)[[\mathtt{id}].[b,c].\uparrow)]\, g(b,a)\big) \\ \overset{(6)}{\to\!\!\!\!\to} & (g(1,2) \twoheadrightarrow h(b,c,1_1,1_2)\, g(b,a)\big) \\ \overset{(7)}{\to\!\!\!\!\to} & h(b,c,1_1,1_2)\, [[b,a].\mathtt{id}] \\ \overset{(8)}{\to\!\!\!\!\to} & h(b,c,b,a) \end{array}$$

where the step labelled (1) reduces the outer redex to the explicit substitution application, the step (2) propagates the substitution over the application, the step (3) propagate the substitution over the abstraction by doing a single lift operation, the step (5) simplifies the composition of the two substitutions. The other steps are explicit substitution applications.

Thanks to the generalized de Bruijn indices and substitutions, only one lift operation is needed when traversing an abstraction.

We conjecture that the full theory of the explicit substitution calculus that we sketched in this section is quite of interest and has the expected properties (such as the Church-Rosser property).

Conclusion and future works

We have proposed a ρ-calculus that handles explicitly the (syntactic) matching constraints and the application of the resulted substitutions. We have proved that it enjoys the classical properties of such a formalism, *i.e.*, confluence of the calculus and the termination and confluence of the constraint handling part. We have seen that the calculus is modular and can be adapted to other matching theories for the defined constants and for the structure operator "ι". We have shown that we can either choose to be atomic and give a simple definition of substitutions (as in ρ_{s}), or more general and efficient and define a calculus that handles substitution composition (as in ρ°_{x}).

ρ-calculi and especially the ρ°_{x}, are new frameworks that provide us with theoretical foundations for a new family of programming languages. Different extensions/variations of the ρ-calculus are now available: in [LS04] an imperative version of the calculus has been proposed and in [FK02] exceptions in the ρ-calculus were studied. The implementation briefly described here can be seen as the basis for programming language incorporating the features introduced in these different extensions.

Related work: In [BKK98a], a calculus called the PSA-Calculus was introduced. The explicit application of a rewrite rule and the *explicit matching handling* were coined for the first time in this ancestor of the ρ-calculus. Nevertheless, it was a first approach to make explicit rewriting and thus less powerful than the current ρ-calculus. For example, the PSA-Calculus is not powerful enough to express strategies as explicit objects and thus there is a hierarchy between rules and strategies.

A rewriting calculus with explicit substitutions has been already proposed in [Cir00]. This calculus is mainly an extension of the λσ-calculus and is called the ρσ-calculus. The approach is less general than the one presented here since this calculus makes explicit the substitution application but not the computations to go from constraints to substitutions. In [NKK02], the cooperation between Coq and Elan that automates the use of AC-rewriting is the proof assistant makes an intensive usage of the ρσ-calculus to represent proof terms of rewrite derivations. The explicit treatment of matching in the ρ°_{x} should be a useful tool to obtain normalization traces in some non-trivial matching theories.

Our work follows the line of the works on explicit substitution calculi and more generally on the way to represent higher-order languages [Bru72, BF82, Ros96, Pag98]. This chapter shows that these works need to be extended to deal with the interaction of matching and substitution. Even the works on generic calculi of explicit substitutions [Ste00] cannot be used since, to the best of our knowledge, they do not handle composition of substitutions. Of course, if such works would be extended, the embedding of our calculus should be of interest.

Future work: There are different points that should be studied. Namely, we should understand how the approach proposed in [KL05, Kes07] and, in particular, the permutation of substitutions can be applied in order to simplify the notion of substitution for the ρ-calculus. Moreover, taking advantage of the very general management of errors we should propose a powerful named exception mechanism.

Finally, we should note that that confluence of the ρ_x° was proved by splitting the set of evaluation rules as follows: the ρ and δ rules, the rules dealing explicitly with matching and the rules dealing with explicit substitutions. The set of matching rules deals with first-order linear structure patterns while the two other set of rules are independent from the matching theory (and from the set of patterns). A more general approach would have been to study all the properties independently of the set of patterns and of the matching algorithm. This is what we do in the following chapter for a plain ρ-calculus with dynamic patterns.

Chapter 6

Confluence of pattern-based λ-calculi

Context There are several formalisms that address the integration of pattern matching capabilities with the λ-calculus; we can mention the λ-calculus with patterns [Oos90, KOV07], the rewriting calculus [CK01], the pure pattern calculus [JK06] and the λ-calculus with constructors [AMR06].

Each of these pattern-based calculi differs on the way patterns are defined and on the way pattern-abstractions are applied. Thus, patterns can be simple variables like in the λ-calculus, algebraic terms like in the algebraic ρ-calculus [CLW03], special (static) patterns that satisfy certain (semantic or syntactic) conditions like in the λ-calculus with patterns or dynamic patterns that can be instantiated and possibly reduced like in the pure pattern calculus and some versions of the ρ-calculus. The underlying matching theory strongly depends on the form of the patterns and can be syntactic, equational or more sophisticated [JK06, BCKL03].

Although some of these calculi just extend the λ-calculus by pattern-abstractions instead of variable-abstractions the confluence of these formalisms is lost when no restrictions are imposed.

Several approaches are then used to recover confluence. One of these techniques consists in syntactically restricting the set of patterns and then showing that the reduction relation is confluent for the chosen subset. This is done for example in the λ-calculus with patterns and in the ρ-calculus (with algebraic patterns). The second technique considers a restriction of the initial reduction relation (that is, a strategy) to guarantee that the calculus is confluent on the whole set of terms. This is done for example in the pure pattern calculus where the matching algorithm is a partial function (whereas any term is a pattern).

Nevertheless we can notice that in practice, the proof methods share the same structure and that each variation on the way pattern-abstractions are applied needs another proof of confluence. There is thus a need for a more abstract and more modular approach in the same spirit as in [Mel02, Kes00]. A possible way to have a unified approach for proving confluence is the application of the general and powerful results on confluence of higher-order rewrite systems [KOR93, MN98, Ter03]. Although these results have already been applied for some particular pattern-calculi [BK07] the encoding seems to be rather complex for some calculi and in particular for the general setting proposed here. Moreover, it would be interesting to have a framework where the expressiveness and (confluence) properties of the different pattern calculi can be compared.

Contributions We show that all the pattern-based calculi can be expressed in a general calculus parameterized by a function that defines the underlying matching algorithm and thus the way pattern abstractions are applied. This function can be instantiated (implemented) by a unitary matching algorithm as in [CK01, JK06] but also by an anti-pattern matching algorithm [KKM07] or it can be even more general [LHL07]. We propose a generic confluence proof where the way pattern abstractions are applied is axiomatized. Intuitively, the sufficient conditions to ensure confluence guarantee that the (matching) function is stable by substitution and by reduction.

We apply our approach to several classical pattern calculi, namely the λ-calculus with patterns, the pure pattern calculus, the rewriting calculus and the λ-calculus with constructors. For all these calculi, we give the encodings in the general framework and we obtain proofs of confluence. This approach does not provide confluence proofs for free but it establishes a proof methodology and isolates the key points that make the calculi confluent. It can also point out some matching algorithms that although natural at the first sight can lead to non confluent reduction in the corresponding calculus.

Outline of the chapter In Section 6.1 we give the syntax and semantics of the dynamic pattern λ-calculus. The hypotheses under which the calculus is confluent and the main theorems are stated and proved in Section 6.2. A non-confluent calculus is given at the end of this section. In Section 6.3 we give the encoding of different pattern-calculi and the corresponding confluence proofs. Note that the sections 6.2 and 6.3 are fairly independent (except for the confluence proofs) and can be read in any order.

6.1 The dynamic pattern λ-calculus

In this section, we first define the syntax and the operational semantics of the core dynamic pattern λ-calculus. We then give the general definition of the dynamic pattern λ-calculus.

6.1.1 Syntax

The syntax of the core dynamic pattern λ-calculus is defined in Figure 6.1. We use the syntactical conventions of the previous chapters. In particular, variables are denoted by $x, y, z, \ldots$, constants by $a, b, c, d, e, f, \ldots$ and we sometimes use an algebraic notation $f(A_1, \ldots, A_n)$ for the term $((f\,A_1)\ldots)\,A_n)$. In an abstraction $A \twoheadrightarrow_\theta B$, the set θ is a subset of the set of variables of A and represents the set of variables bound by the abstraction. This set is often omitted when it is exactly the set of free variables of the pattern.

Comparing to the λ-calculus we abstract thus not only on variables but on general terms and the set of variables bound by an abstraction is not necessarily the same as the set of (free) variables of the corresponding pattern. We say thus that the patterns are dynamic since they can be instantiated and possibly reduced.

Since the binders are somehow unusual, we give precisely the definition of the free and bound variables.

$$
\begin{array}{rcll}
A, B & ::= & x & \text{(Variable)} \\
& | & c & \text{(Constant)} \\
& | & A \twoheadrightarrow_\theta B & \text{(Abstraction)} \\
& | & A\ B & \text{(Application)}
\end{array}
$$

Figure 6.1: Syntax of the (core) dynamic pattern λ-calculus

Definition 6.1 (Free and bound variables) *The set of free and bound variables of a term A, denoted $\mathtt{fv}(A)$ and $\mathtt{bv}(A)$, are defined inductively by:*

$$
\begin{array}{llll}
\mathtt{fv}(c) \triangleq \emptyset & \mathtt{bv}(c) \triangleq \emptyset & \mathtt{fv}(x) \triangleq \{x\} & \mathtt{bv}(x) \triangleq \emptyset \\
\mathtt{fv}(A\,B) \triangleq \mathtt{fv}(A) \cup \mathtt{fv}(B) & & \mathtt{fv}(A \twoheadrightarrow_\theta B) \triangleq (\mathtt{fv}(A) \cup \mathtt{fv}(B)) \backslash \theta & \\
\mathtt{bv}(A\,B) \triangleq \mathtt{bv}(A) \cup \mathtt{bv}(B) & & \mathtt{bv}(A \twoheadrightarrow_\theta B) \triangleq \mathtt{bv}(A) \cup \mathtt{bv}(B) \cup \theta &
\end{array}
$$

Open and closed terms, substitutions and substitution application are defined as usually (see for example Chapter 4). Also, we work modulo α-conversion and as before α-equivalence is denoted by $\equiv$.

6.1.2 Operational semantics

The operational semantics of the core dynamic pattern λ-calculus is given by a single reduction rule that defines the way pattern-abstractions are applied. This rule, given in Figure 6.2, is parameterized by a partial function, denoted $\mathcal{S}\mathtt{ol}(A \prec\!\!\!\prec_\theta B)$, that takes as parameters two terms A and B and a set θ of variables and returns a substitution.

Different instances of the core dynamic pattern λ-calculus are obtained when we give concrete definitions to $\mathcal{S}\mathtt{ol}$. For example, the λ-calculus can be seen as the core dynamic pattern λ-calculus such that

$$\sigma = \mathcal{S}\mathtt{ol}(A \prec\!\!\!\prec_\theta C) \text{ iff } A \text{ is a variable } x,\ \theta = \{x\} \text{ and } A\sigma \equiv C$$

Example 6.2 (Syntactic matching) *Consider the rules to solve syntactic matching problem given in Section4.2.1.We can define $\mathcal{S}\mathtt{ol}(A \prec\!\!\!\prec_\theta B)$ as the function that normalizes the matching problem $A \ll^? B$ w.r.t. the above rewrite rules and according to the obtained result returns:*

- *nothing (i.e. is not defined) if $\mathtt{fv}(A) \neq \theta$ or if the result is $\mathbb{F}$,*
- *the substitution $\{x_i \leftarrow A_i\}_{i \in I}$ if the result is of the form $\bigwedge_{i \in I \neq \emptyset} x_i \ll^? A_i$,*
- *$\mathtt{id}$, if the result is empty.*

Example 6.3 (Case branchings) *Consider a pattern-based calculus with a case construct denoted using $|$ (a.k.a. a $\mathtt{match}$ operator as in functional programming languages). It can be encoded in the core dynamic pattern λ-calculus as follows:*

$$(A_1 \mapsto B_1 | \ldots | A_n \mapsto B_n)\, C \quad \triangleq \quad \big((A_1 \hookrightarrow B_1 / \ldots / A_n \hookrightarrow B_n) \twoheadrightarrow_x x\big)\, C$$

(ρ)	$(A \twoheadrightarrow_\theta B)C$	$\to$	$B\sigma$ where $\sigma = \mathcal{S}ol(A \lessdot_\theta C)$

Figure 6.2: Operational semantics of the core dynamic pattern λ-calculus

where x *is a fresh variable, the symbols* $\hookrightarrow$ *and* $/$ *are constants of the core dynamic pattern* λ*-calculus (infix notation) and the function* $\mathcal{S}ol$ *may be defined by*

$$\mathcal{S}ol\big((A_1 \hookrightarrow B_1/\ldots/A_n \hookrightarrow B_n) \lessdot_x A_i\sigma\big) \quad \triangleq \quad \{x \leftarrow B_i\sigma\}$$

Some pattern-calculi come with additional features and cannot be expressed as instances of the core dynamic pattern λ-calculus. For example, the pure pattern calculus [JK06] and some versions of the rewriting calculus [CLW03] reduce the application of a pattern-abstraction to a special term when the corresponding matching problem has not and will never have a solution. In the rewriting calculus [CLW03] there is also a construction that aggregates terms and then distributes them over applications.

We define thus the dynamic pattern λ-calculus as the core dynamic pattern λ-calculus extended by a set of rewrite rules. As for $\mathcal{S}ol$ this set, denoted ξ, is not made precise and can be considered as a parameter of the calculus. It can include for example some rules to reduce particular pattern-abstractions to a special constant representing a definitive matching failure or some extra rules describing the distributivity of certain symbols (like the structure operator of the ρ-calculus) over the applications.

In what follows, $\to_\rho$ denotes the compatible closure of the relation induced by the rule ρ and $\twoheadrightarrow_\rho$ denotes the transitive closure of $\to_\rho$. Similarly, we will denote by $\to_{\rho\cup\xi}$ the compatible closure of the relation induced by the rules ρ and ξ. $\twoheadrightarrow_{\rho\cup\xi}$ denotes the transitive closure of $\to_{\rho\cup\xi}$.

6.2 Confluence of the dynamic pattern λ-calculus

The calculus is not confluent when no restrictions are imposed on the function $\mathcal{S}ol$. This is for example the case when we consider the decomposition of applications containing free active variables (the term $(x\ a \twoheadrightarrow_x x)\ ((y \twoheadrightarrow_y y)\ a)$ can be reduced either to $y \twoheadrightarrow_y y$ or to $(x\ a \twoheadrightarrow_x x)\ a$ which are not joinable) or when we deal with non-linear patterns (see Section 4.3.3). Nevertheless, confluence is recovered for some specific definitions of $\mathcal{S}ol$ like the one used in Section 6.1.2 when defining the λ-calculus as a core dynamic pattern λ-calculus.

In this section we give some sufficient conditions that guarantee the confluence of the core dynamic pattern λ-calculus. Intuitively, the hypotheses introduced in Section 6.2.2 under which we prove confluence of the calculus guarantee the coherence between $\mathcal{S}ol$ and the underlying relation of the calculus. The obtained results can then be generalized for a dynamic pattern λ-calculus with an extended set of rules ξ that satisfies some classical coherence conditions.

$$\frac{}{A \Rightarrow_\rho A} \qquad \frac{A \Rightarrow_\rho A' \quad B \Rightarrow_\rho B'}{AB \Rightarrow_\rho A'B'} \qquad \frac{A \Rightarrow_\rho A' \quad B \Rightarrow_\rho B'}{(A \twoheadrightarrow_\theta B) \Rightarrow_\rho (A' \twoheadrightarrow_\theta B')}$$

$$\frac{A \Rightarrow_\rho A' \quad B \Rightarrow_\rho B' \quad C \Rightarrow_\rho C'}{(A \twoheadrightarrow_\theta B)C \Rightarrow_\rho B'\sigma'} \text{ IF } \sigma' \in \mathcal{S}ol(A' \prec_\theta C')$$

Figure 6.3: Parallel reduction for the dynamic pattern λ-calculus

6.2.1 The parallel reduction

We use here a proof method introduced by Martin-Löf that consists in defining a so-called parallel reduction that, intuitively, can reduce all the redexes initially present in the term and that is strongly confluent (even if the one-step reduction is not) under some hypotheses. A detailed discussion on parallel reductions in the λ-calculus has been given in Chapter 1.

Definition 6.4 (Parallel reduction) *The parallel reduction is inductively defined on the set of terms in Figure 6.3 .*

Note that the parallel reduction is compatible.

Remark 6.5 (Parallel reduction and developments) *The given definition of parallel reduction does not coincide with the classical notion of developments. For example, if we use a $\mathcal{S}ol$ function that computes the substitution solving the matching between its two arguments (for example, like in Example 6.2 but without using the last rule) then we have*

$$\frac{f\,x \Rightarrow_\rho f\,x \quad x \Rightarrow_\rho x \quad (y \twoheadrightarrow f\,y)\,a \Rightarrow_\rho f\,a}{((f\,x) \twoheadrightarrow x)((y \twoheadrightarrow f\,y)\,a) \Rightarrow_\rho a}$$

The substitution $\{x \leftarrow a\}$ solves the syntactic matching between the terms $f\,x$ and $f\,a$ and thus, even if the initial term contains no head redex (because the argument $(y \twoheadrightarrow f\,y)\,a$ must be reduced before) it can still be reduced using the parallel reduction.

We extend the definition of parallel reduction to substitutions having the same domain by setting $\sigma \Rightarrow_\rho \sigma'$ as soon as for all x in the domain of σ, we have $x\sigma \Rightarrow_\rho x\sigma'$.

6.2.2 Stability of $\mathcal{S}ol$

Preservation of free variables First of all, when defining a higher-order calculus it is natural to ask that the set of free variables is preserved by reduction (some free variables can be lost but no free variables can appear during reduction). For example, the free variables of the term $(A \twoheadrightarrow_\theta B)C$ should be include the ones of the term $B\sigma$ with $\sigma = \mathcal{S}ol(A \prec_\theta C)$. Thus, the substitution σ should instantiate all the variables bound

(by the abstraction) in B, that is, all the variables in θ. Moreover, the free variables of σ should already be present in C or free in A. These conditions are enforced by the hypothesis $\mathcal{H}_0$ in Figure 6.4.

If we think of $\mathcal{S}ol$ as a unitary matching algorithm, the examples that do not verify $\mathcal{H}_0$ are often peculiar algorithms (for example the function that returns the substitution $\{x \leftarrow y\}$ for any problem). When considering non-unitary matching (not handled here), there are several examples that do not verify $\mathcal{H}_0$. For instance, the algorithms solving higher-order matching problems or matching problems in non-regular theories (*e.g.*, such that $x \times 0 = 0$) do not verify $\mathcal{H}_0$.

Stability by substitution In the core dynamic pattern λ-calculus, when a pattern-abstraction is applied the argument may be open. One can wait for the argument to be instantiated and only then compute the corresponding substitution (if it exists) and reduce the application. On the other hand, one might not want to sequentialize the reduction but to perform the reduction as soon as possible. Nevertheless, the same result should be obtained for both reduction strategies. This is enforced by the hypothesis $\mathcal{H}_1$ in Figure 6.4.

If we consider that $\mathcal{S}ol$ performs a naive matching algorithm that does not take into account the variables in θ and such that $\mathcal{S}ol(a \prec_\emptyset b)$ has no solution and $\mathcal{S}ol(x \prec_\emptyset y) = \{x \leftarrow y\}$, then the hypothesis $\mathcal{H}_1$ is clearly not verified (take $\tau = \{x \leftarrow a, y \leftarrow b\}$).

Stability by reduction When applying a pattern-abstraction, the argument may also be not fully reduced. Once again, if $\mathcal{S}ol$ succeeds and produces a substitution σ then subsequent reductions should not lead to a definitive failure for $\mathcal{S}ol$. Moreover, the substitution that is eventually obtained should be derivable from σ. This is formally defined in hypothesis $\mathcal{H}_2$.

The function $\mathcal{S}ol$ proposed in Example 6.2 does not satisfy this hypothesis. If we take $I \triangleq (y \twoheadrightarrow y)$ then $\mathcal{S}ol(f(x,x) \prec_x f(I\,I, I\,I)) = \{x \leftarrow I\,I\}$ but $\mathcal{S}ol(f(x,x) \prec_x f(I\,I, I))$ has no solution. Similarly, this hypothesis is not satisfied by a matching algorithm that allows the decomposition of applications containing a so-called free *active* variable (*i.e.* a variable in applicative position). For example, we can have $\mathcal{S}ol(x\,a \prec_x (y \twoheadrightarrow y)a) = \{x \leftarrow y \twoheadrightarrow y\}$ but $\mathcal{S}ol(x\,a \prec_x a)$ has no solution for any classical (first-order) matching algorithm.

6.2.3 Confluence of the dynamic pattern λ-calculus

In this section, we prove confluence of the dynamic pattern λ-calculus under the hypotheses $\mathcal{H}_0, \mathcal{H}_1, \mathcal{H}_2$. The proof uses the standard techniques of parallel reduction first introduced by Tait and Martin-Löf. We first show that the reflexive and transitive closure of the parallel and one-step reductions are the same. Then we show that the parallel reduction has the diamond property and we deduce confluence of the one-step reduction.

The three hypotheses given in the previous section are used for showing the strong confluence of the parallel reduction and in particular the Lemma 6.7. On the other hand,

$\forall A, C, A', C'$

$$\mathcal{H}_0: \quad \mathcal{S}ol(A \prec\!\!\!\prec_\theta C) = \sigma \quad \text{implies} \quad \begin{cases} \mathcal{D}om(\sigma) = \theta \\ \mathcal{R}an(\sigma) \subseteq \mathtt{fv}(C) \cup (\mathtt{fv}(A) \backslash \theta) \end{cases}$$

$$\mathcal{H}_1: \quad \mathcal{S}ol(A \prec\!\!\!\prec_\theta C) = \sigma \quad \text{implies} \quad \begin{cases} \forall \tau \text{ s.t. } \mathcal{V}ar(\tau) \cap \theta = \emptyset, \text{ we have} \\ \mathcal{S}ol(A\tau \prec\!\!\!\prec_\theta C\tau) = (\tau \circ \sigma)_{|\theta} \end{cases}$$

$$\mathcal{H}_2: \quad \begin{cases} \mathcal{S}ol(A \prec\!\!\!\prec_\theta C) = \sigma \\ A \Rightarrow_\rho A' \quad C \Rightarrow_\rho C' \end{cases} \quad \text{implies} \quad \begin{cases} \mathcal{S}ol(A' \prec\!\!\!\prec C') = \sigma' \\ \sigma \Rightarrow_\rho \sigma' \end{cases}$$

Figure 6.4: Conditions to ensure confluence of the core dynamic pattern λ-calculus

we can show that the reflexive and transitive closure of $\Rightarrow_\rho$ is equal to $\twoheadrightarrow_\rho$ independently of the properties of $\mathcal{S}ol$.

Lemma 6.6 *The following inclusions holds:* $\rightarrow_\rho \subseteq \Rightarrow_\rho \subseteq \twoheadrightarrow_\rho$.

Proof. For the first inclusion suppose that the reduction $A \rightarrow B$ occurs at the head position. This means that $A \equiv (A_1 \rightarrowtriangle_\theta A_2)A_3$ and $B \equiv A_2\sigma$ where $\sigma = \mathcal{S}ol(A_1 \prec\!\!\!\prec_\theta A_3)$. Then we trivially have $A \Rightarrow_\rho B$ since the relation $\Rightarrow_\rho$ is reflexive. In the other cases (the reduction does not occur at the head position), the proof directly follows from the compatibility of the relation $\Rightarrow_\rho$.

For the second inclusion, we prove by induction that if $A \Rightarrow_\rho B$ then $A \twoheadrightarrow B$.

- If $A \equiv B$ then the result follows since the relation $\twoheadrightarrow$ is reflexive.
- If $A \equiv A_1A_2$ and $B \equiv B_1B_2$ with $A_1 \Rightarrow_\rho B_1$ and $A_2 \Rightarrow_\rho B_2$ then by induction hypothesis applied to A_1 and A_2 we have, $A_1 \twoheadrightarrow B_1$ and $A_2 \twoheadrightarrow B_2$. We conclude that $A_1A_2 \twoheadrightarrow B_1B_2$.
- If $A \equiv A_1 \rightarrowtriangle_\theta A_2$ and $B \equiv B_1 \rightarrowtriangle_\theta B_2$ with $A_1 \Rightarrow_\rho B_1$ and $A_2 \Rightarrow_\rho B_2$ we proceed as in the previous case.
- If $A \equiv (A_1 \rightarrowtriangle_\theta A_2)A_3$ and $B \equiv B_2\sigma$ with $A_1 \Rightarrow_\rho B_1$ and $A_2 \Rightarrow_\rho B_2$ and $A_3 \Rightarrow_\rho B_3$ and $\sigma = \mathcal{S}ol(B_1 \prec\!\!\!\prec_\theta B_3)$ then by induction hypothesis, we have

  $$A_1 \twoheadrightarrow B_1 \text{ and } A_2 \twoheadrightarrow B_2 \text{ and } A_3 \twoheadrightarrow B_3$$

  from which we conclude

  $$\begin{array}{ll} & (A_1 \rightarrowtriangle_\theta A_2)\, A_3 \\ \twoheadrightarrow & (B_1 \rightarrowtriangle_\theta B_2)\, B_3 \\ \rightarrow & B_2\sigma. \end{array}$$

□

The proof of the diamond property of the parallel reduction relies on the following lemma:

Lemma 6.7 (Fundamental lemma) *For all terms* C *and* C' *and for all substitutions* σ *and* σ'*,such that* $C \Rightarrow_\rho C'$ *and* $\sigma \Rightarrow_\rho \sigma'$ *we have* $C\sigma \Rightarrow_\rho C'\sigma'$.

Proof. The proof is by induction.

- If $C \equiv C'$ then the result follows by $\mathcal{H}_2$ using a trivial induction on C.
- If $C \equiv C_1C_2$ and $C' \equiv C'_1C'_2$ with $C_1 \Rightarrow_\rho C'_1$ and $C_2 \Rightarrow_\rho C'_2$. By induction hypothesis applied to C_1 and C_2, we have

  $$C_1\sigma \Rightarrow_\rho C'_1\sigma' \text{ and } C_2\sigma \Rightarrow_\rho C'_2\sigma'$$

  and then we have

  $$(C_1C_2)\sigma \Rightarrow_\rho (C'_1C'_2)\sigma'$$

  which means that $C\sigma \Rightarrow_\rho C'\sigma'$. This concludes the case.
- If $C \equiv (C_1 \rightarrowtriangle_\theta C_2)$ and $C' \equiv (C'_1 \rightarrowtriangle_\theta C'_2)$ with $C_1 \Rightarrow_\rho C'_1$ and $C_2 \Rightarrow_\rho C'_2$. The case is similar to the previous one. In fact, by induction hypothesis applied to C_1 and C_2, we have $C_1\sigma \Rightarrow_\rho C'_1\sigma'$ and $C_2\sigma \Rightarrow_\rho C'_2\sigma'$ and then we have $(C_1 \rightarrowtriangle_\theta C_2)\sigma \Rightarrow_\rho (C'_1 \rightarrowtriangle_\theta C'_2)\sigma'$ which means that $C\sigma \Rightarrow_\rho C'\sigma'$. This concludes the case.
- If $C \equiv (C_1 \rightarrowtriangle_{\theta'} C_2)C_3$ and $C' \equiv C'_2\tau'$ with $C_1 \Rightarrow_\rho C'_1$ and $C_2 \Rightarrow_\rho C'_2$ and $C_3 \Rightarrow_\rho C'_3$ and $\tau' = \mathcal{S}ol(C'_1 \lessdot_{\theta'} C'_3)$. We want to prove that

  $$C\sigma \equiv (C_1\sigma \rightarrowtriangle_{\theta'} C_2\sigma)C_3\sigma \quad \Rightarrow_\rho \quad (C'_2\tau')\sigma' \equiv C'\sigma'$$

  By induction hypothesis applied to C_1, C_2 and C_3 we have

  $$C_1\sigma \Rightarrow_\rho C'_1\sigma' \quad \text{and} \quad C_2\sigma \Rightarrow_\rho C'_2\sigma' \quad \text{and} \quad C_3\sigma \Rightarrow_\rho C'_3\sigma'.$$

  Since by α-conversion we can assume that $\mathcal{D}om(\sigma') \cap \theta' = \emptyset$, we can apply $\mathcal{H}_1$ and we get $\mathcal{S}ol(C'_1\sigma' \lessdot_{\theta'} C'_3\sigma') = (\sigma' \circ \tau')_{\theta'}$. Then, we have

  $$(C_1\sigma \rightarrowtriangle_{\theta'} C_2\sigma)C_3\sigma \quad \Rightarrow_\rho \quad (C'_2\sigma')(\sigma' \circ \tau')_{|\theta'}$$

  So, to conclude the proof it remains to show that $\sigma' \circ \tau' \equiv (\sigma' \circ \tau')_{|\theta'} \circ \sigma'$. The proof is depicted as follows. There are two cases.

  1. If x belongs to θ' then

     $$\sigma' \circ \tau' \quad : \quad x \xrightarrow{\tau'} x\tau' \xrightarrow{\sigma'} x(\sigma' \circ \tau')$$

     $$(\sigma' \circ \tau')_{|\theta'} \circ \sigma' \quad : \quad x \xrightarrow[(1)]{\sigma'} x \xrightarrow{(\sigma' \circ \tau')_{|\theta'}} x(\sigma' \circ \tau')$$

2. If x does not belong to θ' then

$$\sigma' \circ \tau' \quad : \quad x \xrightarrow[(3)]{\tau'} x \xrightarrow{\sigma'} x\sigma'$$

$$(\sigma' \circ \tau')_{|\theta'} \circ \sigma' \quad : \quad x \xrightarrow{\sigma'} x\sigma' \xrightarrow[(2)]{(\sigma' \circ \tau')_{|\theta'}} x\sigma'$$

The steps labeled (1) and (2) are true since by α-conversion we can assume that $\mathcal{D}om(\sigma') \cap \theta' = \emptyset$ and $\mathcal{R}an(\sigma') \cap \theta' = \emptyset$. The step labeled (3) is true since by applying $\mathcal{H}_0$, we have $\mathcal{D}om(\tau') = \theta'$.

□

Lemma 6.8 (Diamond property for $\Rightarrow_\rho$) *The relation $\Rightarrow_\rho$ satisfies the diamond property that is, for all terms A, B and C if $A \Rightarrow_\rho B$ and $A \Rightarrow_\rho C$ then there exists a term D such that $B \Rightarrow_\rho D$ and $C \Rightarrow_\rho D$.*

Proof. The proof is by induction on the structure of A.

- If $A \equiv x$ or $A \equiv c$ then the result holds since we necessarily have $A \equiv B \equiv C$.
- If $A \equiv (A_1 \twoheadrightarrow_\theta A_2)$ then necessarily

  we have $B \equiv (B_1 \twoheadrightarrow_\theta B_2)$ with $A_1 \Rightarrow_\rho B_1$ and $A_2 \Rightarrow_\rho B_2$
  and $C \equiv (C_1 \twoheadrightarrow_\theta C_2)$ with $A_1 \Rightarrow_\rho C_1$ and $A_2 \Rightarrow_\rho C_2$.

  Applying the induction hypothesis to A_1 and to A_2 we get that there exist two terms D_1 and D_2 such that

  we have $B_1 \Rightarrow_\rho D_1$ and $C_1 \Rightarrow_\rho D_1$
  and $B_2 \Rightarrow_\rho D_2$ and $C_2 \Rightarrow_\rho D_2$.

  We conclude that

$$\begin{array}{rcl} & B \equiv (B_1 \twoheadrightarrow_\theta B_2) & \Rightarrow_\rho \quad (D_1 \twoheadrightarrow_\theta D_2) \equiv D \\ \text{and} & C \equiv (C_1 \twoheadrightarrow_\theta C_2) & \Rightarrow_\rho \quad (D_1 \twoheadrightarrow_\theta D_2) \equiv D. \end{array}$$

- If $A \equiv A_1 A_2$ then there are three cases depending on the last rules used to prove that $A \Rightarrow_\rho B$ and $A \Rightarrow_\rho C$.
  1. If $B \equiv B_1 B_2$ and $C \equiv C_1 C_2$ with $A_1 \Rightarrow_\rho B_1$ and $A_2 \Rightarrow_\rho B_2$ and $A_1 \Rightarrow_\rho C_1$ and $A_2 \Rightarrow_\rho C_2$ then the proof is similar to the previous case.
  2. If $A \equiv (A_1 \twoheadrightarrow_\theta A_2)A_3$ and $B \equiv B_2\sigma$ and $C \equiv C_2\sigma'$ with

$$\begin{cases} A_1 \Rightarrow_\rho B_1 \text{ and } A_1 \Rightarrow_\rho C_1 \\ A_2 \Rightarrow_\rho B_2 \text{ and } A_2 \Rightarrow_\rho C_2 \\ A_3 \Rightarrow_\rho B_3 \text{ and } A_3 \Rightarrow_\rho C_3 \end{cases} \quad \text{and} \quad \begin{cases} \mathcal{S}ol(B_1 \twoheadleftarrow_\theta B_3) = \sigma \\ \mathcal{S}ol(C_1 \twoheadleftarrow_\theta C_3) = \sigma' \end{cases}$$

then we apply the induction hypothesis on A_1, A_2 and A_3 and we get

$$\begin{array}{lll} B_1 \Rightarrow_\rho D_1 & \text{and} & C_1 \Rightarrow_\rho D_1 \\ B_2 \Rightarrow_\rho D_2 & \text{and} & C_2 \Rightarrow_\rho D_2 \\ B_3 \Rightarrow_\rho D_3 & \text{and} & C_3 \Rightarrow_\rho D_3 \end{array}$$

By applying $\mathcal{H}_2$ we have $\sigma'' = \mathcal{S}ol(D_1 \prec_\theta D_3)$ with $\sigma \Rightarrow_\rho \sigma''$. Then by applying Lemma 6.7 we conclude that $B_2\sigma \Rightarrow_\rho D_2\sigma''$ and $C_2\sigma \Rightarrow_\rho D_2\sigma''$.

3. The last case where $A \equiv (A_1 \twoheadrightarrow_\theta A_2)A_3$ and $B \equiv B_2\sigma$ and $C \equiv (C_1 \twoheadrightarrow_\theta C_2)C_3$ with

$$\begin{array}{llll} A_1 \Rightarrow_\rho B_1 & \text{and} & A_1 \Rightarrow_\rho C_1 \text{ and} & \\ A_2 \Rightarrow_\rho B_2 & \text{and} & A_2 \Rightarrow_\rho C_2 \text{ and} & \\ A_3 \Rightarrow_\rho B_3 & \text{and} & A_3 \Rightarrow_\rho C_3 \text{ and} & \mathcal{S}ol(B_1 \prec_\theta B_3) = \sigma \end{array}$$

is similar to the previous one.

□

Theorem 6.9 (Confluence) *The core dynamic pattern λ-calculus with $\mathcal{S}ol$ satisfying $\mathcal{H}_0$, $\mathcal{H}_1$ and $\mathcal{H}_2$ is confluent.*

Proof. By Lemma 6.6 the reflexive and transitive closure of the relations $\rightarrow_\rho$ and $\Rightarrow_\rho$ are the same. By Lemma 6.8 , the relation $\Rightarrow_\rho$ is confluent. We conclude that the relation $\rightarrow_\rho$ is confluent. □

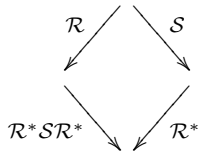
As we have already said most of the pattern-calculi extend the basic β rule (or its equivalent) by a set of rules. We will state here the conditions that should be imposed in order to prove the confluence of the dynamic pattern λ-calculus, conditions that turn out to be satisfied by most of the different calculi that can be expressed as instances of the dynamic pattern λ-calculus.

We show in this section that confluence of extensions of the core dynamic pattern λ-calculus with an appropriate set of rules is easy to deduce using Yokouchi-Hikita's lemma [YH90] (see also [CHL96]).

Lemma 6.10 (Yokouchi-Hikita) *Let $\mathcal{R}$ and $\mathcal{S}$ be two relations defined on the same set $\mathcal{T}$ of terms such that*

- *$\mathcal{R}$ is strongly normalizing and confluent,*
- *$\mathcal{S}$ has the diamond property,*
- *for all A, B and C in $\mathcal{T}$ such that $A \rightarrow_\mathcal{R} B$ and $A \rightarrow_\mathcal{S} C$ then there exists D such that $B \rightarrow_{\mathcal{R}^*\mathcal{S}\mathcal{R}^*} D$ and $C \rightarrow_{\mathcal{R}^*} D$ (often mentioned as Yokouchi-Hikita's diagram).*

The following diagram commutes



Then the relation $\mathcal{R}^*\mathcal{S}\mathcal{R}^*$ is confluent.

Theorem 6.11 *The dynamic pattern λ-calculus is confluent when*

- *$\mathcal{S}\mathtt{ol}$ satisfies $\mathcal{H}_0, \mathcal{H}_1, \mathcal{H}_2$,*
- *the set ξ of reduction rules is strongly normalizing and confluent,*
- *the relations $\Rightarrow_\rho$ and ξ satisfy Yokouchi-Hikita's diagram.*

Proof. We apply Yokouchi-Hikita's lemma by taking for the relation $\mathcal{R}$ the compatible relation induced by ξ and for the relation $\mathcal{S}$ the relation $\Rightarrow_\rho$. The diamond property of the relation $\Rightarrow_\rho$ is given by Lemma 6.8. To conclude the proof, it is sufficient to remark that the reflexive and transitive closure of

$$\rightarrow_{\rho\cup\xi} \qquad \text{and} \qquad \twoheadrightarrow_\xi \Rightarrow_\rho \twoheadrightarrow_\xi$$

are equal (as a consequence of Lemma 6.6).

□

In fact this theorem states that any pattern-calculus defined as a dynamic pattern λ-calculus with a particular $\mathcal{S}\mathtt{ol}$ that satisfies $\mathcal{H}_0$, $\mathcal{H}_1$ and $\mathcal{H}_2$ is confluent.

6.2.4 Confluence issues with linear patterns

The different results we give in the previous sections may seem to be limited because the conditions imposed on the matching algorithm are strong. Nevertheless, these conditions are respected by most of the pattern-calculi we have explored and relaxing them leads to classical counter-examples for confluence.

For example, if the matching can be performed on active variables then non-confluent reductions can be obtained in both the lambda-calculus with patterns [Oos90, KOV07] and in the rewriting calculus [CLW03]. Similarly, non-linear patterns lead to non-confluent reductions that are variations of the Klop's counter-example [Klo80] for higher-order systems dealing with non-linear matching.

When using dynamic patterns containing variables that are not bound in the abstraction the confluence hypotheses should be carefully verified. More precisely, the behavior of non-linear rules can be encoded using *linear* and dynamic patterns. Consequently, Klop's counter-example can be encoded in the corresponding calculus that is therefore non-confluent.

$$(\mathsf{d}\,x\,x) \twoheadrightarrow_x \clubsuit \quad \triangleq \quad (\mathsf{d}\,x\,y) \twoheadrightarrow_{x,y} (x \twoheadrightarrow_\emptyset \clubsuit) y$$

where $\clubsuit$ denotes an arbitrary term.

Figure 6.5: Klop's counter example using dynamic linear patterns

Proposition 6.12 (Non-confluence) *The core dynamic pattern λ-calculus with* $\mathcal{S}ol$ *such that for all terms* A *and* B *and for some constant* d

$$\mathcal{S}ol(A \prec\!\!\!\prec_\emptyset A) = id$$

$$\mathcal{S}ol(x \prec\!\!\!\prec_x A) = \{x \leftarrow A\}$$

$$\mathcal{S}ol(\mathsf{d}(x,y) \prec\!\!\!\prec_{x,y} \mathsf{d}(A,B)) = \{x \leftarrow A, y \leftarrow B\}$$

is not confluent.

Proof. The main idea of the proof is to remark that we can encode non-linear patterns using dynamic linear patterns indicated in Figure 6.5. Then, the proof is a straightforward adaptation of the encoding of Klop's counter example in the ρ-calculus given in Section 4.3.3.

Let d and e be two constants. Recall that we omit the set θ of variables bound in an abstraction when it coincides with set of free variables of the pattern. We denote $\twoheadrightarrow_h$ head reductions.

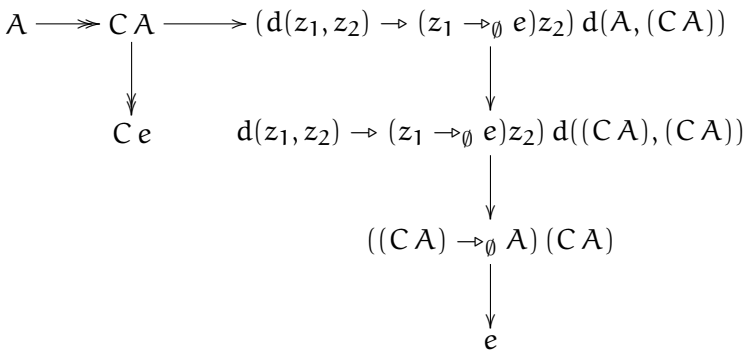
We define the fix-point combinator Y as in the λ-calculus by

$$Y \quad \triangleq \quad \Big(y \twoheadrightarrow x \twoheadrightarrow (x\,(y\,y\,x))\Big)\,\Big(y \twoheadrightarrow x \twoheadrightarrow (x\,(y\,y\,x))\Big)$$

As usually, for all term A we have $Y\,A \twoheadrightarrow_h A\,(Y\,A)$. We define the following terms:

$$\begin{array}{rcl} C' & \equiv & \big(y \twoheadrightarrow x \twoheadrightarrow (\mathsf{d}(z_1,z_2) \twoheadrightarrow (z_1 \twoheadrightarrow_\emptyset e) z_2)\,\mathsf{d}(x,(y\,x))\big) \\ C & \equiv & Y C' \\ C & \twoheadrightarrow_h & \big(y \twoheadrightarrow x \twoheadrightarrow (\mathsf{d}(z_1,z_2) \twoheadrightarrow (z_1 \twoheadrightarrow_\emptyset e) z_2)\,\mathsf{d}(x,(y\,x))\big)\ C \\ & \to_h & x \twoheadrightarrow (\mathsf{d}(x,z) \twoheadrightarrow (z_1 \twoheadrightarrow_\emptyset e) z_2)\,\mathsf{d}(x,(C\,x)) \\ A & \equiv & Y\,C \end{array}$$

and we have the following reductions:



The constant e is in normal form. Let us consider the smallest reduction from $C\,e$ to e. In this reduction, head redexes must be reduced:

$$\begin{array}{lll} C\,e & \equiv & (Y C')\,e \\ & \twoheadrightarrow_h & C'\,(Y\,C')\,e \\ & \twoheadrightarrow_h & ((d(z_1, z_2) \triangleright (z_1 \triangleright_\emptyset e)z_2)(d(e, (C\,e)) \\ & \rightarrow_h & (e \triangleright_\emptyset e)(C\,e) \end{array}$$

But the term $(e \triangleright_\emptyset e)(C\,e)$ can be head-reduced only after reducing $C\,e$ to e. This contradicts the minimality of the reduction and thus we conclude that $C\,e$ does not reduce to e. □

As a consequence we obtain that any pattern-calculus defined using a function $\mathcal{S}ol$ that satisfies the conditions in Proposition 6.12 is not confluent. This is somewhat surprising since the last two computations are clearly satisfied by any classical syntactic matching algorithm and the first one seems to be a reasonable choice. On the other hand, ignoring the information given by the set θ of bound variables when performing matching can lead to strange behaviors and in particular it allows for an encoding of the equality of terms.

There were some attempts [BCKL03] to develop calculi that combine the three conditions of Proposition 6.12. Of course, they do not succeed. In fact, the solution proposed in [BCKL03] validates the first condition but validates neither the second condition nor the third one: the syntactical restriction does not allow in an abstraction $(A \triangleright_\theta B)$ patterns with variables in θ and is thus not conservative with the λ-calculus. We study in more details this syntactical restriction in Section 6.3.2

6.3 Instantiations of the dynamic pattern λ-calculus

In this section, we give some instantiations of the dynamic pattern λ-calculus. For the sake of simplicity, we only give the key points for each of the pattern based calculi. All these calculi have been proved confluent under appropriate conditions; we give here confluence proofs based on these conditions and using the general confluence proof for

$$(\rho_{\lambda_P}) \qquad (A \twoheadrightarrow B)(A\sigma) \qquad \to \qquad B\sigma$$

Figure 6.6: Operational semantics of the λ-calculus with patterns

the dynamic pattern λ-calculus. This latter approach does not provide confluence proofs for free but gives a proof methodology that focuses on the fundamental properties of the underlying matching that can be thus seen as the key issue of pattern based calculi.

6.3.1 λ-calculus with Patterns

The λ-calculus with patterns was introduced in [Oos90, KOV07] as an extension of the λ-calculus with pattern-matching facilities. The set of terms is parameterized by a set of patterns Φ on which we can abstract.

The syntax of the λ-calculus with patterns is thus the one of the core dynamic pattern λ-calculus but where patterns are taken in a given set and abstractions always bind all the (free) pattern variables. Its operational semantics is given by the rule in Figure 6.6.

Instead of considering syntactical restrictions, we can equivalently consider that a matching problem $A \lessdot_\theta A\sigma$ has a solution only when $A \in \Phi$. The λ-calculus with patterns can thus be seen as an instance of the core dynamic pattern λ-calculus.

The calculus is not confluent (see [Oos90] Ex. 4.18) in general but some restrictions can be imposed on the set of patterns to recover confluence. This restriction is called the Rigid Pattern Condition (RPC) and can be defined using the parallel reduction of the calculus.

Definition 6.13 (RPC) *A term P satisfies the RPC condition if*

$$\forall\sigma,\ \forall A,\ P\sigma \Rightarrow_\rho A \qquad \textit{implies} \qquad A \equiv P\sigma' \textit{ with } \sigma \Rightarrow_\rho \sigma'$$

The definition allows for patterns which are extensionally but not intensionally rigid, such as $\Omega\, x\, y$ with $\Omega = (x \twoheadrightarrow xx)\,(x \twoheadrightarrow xx)$. We choose a (less general) syntactical characterization [Oos90, KOV07] which we also call RPC and that excludes these pathological cases.

Definition 6.14 (RPC) *The set of terms satisfying RPC is the set of all terms of the λ-calculus which*

- *are linear (each free variable occur at most once),*
- *are in normal form,*
- *have no active variables (i.e., no sub-terms of the form xA where x is free).*

Note that reformulating $\mathcal{H}_2$ in the particular case of a $\mathcal{S}ol$ function that performs matching on closed patterns leads to a condition close to the original RPC (the parallel reduction used in the definition of RPC is slightly different). Nevertheless, the hypothesis

$\mathcal{H}_2$ allows the matching to be performed on patterns that are not in normal form (or not reducible to themselves) while this is not the case for the RPC.

Example 6.15 *Pairs and projections can be encoded in the λ-calculus with patterns by directly matching on the pair encoding.*

$$\begin{array}{lll} \big((z \twoheadrightarrow (z\,x)\,y) \twoheadrightarrow x\big)\big(z \twoheadrightarrow (z\,A)\,B\big) & \to_{\rho_{\lambda_P}} & x\{x \leftarrow A,\ y \leftarrow B\} \\ & \equiv & A \end{array}$$

Proposition 6.16 *The λ-calculus with patterns is confluent if the patterns are taken in the set defined by the RPC.*

Proof. The hypotheses $\mathcal{H}_0$ and $\mathcal{H}_1$ follow immediately. To prove $\mathcal{H}_2$, we can remark that if $P\sigma \Rightarrow_\rho B$ with $P \in$ RPC then $\exists B', \sigma'$ s.t. $B' \equiv P\sigma'$ with $\sigma \Rightarrow_\rho \sigma'$ This proves that a redex cannot overlap with P in $P\sigma$ if $P \in$ RPC and thus that the condition $\mathcal{H}_2$ is satisfied. We conclude the proof by applying Thm. 6.9. □

6.3.2 Rewriting calculi

In this section, we review the confluence of the extensions of the ρ-calculus presented in Section 4.5: namely the ρ_{stk}-calculus and the ρ_d-calculus.

The ρ_{stk}-calculus

We can consider $\wr$ and stk as constants of the dynamic pattern λ-calculus and thus we can see the syntax of the ρ_{stk}-calculus as an instance of the syntax of the dynamic pattern λ-calculus. The rule (ρ_{stk}) can be considered as an instance of the (ρ) rule of the dynamic pattern λ-calculus.

$$\begin{array}{llll} (\rho) & (P \twoheadrightarrow A)B & \to & A\sigma \\ & & & \text{if } \sigma = \mathcal{S}ol(P \lessdot B) \end{array}$$

where $\mathcal{S}ol(P \lessdot B)$ has a solution only when P is a pattern. Since patterns are linear algebraic terms then the $\mathcal{S}ol$ function can be implemented using first-order linear matching *à la* Huet (see Example 6.2).

Proposition 6.17 *The ρ_{stk}-calculus with linear algebraic patterns is confluent.*

Proof. Confluence of the (ρ_{stk}) rule is obtained as the confluence of the λ-calculus with patterns since the set of patterns of the ρ_{stk}-calculus is a subset of patterns satisfying RPC. The relation $\delta \cup \mathsf{stk}$ induces a terminating relation. It is also locally confluent (this can be proved by an easy induction on the structure of terms).

To conclude that the ρ_{stk}-calculus is confluent using Theorem 6.9 it is sufficient to remark that $\Rightarrow_{\rho\ \rho_{stk}}$ and $(\delta \cup \mathsf{stk})$ satisfy the Yokoutchi-Hikita's diagram (easy induction of the structure of terms).

□

The ρ_d-calculus

The encoding is similar to the one of the ρ_{stk}-calculus. Note that for the (ρ_d) rule, the corresponding function $\mathcal{S}\mathtt{ol}$ has solutions only when the argument is a stuck value.

Proposition 6.18 *The ρ_d-calculus is confluent.*

Proof. The proof is similar to the one of the ρ_{stk}-calculus. Yokoutchi-Hikita's diagram is satisfied since the use of values eliminate most of the critical pairs. □

Pure Pattern Type Systems

In [BCKL03] to obtain confluence, the condition RPC is adapted to deal with dynamic patterns (that can be instantiated and reduced). The original definition of RPC difficultly fits into dynamic patterns since the RPC condition prevents reduction in patterns (except for some pathological cases).

An other alternative close to the one presented in [BCKL03] is to define the following restriction: P satisfies RPC^+

$$\begin{array}{l}\text{if for all } \mathsf{A},\mathsf{B} \text{ and for all } \sigma_1,\sigma_2 \text{ and } \sigma_3 \text{ such that} \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{then} \quad \mathsf{B} \Rightarrow_\rho \mathsf{P}\sigma_3 \\ \mathsf{A} \Rightarrow_\rho \mathsf{P}\sigma_2, \mathsf{A} \Rightarrow_\rho \mathsf{B}, \sigma_1 \Rightarrow_\rho \sigma_2 \text{ and } \sigma_1 \Rightarrow_\rho \sigma_3\end{array}$$

Nevertheless, the RPC^+ condition does not allow variables in patterns. In fact, let x be a variable. We denote by $\mathtt{I} = (\mathsf{y} \to_{\mathsf{y}} \mathsf{y})$ the identity function. We show that the variable x does not satisfy RPC^+ by taking:

$$\begin{array}{lcllcl}\sigma_1 & \equiv & \sigma_3 \equiv \{\mathsf{x} \leftarrow \mathtt{I}\,\mathtt{I}\} & \sigma_2 & \equiv & \{\mathsf{x} \leftarrow \mathtt{I}\} \\ \mathsf{A} & \equiv & \mathtt{I}\,\mathtt{I} & \mathsf{B} & \equiv & \mathtt{I}\end{array}$$

6.3.3 Pure pattern calculus

In the λ-calculus, data structures such as pairs of lists can be encoded. Although the λ-calculus supports some functions that act uniformly on data structures, it cannot support operations that exploit characteristics common to all data structures such as an update function that traverses any data structures to update its atoms. In the pure pattern calculus [JK06] where any term can be a pattern the focus is put on the dynamic matching of data structures.

The syntax of the pure pattern calculus is the same as the one of the dynamic pattern λ-calculus (except that the pure pattern calculus defines a single constructor).

Pattern-abstractions are applied using a particular matching algorithm. Although the original paper uses a single rule to describe application of pattern-abstractions we present it here using two rules. First, a rule that is an instance of the (ρ) rule of the dynamic pattern λ-calculus. Secondly, a rule that reduces the corresponding pattern-abstraction application to the identity (the motivation for this second rule is given in [JK06]) when the pattern-matching does not succeed.

ϕ-data structures and ϕ-matchable forms

$$\begin{array}{lll} D & ::= & x\,(x \in \phi) \mid c \mid D\,A \\ E & ::= & D \mid A \twoheadrightarrow_\theta B \end{array}$$

where A and B are arbitrary terms

Semantics of the pure pattern calculus

$$\begin{array}{lll} (\rho_{pc}) & (A \twoheadrightarrow_\theta B)C & \to \quad B\sigma \text{ if } \sigma = \mathcal{S}ol(A \prec\!\!\!\prec_\theta C) \\ (\rho^{stk}_{pc}) & (A \twoheadrightarrow_\theta B)C & \to \quad x \twoheadrightarrow x \text{ if none} = \mathcal{S}ol(A \prec\!\!\!\prec_\theta C) \end{array}$$

Figure 6.7: The pure pattern calculus

The matching algorithm of the pure pattern calculus is based on the notions of ϕ-data structures (denoted D) and ϕ-matchable forms (denoted E) that are given in Figure 6.7.

The operational semantics of the pure pattern calculus is given in Figure 6.7 where the partial function $\mathcal{S}ol$ is defined by the following equations that are applied respecting the order below

$$\begin{array}{lll} \mathcal{S}ol(x \prec\!\!\!\prec_\theta A) & = & \{x \leftarrow A\} \quad \text{if } x \in \theta \\ \mathcal{S}ol(c \prec\!\!\!\prec_\theta c) & = & \mathtt{id} \\ \mathcal{S}ol(A_1A_2 \prec\!\!\!\prec_\theta B_1B_2) & = & \mathcal{S}ol(A_1 \prec\!\!\!\prec_\theta B_1) \Cup \mathcal{S}ol(A_2 \prec\!\!\!\prec_\theta B_2) \\ & & \quad \text{if } A_1A_2 \text{ is a } \theta\text{-data structure} \\ & & \quad \text{if } B_1B_2 \text{ is a data structure} \\ \mathcal{S}ol(A_1 \prec\!\!\!\prec_\theta B_1) & = & \text{none} \\ & & \quad \text{if } A_1 \text{ is a } \theta\text{-matchable form} \\ & & \quad \text{if } B_1 \text{ is a matchable form} \end{array}$$

Note that the union $\Cup$ is only defined for substitutions of disjoint domains and that the union of none and σ is always none.

At first sight, the matching algorithm may seem surprising because one decomposes application syntactically whereas it is a higher-order symbol. This is sound because the decomposition is done only on data-structures, which consist of head normal forms.

Example 6.19 *[JK06] Define* $\mathtt{elim} \triangleq x \twoheadrightarrow (xy) \twoheadrightarrow_y y$ *to be the generic eliminator. For example, suppose given two constants* `Cons` *and* `Nil` *representing list constructs. We define the function* $\mathtt{singleton} \triangleq x \twoheadrightarrow (\mathtt{Cons}\ x\ \mathtt{Nil})$ *and we check that*

$$\begin{array}{lll} \mathtt{elim}\ \mathtt{singleton} & \equiv & \big(x \twoheadrightarrow (xy) \twoheadrightarrow_y y\big)\big(x \twoheadrightarrow (\mathtt{Cons}\ x\ \mathtt{Nil})\big) \\ & \to_{\rho_{pc}} & \big((x \twoheadrightarrow \mathtt{Cons}\ x\ \mathtt{Nil})y\big) \twoheadrightarrow y \\ & \to_{\rho_{pc}} & (\mathtt{Cons}\ y\ \mathtt{Nil}) \twoheadrightarrow y \end{array}$$

Proposition 6.20 *The pure pattern calculus is confluent.*

Proof. The hypothesis $\mathcal{H}_0$ is true. The hypotheses $\mathcal{H}_1$ and $\mathcal{H}_2$ are not surprisingly intermediate results in [JK06]. In particular, Lemma 7 [JK06] states that the function $\mathcal{S}\mathtt{ol}$ is stable by substitution and Lemma 8 [JK06] proves that the function $\mathcal{S}\mathtt{ol}$ is stable by reduction (when it returns a substitution and when it returns none). We use this property to prove that the relation $(\rho_{\mathsf{pc}}^{\mathsf{stk}})$ is locally confluent (simple induction). It is trivially terminating, and thus confluent. The fact that the relations $\Rightarrow_{\rho_{\mathsf{pc}}}$ and $(\rho_{\mathsf{pc}}^{\mathsf{stk}})$ verify Yokouchi-Hikita's diagram is obtained again by a simple induction. The only interesting case is for the term $(A_0 \twoheadrightarrow A_1)A_2$ but it is easy to conclude using the stability by reduction of the function $\mathcal{S}\mathtt{ol}$. □

6.3.4 λ-calculus with constructors

The λ-calculus with constructors was introduced in [AMR06] to address the problem of the interaction of functions and constructed values. We show that it can be encoded in the dynamic pattern λ-calculus. The encoding gives a nice example where computations and matching cohabits. In fact, a matching problem may have a solution even if the pattern is not in normal form. To the knowledge of the authors the dynamic pattern λ-calculus is the first confluent calculus where such encodings can be done.

We recall the syntax and the operational semantics of the λ-calculus with constructors in Figure 6.8. For the sake of simplicity, we have not included η-conversion rules but the same approach applied also in this case. Note that in the rule (CaseCase) of Figure 6.8, the binding composition denoted $\circ$ is an external operation defined as follows:

$$\phi \circ (c_1 \mapsto A_1; \ldots; c_n \mapsto A_n) \quad \equiv \quad (c_1 \mapsto \{\!|\phi|\!\}.\, A_1; \ldots; c_n \mapsto \{\!|\phi|\!\}.\, A_n)$$

Example 6.21 *[AMR06] In the λ-calculus with constructors the predecessor function is implemented as*

$$\mathrm{pred} \equiv \big(n \twoheadrightarrow \{\!|0 \mapsto 0;\ s \mapsto (z \twoheadrightarrow z)|\!\}.\, n\big)$$

We can check that

$$\begin{array}{rl} & \mathrm{pred}\, 0 \\ \to & \{\!|0 \mapsto 0;\ s \mapsto (z \twoheadrightarrow z)|\!\}.\, 0 \\ \to & 0 \end{array}$$

and

$$\begin{array}{rl} & \mathrm{pred}\, (s\, N) \\ \to & \{\!|0 \mapsto 0;\ s \mapsto (z \twoheadrightarrow z)|\!\}.\, (s\, N) \\ \to & (\{\!|0 \mapsto 0;\ s \mapsto (z \twoheadrightarrow z)|\!\}.\, s)\, N \\ \to & (z \twoheadrightarrow z)\, N \\ \to & N \end{array}$$

Encoding in the dynamic pattern λ-calculus The following encoding is inspired by Example 6.3. We propose the following encoding in the dynamic pattern λ-calculus. The syntax of the λ-calculus with constructors can be translated in the syntax of the

Syntax of the λ-calculus with constructors

Terms	A, B	$::=$	$x \mid c \mid \maltese \mid A\,B \mid x \rightarrowtriangle A \mid \{\!\{\phi\}\!\}.\,A$	
Case bindings	ϕ	$::=$	$c_1 \mapsto A_1; \ldots; c_n \mapsto A_n$	$(c_i \neq c_j \text{ for } i \neq j)$

Semantics of the λ-calculus with constructors

(AppLam)	$(x \rightarrowtriangle A)B$	$\rightarrow$	$A\{x \leftarrow B\}$	
(AppDai)	$\maltese N$	$\rightarrow$	$\maltese$	
(CaseCons)	$\{\!\{\phi\}\!\}.\,c$	$\rightarrow$	A	$((c \mapsto A) \in \phi)$
(CaseDai)	$\{\!\{\phi\}\!\}.\,\maltese$	$\rightarrow$	$\maltese$	
(CaseApp)	$\{\!\{\phi\}\!\}.\,(A\,B)$	$\rightarrow$	$(\{\!\{\phi\}\!\}.\,A)\,B$	
(CaseLam)	$\{\!\{\phi\}\!\}.\,(x \rightarrowtriangle A)$	$\rightarrow$	$x \rightarrowtriangle \{\!\{\phi\}\!\}.\,A$	$(x \notin \mathtt{fv}(\phi))$
(CaseCase)	$\{\!\{\phi\}\!\}.\,\{\!\{\psi\}\!\}.\,A$	$\rightarrow$	$\{\!\{\phi \circ \psi\}\!\}.\,A$	

Figure 6.8: The λ-calculus with constructors

dynamic pattern λ-calculus by defining (the other cases are straightforward, considering that $\maltese$ is a constant)

$$\{\!\{c_1 \mapsto A_1; \ldots; c_n \mapsto A_n\}\!\}.\,B \quad \triangleq \quad \big((c_1 \hookrightarrow A_1/\ldots/c_n \hookrightarrow A_n) \rightarrowtriangle_x x\big)B$$

Note that the symbols used in the encoding (for example $\hookrightarrow$ or $/$) are considered as constants of the dynamic pattern λ-calculus. We can encode the (AppLam) and (CaseCons) rules as a single (ρ) rule with a function $\mathcal{S}\mathtt{ol}$ defined by

$$\begin{array}{lcl} \mathcal{S}\mathtt{ol}(x \prec_x A) & = & \{x \leftarrow A\} \\ \mathcal{S}\mathtt{ol}\big((c_1 \hookrightarrow A_1/\ldots/c_n \hookrightarrow A_n) \prec_x c_{i_0}\big) & = & \{x \leftarrow A_{i_0}\} \end{array}$$

The (ρ) rule induced by this function $\mathcal{S}\mathtt{ol}$ gives a direct encoding of the (AppLam) and (CaseCons) rules.

Proposition 6.22 *The λ-calculus with constructors is confluent.*

Proof. The (ρ) rule defined using the above function $\mathcal{S}\mathtt{ol}$ is confluent since the function $\mathcal{S}\mathtt{ol}$ clearly verifies $\mathcal{H}_0, \mathcal{H}_1$ and $\mathcal{H}_2$. Consider the set of rules ξ that consists of all the rules given in Figure 6.8 but (AppLam) and (CaseCons). The compatible relation induced by this set of rules is confluent since it is strongly normalizing and locally confluent (the proof of strong normalization and local confluence are given in [AMR06] and are straightforward verifications). Moreover, it is easy to check that the relations $\rightarrow_\xi$ and $\Rightarrow_\rho$ verify Yokoutchi-Hikita's lemma. By applying Theorem 6.9 we obtain that the λ-calculus with constructors is confluent. □

Conclusion and future works

We propose here a different formulation for different pattern-based calculi and we use the confluence properties of the general formalism to give alternative confluence proofs for these calculi. The general confluence proof uses the standard techniques of parallel reduction of Tait and Martin-Löf and Yokouchi-Hikita's lemma. The method proposed by M. Takahashi [Tak95] gives in general more elegant and shorter proofs by using the notion of complete developments. Reformulating the hypotheses $\mathcal{H}_0, \mathcal{H}_1$ and $\mathcal{H}_2$ and adapting the proofs of this chapter is easy but given a pattern-calculus the reformulated hypotheses are often more difficult to prove than for the original case.

Moreover, we show that the proof of confluence of the ρ-calculus is easy to deduce from our general result as soon as the structure operator has no equational theory. Nevertheless, if one wants to switch to non-unitary matching (and this is very useful is practice [BBK+06, CDE+02]) then the ρ rule should return a collection of results and the structure operator should be (at least) associative and commutative. This extension is syntactically and semantically non-trivial and opens new challenging problems. Even the extension of the λ-calculus with term collections is not trivial as it is shown in the following chapters.

Part III

Categorical semantics of the parallel λ-calculus

Chapter 7

Categorical semantics of the λ-calculus

Context The categorical semantics of the untyped λ-calculus, which is based on Cartesian closed categories (ccc) with reflexive objects [Mac98], provides a rich framework to classify all the existing models of the λ-calculus [Bar84]. One of the most striking properties of this semantics is that it is complete in a very strong sense [Sco80a, Sco80b]: for each extension $\mathcal{T}$ of the basic equational theory of the λ-calculus, it is possible to find a reflexive object such that the equational theory induced by the interpretation of pure λ-terms in this object is exactly the theory $\mathcal{T}$.

Consequently, there exists a reflexive object, in a 'syntactical' ccc of partial equivalence relations, which captures β-conversion, so that two terms having the same denotation in all reflexive objects have the same denotation in this particular object, and thus are β-convertible.

In other words, we obtain the completeness property by showing that the syntax of the λ-calculus has already the structure of a Cartesian closed category with reflexive objects, i.e. it is a λ-model.

Contributions This chapter recalls some basic material used in the categorical semantics of the λ-calculus and sketches the completeness proof. All the proof details will be given in Chapter 9 for the case of the parallel λ-calculus.

The reader may found more details on categories in [Mac98, AL91]. Several good introductions on the usual interpretation of the simply typed λ-calculus in cccs are available, see for example [AC98] or the master course notes of P.-A. Melliès. More on models for the untyped λ-calculus can be found in [Sco96, Bar84, AC98, AL91].

Part III of this thesis is a joint work with Alexandre Miquel. The following chapters are directly inspired from our official [FM07] and non official documents.

Outline of the chapter In Section 7.1 we recall basic definitions related to Cartesian closed categories and reflexive objects in order to fix the notations used in the following. In Section 7.2 we give the interpretation of the pure λ-terms in a given Cartesian closed category and state its soundness. In Section 7.4 we sketch the proof of the completeness property for the interpretation of pure λ-terms.

7.1 Cartesian closed categories and reflexive objects

We first recall the definition of a category and of a Cartesian closed category.

Definition 7.1 (Category) *A category* $\mathfrak{C}$ *is given by*

- *A class of objects, denoted* $\mathbf{Obj}\mathfrak{C}$.
- *A class of morphisms. Each morphism* f *has a unique source object* A *and target object* B *where* A *and* B *are objects of* $\mathfrak{C}$. *We write* $f : A \to B$, *and we say "*f *is a morphism from* A *to* B*" or equivalently that "*f *is an arrow from* A *to* B*". We write* $\mathfrak{C}[A; B]$ *to denote the class of all morphisms from the object* A *to the object* B.
- *A composition law* $\circ : \mathfrak{C}[B; C] \times \mathfrak{C}[A; B] \to \mathfrak{C}[A; C]$ *which is associative, that is*

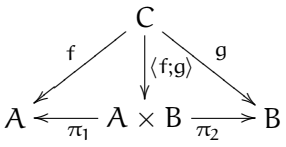
$$\forall (f; g; h) \in \mathfrak{C}[C; D] \times \mathfrak{C}[B; C] \times \mathfrak{C}[A; B] \qquad f \circ (g \circ h) = (f \circ g) \circ h.$$

- *An identity morphism* $\mathrm{id}_A \in \mathfrak{C}[A; A]$ *for all objects* A *which is a neutral element for* $\circ$, *that is*

$$\forall f \in \mathfrak{C}[A; B] \qquad f \circ \mathrm{id}_A = f = \mathrm{id}_B \circ f.$$

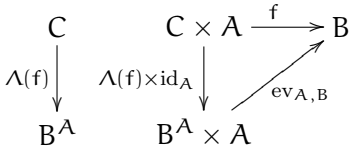
Definition 7.2 (Cartesian closed) *A category* $\mathfrak{C}$ *is called Cartesian closed if it has*

1. *A* terminal object *written* 1 *that is, for each object* $A \in \mathfrak{C}$ *there is exactly one arrow from* A *to* 1 *written* $\diamond_A : A \to 1$.
2. *For each object* $A, B \in \mathfrak{C}$ *a* Cartesian product *given by an object written* $A \times B$ *and two morphisms (projections)* $\pi_1 : A \times B \to A$ *and* $\pi_2 : A \times B \to B$ *such that for all objects* $C \in \mathfrak{C}$ *and arrows* $f : C \to A$ *and* $g : C \to B$, *there exists a unique arrow* $\langle f; g \rangle : C \to A \times B$ *that makes the following diagram commute*



$$\pi_1 \circ \langle f; g \rangle = f \quad (\textit{prod}\text{-}1)$$
$$\pi_2 \circ \langle f; g \rangle = g \quad (\textit{prod}\text{-}2)$$

3. *For each object* $A, B \in \mathfrak{C}$ *an* exponent *given by an object written* B^A *and a corresponding evaluation function written* $\mathrm{ev}_{A,B} : B^A \times A \to B$ *such that for all objects* $C \in \mathfrak{C}$ *and for all morphisms* $f : C \times A \to B$, *there exists a unique morphism* $\Lambda(f) : C \to B^A$ *that makes the following diagram commute*



$$\mathrm{ev}_{A,B} \circ (\Lambda_{C,A,B}(f) \times \mathrm{id}_A) = f \quad (\textit{exp}\text{-}\beta)$$

It is easy to remark that in all Cartesian closed categories (see for example [AC98]) the following equations hold

$$\Lambda(f) \circ h = \Lambda(f \circ (h \times \mathrm{id})) \qquad (\text{exp-}\circ)$$
$$\Lambda(\mathrm{ev}) = \mathrm{id} \qquad (\text{exp-}\eta)$$

Definition 7.3 (Reflexive object) *A reflexive object of a ccc* $\mathfrak{C}$ *is a triple* $(D, \mathrm{lam}, \mathrm{app})$ *formed by an object* $D \in \mathfrak{C}$ *and two arrows* $\mathrm{lam} : D^D \to D$ *and* $\mathrm{app} : D \to D^D$ *such that* $\mathrm{app} \circ \mathrm{lam} = \mathrm{id}_{D^D}$.

7.2 Interpretation of pure λ-terms

The syntax of the λ-calculus used in this chapter was defined in Part I. Terms are denoted here M, N (instead of A, B) because it fits better with the notations used in the following chapters. We recall that the syntax of the λ-calculus is defined by

$$M, N \quad ::= \quad x \;\mid\; \lambda x\,.\,M \;\mid\; MN$$

The corresponding equational theory is given by the equational axiom

$$(\beta) \qquad (\lambda x\,.\,M)N \quad = \quad M\{x := N\}$$

In this section, we work in a fixed Cartesian closed category $\mathfrak{C}$. Let $(D, \mathrm{lam}, \mathrm{app})$ be a reflexive object of $\mathfrak{C}$. The idea of the interpretation is to consider each pure λ-term as a simply typed λ-term expressed in a degenerate type system with only two 'types' D and D^D, plus coercions $\mathrm{lam} : D^D \to D$ and $\mathrm{app} : D \to D^D$ relating both types.

In this setting, typing contexts degenerate into (ordered) lists of variables

$$\ell \quad ::= \quad [] \;\mid\; \ell, x$$

in which variables are implicitly declared with type D. Each list (context) ℓ is interpreted as an object $[\![\ell]\!]$ of $\mathfrak{C}$ which is defined by:

$$[\![[]]\!] = 1 \qquad \text{and} \qquad [\![\ell, x]\!] = [\![\ell]\!] \times D\,.$$

To each pair (ℓ, x) formed by a list of variables ℓ and a variable x that belongs to ℓ, we define the *projection* $\pi^x_\ell : [\![\ell]\!] \to D$ by setting

$$\begin{array}{lcll} \pi^x_{\ell,x} & = & \pi_2 & \in \quad [\![\ell]\!] \times D \to D \\ \pi^x_{\ell,y} & = & \pi^x_\ell \circ \pi_1 & \in \quad [\![\ell]\!] \times D \to D \qquad (\text{if } y \neq x) \end{array}$$

Each term M whose free variables belong to a list of variables ℓ is interpreted by an arrow $[\![M]\!]_\ell : [\![\ell]\!] \to D$ which is defined by the equations given in Figure 7.1.

With a little abuse of notations, we write $FV(N) \subset \ell$ to express that all the free variables of N occur (at least once) in the list ℓ. Notice that a variable x may occur several times in a list ℓ, but that the interpretation $[\![M]\!]_\ell : [\![\ell]\!] \to D$ systematically binds the last occurrence of x in ℓ. The reader is invited to check that this choice (which is implemented by the definition of the projections $\pi^x_\ell : [\![\ell]\!] \to D$) is consistent with the interpretation of λ-abstractions.

The soundness of this interpretation relies on the following propositions.

(Var)	$[\![x]\!]_\ell$	$=$	π^x_ℓ
(Lam)	$[\![\lambda x\,.\,M]\!]_\ell$	$=$	$\mathrm{lam} \circ \Lambda([\![M]\!]_{\ell,x})$
(App)	$[\![MN]\!]_\ell$	$=$	$\mathrm{ev} \circ \langle \mathrm{app} \circ [\![M]\!]_\ell ; [\![N]\!]_\ell \rangle$

Figure 7.1: Interpretation of pure λ-terms

Proposition 7.4 (Soundness w.r.t. substitution) *Given a list of variables ℓ and a variable x, for all terms M and N such that $FV(M) \subset (\ell, x)$ and $FV(N) \subset \ell$,we have:*

$$[\![M\{x := N\}]\!]_\ell \quad = \quad [\![M]\!]_{\ell,x} \circ \langle \mathrm{id}; [\![N]\!]_\ell \rangle$$

Proposition 7.5 (Soundness w.r.t. β-reduction) *Given a list of variables ℓ, for all terms M and M' whose free variables occur in ℓ we have*

$$M \to_\beta M' \quad \Rightarrow \quad [\![M]\!]_\ell = [\![M']\!]_\ell$$

As a corollary, it is clear that β-convertible terms have the same denotation in any reflexive object of any ccc. Notice that if we want to extend this result to η-conversion, we must assume that $\mathrm{lam} \circ \mathrm{app} = \mathrm{id}_D$ too, which means that D^D is not only a retract of D, but that D^D is actually isomorphic to D.

What is more interesting is that the converse of the soundness property is true: if two λ-terms have the same denotation in all reflexive objects of all cccs, then these λ-terms are β-convertible. This property, studied in Section 7.4, is often mentioned as the completeness property.

In the following section, we give examples of models of the untyped λ-calculus taken in the category of Scott domains.

7.3 Examples in Scott domains

In this section, we give a smooth presentation of the first historical model of the untyped λ-calculus given by Scott [Sco80a, Sco80b].

A Scott domain is a poset $(D, \leq)$ satisfying particular axioms (see below for a precise definition). Domain equations such as $D \simeq (D \to D)$ have non trivial solutions when working with *continuous* functions.

The following definitions should be read with the following intuitions. Each point of a Scott domain D represents some amount of information. The elements of D are ordered by the partiality order: $x \leq y$ means that x is less defined than y. The least element $\bot$ denotes thus the lack of observable information, which corresponds in computer science to non-termination. The restriction to continuous functions (that makes the things work) that is, to functions satisfying the monotonicity as well as the finiteness property ensure respectively that we cannot extract information from non-termination (the halting

problem) and that a finite piece of output is only produced by a finite piece of input (commutation with directed limits).

Definition 7.6 (Directed set — cpo) *Given a partial order set* $(\mathsf{D}, \leq)$ *a non-empty subset* $\Delta \subseteq \mathsf{D}$ *is called directed if for all* $\mathsf{x}, \mathsf{y} \in \Delta$ *there exists a* $\mathsf{z} \in \Delta$ *such that* $\mathsf{x} \leq \mathsf{z}$ *and* $\mathsf{y} \leq \mathsf{z}$.

A cpo $(\mathsf{D}, \leq)$ *is a partial order with a least element denoted* $\bot_{\mathsf{D}}$ *and such that each directed set* $\Delta \subseteq \mathsf{D}$ *has a least upper bound denoted* $\bigvee \Delta$.

Definition 7.7 (Compact—Algebraic cpo) *An element* $\mathsf{d} \in \mathsf{D}$ *is called compact if for each directed set* $\Delta \subseteq \mathsf{D}$ *the following implication holds:*

$$\mathsf{d} \leq \bigvee \Delta \Rightarrow \exists \mathsf{x} \in \Delta \quad \mathsf{d} \leq \mathsf{x}$$

We write $\mathcal{K}(\mathsf{D})$ *for the set of all compact elements of* D.

A cpo $(\mathsf{D}, \leq)$ *is called algebraic is for all elements* $\mathsf{x} \in \mathsf{D}$ *the set* $\{\mathsf{d} \in \mathcal{K}(\mathsf{D}) | \mathsf{d} \leq \mathsf{x}\}$ *is directed and has least upper bound* x.

Definition 7.8 (Scott domain) *A Scott domain is an algebraic cpo* $(\mathsf{D}, \leq)$ *such that every set bounded* $\mathsf{B} \subseteq \mathsf{D}$ *has a least upper bound.*

Definition 7.9 (Monotonic — Continuous) *Let* $(\mathsf{D}, \leq)$ *and* $(\mathsf{D}', \leq')$ *be cpos. A function* $\mathsf{f} : \mathsf{D} \to \mathsf{D}'$ *is called monotonic if*

$$\forall \mathsf{x}, \mathsf{y} \in \mathsf{D} \quad \mathsf{x} \leq \mathsf{y} \Rightarrow \mathsf{f}(\mathsf{x}) \leq' \mathsf{f}(\mathsf{y})\,.$$

It is continuous if moreover it is finite that is, for all directed subset Δ *of* D *we have*

$$\mathsf{f}(\bigvee \Delta) = \bigvee \mathsf{f}(\Delta)$$

Theorem 7.10 *The category* **SCOTT** *of Scott domains and continuous functions is a Cartesian closed category with the following structure. For two Scott domains* $(\mathsf{D}, \leq_{\mathsf{D}})$ *and* $(\mathsf{E}, \leq_{\mathsf{E}})$*, we define*

- *the Scott domain* $(\mathsf{D} \times \mathsf{E}, \leq_{\times})$ *where* $\mathsf{D} \times \mathsf{E} = \{(\mathsf{x}, \mathsf{y}) \mid \mathsf{x} \in \mathsf{D}$ *and* $\mathsf{y} \in \mathsf{E}\}$ *and* $(\mathsf{x}, \mathsf{y}) \leq (\mathsf{x}', \mathsf{y}')$ *if* $(\mathsf{x} \leq_{\mathsf{D}} \mathsf{x}', \mathsf{y} \leq_{\mathsf{E}} \mathsf{y}')$.
- *the Scott domain* $(\mathsf{D} \Rightarrow \mathsf{E}, \leq_{\Rightarrow})$ *where* $\mathsf{D} \Rightarrow \mathsf{E}$ *is the set of continuous functions from* D *to* E *and* $\mathsf{f} \leq_{\Rightarrow} \mathsf{g}$ *if for all* $\mathsf{x} \in \mathsf{D}$ *we have* $\mathsf{f}(\mathsf{x}) \leq_{\mathsf{E}} \mathsf{g}(\mathsf{x})$.

Theorem 7.11 *There exist reflexive objects in the category* **SCOTT**.

The historical reflexive objects are the well-known D_∞-models. We refer to [AC98] for their constructions.

7.4 Completeness

The completeness result mentioned in the section 7.2 is an immediate consequence of a more general result, which says that for each λ-theory $\mathcal{T}$, there is a reflexive object that captures the equational axioms of $\mathcal{T}$ exactly.

Definition 7.12 (λ-theory) *A* λ-theory *is an equivalence relation $\mathcal{T}$ over the set Λ of open λ-terms (taken up to α-conversion) which is closed under the following rules:*

$$\frac{}{(\lambda x\,.\,M) =_{\mathcal{T}} M\{x := N\}}$$

$$\frac{M =_{\mathcal{T}} M'}{MN =_{\mathcal{T}} M'N} \qquad \frac{N =_{\mathcal{T}} N'}{MN =_{\mathcal{T}} MN'} \qquad \frac{M =_{\mathcal{T}} M'}{\lambda x\,.\,M =_{\mathcal{T}} \lambda x\,.\,M'}$$

As we already mentioned above, any reflexive object $(D, \mathrm{lam}, \mathrm{app})$ defines a λ-theory written $=_D$ and defined as follows

$$M =_D M' \qquad \text{iff} \qquad [\![M]\!]^D = [\![M']\!]^D$$

for all $M, M' \in \Lambda$. The interesting point is that we can define a 'syntactic' category in which each λ-theory is represented by a reflexive object. This category is the category based on the following notion of λ-per. Let us first recall that a partial equivalence relation (per) on Λ is a symmetric and transitive relation on Λ.

Definition 7.13 (λ-per) *A* λ-per *is a partial equivalence relation $A \subset \Lambda^2$ such that $\{=_\beta\} \circ A \subset A$, that is, such that $(M, M') \in A$ and $M' =_\beta M''$ entail $(M, M'') \in A$ for all $M, M', M'' \in \Lambda$.*

Since a per is not required to be reflexive, it is natural to define the domain of a per A as follows.

Definition 7.14 (Domain) *The domain of a per is defined by*

$$\mathrm{dom}(A) \quad = \quad \{M \in \Lambda \mid (M, M) \in A\}\,.$$

Let A be a per, we write $M \in A$ when $M \in \mathrm{dom}(A)$. It is straightforward to check that $A \subset \mathrm{dom}(A)^2$, and that the relation A is but a total equivalence relation on $\mathrm{dom}(A)$. Conversely, any equivalence relation defined on a subset of Λ is a per on Λ, which gives an equivalent definition to the notion of per. Moreover, the class of all pers (on Λ) is closed under arbitrary intersection. The following lemma defines the realisability arrow over λ-pers.

Lemma 7.15 *If A and B are pers, then the following binary relation is also a per.*

$$A \to B \quad = \quad \{(M_1, M_2) \in \Lambda^2 \mid \forall (N_1, N_2) \in A \quad (M_1N_1, M_2N_2) \in B\}$$

In this framework, a λ-per is but a per which is compatible with the β-conversion equivalence relation. As for the class of pers, the class of λ-pers is closed under arbitrary intersection as well as it is closed under the realisability arrow $(A, B) \mapsto (A \to B)$. In particular, the per $A \to B$ is a λ-per as soon as A and B are λ-pers. In what follows, we will frequently use λ-pers of the form

$$N \to A \quad = \quad \{(N, N)\} \to A$$

where N is a fixed term and A an arbitray λ-per.

Definition 7.16 *The category* **λPER** *is defined as follows.*

- *The objects of* **λPER** *are* λ*-pers.*
- *Given two* λ*-PERs* A *and* B*, the hom-set* $\mathbf{\lambda PER}[A; B]$ *is defined by*

$$\mathbf{\lambda PER}[A; B] \quad = \quad \mathrm{dom}(A \to B)/{\approx_{A \to B}}$$

- *Identity and composition are defined for all objects* $A, B, C \in \mathbf{\lambda PER}$ *and for all arrows* $M \in \mathbf{\lambda PER}[B; C]$ *and* $N \in \mathbf{\lambda PER}[A; B]$ *by*

$$\begin{array}{rcll} \mathrm{id}_A & = & \lambda x\,.\,x & \in \ \mathbf{\lambda PER}[A; A] \\ M \circ N & = & \lambda x\,.\,M(Nx) & \in \ \mathbf{\lambda PER}[A; C] \end{array}$$

  (taking the corresponding operations on equivalence classes).

We can check that:

Proposition 7.17 *The category* **λPER** *is a ccc with the following structure.*

- $A \times B = (\mathbf{true} \to A) \cap (\mathbf{false} \to B)$ *and* $1 = \Lambda^2$
- $\pi_1 = \lambda p\,.\,p\ \mathbf{true}$, $\pi_2 = \lambda p\,.\,p\ \mathbf{false}$ *and* $\langle M; N\rangle = \lambda x p\,.\,p\,(Mx)\,(Nx)$
- $B^A = A \to B$
- $\mathrm{ev} = \lambda p\,.\,(\pi_1\,p)\,(\pi_2\,p)$ *and* $\Lambda(M) = \lambda x y\,.\,M\,(\lambda p\,.\,pxy)$.

Noticing that any λ-theory $\mathcal{T} \subset \Lambda^2$ is a λ-per of domain Λ, we can easily check that:

Proposition 7.18 (λ-theories as reflexive objects) — *For all* λ*-theories* $\mathcal{T}$*, the triple* $(\mathcal{T}, \lambda xy\,.\,xy, \lambda x\,.\,x)$ *is a reflexive object in the category* **λPER**.

Notice that the embedding-retraction pair $(\mathrm{lam}, \mathrm{app}) = (\lambda xy\,.\,xy, \lambda x\,.\,x)$ does not depend on the λ-theory $\mathcal{T}$. Consequently, the denotation $[\![M]\!]^{\mathcal{T}}_{\ell}$ of a term M (with $FV(M) \subset \ell$) does not depend on $\mathcal{T}$ too. This denotation is actually characterized by the following lemma:

Lemma 7.19 *Let $\mathcal{T}$ be a λ-theory (that we also consider as a reflexive object of* **λPER***). For all lists of variables $\ell = (x_1, \ldots, x_n)$ and for all terms M such that $FV(M) \subset \ell$, one has*

$$[\![M]\!]^{\mathcal{T}}_{\ell} \quad =_{\beta} \quad \lambda z \,.\, M\{x_1 := \pi^{x_1}_{\ell}(z); \ldots; x_n := \pi^{x_n}_{\ell}(z)\}\,.$$

We can now easily conclude that:

Proposition 7.20 *Let $\mathcal{T}$ be a λ-theory (that we also consider as a reflexive object of* **λPER***). For all lists of variables ℓ, and for all terms M_1 and M_2 such that $FV(M_1) \cup FV(M_2) \subset \ell$, one has*

$$[\![M_1]\!]^{\mathcal{T}}_{\ell} = [\![M_2]\!]^{\mathcal{T}}_{\ell} \qquad \textit{iff} \qquad M_1 =_{\mathcal{T}} M_2\,.$$

Conclusion

In this chapter, we recall the sound and complete categorical semantics for the λ-calculus. In the following chapters, we extend this result to the parallel λ-calculus.

Chapter 8

Categorical semantics of the parallel λ-calculus

Context The starting point of this work was the semantical study of the ρ-calculus. A Scott semantics revealed the similarity between term collections of the ρ-calculus interpreted as the binary join and Boudol's parallel construct of the parallel λ-calculus [Bou94] (up to the fact that term collections are not systematically required to be associative, commutative and/or idempotent).

Formally, the parallel λ-calculus is obtained by extending the pure λ-calculus with a binary operator $M /\!/ N$, that intuitively represents the parallel execution of M and N. The parallel λ-calculus adds to the equational theory of the pure λ-calculus the single equational axiom (δ) expressing the distributivity of function application w.r.t. parallel aggregation.

The parallel λ-calculus was initially introduced as a tool to study full-abstraction of the interpretation of λ-terms in Scott domains. In this framework, Boudol extended the interpretation of pure λ-terms to the parallel construction using the join operation.

Scott semantics is well-suited to achieve full-abstraction but it is not sufficient to capture neither the basic equational theory of the calculus nor many interesting extensions of it – typically when dealing with extensionality. In the same way, interpreting the parallel operator as the binary join automatically validates associativity, commutativity and idempotence, although on a purely syntactical level, these equations are clearly independent from the basic equations (β) and (δ). For these reasons, there is a need for a more general and more modular semantics.

The semantical study of the ρ-calculus also pointed out many interesting problems related to the interaction between a mechanism of pattern-matching and the parallel construct. These problems helped us to grasp the importance of linear terms which play a central rôle in the proof of completeness (Boudol's shift towards the λ-calculus with resources [BCL99] seems to be motivated by similar reasons).

Contributions We first give a modular presentation of the parallel λ-calculus: the core calculus consists of the (β) and the (δ) equational axioms while some extensions are possible including some equational axioms for extensionnality and some equational axioms expressing associativity, commutativity and idempotence of the parallel operator. This gives 24 variants of the calculus (see below for more details).

We define a sound categorical semantics for the parallel λ-calculus, based on a notion

of aggregation monad which is modular w.r.t. the 24 equational theories of the calculus. Particular cases of this semantics are given by Boudol's models.

The given semantics is complete in a very strong sense: for each congruence extending one of the 24 equational theories of the parallel λ-calculus there exists a reflexive object, adapted to the considered equational theory, such that the equational theory induced by the interpretation of parallel λ-terms in this object is exactly this congruence.

To prove completeness, we introduce a category of partial equivalence relations adapted to parallelism, in which any extension of the basic equational theory of the calculus is induced by some model (see Chapter 9).

We also present abstract methods to construct models of the parallel λ-calculus in categories where particular equations have solutions, such as the category of Scott domains and its variants (see Chapter 10).

We think that the categorical semantics introduced here will contribute to a better understanding of the interaction between pattern-matching and the parallel construct, and thus will constitute a significant step towards a denotational semantics for the ρ-calculus.

Outline of the chapter In Section 8.1, we present the syntax and equational theory of the parallel λ-calculus and define the notion of linear terms. In Section 8.2, we recall some basic definitions for monads and algebras. In Section 8.3, we introduce and study the notion of aggregation monads. In Section 8.4, we first give the definition of the categorical models of the parallel λ-calculus and we then define the interpretation of parallel λ-terms in this model and we show its soundness. The last section (Section 8.5) gives examples of such models in Scott domains.

8.1 The parallel λ-calculus

We first give a detailed presentation of the syntax and the equational theory of the parallel λ-calculus which is summarized in Figure 8.1 page 158.

8.1.1 The core calculus

The parallel λ-calculus is obtained by extending the pure λ-calculus with a binary operator $M /\!/ N$ representing the parallel execution of M and N. Formally, the terms of the parallel λ-calculus are given by

$$M, N \quad ::= \quad x \mid \lambda x.M \mid MN \mid M /\!/ N$$

and the corresponding equational theory is defined from the two equational axioms

$$(\beta) \qquad (\lambda x.M)N = M\{x := N\}$$
$$(\delta) \qquad (M_1 /\!/ M_2)N = M_1N /\!/ M_2N$$

In the following, we will consider parallel λ-terms from the point of view of equational reasoning rather than from the point of view of reduction. However, both equational

axioms can also be presented as rewrite rules (orienting them from left to right), and it can be checked that the rewrite systems induced by β, δ and $\beta\delta$ are confluent.

8.1.2 Extensions of the equational theory

In many situations, it is desirable to extend the core calculus with one of the following equational axioms:

$$(\epsilon) \qquad \lambda x\,.\,(M_1 /\!\!/ M_2) = (\lambda x\,.\,M_1) /\!\!/ (\lambda x\,.\,M_2)$$
$$(\eta) \qquad \lambda x\,.\,Mx = M \qquad (\text{if } x \notin FV(M))$$

Again, both equation axioms can be presented as reduction rules, orienting them from left to right. Notice that in the presence of equational axiom δ, the equational axiom η subsumes ϵ, that is: $\epsilon \subset \delta\eta$ (equationally). In fact, we can check that

$$\begin{array}{rll} (\lambda x\,.\,M_1) /\!\!/ (\lambda x\,.\,M_2) & =_\eta & \lambda y\,.\,\big((\lambda x\,.\,M_1) /\!\!/ (\lambda x\,.\,M_2)\big)y \\ & =_\delta & \lambda y\,.\,\big((\lambda x\,.\,M_1)y /\!\!/ (\lambda x\,.\,M_2)y\big) \\ & =_\beta & \lambda y\,.\,\big(M_1\{x := y\} /\!\!/ M_2\{x := y\}\big) \\ & =_\alpha & \lambda x\,.\,(M_1 /\!\!/ M_2) \end{array}$$

From the point of view of the corresponding rewrite systems, both reduction rules δ and η make a critical pair which is closed with the ϵ-reduction rule.

Finally, the parallel λ-calculus can be extended with any combination of the three equational axioms expressing associativity, commutativity and idempotence of the parallel operator:

$$(A) \qquad (M_1 /\!\!/ M_2) /\!\!/ M_3 = M_1 /\!\!/ (M_2 /\!\!/ M_3)$$
$$(C) \qquad M_1 /\!\!/ M_2 = M_2 /\!\!/ M_1$$
$$(I) \qquad M /\!\!/ M = M$$

In what follows, most definitions will be modularized in order to handle the 24 variants of the calculus obtained by combining each of the 3 basic calculi $\beta\delta$, $\beta\delta\epsilon$ and $\beta\delta\eta$ with the $2^3 = 8$ possible combinations of A, C, and I.

8.1.3 Linearity

Definition 8.1 (Linear term) *Let $\mathcal{T}$ be an equational theory of the parallel λ-calculus which contains at least β and δ. We say that a term M is* linear *(w.r.t. its first argument) in the theory $\mathcal{T}$ when*

$$M(N_1 /\!\!/ N_2) =_{\mathcal{T}} MN_1 /\!\!/ MN_2$$

for all terms N_1, N_2.

By substitutivity, it is equivalent to say that $M(x /\!\!/ y) =_{\mathcal{T}} Mx /\!\!/ My$, where x and y are fresh variables.

When a term M is linear in the theory $\beta\delta$, we simply say that M is linear. Examples of linear terms are the identity term $\lambda x\,.\,x$ and more generally all the terms of the form $\lambda x\,.\,x\vec{N}$ where $x \notin FV(\vec{N})$ (i.e. tuples).

The core calculus

$$M, N \quad ::= \quad x \mid \lambda x\,.\,M \mid MN \mid M /\!\!/ N$$

(β)	$(\lambda x\,.\,M)N$	$=$	$M\{x := N\}$
(δ)	$(M_1 /\!\!/ M_2)N$	$=$	$M_1N /\!\!/ M_2N$

Extensions of the equational theory

(ϵ)	$\lambda x\,.\,(M_1 /\!\!/ M_2)$	$=$	$\lambda x\,.\,M_1 /\!\!/ \lambda x\,.\,M_2$
(η)	$\lambda x\,.\,Mx$	$=$	$M \quad (\text{if } x \notin FV(M))$
(A)	$(M_1 /\!\!/ M_2) /\!\!/ M_3$	$=$	$M_1 /\!\!/ (M_2 /\!\!/ M_3)$
(C)	$M_1 /\!\!/ M_2$	$=$	$M_2 /\!\!/ M_1$
(I)	$M /\!\!/ M$	$=$	M

Figure 8.1: The parallel λ-calculus

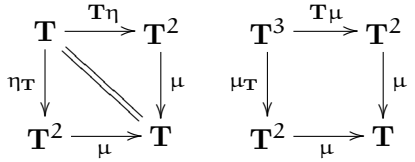
It is important not to confuse the notion of linearity given here with the more syntactic notion of linear λ-terms (referred below as syntactic linearity) that says that a λ-term is said to be linear when all its free and bound variables occur exactly once. In particular, there is no inclusion between both notions:

- The term $\lambda x\,.\,x(\lambda y\,.\,yy)$ is linear but not syntactically linear,
- The term $\lambda xy\,.\,yx$ is syntactically linear but not linear.

8.2 Monads and algebras

We first give the categorical definition of a monad. Then, we give some intuitions coming from functional programming monads.

Definition 8.2 (Monad) *A* monad *over a category $\mathfrak{C}$ is a triple $(\mathbf{T}, \eta, \mu)$ formed by an endofunctor $\mathbf{T}$ in the category $\mathfrak{C}$ equipped with two natural transformations $\eta : \mathbf{I} \to \mathbf{T}$ and $\mu : \mathbf{T}^2 \to \mathbf{T}$ such that the following diagrams commute:*



$$\mu_X \circ \mathbf{T}\eta_X = \mathrm{id}_{\mathbf{T}X} \qquad (\textit{mon-1})$$
$$\mu_X \circ \eta_{\mathbf{T}X} = \mathrm{id}_{\mathbf{T}X} \qquad (\textit{mon-2})$$
$$\mu_X \circ \mathbf{T}\mu_X = \mu_X \circ \mu_{\mathbf{T}X} \qquad (\textit{mon-3})$$

To make the definition complete, let us recall the axioms which express the functoriality of $\mathbf{T}$ and the naturality of the transformations η and μ:

$$\begin{array}{rcl} \mathbf{T}\mathrm{id}_X & = & \mathrm{id}_{\mathbf{T}X} \\ \mathbf{T}(g \circ f) & = & \mathbf{T}g \circ \mathbf{T}f \end{array} \qquad \begin{array}{rcll} \eta_Y \circ f & = & \mathbf{T}f \circ \eta_X & \text{(nat-}\eta\text{)} \\ \mu_Y \circ \mathbf{T}^2 f & = & \mathbf{T}f \circ \mu_X & \text{(nat-}\mu\text{)} \end{array}$$

In what follows, we identify the monad with its underlying functor, and simply write 'the monad $\mathbf{T}$' instead of 'the monad $(\mathbf{T}, \eta, \mu)$'.

The definition of a monad is based upon the notion of functoriality, natural transformations that can be modelled in functional languages (Walder's starting point [Wad93] was to adapt Moggi's ideas [Mog89, Mog91] in order to use monads to structure the semantics of computations).

A functor is based upon an object mapping and a morphism mapping. The object part of a functor is represented in a functional language by a type-constructor. For example, if an object A has type α, then the object $\mathbf{T}A$ has type $M\alpha$ where M is a type constructor (typically the list type constructor). The morphism mapping of the functor is modelled by a function $\mathrm{map} : (\alpha \to \beta) \to (M\alpha \to M\beta)$ and then the corresponding morphism of f via the functor $\mathbf{T}$ (that is $\mathbf{T}f$) is modelled by $\mathrm{map}\, f$.

Moreover, a natural transformation can be thought as a family of arrows from each object in a category to objects in an over category. It is thus somehow similar to a polymorphic function. Therefore, the natural transformations η and μ can be written as the polymorphic functions $\mathrm{unit} : \alpha \to M\alpha$ and $\mathrm{join} : M(M\alpha) \to M\alpha$ such that

$$\begin{array}{lcl} \mathrm{join} \circ \mathrm{unit} & = & \mathrm{id} \\ \mathrm{join} \circ (\mathrm{map}\, \mathrm{unit}) & = & \mathrm{id} \\ \mathrm{join} \circ (\mathrm{map}\, \mathrm{join}) & = & \mathrm{join} \circ \mathrm{join} \end{array}$$

Note that functional programming monads are in practice based on the (equivalent) notion of Kleisli triples.

When working with a monad $\mathbf{T}$ defined in a Cartesian category, it is usually necessary to make the following stronger assumption on $\mathbf{T}$.

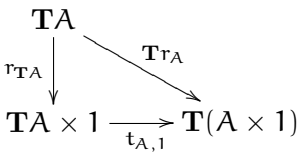
Definition 8.3 (Strong monad) *A* strong monad *over a Cartesian category $\mathfrak{C}$ is a monad $\mathbf{T} = (\mathbf{T}, \eta, \mu)$ equipped with a (bi-)natural transformation*

$$t_{A,B} \quad : \quad \mathbf{T}A \times B \to \mathbf{T}(A \times B)$$

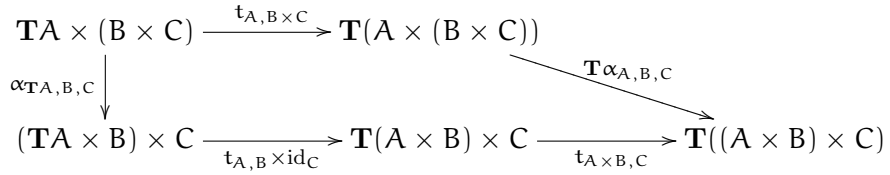
such that the diagrams given in Figure 8.2 commute (where $\alpha_{A,B,C}$ is the natural isomorphism from $A \times (B \times C)$ to $(A \times B) \times C$ and r_A is the natural isomorphism from A to $A \times 1$.

Intuitively, the transformation t distributes the second component of its input to all the elements of the first component—thinking of $\mathbf{T}A$ as a type of lists or sets.

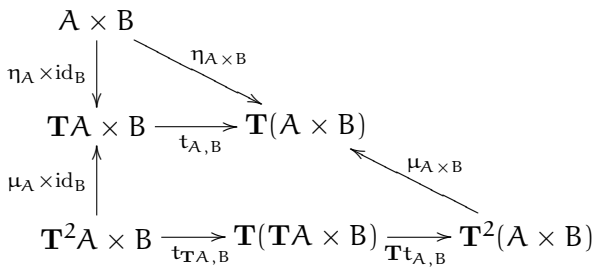
We conclude this section by the definition of an algebra.



$$t_{A,1} \circ r_{\mathbf{T}A} = \mathbf{T}r_A \qquad \text{(str-1)}$$



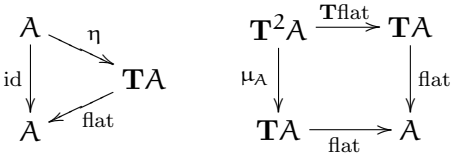
$$\mathbf{T}\alpha_{A,B,C} \circ t_{A,B\times C} = t_{A\times B,C} \circ (t_{A,B} \times \mathrm{id}_C) \circ \alpha_{\mathbf{T}A,B,C} \qquad \text{(str-}\alpha\text{)}$$



$$t_{A,B} \circ (\eta_A \times \mathrm{id}_B) = \eta_{A\times B} \qquad \text{(str-}\eta\text{)}$$
$$\mu_{A\times B} \circ \mathbf{T}t_{A,B} \circ t_{\mathbf{T}A,B} = t_{A,B} \circ (\mu_A \times \mathrm{id}_B) \qquad \text{(str-}\mu\text{)}$$

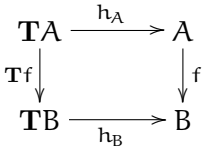
Figure 8.2: Diagrams of strong monadicity

Definition 8.4 (Algebra) *Let* $\mathbf{T}$ *be a monad over a category* $\mathfrak{C}$*. A* $\mathbf{T}$-algebra *is an object* $A \in \mathbf{Obj}\mathfrak{C}$ *equipped with an arrow* $\mathrm{flat} \in \mathfrak{C}[T(A); A]$ *such that the following diagrams commute:*



$$\begin{array}{rcll} \mathrm{flat} \circ \eta & = & \mathrm{id}_A & (\textit{alg-1}) \\ \mathrm{flat} \circ T\mathrm{flat} & = & \mathrm{flat} \circ \mu & (\textit{alg-2}) \end{array}$$

A morphism $f : \langle A, h_A \rangle \to \langle B, h_B \rangle$ *of* $\mathbf{T}$*-algebras is an arrow* $f : A \to B$ *such that the following diagram commutes*



Notice the similarity between the second diagram above (the one that corresponds to (agg-2)) and the second diagram of monadicity. Actually, any pair of the form $(\mathbf{T}(X), \mu_X)$ is a $\mathbf{T}$-algebra.

8.3 Aggregation monads

We now present a notion of *aggregation monad* which is the categorical counterpart of the syntactical notion of parallel execution. We use here the terminology of 'aggregation' to emphasize the fact that this notion exists independently from the properties of associativity, commutativity and idempotence that are usually associated with the idea of parallelism (however, we keep the name of parallel λ-calculus, for obvious historical reasons.)

8.3.1 Notion of aggregation

Definition 8.5 (Aggregation monad) *Let* $\mathfrak{C}$ *be a Cartesian category. A* notion of aggregation *(in* $\mathfrak{C}$*) is a monad* $\mathbf{T} = (\mathbf{T}, \eta, \mu)$ *equipped with a natural transformation*

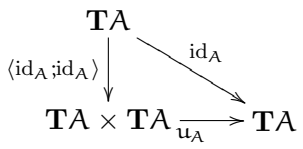
$$\mathfrak{u} \quad : \quad \mathbf{T}X \times \mathbf{T}X \quad \Rightarrow \quad \mathbf{T}X$$

such that the following diagram commutes

$$\mu_A \circ \mathfrak{u}_{\mathbf{T}A} \quad = \quad \mathfrak{u}_A \circ (\mu_A \times \mu_A) \qquad (\textit{agg-}\mu)$$

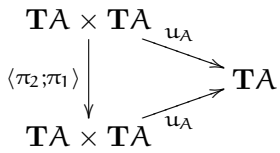
An aggregation monad *is simply a monad equipped with a notion of aggregation. We say that a notion of aggregation* $\mathfrak{u}$ *is* associative *(A),* commutative *(C) or* idempotent *(I) depending on the corresponding diagram commutes in Figure 8.3 (where* α *denotes the associativity isomorphism of* $\times$*).*

Idempotence



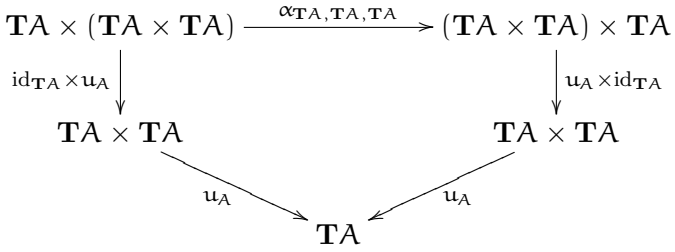
$$u_A \circ \langle id_A; id_A \rangle = id_A \qquad \text{(I-u)}$$

Commutativity



$$u_A \circ \langle \pi_2; \pi_1 \rangle = u_A \qquad \text{(C-u)}$$

Associativity



$$u_A \circ (id_{\mathbf{T}A} \times u_A) = u_A \circ (u_A \times id_{\mathbf{T}A}) \circ \alpha_{\mathbf{T}A,\mathbf{T}A,\mathbf{T}A} \qquad \text{(A-u)}$$

Figure 8.3: ACI diagrams of aggregation

The diagram (agg-μ) describes the interaction between the notion of aggregation $\mathfrak{u}$ and μ. A useful consequence of this diagram is that the arrow $\mathfrak{u}_A$ can always be defined in terms of the arrow $\mathfrak{u}_{\mathbf{T}A}$. This can also be seen as a description of the interaction between $\mathfrak{u}$ and η.

Lemma 8.6 *If* $(\mathbf{T}, \mathfrak{u})$ *is a notion of aggregation then the following diagram commutes*

$$\begin{array}{ccc} \mathbf{T}A \times \mathbf{T}A & \xrightarrow{\mathfrak{u}_A} & \mathbf{T}A \\ {\scriptstyle \eta_{\mathbf{T}A} \times \eta_{\mathbf{T}A}} \downarrow & & \uparrow {\scriptstyle \mu_A} \\ \mathbf{T}^2A \times \mathbf{T}^2A & \xrightarrow[\mathfrak{u}_{\mathbf{T}A}]{} & \mathbf{T}^2A \end{array} \qquad \mu_A \circ \mathfrak{u}_{\mathbf{T}A} \circ (\eta_{\mathbf{T}A} \times \eta_{\mathbf{T}A}) \;=\; \mathfrak{u}_A \quad (\textit{agg}\text{-}\eta)$$

Proof. It is easy to check that

$$\begin{array}{lcll} \mu_A \circ \mathfrak{u}_{\mathbf{T}A} \circ (\eta_{\mathbf{T}A} \times \eta_{\mathbf{T}A}) & = & \mathfrak{u}_A \circ (\mu_A \times \mu_A) \circ (\eta_{\mathbf{T}A} \times \eta_{\mathbf{T}A}) & (\text{agg-}\mu) \\ & = & \mathfrak{u}_A & (\text{mon-2}) \end{array}$$

□

8.3.2 Typical examples

Typical notions of aggregation monads are the following.

In the category of sets

- The powerset monad with union (ACI)
- The multiset monad with multi-union (AC)
- The list monad with concatenation (A)
- The free group monad with composition (A)

In the category of Scott domains

- The lower powerdomain monad with join (ACI)
- The upper powerdomain monad with meet (ACI)

In Ab**-categories (with finite products),** the fundamental aggregation monad is the binary sum, given as the identity monad equipped with the arrow

$$\pi_1 + \pi_2 \;:\; A \times A \to A \qquad (\text{ACI})$$

8.3.3 Algebras and linear morphisms

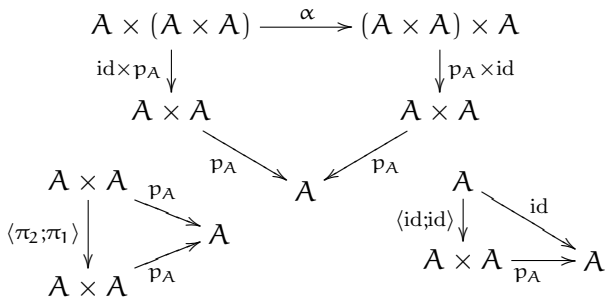
Let $\mathfrak{C}$ be a Cartesian category equipped with an aggregation monad $\langle \mathbf{T}, \mathfrak{u} \rangle$.

Definition 8.7 (Aggregation operator) *Each* $\mathbf{T}$*-algebra* $\langle A, h_A \rangle$ *can be given an* aggregation operator $p_A : A \times A \to A$ *defined by*

$$\begin{array}{ccc} A \times A & \xrightarrow{p_A} & A \\ {\scriptstyle \eta_A \times \eta_A}\downarrow & & \uparrow{\scriptstyle h_A} \\ \mathbf{T}(A) \times \mathbf{T}(A) & \xrightarrow[\mathfrak{u}_A]{} & \mathbf{T}(A) \end{array} \qquad p_A \;=\; h_A \circ \mathfrak{u}_A \circ (\eta_A \times \eta_A)$$

The aggregation operator p_A inherits the properties of associativity, commutativity and idempotence from the underlying aggregation monad $\langle \mathbf{T}, \mathfrak{u} \rangle$ in the sense of the following lemma.

Lemma 8.8 *If the aggregation monad* $\langle \mathbf{T}, \mathfrak{u} \rangle$ *is associative, commutative, and/or idempotent, then for all* $\mathbf{T}$*-algebras* $\langle A, h_A \rangle$ *the aggregation operator* $p_A : A \times A \to A$ *is associative, commutative, and/or idempotent in the sense of the following diagrams:*



Proof. First, let us suppose that $\mathfrak{u}$ is idempotent that is, it satisfies (I-$\mathfrak{u}$). Then we have:

$$\begin{array}{lcll} h_A \circ \mathfrak{u}_A \circ (\eta_A \times \eta_A) \circ \langle id_A; id_A \rangle & = & h_A \circ \mathfrak{u}_A \circ \langle id_{\mathbf{T}A}; id_{\mathbf{T}A} \rangle \circ \eta_A & \\ & = & h_A \circ id_{\mathbf{T}A} \circ \eta_A & \text{(I-}\mathfrak{u}\text{)} \\ & = & id_A & \text{(alg-1)} \end{array}$$

This shows that p_A is idempotent.

We suppose now that $\mathfrak{u}$ is commutative that is, it satisfies (C-$\mathfrak{u}$). Then we have:

$$\begin{array}{lcll} h_A \circ \mathfrak{u}_A \circ (\eta_A \times \eta_A) \circ \langle \pi_2; \pi_1 \rangle & = & h_A \circ \mathfrak{u}_A \circ \langle \pi_2; \pi_1 \rangle \circ (\eta_A \times \eta_A) & \\ & = & h_A \circ \mathfrak{u}_A \circ (\eta_A \times \eta_A) & \text{(C-}\mathfrak{u}\text{)} \end{array}$$

This shows that p_A is commutative.

We suppose now that $\mathfrak{u}$ is associative that is, it satisfies (A-$\mathfrak{u}$). The proof of associativity of p_A is given in Figure 8.4. The innermost diagram is the associativity of p_A. To show that this diagram commutes we prove the commutativity of the other ones. The outermost diagram is given by (A-$\mathfrak{u}$). Diagram (1) is true by the naturality of η. Diagrams (2) and (3) are true by definition of p_A for the first component of the product

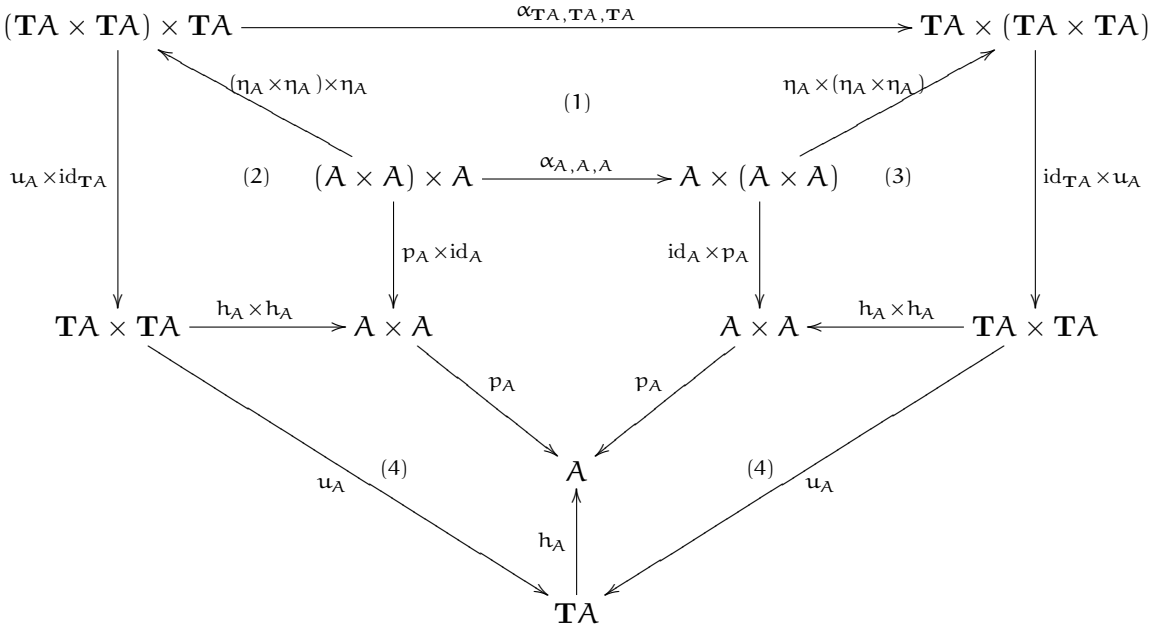


Figure 8.4: Associativity of the aggregation operator

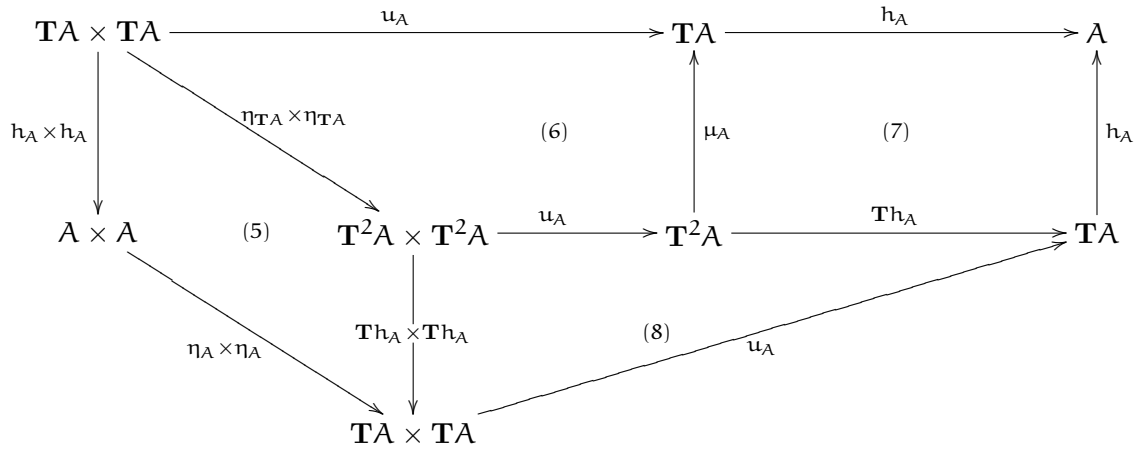


Figure 8.5: Commutation of the notion of aggregation and the aggregation operator

and by (alg-1) for the second one. The proof of diagrams (4) is a bit more elaborated and is detailed in Figure 8.5. The diagram (5) is given by the naturality of η. The diagram (6) was proved before and is known as (agg-η). The diagram (7) is given by (alg-2). The diagram (8) is given by the naturality of u.

□

A consequence of diagram (agg-μ) is that morphisms of algebras are linear in the sense that they commute with the aggregation operator:

Lemma 8.9 *If* $\langle A, h_A\rangle$ *and* $\langle B, h_B\rangle$ *are* $\mathbf{T}$*-algebras, then for all morphisms of* $\mathbf{T}$*-algebras* $f : \langle A, h_A\rangle \to \langle B, h_B\rangle$ *the following diagram commutes*

$$\begin{array}{ccc} A \times A & \xrightarrow{p_A} & A \\ {\scriptstyle f\times f}\downarrow & & \downarrow{\scriptstyle f} \\ B \times B & \xrightarrow[p_B]{} & B \end{array}$$

Proof. The proof is given by the following diagram

$$\begin{array}{ccccccc} A \times A & \xrightarrow{\eta_A\times\eta_A} & \mathbf{T}A \times \mathbf{T}A & \xrightarrow{u_A} & \mathbf{T}A & \xrightarrow{h_A} & A \\ {\scriptstyle f\times f}\downarrow & (1) & \downarrow{\scriptstyle \mathbf{T}f\times\mathbf{T}f} & (2) & \downarrow{\scriptstyle \mathbf{T}f} & (3) & \downarrow{\scriptstyle f} \\ B \times B & \xrightarrow[\eta_B\times\eta_B]{} & \mathbf{T}B \times \mathbf{T}B & \xrightarrow[u_B]{} & \mathbf{T}B & \xrightarrow[h_B]{} & B \end{array}$$

where the diagram (1) commutes by the naturality of η, the diagram (2) commutes by the naturality of u and the diagram (3) commutes since f is a morphism of $\mathbf{T}$-algebra.

□

Remember that for all objects A, B and for all morphisms $f : A \to B$, the morphism $\mathbf{T}f$ is a morphism of algebras from $\langle \mathbf{T}A, \mu_A\rangle$ to $\langle \mathbf{T}B, \mu_B\rangle$.

8.3.4 Strong notion of aggregation

Definition 8.10 (Strong notion of aggregation) *A* strong notion of aggregation *is a notion of aggregation such that the underlying notion of monad is strong and that satisfies the following diagram:*

$$\begin{array}{ccc} (\mathbf{T}A \times \mathbf{T}A) \times B & \xrightarrow{\langle \pi_1\times id_B ; \pi_2\times id_B\rangle} & (\mathbf{T}A \times B) \times (\mathbf{T}A \times B) \\ {\scriptstyle u_A\times id_B}\downarrow & & \downarrow{\scriptstyle t_{A,B}\times t_{A,B}} \\ \mathbf{T}A \times B & & \mathbf{T}(A \times B) \times \mathbf{T}(A \times B) \\ & {\scriptstyle t_{A,B}}\searrow \quad \swarrow{\scriptstyle u_{A\times B}} & \\ & \mathbf{T}(A \times B) & \end{array}$$

$$u_{A\times B} \circ (t_{A,B} \times t_{A,B}) \circ \langle \pi_1 \times id_B ; \pi_2 \times id_B\rangle = t_{A,B} \circ u_A \times id_B \quad (agg\text{-}t)$$

A strong aggregation monad *is simply a strong monad equipped with a strong notion of aggregation.*

In particular, all the examples of aggregation monads given in Section 8.3.2 are strong aggregation monads.

8.4 Model of the parallel λ-calculus

8.4.1 Definition

Definition 8.11 (Model of parallel λ-calculus) *A model of the parallel* λ*-calculus is a ccc* $\mathfrak{C}$ *equipped with a strong aggregation monad* $\langle \mathbf{T}, \mathfrak{u} \rangle$ *and a quadruple* $\langle \mathsf{D}, \mathrm{lam}, \mathrm{app}, \mathrm{flat} \rangle$ *such that*

- $\langle \mathsf{D}, \mathrm{lam}, \mathrm{app} \rangle$ *is a reflexive object of* $\mathfrak{C}$*;*
- $\langle \mathsf{D}, \mathrm{flat} \rangle$ *is a* $\mathbf{T}$*-algebra;*

and such that the following diagram commutes:

$$(\delta) \qquad \begin{array}{ccc} \mathbf{T}\mathsf{D} & \xrightarrow{\quad\mathrm{flat}\quad} & \mathsf{D} \\ {\scriptstyle \mathbf{T}\mathrm{app}}\downarrow & & \downarrow{\scriptstyle \mathrm{app}} \\ \mathbf{T}(\mathsf{D}^{\mathsf{D}}) & \xrightarrow[\Lambda(\mathrm{flat} \circ \mathbf{T}\mathrm{ev} \circ \mathrm{t})]{} & \mathsf{D}^{\mathsf{D}} \end{array}$$

where the bottom arrow is built by curryfying the following sequence of morphisms:

$$\mathbf{T}(\mathsf{D}^{\mathsf{D}}) \times \mathsf{D} \xrightarrow[\mathrm{t}]{} \mathbf{T}(\mathsf{D}^{\mathsf{D}} \times \mathsf{D}) \xrightarrow[\mathbf{T}\mathrm{ev}]{} \mathbf{T}(\mathsf{D}) \xrightarrow[\mathrm{flat}]{} \mathsf{D}$$

Definition 8.12 (ϵ/η/ACI-model) *We say that a model of the parallel* λ*-calculus is*

- *An* ϵ-model *when the following diagram commutes:*

$$(\epsilon) \qquad \begin{array}{ccc} \mathbf{T}\mathsf{D} & \xrightarrow{\quad\mathrm{flat}\quad} & \mathsf{D} \\ {\scriptstyle \mathbf{T}\mathrm{lam}}\uparrow & & \uparrow{\scriptstyle \mathrm{lam}} \\ \mathbf{T}(\mathsf{D}^{\mathsf{D}}) & \xrightarrow[\Lambda(\mathrm{flat} \circ \mathbf{T}\mathrm{ev} \circ \mathrm{t})]{} & \mathsf{D}^{\mathsf{D}} \end{array}$$

- *An* η-model *when* $\mathrm{lam} \circ \mathrm{app} = \mathrm{id}$ (η)*, that is, when the arrows* app *and* lam *are converse isomorphisms;*
- *Associative (A), commutative (C) or idempotent (I) when the underlying aggregation monad* $\langle \mathbf{T}, \mathfrak{u} \rangle$ *is.*

We have seen that in the presence of the equational axiom (δ), the equational axiom (η) subsumes (ϵ), that is $\epsilon \subseteq \delta\eta$. The definition of η and ϵ models is sound for this inclusion in the following sense.

(Var)	$[\![x]\!]_\ell$	$=$	π^x_ℓ
(Lam)	$[\![\lambda x\,.\,M]\!]_\ell$	$=$	$\mathrm{lam} \circ \Lambda([\![M]\!]_{\ell,x})$
(App)	$[\![MN]\!]_\ell$	$=$	$\mathrm{ev} \circ \langle \mathrm{app} \circ [\![M]\!]_\ell ; [\![N]\!]_\ell \rangle$
(Par)	$[\![M \mathbin{/\!/} N]\!]_\ell$	$=$	$\mathsf{par} \circ \langle [\![M]\!]_\ell ; [\![N]\!]_\ell \rangle$

Figure 8.6: Interpretation of parallel λ-terms

Lemma 8.13 *Any η-model of the parallel λ-calculus is also an ϵ-model.*

Proof. The diagram (δ) gives

$$\mathrm{app} \circ \mathrm{flat} \quad = \quad \Lambda(\mathrm{flat} \circ \mathbf{T}\mathrm{ev} \circ \mathsf{t}) \circ \mathbf{T}\mathrm{app}$$

That is

$$\underbrace{\mathrm{lam} \circ \mathrm{app}} \circ \mathrm{flat} \quad = \quad \mathrm{lam} \circ \Lambda(\mathrm{flat} \circ \mathbf{T}\mathrm{ev} \circ \mathsf{t}) \circ \mathbf{T}\mathrm{app}$$

Since by hypothesis, we consider an η-model we have

$$\mathrm{flat} \quad = \quad \mathrm{lam} \circ \Lambda(\mathrm{flat} \circ \mathbf{T}\mathrm{ev} \circ \mathsf{t}) \circ \mathbf{T}\mathrm{app}$$

from which we conclude that

$$\begin{aligned} \mathrm{flat} \circ \mathbf{T}\mathrm{lam} \quad &= \quad \mathrm{lam} \circ \Lambda(\mathrm{flat} \circ \mathbf{T}\mathrm{ev} \circ \mathsf{t}) \circ \underbrace{\mathbf{T}\mathrm{app} \circ \mathbf{T}\mathrm{lam}} \\ &= \quad \mathrm{lam} \circ \Lambda(\mathrm{flat} \circ \mathbf{T}\mathrm{ev} \circ \mathsf{t}) \end{aligned}$$

□

Definition 8.14 (Parallel operator) *Each model of the parallel λ-calculus comes with a* parallel operator $\mathsf{par} : D \times D \to D$ *given by*

$$\mathsf{par} \quad = \quad \mathfrak{p}_D \quad = \quad \mathrm{flat} \circ \mathfrak{u}_D \circ (\eta_D \times \eta_D)\,.$$

8.4.2 Interpreting parallel λ-terms

Let $\mathfrak{C}$ be a ccc with a strong aggregation monad $\langle \mathsf{T}, \mathfrak{u} \rangle$, and $\langle D, \mathrm{lam}, \mathrm{app}, \mathrm{flat} \rangle$ a model of the parallel λ-calculus in this category. Context (list of variables) and projections are defined in the same as in Section 7.2. Each parallel λ-term M whose free variables belong to a list ℓ is interpreted as an arrow $[\![M]\!]_\ell : D^\ell \to D$ which is defined by the equations given in Figure 8.6. The soundness of this interpretation relies on the following lemma.

Lemma 8.15 (Substitutivity) — *Given a list of variables ℓ and a variable x, then for all terms M and N such that $FV(M) \subset (\ell, x)$ and $FV(N) \subset \ell$, we have:*

$$[\![M\{x := N\}]\!]_\ell \quad = \quad [\![M]\!]_{\ell,x} \circ \langle \mathrm{id} ; [\![N]\!]_\ell \rangle$$

Definition 8.16 (Adapted model) *Let $\mathcal{T}$ denote one of the 24 equational theories obtained by combining the three basic theories $\beta\delta$, $\beta\delta\epsilon$, $\beta\delta\eta$ with all possible combinations of A, C and I. We say that the model D is* adapted *to the theory $\mathcal{T}$ when*

- *if $\mathcal{T}$ contains the equational axiom ϵ (resp. η), then D is an ϵ-model (resp. an η-model);*
- *if $\mathcal{T}$ contains the equational axiom A (resp. C, I), then the underlying aggregation monad $\langle \mathbf{T}, \mathfrak{u} \rangle$ is associative (resp. commutative, idempotent).*

Proposition 8.17 (Soundness) — *If the model is adapted to the theory $\mathcal{T}$, then for all lists of variables ℓ and for all terms M, M' whose free variables occur in ℓ, we have:*

$$M =_{\mathcal{T}} M' \quad \Rightarrow \quad [\![M]\!]_\ell = [\![M']\!]_\ell \,.$$

Proof. It suffices to check the equality for each equational axiom of $\mathcal{T}$. The soundness of the β rule is inherited from the soundness property for the λ-calculus.

To check that the δ-rule is sound we can first remark that the following diagram commutes

$$\begin{array}{ccc} \mathsf{D} \times \mathsf{D} & \xrightarrow{\quad \mathsf{par} \quad} & \mathsf{D} \\ {\scriptstyle \mathrm{app}\times\mathrm{app}}\downarrow & & \downarrow{\scriptstyle \mathrm{app}} \\ \mathsf{D}^{\mathsf{D}} \times \mathsf{D}^{\mathsf{D}} & \xrightarrow[\Lambda(\mathsf{par}\circ\langle \mathrm{ev}\circ(\pi_1\times\mathrm{id});\mathrm{ev}\circ(\pi_2\times\mathrm{id})\rangle)]{} & \mathsf{D}^{\mathsf{D}} \end{array}$$

We can check that

$$\begin{array}{lll} & \mathrm{app} \circ \mathsf{par} & \\ = & \mathrm{app} \circ \mathrm{flat} \circ \mathfrak{u} \circ (\eta \times \eta) & \\ = & \Lambda(\mathrm{flat} \circ \mathbf{T}\mathrm{ev} \circ \mathrm{t}) \circ \mathbf{T}\mathrm{app} \circ \mathfrak{u} \circ (\eta \times \eta) & (\delta) \\ = & \Lambda(\mathrm{flat} \circ \mathbf{T}\mathrm{ev} \circ \mathrm{t} \circ (\mathbf{T}\mathrm{app} \times \mathrm{id}) \circ (\mathfrak{u} \times \mathrm{id}) \circ (\eta \times \eta) \times \mathrm{id}) & (\text{exp-}\circ) \\ = & \Lambda(\mathrm{flat} \circ \mathfrak{u} \circ (\eta \times \eta) \circ \mathrm{ev} \times \mathrm{ev} \circ \langle \pi_1 \times \mathrm{id}; \pi_2 \times \mathrm{id} \rangle \circ (\mathrm{app} \times \mathrm{app}) \times \mathrm{id}) & (\text{Fig. 8.7}) \\ = & \Lambda(\mathsf{par} \circ \langle \mathrm{ev} \circ (\pi_1 \times \mathrm{id}); \mathrm{ev} \circ (\pi_2 \times \mathrm{id}) \rangle) \circ (\mathrm{app} \times \mathrm{app}) & (\text{exp-}\circ) \end{array}$$

The diagram in Figure 8.7 commutes essentially thanks to the strong aggregation monad property (agg-t). More precisely, in Figure 8.7, diagram (1) is given by the naturality of η. Diagram (2) is given by the naturality of $\mathfrak{u}$. Diagram (3) is given by some obvious properties of products. Diagram (4) is given by (agg-t). Diagram (5) is given by (str-η) and the naturality of η. Finally, diagram (6) is given by the naturality of $\mathfrak{u}$.

We now check that

$$
\begin{aligned}
& [\![(M_1 /\!/ M_2)N]\!]_\ell \\
= \;& \mathrm{ev} \circ \langle \mathrm{app} \circ [\![M_1 \;\mathsf{par}\; M_2]\!];\; [\![N]\!]_\ell \rangle \\
= \;& \mathrm{ev} \circ \langle \mathrm{app} \circ \mathsf{par} \langle [\![M_1]\!]_\ell ; [\![M_2]\!]_\ell \rangle;\; [\![N]\!]_\ell \rangle \\
= \;& \mathrm{ev} \circ \langle \Lambda(\underbrace{\mathsf{par} \circ (\mathrm{ev} \times \mathrm{ev}) \circ \langle \pi_1 \times \mathrm{id}; \pi_2 \times \mathrm{id} \rangle}_{f}) \circ (\mathrm{app} \times \mathrm{app}) \circ \langle [\![M_1]\!]_\ell;\; [\![M_2]\!]_\ell \rangle;\; [\![N]\!]_\ell \rangle \\
= \;& \mathrm{ev} \circ \langle \Lambda(f) \circ (\mathrm{app} \times \mathrm{app}) \circ \pi_1;\; \pi_2 \rangle \circ \langle \langle [\![M_1]\!]_\ell;\; [\![M_2]\!]_\ell \rangle;\; [\![N]\!]_\ell \rangle \\
= \;& \mathrm{ev} \circ \langle \Lambda(f \circ (\mathrm{app} \times \mathrm{app}) \times \mathrm{id}) \circ \pi_1;\; \pi_2 \rangle \circ \langle \langle [\![M_1]\!]_\ell;\; [\![M_2]\!]_\ell \rangle;\; [\![N]\!]_\ell \rangle \\
= \;& \mathsf{par} \circ (\mathrm{ev} \times \mathrm{ev}) \circ \langle \pi_1 \times \mathrm{id};\; \pi_2 \times \mathrm{id} \rangle \circ (\mathrm{app} \times \mathrm{app}) \times \mathrm{id}) \circ \langle \langle [\![M_1]\!]_\ell;\; [\![M_2]\!]_\ell \rangle;\; [\![N]\!]_\ell \rangle \\
= \;& \mathsf{par} \circ \langle \mathrm{ev} \circ \pi_1 \times \mathrm{id};\; \mathrm{ev} \circ \pi_2 \times \mathrm{id} \rangle \circ (\mathrm{app} \times \mathrm{app}) \times \mathrm{id}) \circ \langle \langle [\![M_1]\!]_\ell;\; [\![M_2]\!]_\ell \rangle;\; [\![N]\!]_\ell \rangle \\
= \;& \mathsf{par} \circ \langle \mathrm{ev} \circ \langle \mathrm{app} \circ [\![M_1]\!]_\ell;\; [\![N]\!]_\ell \rangle;\; \mathrm{ev} \circ \langle \mathrm{app} \circ [\![M_2]\!]_\ell;\; [\![N]\!]_\ell \rangle \rangle \\
= \;& \mathsf{par} \circ \langle [\![M_1 N]\!]_\ell;\; [\![M_2 N]\!]_\ell \rangle \\
= \;& [\![M_1 N /\!/ M_2 N]\!]_\ell
\end{aligned}
$$

If $\mathcal{T}$ contains the ϵ rule, then the soundness of the (ϵ) equational axiom is proved in the same way as for the soundness of the δ-rule: we check that the following diagram commutes

$$
\begin{array}{ccc}
D \times D & \xrightarrow{\quad\mathsf{par}\quad} & D \\
\uparrow{\scriptstyle \mathrm{lam}\times\mathrm{lam}} & & \uparrow{\scriptstyle \mathrm{lam}} \\
D^D \times D^D & \xrightarrow[\Lambda(\mathsf{par}\circ\langle \mathrm{ev}\circ(\pi_1\times\mathrm{id});\mathrm{ev}\circ(\pi_2\times\mathrm{id})\rangle)]{} & D^D
\end{array}
$$

Suppose that $\mathcal{T}$ contains the equational axiom (A) respectively the equational axiom (C) respectively the equational axiom (I). All models adapted to $\mathcal{T}$ validates these equations by Lemma 8.8.

□

8.5 Examples in Scott domains

In the category of Scott domains [Sco82, AC98], the ACI-aggregation monad $\langle \mathcal{P}_\iota; \vee \rangle$ (where $\mathcal{P}_\iota$ denotes the lower powerdomain [Plo76, Smy78]) is the source of a plethora of models for the parallel λ-calculus, due to the fact that:

Proposition 8.18 — *Any Scott domain* D *with a top element is a* $\mathcal{P}_\iota$*-algebra whose aggregation operator* p_D *is the binary join:* $p_D(x, y) = x \vee y$ *(for all* $x, y \in D$*).*

Proof. Remember that a Scott domain has a top element iff it has all its joins (the converse already holds for cpos). We define the morphism $h_D : \mathcal{P}_\iota(D) \to D$ from the map

$$
\begin{array}{rccc}
h_0 \;:\; \mathcal{K}(\mathcal{P}_\iota(D)) = \mathfrak{P}^+_{\mathrm{fin}}(\mathcal{K}(D)) & \to & D \\
\{k_1; \ldots; k_n\} & \mapsto & k_1 \vee \cdots \vee k_n
\end{array}
$$

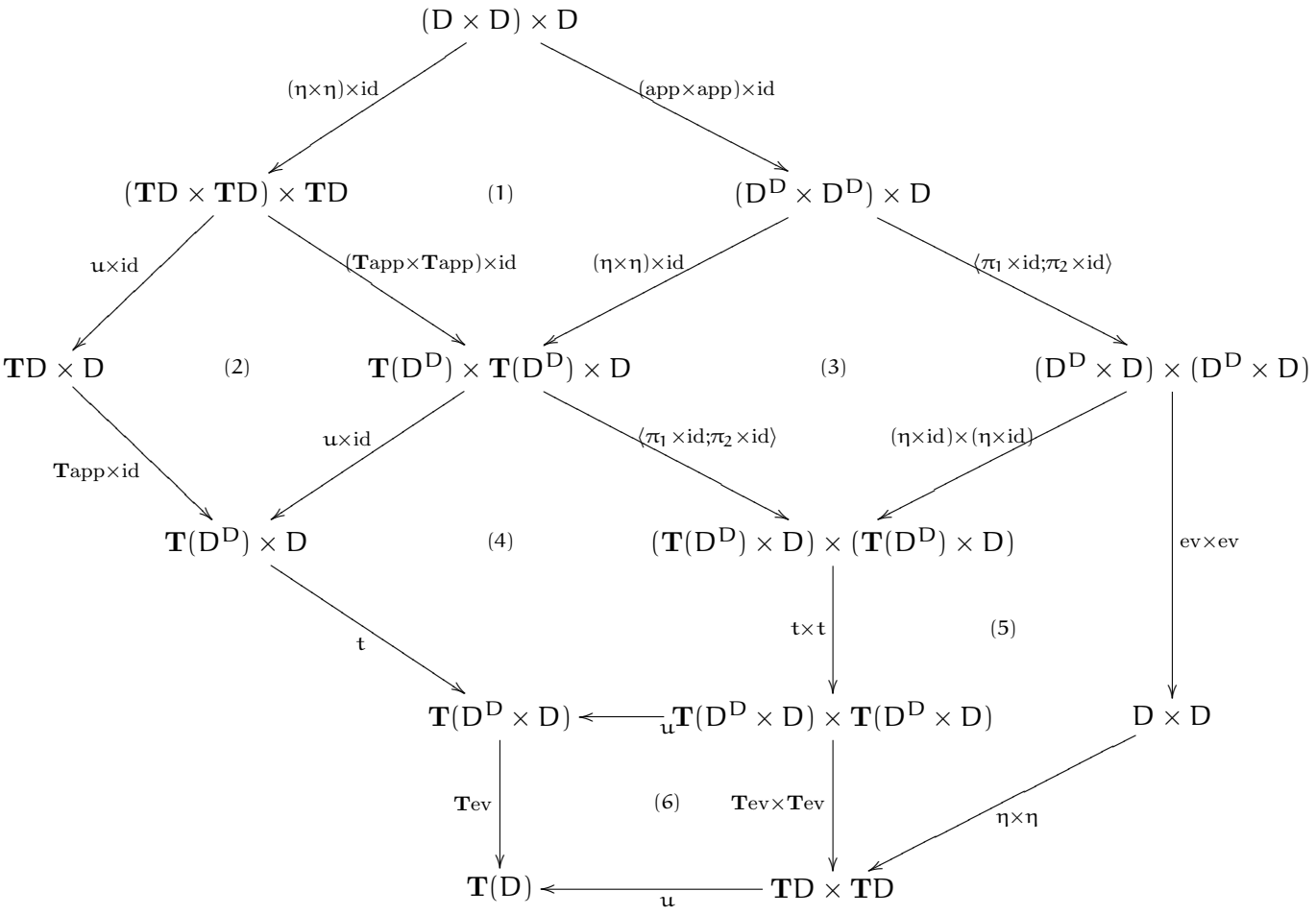


Figure 8.7: Soundness of the (δ) rule

(writing $\mathcal{K}(D)$ the set of finite elements of D) by setting

$$h_D(x) = \sup_{k \ll x} h_0(k)$$

(where k ranges over all finite approximants of x) for all $x \in \mathcal{P}_l(D)$. From this construction, it is obvious that the corresponding aggregation operator is the binary join. □

Moreover, the notion of morphism of algebras (cf subsection 8.3.3) exactly corresponds to the notion of additive functions in Scott domains:

Proposition 8.19 *Let D and E be two Scott domains with a top element. A continuous function $f : D \to E$ is a morphism of algebras iff it is additive, namely:*

$$f(\bot) = \bot \qquad \textit{and} \qquad f(x \vee y) = f(x) \vee f(y)$$

for all $x, y \in D$.

(Notice that we require that additive functions are strict.)

Proposition 8.20 *If D is a Scott domain with a top element equipped with a reflexive structure $(\mathrm{lam}, \mathrm{app})$ where $\mathrm{app} : D \to D^D$ is an additive continuous function, then D is a model of the parallel λ-calculus w.r.t. the aggregation monad $\langle \mathcal{P}_l; \vee \rangle$.*

Proof. To check that the diagram (δ) commutes, it suffices to check that for all $\bar{k} = \{k_1; \ldots; k_n\} \in \mathcal{K}(\mathcal{P}_l(D))$ and for all $x \in D$ we have

$$\begin{aligned} &\mathrm{app}(k_1 \vee \cdots \vee k_n)(x) \;= \\ &\quad \mathrm{app}(k_1)(x) \vee \cdots \vee \mathrm{app}(k_n)(x)\,, \end{aligned}$$

which follows from the hypothesis and the fact that function application is additive on its first argument. □

An obvious example of such a model is Scott's D_∞ domain, which is built from the domain $D_0 = \{\top_\bot\}$ by taking the colimit of the sequence $D_{i+1} = (D_i \to D_i)$ $(i \geq 0)$.

Application: Boudol's models In [Bou94], Boudol presents two models D_* and D_s for λ-calculi with a parallel construct, as the initial solutions of both equations

$$D_* = (D_* \to D_*)_\bot \quad \text{and} \quad D_s = (D_s \to_\bot D_s)_\bot$$

(where $(\_)_\bot$ denotes lifting and $(\_ \to_\bot \_)$ the space of strict continuous functions). The first model D_* (due to Abramsky [Abr91, AO93]) is clearly a model of the parallel λ-calculus from Prop. 8.20 due to the existence of a retraction pair

$$(\mathrm{up}_*, \mathrm{down}_*) \;:\; (D_* \to D_*) \lhd D_*$$

whose second component (the projection) is additive. The second model $\mathsf{D_s}$ (which interprets a λ-calculus with call-by-value abstractions) can be decomposed as follows

$$\begin{cases} \mathsf{D_s} & = & \mathsf{V_{s\perp}} \\ \mathsf{V_s} & = & \mathsf{V_{s\perp}} \to_\perp \mathsf{D_s} \quad = \quad \mathsf{V_s} \to \mathsf{D_s} \end{cases}$$

where $\mathsf{V_s}$ is a space of values (as opposed to $\mathsf{D_s}$, which is a space of computations). Here, the space of values $\mathsf{V_s}$ is again a model of the parallel λ-calculus from Prop. 8.20 due to the existence of a retraction pair

$$(\lambda \mathsf{f} \,.\, \mathrm{up_s} \circ \mathsf{f},\ \lambda \mathsf{f} \,.\, \mathrm{down_s} \circ \mathsf{f}) \quad : \quad (\mathsf{V_s} \to \mathsf{V_s}) \lhd \mathsf{V_s}$$

whose second component is additive. (Here, $(\mathrm{up_s}, \mathrm{down_s})$ denotes the retraction $\mathsf{V_s} \lhd \mathsf{D_s}$.)

Conclusion

In this chapter, we define a categorical semantics for the parallel λ-calculus and we prove it sound. We prove in the following chapter its completeness.

Chapter 9

Completeness by syntactical models

Contributions For each of the 24 equational theories $\mathcal{T}_0$ of the parallel λ-calculus obtained by combining each of the 3 basic calculi βδ, βδϵ and βδη with the 8 combinations of A, C and I, we prove the following completeness result: for every congruence $\mathcal{T}$ that contains $\mathcal{T}_0$ there exists a model adapted to $\mathcal{T}_0$ such that the equational theory induced by the interpretation of parallel terms in this object is exactly $\mathcal{T}$.

The proof is based on an adequate notion of pers models as for the proof of completeness of the categorical semantics of the λ-calculus given in Chapter 7. The main difficulty here is to find a suitable aggregation monad. This is achieved by introducing a boxing monad that ensures that in the per models we only have parallel aggregation of *linear* terms (in the sense of definition 8.1).

Outline of the chapter In Section 9.1, we extend the notion of λ-per by defining a general notion of $\mathcal{T}$-per w.r.t. the equational theories of the parallel λ-calculus. We define and study in Section 9.2 the corresponding category and in Section 9.3 the corresponding aggregation monad. The last section (9.4) is devoted to the detailed proof of the completeness result.

9.1 The notion of $\mathcal{T}$-per

Let $\mathcal{T}$ be one of the 24 equational theories of the parallel λ-calculus mentioned in Section 8.1 and obtained by combining each of the 3 basic calculi βδ, βδϵ and βδη with the combinations of A, C and I.

Definition 9.1 ($\mathcal{T}$-per) — *A* $\mathcal{T}$-partial equivalence relation ($\mathcal{T}$-per) *is a partial equivalence relation (per)* A *on the set of parallel* λ*-terms such that* $\mathcal{T} \circ \mathsf{A} \subset \mathsf{A}$*, that is, a symmetric and transitive relation* A *such that*

$$(\mathsf{M}, \mathsf{M}') \in \mathsf{A} \quad \textit{and} \quad \mathsf{M}' =_{\mathcal{T}} \mathsf{M}'' \;\Rightarrow\; (\mathsf{M}, \mathsf{M}'') \in \mathsf{A}$$

for all terms $\mathsf{M}, \mathsf{M}', \mathsf{M}''$*.*

Definition 9.2 (Domain) *The domain of a* $\mathcal{T}$*-per* A *is defined by*

$$\mathrm{dom}(\mathsf{A}) = \{\mathsf{M} \mid (\mathsf{M}, \mathsf{M}) \in \mathsf{A}\}.$$

$\mathcal{T}$-pers are naturally ordered by inclusion: the smallest $\mathcal{T}$-per is the empty per (of domain the empty set) and the largest $\mathcal{T}$-per is the full per (of domain the set of all terms). Moreover, $\mathcal{T}$-pers are closed under arbitrary intersection, and thus form a complete distributive lattice.

Two important constructions of $\mathcal{T}$-pers are the arrow $A \to B$ of two $\mathcal{T}$-pers A and B and the parallel closure A^+ of one $\mathcal{T}$-per defined as follows.

Definition 9.3 (Arrow $\mathcal{T}$-pers) *The* arrow $A \to B$ *of two $\mathcal{T}$-pers A and B is defined for all M, M' by*

$$(M, M') \in (A \to B) \quad \textit{iff} \quad \forall N, N' \, ((N, N') \in A \Rightarrow (MN, M'N') \in B)$$

The arrow construction of pers is a construction which is antimonotonic w.r.t. A and monotonic w.r.t. B.

Definition 9.4 (Parallel closure) *The* parallel closure A^+ *of a $\mathcal{T}$-per A is inductively defined by the two following rules*

$$\frac{(M, M') \in A}{(M, M') \in A^+} \qquad \frac{\begin{array}{ll}(M_1, M_1') \in A^+ & M =_{\mathcal{T}} M_1 \mathbin{/\!\!/} M_2 \\ (M_2, M_2') \in A^+ & M' =_{\mathcal{T}} M_1' \mathbin{/\!\!/} M_2'\end{array}}{(M, M') \in A^+}$$

We first show that the two definitions are sound. Note that in the proof of soundness of the arrow construction, we use the fact that both A and B are $\mathcal{T}$-pers.

Lemma 9.5 *Let A and B be two $\mathcal{T}$-pers. The arrow $A \to B$ is also a $\mathcal{T}$-per.*

Proof. We first prove the symmetry of the relation $A \to B$. Suppose that we have

$$(M, M') \in A \to B.$$

Let $(N', N) \in A$. By the symmetry of A, we have $(N, N') \in A$ and then

$$(MN, M'N') \in B.$$

By the symmetry of B, we get

$$(M'N', MN) \in B.$$

We then show the transitivity of the relation $A \to B$. Suppose that we have

$$(M, M') \in A \to B \qquad \text{and} \qquad (M', M'') \in A \to B.$$

Let $(N, N') \in A$. Then we have $N \in \mathrm{dom}(A)$ and thus $(MN, M'N) \in B$. But we have also

$$(M'N, M''N') \in B.$$

The transitivity of B concludes the proof.

We finally show that the relation $A \to B$ is closed by $\mathcal{T}$. Suppose that we have

$$(M, M') \in A \to B \qquad \text{and} \qquad M' =_{\mathcal{T}} M''.$$

Let $(N, N') \in A$. Then we have

$$(MN, M'N') \in B \qquad \text{and} \qquad M'N' =_{\mathcal{T}} M''N'.$$

Since B is closed by $\mathcal{T}$, we have $MN =_{\mathcal{T}} M''N'$. □

Lemma 9.6 *Let A be a $\mathcal{T}$-per. The parallel closure A^+ is a $\mathcal{T}$-per.*

Proof. Let us show the symmetry of the relation A^+. Suppose that $(M, M') \in A^+$. We proceed by induction on its proof.

- First, if $(M, M') \in A$ then the result follows by the symmetry of A.
- For the induction case, we have

  $$M =_{\mathcal{T}} M_1 /\!\!/ M_2 \qquad \text{and} \qquad M' =_{\mathcal{T}} M'_1 /\!\!/ M'_2$$

  with $(M_1, M'_1) \in A^+$ and $(M_2, M'_2) \in A^+$. By induction, we obtain that

  $$(M'_1, M_1) \in A^+ \qquad \text{and} \qquad (M'_2, M_2) \in A^+.$$

  We conclude $(M', M) \in A^+$.

The proof of the transitivity and the proof of the closure of A^+ are easy and proceed in the same way. □

Fact 9.7 (Parallel closure construction) *The parallel closure construction is a closure operator since it is monotonic and we have*

$$A \subset A^+ \qquad \textit{and} \qquad A^{++} = A^+.$$

Lemma 9.8 *For all $\mathcal{T}$-pers A and B, we have $(A \to B)^+ \subset A \to B^+$*

Proof. Suppose that $(M, M') \in (A \to B)^+$. We prove the result by induction on its proof.

- If $(M, M') \in (A \to B)$. The result is obvious since $A \to B \subseteq A \to B^+$.
- Suppose now that we have

  $$M =_{\mathcal{T}} M_1 /\!\!/ M_2 \qquad \text{and} \qquad M' =_{\mathcal{T}} M'_1 /\!\!/ M'_2$$

  with $(M_1, M'_1) \in (A \to B)^+$ and $(M_2, M'_2) \in (A \to B)^+$. By induction hypothesis, we have

  $$(M_1, M'_1) \in A \to B^+ \qquad \text{and} \qquad (M_2, M'_2) \in A \to B^+.$$

Let $(N, N') \in A$. We check that

$$\begin{array}{lll} (M_1 /\!\!/ M_2)N & =_\delta & M_1N /\!\!/ M_2N \\ & =_{B^+} & M_1'N' /\!\!/ M_2'N' \\ & =_\delta & (M_1' /\!\!/ M_2')N' \end{array}$$

This proves $(M, M') \in A \to B^+$.

□

The previous lemma is obtained from the equation (δ). On the other hand, the equation $(A \to B) \subset (A^+ \to B^+)$ holds only for linear terms (in the sense of definition 8.1). This justifies the importance of linear terms.

Fact 9.9 *If two terms M and M' are linear in the theory $\mathcal{T}$ then*

$$(M, M') \in (A \to B) \qquad \textit{implies} \qquad (M, M') \in (A^+ \to B^+)\,.$$

Proof. Let $(N, N') \in A^+$ we prove that

$$(MN, M'N') \in B^+$$

by induction on the proof of $(N, N') \in A^+$.

- If $(N, N') \in A$ then $(MN, M'N') \in B \subseteq B^+$.
- If $N =_{\mathcal{T}} N_1 /\!\!/ N_2$ and $N' =_{\mathcal{T}} N_1' /\!\!/ N_2'$ with $(N_i, N_i') \in A^+$. Then by induction hypothesis we have $(MN_i, M'N_i') \in B^+$. We can check that

$$\begin{array}{lll} MN & =_{\mathcal{T}} & M(N_1 /\!\!/ N_2) \\ & =_{\mathcal{T}} & MN_1 /\!\!/ MN_2 \qquad \text{Linearity of } M \\ & =_{B^+} & M'N_1' /\!\!/ M'N_2' \\ & =_{\mathcal{T}} & M'(N_1' /\!\!/ N_2') \qquad \text{Linearity of } M' \\ & =_{\mathcal{T}} & M'N' \end{array}$$

□

9.2 The ccc structure of $\mathcal{T}$-PER

Definition 9.10 (Category $\mathcal{T}$-PER) *The category $\mathcal{T}$-**PER** is defined as follows*

- *The objects of $\mathcal{T}$-**PER** are $\mathcal{T}$-pers.*
- *Given two $\mathcal{T}$-pers A and B, the hom-set $\mathcal{T}$-**PER**$[A; B]$ is defined by*

$$\mathcal{T}\text{-}\mathbf{PER}[A; B] \quad = \quad \mathrm{dom}(A \to B)/\!\sim_{(A \to B)}\,,$$

where $\sim_{(A \to B)}$ denotes the (total) equivalence relation induced by the $\mathcal{T}$-per $(A \to B)$ on its domain.

- *Identity and composition are defined for all objects* $A, B, C \in \mathcal{T}$-**PER** *and for all arrows* $M \in \mathcal{T}$-**PER**$[B; C]$ *and* $N \in \mathcal{T}$-**PER**$[A; B]$ *by*

$$\begin{array}{rcll} \mathrm{id}_A & = & \lambda x\,.\,x & \in \ \mathcal{T}\text{-}\mathbf{PER}[A; A] \\ M \circ N & = & \lambda x\,.\,M(Nx) & \in \ \mathcal{T}\text{-}\mathbf{PER}[A; C] \end{array}$$

  (taking the corresponding operations on equivalence classes).

Lemma 9.11 *The category* $\mathcal{T}$-**PER** *is a Cartesian category:*

- *The terminal object is the full* $\mathcal{T}$*-per:* $1 = \top$.
- *The Cartesian product* $A \times B$ *is defined by*

$$\begin{array}{l} (M, M') \in (A \times B) \ \ \textit{iff} \\ \quad (\pi_1 M, \pi_1 M') \in A \ \ \textit{and} \ \ (\pi_2 M, \pi_2 M') \in B\,, \end{array}$$

  where $\pi_1 = \lambda p\,.\,p(\lambda xy\,.\,x)$ *and* $\pi_2 = \lambda p\,.\,p(\lambda xy\,.\,y)$.
- *The pairing operator is given by* $\langle M; N\rangle = \lambda xp\,.\,p(Mx)(Nx)$.

Proof. We prove the following three items.

- The fact that the full $\mathcal{T}$-per is the terminal object is trivial: the full $\mathcal{T}$-per makes all terms equal and then $(A \to \top)$ is also the full $\mathcal{T}$-per.
- We check that for all $M : C \to A$ and $N : C \to B$

$$\begin{array}{rcl} \pi_1 \circ \langle M; N\rangle & =_\beta & \lambda z\,.\,\pi_1(\lambda p\,.\,p(Mz)(Nz)) \\ & =_\beta & \lambda z\,.\,Mz \\ & =_{C\to A} & M \end{array}$$

  and

$$\begin{array}{rcl} \pi_2 \circ \langle M; N\rangle & =_\beta & \lambda z\,.\,\pi_2(\lambda p\,.\,p(Mz)(Nz)) \\ & =_\beta & \lambda z\,.\,Nz \\ & =_{C\to B} & N \end{array}$$

- We show the uniqueness of the pairing operator for every $(M, N) \in \mathrm{dom}(A) \times \mathrm{dom}(B)$. Suppose that there exists $P : C \to A \times B$ such that

$$\begin{array}{rcl} \pi_1 \circ P & =_{C\to A} & M \\ \pi_2 \circ P & =_{C\to B} & N \end{array}$$

  since we have

$$\begin{array}{rcl} M & =_{C\to A} & \pi_1 \circ \langle M; N\rangle \\ N & =_{C\to B} & \pi_2 \circ \langle M; N\rangle \end{array}$$

  This gives $\pi_1 \circ P =_{C\to A} \pi_1 \circ \langle M; N\rangle$ and $\pi_2 \circ P =_{C\to B} \pi_2 \circ \langle M; N\rangle$. By the definition of $A \times B$, we conclude $P =_{C\to A\times B} \langle M; N\rangle$.

□

Lemma 9.12 *In the category* $\mathcal{T}$-**PER**, *for every pairs of objets* A, B *there exists an exponent:*

- *The exponent is given by* $B^A = (A \to B)$.
- *The evaluation map is given by* $\mathrm{ev} = \lambda p \,.\, (\pi_1 p)\,(\pi_2 p)$.
- *The currying operator is given by* $\Lambda(M) = \lambda xy \,.\, M(\lambda p \,.\, pxy)$.

Proof. We prove the following items.

- Let $M \in C \times A \to B$. We check that

$$\begin{aligned} \mathrm{ev} \circ (\Lambda(M) \times \mathrm{id}) &=_\beta && \lambda z \,.\, \mathrm{ev}\big(\langle \Lambda(M)(\pi_1 z);\ \pi_2 z\rangle\big) \\ &=_\beta && \lambda z \,.\, \Lambda(M)(\pi_1 z)(\pi_2 z) \\ &=_\beta && \lambda z \,.\, M(\lambda p \,.\, p(\pi_1 z)(\pi_2 z)) \\ &=_{C\times A\to B} && M \end{aligned}$$

- Suppose given $M \in C \times A \to B$. We now prove the uniqueness of the currying operator. Suppose given a term $M' : C \to A \to B$ such that

$$\mathrm{ev} \circ (M' \times \mathrm{id}) \ =_{C\times A\to B}\ M$$

  We want to show

$$M' =_{C\to A\to B} \Lambda(M).$$

  Let $(N_1, N_1') \in C$ and $(N_2, N_2') \in A$. We want to show that

$$M' N_1 N_2 \ =_C\ \Lambda(M)\, N_1' N_2'$$

  or equivalently:

$$M' N_1 N_2 \ =_B\ M\, \lambda p \,.\, p N_1' N_2'$$

  But since $\mathrm{ev} \circ (M' \times \mathrm{id}) \ =_{C\times A\to B}\ M$ and $\lambda p \,.\, p N_1 N_2 \ =_{C\times A}\ \lambda p \,.\, p N_1' N_2'$ we have

$$(\mathrm{ev} \circ (M' \times \mathrm{id}))\ (\lambda p \,.\, p N_1 N_2) \ =_B\ M(\lambda p \,.\, p N_1' N_2')$$

  The equality $(\mathrm{ev} \circ (M' \times \mathrm{id}))\ (\lambda p \,.\, p N_1 N_2) =_\beta M' N_1 N_2$ concludes the proof.

□

From the two previous results, we conclude the following proposition.

Proposition 9.13 (Ccc of $\mathcal{T}$-PER) *The category of* $\mathcal{T}$-**PER** *can be given the structure of a ccc.*

9.3 The aggregation monad of $\mathcal{T}$-PER

It would be tempting to define the aggregation monad $\mathbf{T}$ of $\mathcal{T}$-**PER** by setting $\mathbf{T}A = A^+$. Unfortunately, the parallel closure operator $A \mapsto A^+$ is not functorial (since $A \subset A^+$ but $(A \to B) \not\subseteq (A^+ \to B^+)$) and thus cannot be given the structure of a monad. To achieve functoriality, we first need to introduce the following boxing mechanism that constructs only linear terms (see Lemma 9.20).

9.3.1 The boxing monad

Definition 9.14 (Boxing terms) *For all terms* M*, we set*

$$[M] = \lambda x\,.\,xM$$

where x *is a fresh variable.*

This construction can be understood as a 1-uple. Unboxing is performed by applying $\mathbf{I} = \lambda x\,.\,x$, since $[M]\mathbf{I} =_\beta M$.

Definition 9.15 (Boxing $\mathcal{T}$-pers) *To each $\mathcal{T}$-per A, we associate a $\mathcal{T}$-per $[A]$ defined by*

$$\begin{aligned}(M, M') \in [A] \quad &\textit{iff}\\ \exists (M_0, M_0') \in A \quad &(M =_{\mathcal{T}} [M_0] \quad \wedge \quad M' =_{\mathcal{T}} [M_0'])\,.\end{aligned}$$

This definition is clearly sound and we have

Lemma 9.16 *Both $\mathcal{T}$-pers $[A]$ and A are isomorphic via the converse isomorphisms*

$$\mathsf{box} = \lambda x\,.\,[x] \in \mathrm{dom}(A \to [A]) \qquad \textit{and} \qquad \mathsf{unbox} = \lambda x\,.\,x\mathbf{I} \in \mathrm{dom}([A] \to A)\,.$$

Proof. As remarked above, we have $\mathsf{unbox} \circ \mathsf{box} =_{A \to A} \mathbf{I}$. We prove the other direction. Suppose that we have $(N, N') \in [A]$ that is,

$$N =_{\mathcal{T}} [N_0] \quad \text{and} \quad N' =_{\mathcal{T}} [N_0'] \quad \text{and} \quad (N_0, N_0') \in A$$

We check that

$$\begin{aligned}(\mathsf{box} \circ \mathsf{unbox})(N) &=_\beta [N\mathbf{I}]\\ &=_{\mathcal{T}} [[N_0]\mathrm{I}]\\ &=_\beta [N_0]\\ &=_{[A]} [N_0']\\ &=_{\mathcal{T}} N'\end{aligned}$$

□

Moreover, the following implication holds.

Lemma 9.17 *For all $\mathcal{T}$-pers A and B, we have $[A \to B] \subset [A] \to B$.*

Proof. Suppose that $(M, M') \in [A \to B]$ that is

$$M =_{\mathcal{T}} [M_0] \text{ and } M' =_{\mathcal{T}} [M'_0] \text{ and } (M_0, M'_0) \in A \to B$$

Let $(N, N') \in [A]$, that is

$$N =_{\mathcal{T}} [N_0] \text{ and } N' =_{\mathcal{T}} [N'_0] \text{ and } (N_0, N'_0) \in A$$

We check that

$$\begin{array}{lll} MN & =_{\mathcal{T}} & [M_0][N_0] \\ & =_{\beta} & [N_0]M_0 \\ & =_{\beta} & M_0N_0 \\ & =_{B} & M'_0N'_0 \\ & =_{\beta} & [M'_0][N'_0] \end{array}$$

□

Definition 9.18 (Boxing functor) *The boxing functor is defined as follows*

- *The object function:* $A \mapsto [A]$.
- *The arrow function:* $\uparrow M = [\lambda z\,.\,[Mz]]$ *for all* M.

The definition is sound since $M \in \mathrm{dom}(A \to B)$ implies

$$\uparrow M \;\in\; \mathrm{dom}([A \to [B]]) \;\subset\; \mathrm{dom}([A] \to [B])\,.$$

It actually defines a functor since

- it preserves identity:

$$\begin{array}{lll} \uparrow \mathrm{id} & =_{[A]\to[A]} & [\lambda z\,.\,[z]] \\ & =_{[A]\to[A]} & \lambda z\,.\,z \end{array}$$

- it preserves composition:

$$\begin{array}{lll} \uparrow(f \circ g) & =_{\beta} & [\lambda y\,.\,[f(gz)]] \\ & =_{[A]\to[A]} & \lambda z\,.\,z(\lambda x\,.\,[gx])(\lambda y\,.\,[fy]) \\ & = & \uparrow f \circ \uparrow g \end{array}$$

Lemma 9.19 (Boxing monad) *The boxing functor can be given the structure of monad by setting:*

- *Unit:* $\eta = \mathsf{box}$.
- *Multiplication:* $\mu = \mathsf{unbox}$.

Proof. Note that all the terms involved in this proof are linear (except in the naturality of η). This will be used in the section related to the aggregation monad. The two transformations η and μ are well-defined and define a monad. We show that there are natural transformations.

- We check that η is natural:

$$\begin{array}{rcl} \eta \circ M & =_\beta & \lambda z\,.\,[Mz] \\ & =_\beta & \lambda z\,.\,[z](\lambda y\,.\,[My]) \\ & =_\beta & \Uparrow M \circ \eta \end{array}$$

- We check that μ is natural. We first check that

$$\Uparrow M \circ \mu \quad =_\beta \quad \lambda z\,.\,zI(\lambda x\,.\,[Mx])$$

$$\mu \circ \Uparrow\Uparrow M \quad =_\beta \quad \lambda z\,.\,z(\lambda x\,.\,[x(\lambda y\,.\,[My])])\mathbf{I}\,.$$

  Let $(N, N') \in [[A]]$ that is $N =_{\mathcal{T}} [[N_0]]$ and $N' =_{\mathcal{T}} [[N'_0]]$ with $(N_0, N'_0) \in A$ we check that

$$\begin{array}{rcl} (\Uparrow M \circ \mu)(N) & =_{\mathcal{T}} & ([[N_0]]\mathbf{I})(\lambda y\,.\,[My]) \\ & =_\beta & [MN_0] \\ & =_{[B]} & [MN'_0] \\ & =_\beta & [[N'_0](\lambda x\,.\,[Mx])]\mathbf{I} \\ & =_{\mathcal{T}} & [[N'_0]](\lambda y\,.\,[y(\lambda x\,.\,[Mx])])\mathbf{I} \\ & = & (\mu \circ \Uparrow\Uparrow M)(N') \end{array}$$

- We check that (mon-1) is satisfied. We have

$$\mu \circ \Uparrow \eta \;=\; \lambda z\,.\,z(\lambda x\,.\,[\eta x])\mathbf{I}$$

  Let $(N, N') \in [A]$ this means that $N =_{\mathcal{T}} [N_0]$ and $N' =_{\mathcal{T}} [N'_0]$ with $(N_0, N'_0) \in A$. We have

$$\begin{array}{rcl} (\mu \circ \Uparrow \eta)(N) & =_{\mathcal{T}} & [N_0](\lambda x\,.\,[[x]])\mathbf{I} \\ & =_\beta & [[N_0]]\mathbf{I} \\ & =_\beta & [N_0] \\ & =_{[A]} & [N'_0] \\ & =_{\mathcal{T}} & N' \end{array}$$

- We check that (mon-2) is satisfied: this is true since η and μ are isomorphic.

- We check that (mon-3) is satisfied. We have

$$\begin{array}{rcl} \mu \circ \Uparrow \mu & =_\beta & \lambda z\,.\,z(\lambda x\,.\,[x\mathbf{I}])\mathbf{I} \\ \mu \circ \mu & =_\beta & \lambda z\,.\,(z\mathbf{I})\mathbf{I} \end{array}$$

Let $(N, N') \in [[[A]]]$ this means that we have $N =_{\mathcal{T}} [[N_0]]$ and $N' =_{\mathcal{T}} [[N'_0]]$ with $(N_0, N'_0) \in [A]$. We have

$$\begin{array}{rll}(\mu \circ \mu)(N) & =_{\mathcal{T}} & [[N_0]]\mathbf{II} \\ & =_{\beta} & N_0 \\ & =_{[A]} & N'_0 \\ & =_{\beta} & [[N'_0]\mathbf{I}]\mathbf{I} \\ & =_{\beta} & [[N'_0]](\lambda x\,.\,[x\mathbf{I}])\mathbf{I} \\ & =_{\beta} & (\mu \circ \Uparrow\mu)(N')\end{array}$$

□

The main property of the boxing mechanism is that boxed objects (including lifted morphisms $\Uparrow M$) are linear w.r.t. their first argument, in the sense that

Lemma 9.20 *For all* $M \in [A]$ *and for all terms* N_1, N_2*,*

$$M(N_1 /\!\!/ N_2) =_{\mathcal{T}} MN_1 /\!\!/ MN_2$$

Proof. Let $M \in [A]$ that is $M =_{\mathcal{T}} [M'_0]$ with $M'_0 \in A$. We check that

$$\begin{array}{rll}M(N_1 /\!\!/ N_2) & =_{\mathcal{T}} & [M_0](N_1 /\!\!/ N_2) \\ & =_{\beta} & (N_1 /\!\!/ N_2)M_0 \\ & =_{\delta} & N_1M_0 /\!\!/ N_2M_0 \\ & =_{\beta} & [M_0]N_1 /\!\!/ [M_0]N_2 \\ & =_{\mathcal{T}} & MN_1 /\!\!/ MN_2\end{array}$$

□

This property is crucial for the definition of the aggregation monad below.

9.3.2 The aggregation monad

Definition 9.21 (Aggregation monad) *The aggregation monad* $\mathbf{T}$ *is defined by*

- *The object function:* $A \mapsto [A]^+$.
- *The arrow function:* $\Uparrow M = [\lambda z\,.\,[Mz]]$ *for all* M.
- *Unit:* $\eta = \mathsf{box}$.
- *Multiplication:* $\mu = \mathsf{unbox}$.
- *Aggregation:* $\mathfrak{u} = \lambda p\,.\,(\pi_1 p) /\!\!/ (\pi_2 p)$.
- *Strength:* $\mathfrak{t} = \lambda x\,.\,(\pi_1 x)(\lambda y\,.\,[\langle y; \pi_2 x\rangle])$.

Of course, one has to check that these constructions fit in their new types, using the property of linearity mentioned above.

Lemma 9.22 *The functor* $\mathbf{T}$ *is well-defined and is a monad.*

Proof. We first show that $\mathbf{T}$ fits in its types: If $M \in A \to B$ then $\mathbf{T}(M) \in [A]^+ \to [B]^+$. Let $M \in A \to B$ and let $N \in [A]^+$. We prove that

$$\mathbf{T}(M)(N) \in [B]^+$$

by induction on the proof of $N \in [A]^+$. Note that since $\mathbf{T}(M)$ is linear, it is sufficient to check the basic case. So let $N \in [A]$. We have $\mathbf{T}(M)(N) \in [B]$. Since

$$[B] \subseteq [B]^+,$$

then we conclude $\mathbf{T}(M)(N) \in [B]^+$. This concludes the proof.

To show that $\mathbf{T}$ defines actually a functor and that we have indeed define a monadic structure, it is sufficient to remark that all terms that are involved in the proof of Lemma 9.19 are linear (except in the naturality of η but it is trivial).

□

Lemma 9.23 $\langle \mathbf{T}, u \rangle$ *is an aggregation monad on the category* $\mathcal{T}$-**PER**.

Proof. We can check that:

$$\begin{array}{rll} \mu \circ u & =_\beta & \lambda z\,.\,(\pi_1 z \mathbin{/\!\!/} \pi_2 z)\mathbf{I} \\ & =_\delta & \lambda z\,.\,(\pi_1 z)\mathbf{I} \mathbin{/\!\!/} (\pi_2 z)\mathbf{I} \\ & =_\beta & u \circ (\mu \times \mu) \end{array}$$

□

Proposition 9.24 $\langle \mathbf{T}, u \rangle$ *is a strong aggregation monad on the category* $\mathcal{T}$-**PER**.

Proof. We check that the diagram (agg-t) commutes:

$$\begin{array}{rll} t \circ (u \times \mathrm{id}) & =_\beta & \lambda z\,.\,((\pi_1\pi_1 z) \mathbin{/\!\!/} (\pi_2\pi_1 z))\lambda x\,.\,[\langle x; \pi_2 z\rangle] \\ & =_\delta & \lambda z\,.\,(\pi_1\pi_1 z)(\lambda x\,.\,[\langle x; \pi_2 z\rangle]) \mathbin{/\!\!/} (\pi_2\pi_1 z)(\lambda x\,.\,[\langle x; \pi_2 z\rangle]) \\ & =_\beta & \lambda z\,.\,t\langle \pi_1\pi_1 z; \pi_2 z\rangle \mathbin{/\!\!/} t\langle \pi_2\pi_1 z; \pi_2 z\rangle \\ & =_\beta & u \circ (t \times t) \circ \langle \pi_1 \times \mathrm{id}; \pi_2 \times \mathrm{id}\rangle \end{array}$$

□

Moreover, it is straightforward to check that the aggregation monad $\langle \mathbf{T}, u \rangle$ is associative, commutative or idempotent as soon as $\mathcal{T}$ contains the corresponding equation.

Lemma 9.25 *If* $\mathcal{T}$ *contains the equation* A*, then the category of* $\mathcal{T}$-**PER** *can be equipped with a strong notion of aggregation that is associative.*

Proof. We check that

$$\begin{array}{lll} \mathsf{u} \circ (\mathsf{u} \times \mathrm{id}) \circ \alpha & =_\beta & \lambda z \,.\, \big(\pi_1(z) \mathbin{/\!\!/} \pi_1(\pi_2(z))\big) \mathbin{/\!\!/} \pi_2(\pi_2(z)) \\ & =_A & \lambda z \,.\, \pi_1(z) \mathbin{/\!\!/} \big(\pi_1(\pi_2(z)) \mathbin{/\!\!/} \pi_2(\pi_2(z))\big) \\ & =_\beta & \mathsf{u} \circ (\mathrm{id} \times \mathsf{u}) \end{array}$$

□

Lemma 9.26 *If $\mathcal{T}$ contains the equation* C*, then the category of $\mathcal{T}$-***PER** *can be equipped with a strong notion of aggregation that is commutative.*

Proof. We check that

$$\begin{array}{lll} \mathsf{u} \circ \langle \pi_2; \pi_1 \rangle & =_\beta & \lambda z \,.\, (\pi_2(z) \mathbin{/\!\!/} \pi_1(z)) \\ & =_C & \lambda z \,.\, (\pi_1(z) \mathbin{/\!\!/} \pi_2(z)) \\ & = & \mathsf{u} \end{array}$$

□

Lemma 9.27 *If $\mathcal{T}$ contains the equation* I*, then the category of $\mathcal{T}$-***PER** *can be equipped with a strong notion of aggregation that is idempotent.*

Proof. We check that

$$\begin{array}{lll} \mathsf{u} \circ \langle \mathrm{id}; \mathrm{id} \rangle & =_\beta & \lambda z \,.\, \mathsf{u}(\langle z, z \rangle) \\ & =_\beta & \lambda z \,.\, (z \mathbin{/\!\!/} z) \\ & =_{\mathbf{I}} & \lambda z \,.\, z \end{array}$$

□

9.4 Completeness

Every model D of the parallel λ-calculus induces a congruence written $=_\mathsf{D}$ over the set of parallel λ-terms, which is defined for all terms M and M' by

$$M =_\mathsf{D} M' \quad \text{iff} \qquad [\![M]\!]^\mathsf{D}_\ell = [\![M']\!]^\mathsf{D}_\ell$$

(where ℓ is such that $FV(MM') \subset \ell$). Of course, since the semantics is sound the congruence $=_\mathsf{D}$ contains $\beta\delta$.

We want to show that the converse holds, in the sense that for every congruence $\mathcal{T}$ containing $\beta\delta$, there exists a model D of the parallel λ-calculus that induces the congruence $\mathcal{T}$ exactly, namely, there exists a model D such that $M =_\mathsf{D} M'$ iff $M =_\mathcal{T} M'$ for all terms M and M'.

Theorem 9.28 (Completeness) — *Let $\mathcal{T}_0$ be one of the 24 equational theories of the parallel λ-calculus mentioned in Section 8.1. For every congruence $\mathcal{T} \supseteq \mathcal{T}_0$, there exists a $\mathcal{T}_0$-per D such that:*

1. D *is a model of the parallel* λ*-calculus in* $\mathcal{T}_0$**-PER**, *which is adapted to the theory* $\mathcal{T}_0$*;*
2. $M =_{\mathsf{D}} M'$ *iff* $M =_{\mathcal{T}} M'$ *(for all terms* M, M'*)*

9.4.1 Proof sketch

The theorem is proved as follows. Consider a congruence $\mathcal{T} \supseteq \mathcal{T}_0 (\supseteq \beta\delta)$. We first notice that as a binary relation, the congruence $\mathcal{T}$ is a $\mathcal{T}_0$-per whose domain is the set of all parallel λ-terms. We choose for the object D of Theorem 9.28

$$\mathsf{D} = \mathcal{T}$$

and we show that

1. D **is a model in** $\mathcal{T}_0$**-PER, which is adapted to the theory** $\mathcal{T}_0$**.** We first check that the model fulfills all the expected properties namely it is a reflexive object (Lemma 9.29) and an algebra (Lemma 9.30) and that the diagram (δ) holds (Lemma 9.31). We check (Lemma 9.32 resp. Lemma 9.33) that the model D defined above is an ϵ-model (resp. an η-model) as soon as the equational theory $\mathcal{T}_0$ contains the equation ϵ (resp. the equation η). Moreover, we know (previous section) that the underlying aggregation monad that comes with the category $\mathcal{T}_0$-**PER** is associative, commutative, idempotent as soon as $\mathcal{T}_0$ contains the corresponding equation.
2. $M =_{\mathsf{D}} M'$ **iff** $M =_{\mathcal{T}} M'$ **for all terms** M **and** M'**.** This is done in Section 9.4.6.

9.4.2 Reflexivity

Lemma 9.29 *The reflexive structure for the congruence* $\mathcal{T}$ *is given by*

$$\mathrm{app} = \lambda x\,.\,x \qquad \mathrm{lam} = \lambda xy\,.\,xy\,.$$

Proof. We first check that $\mathrm{app} \in \mathcal{T} \to \mathcal{T}^{\mathcal{T}}$ and $\mathrm{lam} \in \mathcal{T}^{\mathcal{T}} \to \mathcal{T}$. We easily check that

$$\begin{array}{rcl} \mathrm{app} \circ \mathrm{lam} & \triangleq & \lambda z\,.\,\lambda y\,.\,z\,y \\ & =_{\mathcal{T}^{\mathcal{T}} \to \mathcal{T}^{\mathcal{T}}} & \lambda z\,.\,z \end{array}$$

In fact, let $(M_0, M_1) \in \mathcal{T}^{\mathcal{T}} = \mathcal{T} \to \mathcal{T}$ and $(N_0, N_1) \in \mathcal{T}$ then

$$\begin{array}{rcl} (\lambda z\,.\,(\lambda y\,.\,z\,y))\,M_0)\,N_0 & =_\beta & M_0\,N_0 \\ & =_{\mathcal{T}} & M_1\,N_1 \\ & =_\beta & ((\lambda z\,.\,z)\,M_1)\,N_1 \end{array}$$

□

9.4.3 Algebraicity

Lemma 9.30 *The structure of algebra for the congruence $\mathcal{T}$ is given by*

$$\text{flat} \quad = \quad \mu = \mathsf{unbox} \quad = \quad \lambda x\,.\,xI$$

Proof. We are in the particular case where the arrow of the algebra is the multiplicity of the corresponding monad. Then the two diagram (alg-1) and (alg-2) are given by the monadicity (Lemma 9.22). □

9.4.4 Distributivity axiom

Lemma 9.31 *For the model* D *defined above, the diagram* (δ) *commutes.*

Proof. We prove that the diagram (δ) commutes. We first make the following simplifications:

$$\begin{array}{lll}
\text{flat} \circ \uparrow\!\mathsf{ev} \circ t & =_\beta & \lambda z\,.\,(\text{flat} \circ \uparrow\!\mathsf{ev})\ \big((\pi_1 z)(\lambda x\,.\,[\langle x; \pi_2 z\rangle])\big) \\
& =_\beta & \lambda z\,.\,(\pi_1 z)\ (\lambda x_2\,.\,[\langle x_2; \pi_2 z\rangle])\ (\lambda x_3\,.\,[(\pi_1 x_3)(\pi_2 x_3)])\ \mathbf{I} \\
\Lambda(\text{flat} \circ \uparrow\!\mathsf{ev} \circ t) & =_\beta & \lambda x y\,.\,x(\lambda x_2\,.\,[\langle x_2; y\rangle])\ (\lambda x_3\,.\,[(\pi_1 x_3)(\pi_2 x_3)])\mathbf{I} \\
\Lambda(\text{flat} \circ \uparrow\!\mathsf{ev} \circ t) \circ \uparrow\!\mathsf{app} & =_\beta & \lambda z\,.\,\lambda y\,.\,z(\lambda x_1\,.\,[x_1])\ (\lambda x_2\,.\,[\langle x_2; y\rangle])\ (\lambda x_3\,.\,[(\pi_1 x_3)(\pi_2 x_3)])\ \mathbf{I} \\
\text{app} \circ \text{flat} & = & \lambda x_1\,.\,x_1\mathbf{I}
\end{array}$$

We want to prove that

$$\Lambda(\text{flat} \circ \uparrow\!\mathsf{ev} \circ t) \circ \uparrow\!\mathsf{app} \quad =_{\mathbf{T}(\mathcal{T})\to(\mathcal{T}\to\mathcal{T})} \quad \text{app} \circ \text{flat}$$

that is, for all terms M and M' such that $M =_{\mathbf{T}(\mathcal{T})} M'$ we have

$$\big(\Lambda(\text{flat} \circ \uparrow\!\mathsf{ev} \circ t) \circ \uparrow\!\mathsf{app}\big)\ M \qquad =_{\mathcal{T}\to\mathcal{T}} \qquad \big(\text{app} \circ \text{flat}\big)\ M'$$

We prove the result by induction on the proof of $M =_{\mathbf{T}(\mathcal{T})} M'$. Note that since the two terms corresponding to $\Lambda(\text{flat} \circ \uparrow\!\mathsf{ev} \circ t) \circ \uparrow\!\mathsf{app}$ and $\text{app} \circ \text{flat}$ are linear it is sufficient to check the result for the basic case when $(M, M') \in [\mathcal{T}]$ that is,

$$M =_{\mathcal{T}} [M_0] \quad \text{and} \quad M' =_{\mathcal{T}} [M'_0] \qquad \text{and} \qquad (M_0, M'_0) \in \mathcal{T}$$

We can check that for all terms $(N, N') \in \mathcal{T}$

$$\begin{array}{lll}
\big(\Lambda(\mathrm{flat} \circ \uparrow\mathrm{ev} \circ \mathrm{t}) \circ \uparrow\mathrm{app}\big)\ M\ N & =_{\mathcal{T}} & \big(\Lambda(\mathrm{flat} \circ \uparrow\mathrm{ev} \circ \mathrm{t}) \circ \uparrow\mathrm{app}\big)\ [M_0]\ N \\
& =_{\beta} & [M_0]\ (\lambda x_1 \,.\, [x_1])\ (\lambda x_2 \,.\, [\langle x_2; N\rangle])\ (\lambda x_3 \,.\, [(\pi_1 x_3)\ (\pi_2 x_3)])\ \mathbf{I} \\
& =_{\beta} & [M_0]\ (\lambda x_2 \,.\, [\langle x_2; N\rangle])\ (\lambda x_3 \,.\, [(\pi_1 x_3)\ (\pi_2 x_3)])\ \mathbf{I} \\
& =_{\beta} & [\langle M_0; N\rangle]\ (\lambda x_3 \,.\, [(\pi_1 x_3)\ (\pi_2 x_3)])\ \mathbf{I} \\
& =_{\beta} & [M_0\ N]\ \mathbf{I} \\
& =_{\beta} & M_0\ N \\
& =_{\mathcal{T}} & (\lambda x \,.\, x\mathbf{I})\ [M_0']\ N' \\
& =_{\mathcal{T}} & (\lambda x \,.\, x\mathbf{I})\ M'\ N' \\
& = & \big(\mathrm{app} \circ \mathrm{flat}\big)\ M'\ N'
\end{array}$$

□

9.4.5 Adapted model

Lemma 9.32 *If the equational theory $\mathcal{T}_0$ contains the equation ε, then the diagram (ε) commutes.*

Proof. We prove that the diagram (ε) commutes. Recall (proof of Lemma 9.31) first that

$$\begin{array}{lll}
\Lambda(\mathrm{flat} \circ \uparrow\mathrm{ev} \circ \mathrm{t}) & =_{\beta} & \lambda xy \,.\, x(\lambda x_2 \,.\, [\langle x_2; y\rangle])\ (\lambda x_3 \,.\, [(\pi_1 x_3)(\pi_2 x_3)])\ \mathbf{I} \\
\mathrm{lam} \circ \Lambda(\mathrm{flat} \circ \uparrow\mathrm{ev} \circ \mathrm{t}) & =_{\beta} & \lambda z \,.\, \lambda y \,.\, z\ (\lambda z_2 \,.\, [\langle z_2; y\rangle])\ (\lambda z_3 \,.\, [(\pi_1 z_3)(\pi_2 z)])\ \mathbf{I} \\
\mathrm{flat} \circ \Uparrow\mathrm{lam} & =_{\beta} & \lambda z \,.\, z\ (\lambda x_1 \,.\, [\lambda x_2 \,.\, x_1 x_2])\ \mathbf{I}
\end{array}$$

We want to prove that

$$\mathrm{flat} \circ \Uparrow\mathrm{lam} \quad =_{\mathbf{T}(\mathcal{T}\to\mathcal{T})\to\mathcal{T}} \quad \mathrm{lam} \circ \Lambda(\mathrm{flat} \circ \uparrow\mathrm{ev} \circ \mathrm{t})$$

that is, for all terms M and M' such that $M =_{\mathbf{T}(\mathcal{T}\to\mathcal{T})} M'$ we have

$$\Big(\mathrm{flat} \circ \Uparrow\mathrm{lam}\Big)\ M \quad =_{\mathcal{T}} \quad \Big(\mathrm{lam} \circ \Lambda(\mathrm{flat} \circ \uparrow\mathrm{ev} \circ \mathrm{t})\Big)\ M'$$

We prove the result by induction on the proof of $M =_{\mathbf{T}(\mathcal{T}\to\mathcal{T})} M'$.
Base case If $M =_{[\mathcal{T}\to\mathcal{T}]} M'$ that is,

$$M =_{\mathcal{T}} [M_0] \quad \text{and} \quad M' =_{\mathcal{T}} [M_0'] \quad \text{and} \quad (M_0, M_0') \in (\mathcal{T} \to \mathcal{T})$$

We check that

$$\begin{aligned}
\Big(\mathrm{flat}\circ\Uparrow\mathrm{lam}\Big)\,M &=_{\mathcal{T}} [M_0]\,(\lambda x_1\,.\,[\lambda x_2\,.\,x_1x_2])\,\mathbf{I}\\
&=_{\beta} [\lambda x_2\,.\,M_0x_2]\,\mathbf{I}\\
&=_{\beta} \lambda x_2\,.\,M_0x_2\\
&=_{\mathcal{T}} \lambda y\,.\,M_0'\,y\\
&=_{\beta} \lambda y\,.\,[M_0'\,y]\,\mathbf{I}\\
&=_{\beta} \lambda y\,.\,[\langle M_0';\,y\rangle]\,(\lambda z_3\,.\,[(\pi_1z_3)(\pi_2z_3)])\,\mathbf{I}\\
&=_{\beta} \lambda y\,.\,[M_0']\,(\lambda z_2\,.\,[\langle z_2;\,y\rangle])\,(\lambda z_3\,.\,[(\pi_1z_3)(\pi_2z_3)])\,\mathbf{I}\\
&=_{\mathcal{T}} \Big(\mathrm{lam}\circ\Lambda(\mathrm{flat}\circ\Uparrow\mathrm{ev}\circ t)\Big)\,M'
\end{aligned}$$

Induction case If $M =_{\mathcal{T}} M_1 /\!\!/ M_2$ and $M' =_{\mathcal{T}} M_1' /\!\!/ M_2'$ with $(M_1, M_1') \in [\mathcal{T}\to\mathcal{T}]^+$ and $(M_2, M_2') \in [\mathcal{T}\to\mathcal{T}]^+$ we can check that

$$\begin{aligned}
&\Big(\mathrm{lam}\circ\Lambda(\mathrm{flat}\circ\Uparrow\mathrm{ev}\circ t)\Big)\,(M_1 /\!\!/ M_2)\\
=_{\beta}\ & \lambda y\,.\,(M_1 /\!\!/ M_2)\,(\lambda z_2\,.\,[\langle z_2;y\rangle])\,(\lambda z_3\,.\,[(\pi_1z_3)(\pi_2z)])\,\mathbf{I}\\
=_{\delta}\ & \lambda y\,.\,\big(M_1\,(\lambda z_2\,.\,[\langle z_2;y\rangle])\,(\lambda z_3\,.\,[(\pi_1z_3)(\pi_2z)])\,\mathbf{I}\big) /\!\!/ \big(M_2\,(\lambda z_2\,.\,[\langle z_2;y\rangle])\,(\lambda z_3\,.\,[(\pi_1z_3)(\pi_2z)])\,\mathbf{I}\big)\\
=_{\varepsilon}\ & \Big(\lambda y\,.\,M_1\,(\lambda z_2\,.\,[\langle z_2;y\rangle])\,(\lambda z_3\,.\,[(\pi_1z_3)(\pi_2z)])\,\mathbf{I}\Big) /\!\!/ \Big(\lambda y\,.\,M_2\,(\lambda z_2\,.\,[\langle z_2;y\rangle])\,(\lambda z_3\,.\,[(\pi_1z_3)(\pi_2z)])\,\mathbf{I}\Big)\\
\overset{\text{IH}}{=}\ & (\mathrm{flat}\circ\Uparrow\mathrm{lam})\,M_1' \quad /\!\!/ \quad (\mathrm{flat}\circ\Uparrow\mathrm{lam})\,M_2'\\
=_{\beta}\ & M_1'\,(\lambda x_1\,.\,[\lambda x_2\,.\,x_1x_2])\,\mathbf{I} \quad /\!\!/ \quad M_2'\,(\lambda x_1\,.\,[\lambda x_2\,.\,x_1x_2])\,\mathbf{I}\\
=_{\delta}\ & (M_1' /\!\!/ M_2')\,(\lambda x_1\,.\,[\lambda x_2\,.\,x_1x_2])\,\mathbf{I}\\
=_{\beta}\ & \big(\mathrm{flat}\circ\Uparrow\mathrm{lam}\big)\,(M_1' /\!\!/ M_2')
\end{aligned}$$

where IH indicates the two applications of the induction hypothesis on the proofs of $(M_1, M_1') \in [\mathcal{T}\to\mathcal{T}]^+$ and $(M_2, M_2') \in [\mathcal{T}\to\mathcal{T}]^+$.

□

Lemma 9.33 *If $\mathcal{T}$ contains the equation η, then for all $\mathcal{T}_0$-theories, the triple $(\mathbf{T}, \lambda x\,.\,x, \lambda xy\,.\,xy)$ defines an isomorphism between $\mathbf{T}$ and $\mathbf{T}\to\mathbf{T}$*

Proof. We check first that $\lambda x\,.\,x \in \mathcal{T}^{\mathcal{T}}\to\mathcal{T}$. Let $M_1 =_{\mathcal{T}^{\mathcal{T}}} M_2$. We want to prove

$$(\lambda x\,.\,x)\,M_1 =_{\mathcal{T}} (\lambda x\,.\,x)\,M_2$$

We can equivalently prove that $M_1 =_{\mathcal{T}} M_2$. In fact, $(x,x) \in \mathcal{T}$ and then $M_1x =_{\mathcal{T}} M_2x$. Since $\mathcal{T}$ is a congruence, we have

$$\lambda x\,.\,(M_1x) =_{\mathcal{T}} \lambda x\,.\,(M_2x).$$

Using the η rule we conclude that $M_1 =_{\mathcal{T}} M_2$.

We check that

$$\begin{array}{rll} \mathrm{lam} \circ \mathrm{app} & =_\beta & \lambda z\,.\,\mathrm{lam}(z) \\ & =_\beta & \lambda z\,.\,\lambda y\,.\,zy \\ & =_\eta & \lambda z\,.\,z \end{array}$$

□

9.4.6 Completeness result

To conclude the proof of Theorem 9.28, we need to ensure that the congruence induced by the model D is precisely the congruence $\mathcal{T}$. This relies on the following lemma:

Lemma 9.34 *Let $\ell = [x_1, \ldots, x_n]$ be a list of variables. For all terms M such that $FV(M) \subset \ell$, we have:*

$$[\![M]\!]^{\mathsf{D}}_\ell \;=_{\beta\delta}\; \lambda z\,.\,M\{x_1 := \pi^{x_1}_\ell z; \ldots; x_n := \pi^{x_n}_\ell z\}$$

(where z is a fresh variable).

Proof. We prove the result by induction on M. We only check the result for the last case that is, if $M = M_1 \mathbin{/\!/} M_2$ then we have

$$\begin{array}{rll} [\![M_1 \mathbin{/\!/} M_2]\!]^{\mathsf{D}}_\ell & = & \mathsf{par} \circ \langle [\![M_1]\!]^{\mathsf{D}}_\ell ; [\![M_2]\!]^{\mathsf{D}}_\ell \rangle \\ & \overset{\text{IH}}{=_{\beta\delta}} & \mathsf{par} \circ \langle \lambda z\,.\,M_1\{x_1 := \pi^{x_1}_\ell(z); \ldots; x_n := \pi^{x_n}_\ell(z)\}; \\ & & \qquad \lambda z\,.\,M_2\{x_1 := \pi^{x_1}_\ell(z); \ldots; x_n := \pi^{x_n}_\ell(z)\}\rangle \\ & =_\beta & \lambda y\,.\;\mathsf{par}\,(\lambda p\,.\,p\,(M_1\{x_1 := \pi^{x_1}_\ell(y); \ldots; x_n := \pi^{x_n}_\ell(y)\}) \\ & & \qquad\qquad (M_2\{x_1 := \pi^{x_1}_\ell(y); \ldots; x_n := \pi^{x_n}_\ell(y)\})) \\ & =_\beta & \lambda y\,.\,\mathrm{flat} \circ \mathrm{u}(\lambda p\,.\,p\,(M_1\{x_1 := \pi^{x_1}_\ell(y); \ldots; x_n := \pi^{x_n}_\ell(y)\}) \\ & & \qquad\qquad (M_2\{x_1 := \pi^{x_1}_\ell(y); \ldots; x_n := \pi^{x_n}_\ell(y)\})) \\ & =_\beta & \lambda y\,.\,\Big([M_1\{x_1 := \pi^{x_1}_\ell(y); \ldots; x_n := \pi^{x_n}_\ell(y)\}] \;\;/\!/ \\ & & \qquad [M_2\{x_1 := \pi^{x_1}_\ell(y); \ldots; x_n := \pi^{x_n}_\ell(y)\}]\Big)\;\mathbf{I} \\ & =_\delta & \lambda y\,.\,[M_1\{x_1 := \pi^{x_1}_\ell(y); \ldots; x_n := \pi^{x_n}_\ell(y)\}]\;\mathbf{I} \;\;/\!/ \\ & & \qquad [M_2\{x_1 := \pi^{x_1}_\ell(y); \ldots; x_n := \pi^{x_n}_\ell(y)\}]\;\mathbf{I} \\ & =_\beta & \lambda y\,.\,M_1\{x_1 := \pi^{x_1}_\ell(y); \ldots; x_n := \pi^{x_n}_\ell(y)\} \;\;/\!/ \\ & & \qquad M_2\{x_1 := \pi^{x_1}_\ell(y); \ldots; x_n := \pi^{x_n}_\ell(y)\} \\ & = & \lambda y\,.\,(M_1 \mathbin{/\!/} M_2)\{x_1 := \pi^{x_1}_\ell(y); \ldots; x_n := \pi^{x_n}_\ell(y)\} \end{array}$$

□

Theorem 9.35 (Completeness) *For all lists of variables ℓ, and for all terms M_1 and M_2 such that $FV(M_1) \cup FV(M_2) \subset \ell$, one has*

$$[\![M_1]\!]^D_\ell = [\![M_2]\!]^D_\ell \qquad \textit{iff} \qquad M_1 =_{\mathcal{T}} M_2 \,.$$

Proof. The soudness proposition gives one implication. We show the other one. Let us assume that $[\![M_1]\!]^D_\ell = [\![M_2]\!]^D_\ell$ and $l = [x_1, \ldots, x_n]$. Since D is a congruence we have $\langle x_1, \ldots, x_n \rangle \in D^n$ (by abuse we denote by l this term). We check that

$$\begin{array}{rcl} M_1 & = & M_1\{x_1 := x_1; \ldots; x_n := x_n\} \\ & =_\beta & M_1\{x_1 := \pi^{x_1}_\ell(l); \ldots; x_n := \pi^{x_n}_\ell(l)\} \\ & =_\beta & \Big(\lambda z \,.\, M_1\{x_1 := \pi^{x_1}_\ell(z); \ldots; x_n := \pi^{x_n}_\ell(z)\}\Big) l \\ & =_{\beta\delta} & [\![M_1]\!]^D_\ell l \\ & =_{\mathcal{T}} & [\![M_2]\!]^D_\ell l \\ & =_{\beta\delta} & M_2 \end{array}$$

□

Conclusion

In the previous chapter, we introduce a sound categorical semantics for the parallel λ-calculus that we proved here its completeness. In the following chapters, we give two abstract methods to build such models.

Chapter 10

Constructing a model of the parallel λ-calculus

Context We have defined a sound and complete categorical semantics for the parallel λ-calculus. We have given in Chapter 8 several examples of models of the parallel λ-calculus in the category of Scott domains.

Contributions We present two methods for constructing a model of the parallel λ-calculus from a given ccc and a given strong aggregation monad. Both construction methods rely on the existence of objects satisfying particular equations. They can be fruitfully used in the category of Scott domains where such equations have many interesting solutions.

Outline of the chapter We give a first method in Section 10.1 and a second one in Section 10.2.

10.1 First method

In this section, we assume that $\mathfrak{C}$ is a Cartesian closed category equipped with a strong aggregation monad $(\mathbf{T}, \eta, \mu, \mathsf{t}, \mathsf{u})$. Our aim is to show that

Theorem 10.1 *If* $\mathsf{D} \in \mathbf{Obj}\mathfrak{C}$ *is such that* $\mathsf{D} \cong (\mathbf{T}\mathsf{D})^{\mathsf{D}}$*, then* D *can be given all the structures of a model of the parallel* λ*-calculus.*

The rest of this section is devoted to the proof of Theorem 10.1. We can notice that we will never use the hypothesis (agg-t) and (agg-μ).

In what follows, we assume that D is an object equipped with a pair of arrows

$$\mathsf{D} \underset{\text{fold}}{\overset{\text{unfold}}{\rightleftarrows}} (\mathbf{T}\mathsf{D})^{\mathsf{D}}$$

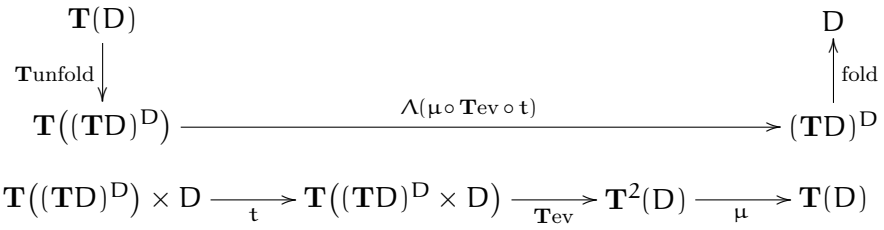
such that $\quad \text{unfold} \circ \text{fold} = \text{id}_{\mathbf{T}\mathsf{D}^{\mathsf{D}}} \quad$ and $\quad \text{fold} \circ \text{unfold} = \text{id}_{\mathsf{D}}$.

10.1.1 Algebraicity

The arrow $\mathsf{flat} \in \mathfrak{C}[\mathbf{T}(\mathsf{D}); \mathsf{D}]$ is defined by

$$\mathsf{flat} \quad = \quad \mathsf{fold} \circ \Lambda(\mu \circ \mathbf{T}\mathsf{ev} \circ \mathsf{t}) \circ \mathbf{T}\mathsf{unfold}$$

Its construction is depicted below:



We check that

$$\begin{array}{rcll}
\mathsf{flat} \circ \eta & = & \mathsf{fold} \circ \Lambda(\mu \circ \mathbf{T}\mathsf{ev} \circ \mathsf{t}) \circ \mathbf{T}\mathsf{unfold} \circ \eta & \\
& = & \mathsf{fold} \circ \Lambda(\mu \circ \mathbf{T}\mathsf{ev} \circ \mathsf{t}) \circ \eta \circ \mathsf{unfold} & (\text{nat-}\eta) \\
& = & \mathsf{fold} \circ \Lambda\big(\mu \circ \mathbf{T}\mathsf{ev} \circ \mathsf{t} \circ (\eta \times \mathsf{id})\big) \circ \mathsf{unfold} & (\text{exp-}\circ) \\
& = & \mathsf{fold} \circ \Lambda\big(\mu \circ \mathbf{T}\mathsf{ev} \circ \eta\big) \circ \mathsf{unfold} & (\text{str-}\eta) \\
& = & \mathsf{fold} \circ \Lambda\big(\mu \circ \eta \circ \mathsf{ev}\big) \circ \mathsf{unfold} & (\text{nat-}\eta) \\
& = & \mathsf{fold} \circ \Lambda\big(\mathsf{ev}\big) \circ \mathsf{unfold} & (\text{mon-2}) \\
& = & \mathsf{fold} \circ \mathsf{unfold} & (\text{exp-}\eta) \\
& = & \mathsf{id} & (\text{fold/unfold})
\end{array}$$

Writing $\mathsf{f} = \Lambda(\mu \circ \mathbf{T}\mathsf{ev} \circ \mathsf{t})$, we check that

$$\begin{array}{rcll}
\mathsf{flat} \circ \mathbf{T}\mathsf{flat} & = & \mathsf{fold} \circ \mathsf{f} \circ \mathbf{T}\mathsf{unfold} \circ \mathbf{T}(\mathsf{fold} \circ \mathsf{f} \circ \mathbf{T}\mathsf{unfold}) & \\
& = & \mathsf{fold} \circ \Lambda(\mu \circ \mathbf{T}\mathsf{ev} \circ \mathsf{t}) \circ \mathbf{T}\mathsf{f} \circ \mathbf{T}^2\mathsf{unfold} & (\text{fold/unfold}) \\
& = & \mathsf{fold} \circ \Lambda(\mu \circ \mathbf{T}\mathsf{ev} \circ \mathsf{t} \circ (\mathbf{T}\mathsf{f} \times \mathsf{id})) \circ \mathbf{T}^2\mathsf{unfold} & (\text{exp-nat}) \\
& = & \mathsf{fold} \circ \Lambda(\mu \circ \mathbf{T}\mathsf{ev} \circ \mathbf{T}(\mathsf{f} \times \mathsf{id}) \circ \mathsf{t}) \circ \mathbf{T}^2\mathsf{unfold} & (\text{t-nat}) \\
& = & \mathsf{fold} \circ \Lambda(\mu \circ \mathbf{T}(\mu \circ \mathbf{T}\mathsf{ev} \circ \mathsf{t}) \circ \mathsf{t}) \circ \mathbf{T}^2\mathsf{unfold} & (\text{exp-}\beta) \\
& = & \mathsf{fold} \circ \Lambda(\mu \circ \mu \circ \mathbf{T}^2\mathsf{ev} \circ \mathbf{T}\mathsf{t} \circ \mathsf{t}) \circ \mathbf{T}^2\mathsf{unfold} & (\text{mon-3}) \\
& = & \mathsf{fold} \circ \Lambda(\mu \circ \mathbf{T}\mathsf{ev} \circ \mu \circ \mathbf{T}\mathsf{t} \circ \mathsf{t}) \circ \mathbf{T}^2\mathsf{unfold} & (\mu\text{-nat}) \\
& = & \mathsf{fold} \circ \Lambda(\mu \circ \mathbf{T}\mathsf{ev} \circ \mathsf{t} \circ (\mu \times \mathsf{id})) \circ \mathbf{T}^2\mathsf{unfold} & (\text{str-}\mu) \\
& = & \mathsf{fold} \circ \Lambda(\mu \circ \mathbf{T}\mathsf{ev} \circ \mathsf{t}) \circ \mu \circ \mathbf{T}^2\mathsf{unfold} & (\text{exp-}\circ) \\
& = & \mathsf{fold} \circ \Lambda(\mu \circ \mathbf{T}\mathsf{ev} \circ \mathsf{t}) \circ \mathbf{T}\mathsf{unfold} \circ \mu \quad = \quad \mathsf{flat} \circ \mu & (\mu\text{-nat})
\end{array}$$

10.1.2 Reflexivity

Once the structure of $\mathbf{T}$-algebra of D has been outlined, it is easy to check that D is a reflexive object of the category. For that, we define two arrows $\mathsf{lam} \in \mathfrak{C}[\mathsf{D}^\mathsf{D}; \mathsf{D}]$ and $\mathsf{app} \in \mathfrak{C}[\mathsf{D}; \mathsf{D}^\mathsf{D}]$ by setting:

$$\mathsf{lam} \;=\; \mathsf{fold} \circ \Lambda(\eta \circ \mathsf{ev}) \qquad \text{and} \qquad \mathsf{app} \;=\; \Lambda(\mathsf{flat} \circ \mathsf{ev}) \circ \mathsf{unfold}\,.$$

The construction of these arrows is depicted below:

$$\mathrm{lam} \;=\; \begin{cases} \mathsf{D}^{\mathsf{D}} \xrightarrow{\Lambda(\eta \circ \mathrm{ev})} (\mathbf{T}\mathsf{D})^{\mathsf{D}} \xrightarrow{\mathrm{fold}} \mathsf{D} \\ \mathsf{D}^{\mathsf{D}} \times \mathsf{D} \xrightarrow[\mathrm{ev}]{} \mathsf{D} \xrightarrow[\eta]{} \mathbf{T}(\mathsf{D}) \end{cases}$$

$$\mathrm{app} \;=\; \begin{cases} \mathsf{D} \xrightarrow{\mathrm{unfold}} (\mathbf{T}\mathsf{D})^{\mathsf{D}} \xrightarrow{\Lambda(\mathrm{flat} \circ \mathrm{ev})} \mathsf{D}^{\mathsf{D}} \\ (\mathbf{T}\mathsf{D})^{\mathsf{D}} \times \mathsf{D} \xrightarrow[\mathrm{ev}]{} \mathbf{T}(\mathsf{D}) \xrightarrow[\mathrm{flat}]{} \mathsf{D} \end{cases}$$

We check that

$$\begin{array}{rcll} \mathrm{app} \circ \mathrm{lam} & = & \Lambda(\mathrm{flat} \circ \mathrm{ev}) \circ \mathrm{unfold} \circ \mathrm{fold} \circ \Lambda(\eta \circ \mathrm{ev}) & \\ & = & \Lambda(\mathrm{flat} \circ \mathrm{ev}) \circ \Lambda(\eta \circ \mathrm{ev}) & (\mathrm{unfold/fold}) \\ & = & \Lambda\big(\mathrm{flat} \circ \mathrm{ev} \circ (\Lambda(\eta \circ \mathrm{ev}) \times \mathrm{id})\big) & (\mathrm{exp}\text{-}\circ) \\ & = & \Lambda(\mathrm{flat} \circ \eta \circ \mathrm{ev}) & (\mathrm{exp}\text{-}\beta) \\ & = & \Lambda(\mathrm{ev}) & (\mathrm{alg}\text{-}1) \\ & = & \mathrm{id} & (\mathrm{exp}\text{-}\eta) \end{array}$$

so that $(\mathrm{lam}, \mathrm{app})$ is a retraction of D onto D^{D}.

10.1.3 Distributivity axiom

We prove that the diagram (δ) commutes. We check that

$$\begin{array}{rcll} \mathrm{app} \circ \mathrm{flat} & = & \Lambda(\mathrm{flat} \circ \mathrm{ev}) \circ \mathrm{unfold} \circ \mathrm{flat} & \\ & = & \Lambda(\mathrm{flat} \circ \mathrm{ev}) \circ \underbrace{\mathrm{unfold} \circ \mathrm{fold}} \circ \Lambda(\mu \circ \mathbf{T}\mathrm{ev} \circ \mathsf{t}) \circ \mathbf{T}\mathrm{unfold} & (\mathrm{def}\text{-}\mathrm{flat}) \\ & = & \Lambda(\mathrm{flat} \circ \mathrm{ev}) \circ \qquad\qquad \Lambda(\mu \circ \mathbf{T}\mathrm{ev} \circ \mathsf{t}) \circ \mathbf{T}\mathrm{unfold} & (\mathrm{unfold/fold}) \\ & = & \Lambda(\mathrm{flat} \circ \mathrm{ev} \circ \Lambda(\mu \circ \mathbf{T}\mathrm{ev} \circ \mathsf{t}) \times \mathrm{id}) \circ \mathbf{T}\mathrm{unfold} & (\mathrm{exp}\text{-}\circ) \\ & = & \Lambda(\mathrm{flat} \circ \mu \circ \mathbf{T}\mathrm{ev} \circ \mathsf{t}) \circ \mathbf{T}\mathrm{unfold} & (\mathrm{exp}\text{-}\beta) \\ & = & \Lambda(\mathrm{flat} \circ \mathbf{T}\mathrm{flat} \circ \mathbf{T}\mathrm{ev} \circ \mathsf{t}) \circ \mathbf{T}\mathrm{unfold} & (\mathrm{alg}\text{-}2) \\ & = & \Lambda(\mathrm{flat} \circ \mathbf{T}\mathrm{ev} \circ \mathbf{T}(\Lambda(\mathrm{flat} \circ \mathrm{ev}) \times \mathrm{id}) \circ \mathsf{t}) \circ \mathbf{T}\mathrm{unfold} & (\mathrm{exp}\text{-}\beta) \\ & = & \Lambda(\mathrm{flat} \circ \mathbf{T}\mathrm{ev} \circ \mathsf{t} \circ (\mathbf{T}\Lambda(\mathrm{flat} \circ \mathrm{ev}) \times \mathrm{id})) \circ \mathbf{T}\mathrm{unfold} & (\mathrm{nat}\text{-}\mathsf{t}) \\ & = & \Lambda(\mathrm{flat} \circ \mathbf{T}\mathrm{ev} \circ \mathsf{t}) \circ \mathbf{T}\Lambda(\mathrm{flat} \circ \mathrm{ev}) \circ \mathbf{T}\mathrm{unfold} & (\mathrm{exp}\text{-}\circ) \\ & = & \Lambda(\mathrm{flat} \circ \mathbf{T}\mathrm{ev} \circ \mathsf{t}) \circ \mathbf{T}\mathrm{app} & (\mathrm{def}\text{-}\mathrm{app}) \end{array}$$

10.1.4 Typical use

The typical use of this theorem in the category of Scott domains is the following: assume that $\langle \mathbf{T}, \mathsf{u} \rangle$ is a strong aggregation monad in the category of Scott domains whose underlying endofunctor $\mathbf{T}$ is ω^{op}-continuous (i.e. preserves limits on ω^{op}-chains). Then the correspondence

$$\mathsf{X} \mapsto (\mathbf{T}\mathsf{X})^{\mathsf{X}}$$

induces an ω-cocontinuous (covariant) endofunctor in the category $\mathbf{Scott}^{\mathrm{ip}}$ of Scott domains equipped with injection-projection pairs. Starting from a domain D_0 equipped

with an injection retraction pair $D_0 \rightarrowtail (\mathbf{T}D_0)^{D_0}$, it is easy to build a smallest fixpoint $D \simeq (\mathbf{T}D)^D$ containing D_0 (in the sense of injection-retraction pairs).

Notice that this way of constructing models in the category of Scott domains is not limited to the lower powerdomain monad $\mathcal{P}_l$, but that it can be also used with:

- The list monad, which defines an associative aggregation monad using the concatenation function;
- The free magma monad $\mathbf{T}X$, defined as the smallest fixpoint of the equation

$$\mathbf{T}X \;=\; \big(X + (\mathbf{T}X \times \mathbf{T}X)\big)_\perp\,,$$

  that induces an aggregation monad which is neither associative, commutative nor idempotent.

10.2 Second method

In this section, we assume that $\mathfrak{C}$ is a Cartesian closed category equipped with a strong aggregation monad (T, η, μ, t, u) and a $\mathbf{T}$-algebra (R, flat_R).

We denote by D^ω the infinite product of D. We use the standard functions coming with infinite products:

$$\begin{array}{lll} \mathrm{hd}: & D^\omega \to D & = \pi_0 \\ \mathrm{tl}: & D^\omega \to D^\omega & = \langle \pi_{i+1} \rangle_{i\in\mathbb{N}} \\ \mathrm{cons}: & D \times D^\omega \to D^\omega & = \langle f_i \rangle_{i\in\mathbb{N}} \\ & f_0 = \pi_1 & \\ & f_{i+1} = \pi_i \circ \pi_2 & \end{array}$$

We will often use the following equation:

$$\begin{array}{rcll} \mathrm{cons} \circ \langle \mathrm{hd}; \mathrm{tl} \rangle & = & \mathrm{id}_{D^\omega} & \text{(prod-i)} \\ \langle \mathrm{hd}; \mathrm{tl} \rangle \circ \mathrm{cons} & = & \mathrm{id}_{D\times D^\omega} & \end{array}$$

Theorem 10.2 *If $D \in \mathbf{Obj}\mathfrak{C}$ is such that $D \cong R^{D^\omega}$ then D can be given the additional structures of an ε-model of the parallel λ-calculus.*

In what follows we assume that D is an object equipped with a pair of arrows

$$D \underset{\text{fold}}{\overset{\text{unfold}}{\rightleftarrows}} R^{D^\omega}$$

such that

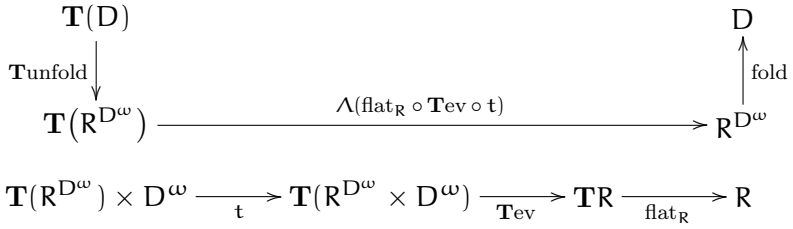
$$\mathrm{unfold} \circ \mathrm{fold} = \mathrm{id}_{R^{D^\omega}} \quad \text{and} \quad \mathrm{fold} \circ \mathrm{unfold} = \mathrm{id}_D\,.$$

10.2.1 Algebraicity

The arrow $\mathsf{flat}_{\mathsf{D}} \in \mathfrak{C}[\mathbf{T}(\mathsf{D}); \mathsf{D}]$ is defined by

$$\mathsf{flat}_{\mathsf{D}} \quad = \quad \mathsf{fold} \circ \Lambda(\mathsf{flat}_{\mathsf{R}} \circ \mathbf{T}\mathsf{ev} \circ \mathsf{t}) \circ \mathbf{T}\mathsf{unfold}$$

Its construction is depicted below:



We check that

$$\begin{array}{rcll}
\mathsf{flat}_{\mathsf{D}} \circ \eta & = & \mathsf{fold} \circ \Lambda(\mathsf{flat}_{\mathsf{R}} \circ \mathbf{T}\mathsf{ev} \circ \mathsf{t}) \circ \mathbf{T}\mathsf{unfold} \circ \eta & \\
& = & \mathsf{fold} \circ \Lambda(\mathsf{flat}_{\mathsf{R}} \circ \mathbf{T}\mathsf{ev} \circ \mathsf{t}) \circ \eta \circ \mathsf{unfold} & (\text{nat-}\eta) \\
& = & \mathsf{fold} \circ \Lambda\big(\mathsf{flat}_{\mathsf{R}} \circ \mathbf{T}\mathsf{ev} \circ \mathsf{t} \circ (\eta \times \mathsf{id})\big) \circ \mathsf{unfold} & (\text{exp-}\circ) \\
& = & \mathsf{fold} \circ \Lambda\big(\mathsf{flat}_{\mathsf{R}} \circ \mathbf{T}\mathsf{ev} \circ \eta\big) \circ \mathsf{unfold} & (\text{str-}\eta) \\
& = & \mathsf{fold} \circ \Lambda\big(\mathsf{flat}_{\mathsf{R}} \circ \eta \circ \mathsf{ev}\big) \circ \mathsf{unfold} & (\text{nat-}\eta) \\
& = & \mathsf{fold} \circ \Lambda\big(\mathsf{ev}\big) \circ \mathsf{unfold} & (\text{alg-1}) \\
& = & \mathsf{fold} \circ \mathsf{unfold} & (\text{exp-}\eta) \\
& = & \mathsf{id} & (\text{fold/unfold})
\end{array}$$

Writing $\mathsf{f} = \Lambda(\mathsf{flat}_{\mathsf{R}} \circ \mathbf{T}\mathsf{ev} \circ \mathsf{t})$, we check that

$$\begin{array}{rcll}
\mathsf{flat}_{\mathsf{D}} \circ \mathbf{T}\mathsf{flat}_{\mathsf{D}} & = & \mathsf{fold} \circ \mathsf{f} \circ \mathbf{T}\mathsf{unfold} \circ \mathbf{T}(\mathsf{fold} \circ \mathsf{f} \circ \mathbf{T}\mathsf{unfold}) & \\
& = & \mathsf{fold} \circ \Lambda(\mathsf{flat}_{\mathsf{R}} \circ \mathbf{T}\mathsf{ev} \circ \mathsf{t}) \circ \mathbf{T}\mathsf{f} \circ \mathbf{T}^2\mathsf{unfold} & (\text{fold/unfold}) \\
& = & \mathsf{fold} \circ \Lambda(\mathsf{flat}_{\mathsf{R}} \circ \mathbf{T}\mathsf{ev} \circ \mathsf{t} \circ (\mathbf{T}\mathsf{f} \times \mathsf{id})) \circ \mathbf{T}^2\mathsf{unfold} & (\text{exp-}\circ) \\
& = & \mathsf{fold} \circ \Lambda(\mathsf{flat}_{\mathsf{R}} \circ \mathbf{T}\mathsf{ev} \circ \mathbf{T}(\mathsf{f} \times \mathsf{id}) \circ \mathsf{t}) \circ \mathbf{T}^2\mathsf{unfold} & (\text{t-nat}) \\
& = & \mathsf{fold} \circ \Lambda(\mathsf{flat}_{\mathsf{R}} \circ \mathbf{T}(\mathsf{flat}_{\mathsf{R}} \circ \mathbf{T}\mathsf{ev} \circ \mathsf{t}) \circ \mathsf{t}) \circ \mathbf{T}^2\mathsf{unfold} & (\text{exp-}\beta) \\
& = & \mathsf{fold} \circ \Lambda(\mathsf{flat}_{\mathsf{R}} \circ \mu \circ \mathbf{T}^2\mathsf{ev} \circ \mathbf{T}\mathsf{t} \circ \mathsf{t}) \circ \mathbf{T}^2\mathsf{unfold} & (\text{alg-2}) \\
& = & \mathsf{fold} \circ \Lambda(\mathsf{flat}_{\mathsf{R}} \circ \mathbf{T}\mathsf{ev} \circ \mu \circ \mathbf{T}\mathsf{t} \circ \mathsf{t}) \circ \mathbf{T}^2\mathsf{unfold} & (\mu\text{-nat}) \\
& = & \mathsf{fold} \circ \Lambda(\mathsf{flat}_{\mathsf{R}} \circ \mathbf{T}\mathsf{ev} \circ \mathsf{t} \circ (\mu \times \mathsf{id})) \circ \mathbf{T}^2\mathsf{unfold} & (\text{str-}\mu) \\
& = & \mathsf{fold} \circ \Lambda(\mathsf{flat}_{\mathsf{R}} \circ \mathbf{T}\mathsf{ev} \circ \mathsf{t}) \circ \mu \circ \mathbf{T}^2\mathsf{unfold} & (\text{exp-}\circ) \\
& = & \mathsf{fold} \circ \Lambda(\mathsf{flat}_{\mathsf{R}} \circ \mathbf{T}\mathsf{ev} \circ \mathsf{t}) \circ \mathbf{T}\mathsf{unfold} \circ \mu \quad = \quad \mathsf{flat}_{\mathsf{D}} \circ \mu & (\mu\text{-nat})
\end{array}$$

Note that the proofs of the last two equations are in a one-to-one correspondence with the one of Theorem 10.1. The morphism μ is replaced here by $\mathsf{flat}_{\mathsf{R}}$.

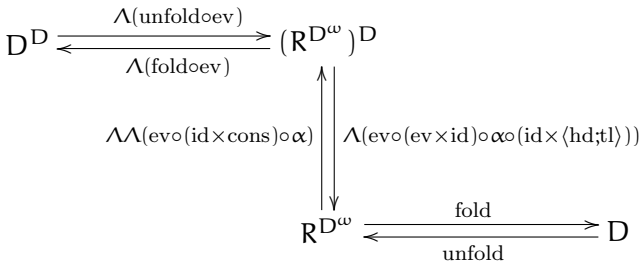
10.2.2 Reflexivity

We check that D is a reflexive object of the category. For that, we define two arrows $\mathsf{lam} \in \mathfrak{C}[\mathsf{D}^{\mathsf{D}}; \mathsf{D}]$ and $\mathsf{app} \in \mathfrak{C}[\mathsf{D}; \mathsf{D}^{\mathsf{D}}]$ by setting:

$$\mathsf{lam} \quad = \quad \mathsf{fold} \circ \Lambda(\mathsf{ev} \circ (\mathsf{ev} \times \mathsf{id}) \circ \alpha \circ (\mathsf{id} \times \langle \mathsf{hd}; \mathsf{tl} \rangle)) \circ \Lambda(\mathsf{unfold} \circ \mathsf{ev})$$

$$\text{and} \quad \mathsf{app} \quad = \quad \Lambda(\mathsf{fold} \circ \mathsf{ev}) \circ \Lambda\Lambda(\mathsf{ev} \circ (\mathsf{id} \times \mathsf{cons}) \circ \alpha) \circ \mathsf{unfold}$$

The construction of these arrows is depicted below:



We first prove that $\mathsf{D}^{\mathsf{D}} \cong (\mathsf{R}^{\mathsf{D}^\omega})^{\mathsf{D}}$. We check that

$$\begin{array}{lll}
& \Lambda(\mathrm{unfold} \circ \mathrm{ev}_{\mathsf{D},\mathsf{D}}) \circ \Lambda(\mathrm{fold} \circ \mathrm{ev}_{\mathsf{R}^{\mathsf{D}^\omega},\mathsf{D}}) & \\
= & \Lambda(\mathrm{unfold} \circ \mathrm{ev}_{\mathsf{D},\mathsf{D}} \circ \Lambda(\mathrm{fold} \circ \mathrm{ev}_{\mathsf{R}^{\mathsf{D}^\omega},\mathsf{D}}) \times \mathrm{id}_{\mathsf{D}}) & (\text{exp-}\circ) \\
= & \Lambda(\mathrm{unfold} \circ \mathrm{fold} \circ \mathrm{ev}_{\mathsf{R}^{\mathsf{D}^\omega},\mathsf{D}}) & (\text{exp-}\beta) \\
= & \Lambda(\mathrm{ev}_{\mathsf{R}^{\mathsf{D}^\omega},\mathsf{D}}) & (\text{fold/unfold}) \\
= & \mathrm{id}_{(\mathsf{R}^{\mathsf{D}^\omega})^{\mathsf{D}}} & (\text{exp-}\eta)
\end{array}$$

and

$$\begin{array}{lll}
& \Lambda(\mathrm{fold} \circ \mathrm{ev}_{\mathsf{R}^{\mathsf{D}^\omega},\mathsf{D}}) \circ \Lambda(\mathrm{unfold} \circ \mathrm{ev}_{\mathsf{D},\mathsf{D}}) & \\
= & \Lambda(\mathrm{fold} \circ \mathrm{ev}_{\mathsf{R}^{\mathsf{D}^\omega},\mathsf{D}} \circ \Lambda(\mathrm{unfold} \circ \mathrm{ev}_{\mathsf{D},\mathsf{D}}) \times \mathrm{id}_{\mathsf{D}}) & (\text{exp-}\circ) \\
= & \Lambda(\mathrm{fold} \circ \mathrm{unfold} \circ \mathrm{ev}_{\mathsf{D},\mathsf{D}}) & (\text{exp-}\beta) \\
= & \Lambda(\mathrm{ev}_{\mathsf{D},\mathsf{D}}) & (\text{fold/unfold}) \\
= & \mathrm{id}_{\mathsf{D}^{\mathsf{D}}} & (\text{exp-}\eta)
\end{array}$$

We now prove that $(\mathsf{R}^{\mathsf{D}^\omega})^{\mathsf{D}} \cong \mathsf{R}^{\mathsf{D}^\omega}$. Writing $\mathsf{f} = \mathrm{ev} \circ (\mathrm{ev} \times \mathrm{id}) \circ \alpha \circ (\mathrm{id} \times \langle \mathrm{hd}; \mathrm{tl} \rangle)$ and $\mathsf{g} = \mathrm{ev} \circ (\mathrm{id} \times \mathrm{cons}) \circ \alpha$ we check that

$$\begin{array}{lll}
& \Lambda\mathsf{f} \circ \Lambda\Lambda\mathsf{g} & \\
\overset{(\mathrm{i})}{=} & \Lambda(\mathrm{ev} \circ (\mathrm{ev} \times \mathrm{id}) \circ \alpha \circ (\mathrm{id} \times \langle \mathrm{hd}; \mathrm{tl} \rangle) \circ \Lambda\Lambda\mathsf{g} \times \mathrm{id}) & (\text{exp-}\circ) \\
= & \Lambda(\mathrm{ev} \circ (\mathrm{ev} \times \mathrm{id}) \circ \alpha \circ (\Lambda\Lambda\mathsf{g} \times \mathrm{id}) \circ (\mathrm{id} \times \langle \mathrm{hd}; \mathrm{tl} \rangle)) & \\
= & \Lambda(\mathrm{ev} \circ (\mathrm{ev} \times \mathrm{id}) \circ ((\Lambda\Lambda\mathsf{g} \times \mathrm{id}) \times \mathrm{id}) \circ \alpha \circ (\mathrm{id} \times \langle \mathrm{hd}; \mathrm{tl} \rangle)) & (\text{nat-}\alpha) \\
\overset{(\mathrm{v})}{=} & \Lambda(\mathrm{ev} \circ (\Lambda\mathsf{g} \times \mathrm{id}) \circ \alpha \circ (\mathrm{id} \times \langle \mathrm{hd}; \mathrm{tl} \rangle)) & (\text{exp-}\beta) \\
\overset{(\mathrm{vi})}{=} & \Lambda(\underbrace{\mathrm{ev} \circ (\mathrm{id} \times \mathrm{cons}) \circ \alpha}_{\mathsf{g}} \circ \alpha \circ (\mathrm{id} \times \langle \mathrm{hd}; \mathrm{tl} \rangle)) & (\text{exp-}\beta) \\
\overset{(\mathrm{vii})}{=} & \Lambda(\mathrm{ev}) & (\text{prod-i}) \\
= & \mathrm{id} & (\text{exp-}\eta)
\end{array}$$

The steps from (i) to (vi) are described in Figure 10.1. In this figure, diagram (1) is given by trivial properties of products. Diagram (2) is given by the naturality of α. Diagram (3) is given by (exp-β) and (exp-$\circ$).

The step (vii) is described in Figure 10.2, which expresses easy properties of products. More precisely, this diagram is given by (prod-i).

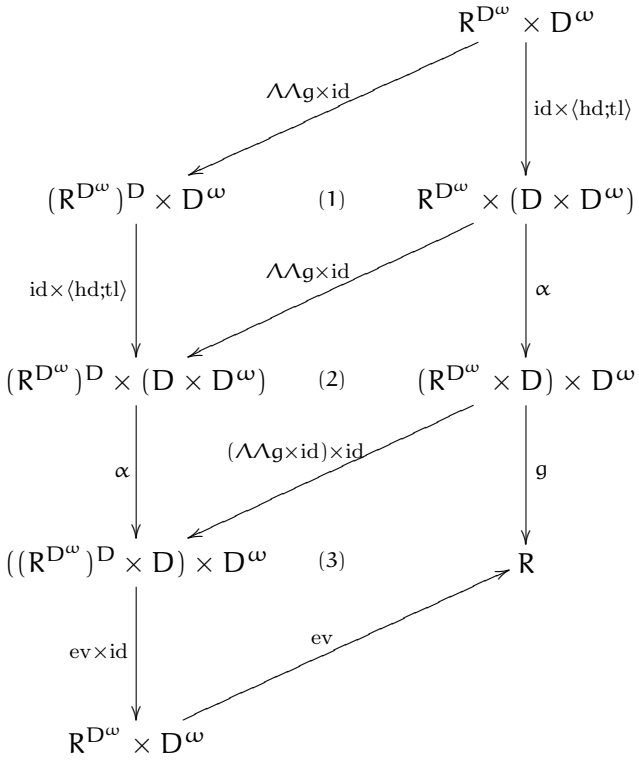


Figure 10.1: Auxiliary diagram for the reflexivity

We finally check that

$$
\begin{aligned}
& \Lambda\Lambda g \circ \Lambda f \\
= \; & \Lambda\Lambda(g \circ ((\Lambda f \times id) \times id)) && \text{(exp-}\circ\text{)} \\
= \; & \Lambda\Lambda(ev \circ (id \times cons) \circ \alpha \circ ((\Lambda f \times id) \times id)) \\
= \; & \Lambda\Lambda(ev \circ (id \times cons) \circ (\Lambda f \times id) \circ \alpha) && \text{(nat-}\alpha\text{)} \\
= \; & \Lambda\Lambda(ev \circ (\Lambda f \times id) \circ (id \times cons) \circ \alpha) \\
= \; & \Lambda\Lambda(\underbrace{ev \circ (ev \times id) \circ \alpha \circ (id \times \langle hd;tl \rangle)}_{f}) \circ (id \times cons) \circ \alpha) && \text{(exp-}\beta\text{)} \\
= \; & \Lambda\Lambda(ev \circ (ev \times id) \circ \alpha \circ \alpha) && \text{(prod-i)} \\
= \; & \Lambda\Lambda(ev \circ (ev \times id)) \;=\; \Lambda(ev) \;=\; id
\end{aligned}
$$

10.2.3 Distributivity axiom

Since we have an isomorphism between D^D and D then the two diagrams (δ) and (ε) are equivalent. So we only check the first one.

We isolate two parts of the proofs in Figure 10.3 and in Figure 10.4. The proof of commutation of the first one is immediate. In Figure 10.4 the first diagram is given

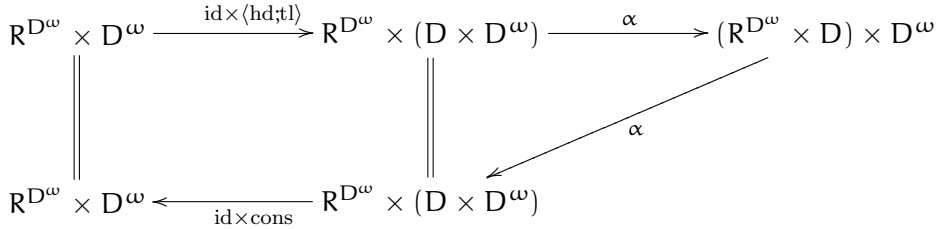


Figure 10.2: Auxiliary diagram for reflexivity

by (str-α). Diagram (2) is given by the naturality of t. Diagram (3) is given by the definition of g.

We often use the equation

$$\mathbf{T}\mathrm{ev} \circ \mathsf{t} \circ \mathbf{T}\Lambda \mathsf{f} \times \mathrm{id} \quad = \quad \mathbf{T}\mathsf{f} \circ \mathsf{t} \tag{10.1}$$

that can be proved using first (nat-t) and then using (exp-β). Writing $\mathsf{h} = \mathrm{flat}_{\mathsf{R}} \circ \mathbf{T}\mathrm{ev} \circ \mathsf{t}$, we check that:

$$\begin{array}{lll}
& \mathrm{app} \circ \mathrm{flat}_{\mathsf{D}} & \\
= & \Lambda(\mathrm{fold} \circ \mathrm{ev}) \circ \Lambda\Lambda \mathsf{g} \circ \mathrm{unfold} \circ \mathrm{fold} \circ \Lambda \mathsf{h} \circ \mathbf{T}\mathrm{unfold} & \\
= & \Lambda(\mathrm{fold} \circ \mathrm{ev} \circ (\Lambda\Lambda \mathsf{g} \times \mathrm{id}) \circ \Lambda \mathsf{h} \times \mathrm{id}) \circ \mathbf{T}\mathrm{unfold} & (\text{exp-}\circ) \\
= & \Lambda(\mathrm{fold} \circ \Lambda \mathsf{g} \circ \Lambda \mathsf{h} \times \mathrm{id}) \circ \mathbf{T}\mathrm{unfold} & (\text{exp-}\beta) \\
= & \Lambda(\mathrm{fold} \circ \Lambda(\mathsf{g} \circ ((\Lambda \mathsf{h} \times \mathrm{id}) \times \mathrm{id}))) \circ \mathbf{T}\mathrm{unfold} & (\text{exp-}\circ) \\
= & \Lambda(\mathrm{fold} \circ \Lambda(\mathsf{h} \circ (\mathrm{id} \times \mathrm{cons}) \circ \alpha)) \circ \mathbf{T}\mathrm{unfold} & (\text{Fig. 10.3}) \\
= & \Lambda(\mathrm{fold} \circ \Lambda(\mathrm{flat}_{\mathsf{R}} \circ \mathbf{T}\mathsf{g} \circ \mathsf{t} \circ (\mathsf{t} \times \mathrm{id}))) \circ \mathbf{T}\mathrm{unfold} & (\text{Fig. 10.4}) \\
= & \Lambda(\mathrm{fold} \circ \Lambda(\mathrm{flat}_{\mathsf{R}} \circ \mathbf{T}\mathrm{ev} \circ \mathsf{t} \circ \mathbf{T}\Lambda \mathsf{g} \times \mathrm{id} \circ (\mathsf{t} \times \mathrm{id}))) \circ \mathbf{T}\mathrm{unfold} & (\text{Eq. 10.1}) \\
= & \Lambda(\mathrm{fold} \circ \Lambda(\mathsf{h} \circ \mathbf{T}\Lambda \mathsf{g} \times \mathrm{id} \circ (\mathsf{t} \times \mathrm{id}))) \circ \mathbf{T}\mathrm{unfold} & \\
= & \Lambda(\mathrm{fold} \circ \Lambda(\mathsf{h}) \circ \mathbf{T}\mathrm{unfold} \circ \mathbf{T}\mathrm{fold} \circ \mathbf{T}\mathrm{ev} \circ \mathsf{t} \circ \mathbf{T}\Lambda\Lambda \mathsf{g} \times \mathrm{id}) \circ \mathbf{T}\mathrm{unfold} & (\text{Eq. 10.1}) \\
= & \Lambda(\mathrm{fold} \circ \Lambda(\mathsf{h}) \circ \mathbf{T}\mathrm{unfold} \circ \mathbf{T}\mathrm{ev} \circ \mathsf{t} \circ (\mathbf{T}\Lambda(\mathrm{fold} \circ \mathrm{ev}) \times \mathrm{id}) \circ \mathbf{T}\Lambda\Lambda \mathsf{g} \times \mathrm{id}) \circ \mathbf{T}\mathrm{unfold} & (\text{Eq. 10.1}) \\
= & \Lambda(\mathrm{flat}_{\mathsf{D}} \circ \mathbf{T}\mathrm{ev} \circ \mathsf{t}) \circ \mathbf{T}\Lambda(\mathrm{fold} \circ \mathrm{ev}) \circ \mathbf{T}\Lambda\Lambda \mathsf{g} \circ \mathbf{T}\mathrm{unfold} \;=\; \Lambda(\mathrm{flat}_{\mathsf{D}} \circ \mathbf{T}\mathrm{ev} \circ \mathsf{t}) \circ \mathbf{T}\mathrm{app} & (\text{exp-}\circ)
\end{array}$$

10.2.4 Typical use

In Scott domains, the equation $\mathsf{D} = \mathsf{R}^{(\mathsf{D}^\omega)}$ always has solutions, since the endofunctor $\mathsf{X} \mapsto \mathsf{R}^{(\mathsf{X}^\omega)}$ is ω-cocontinuous in the category of injection-projection pairs. Notice that the least fixpoint D of this functor is not trivial as soon as the algebra R is not trivial. Intuitively, the smallest solution D can be understood as the smallest η-model of the parallel λ-calculus which contains R (in the sense of injection-projection pairs).

Conclusion and Future Works

We have defined a sound and complete categorical semantics for the parallel λ-calculus, based on a notion of aggregation monad which is modular w.r.t. associativity, commu-

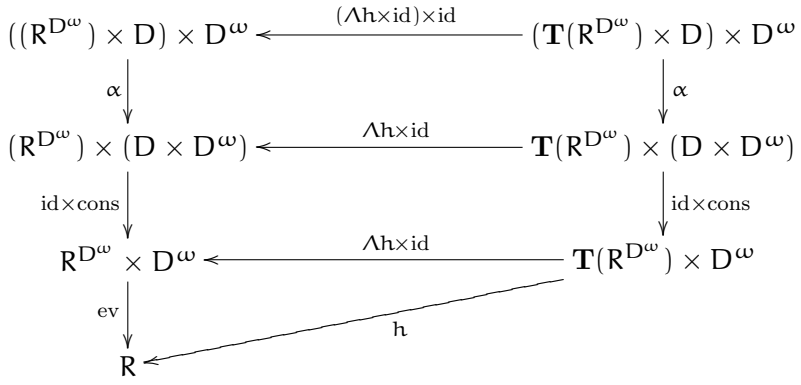


Figure 10.3: Auxiliary diagram for the distributivity

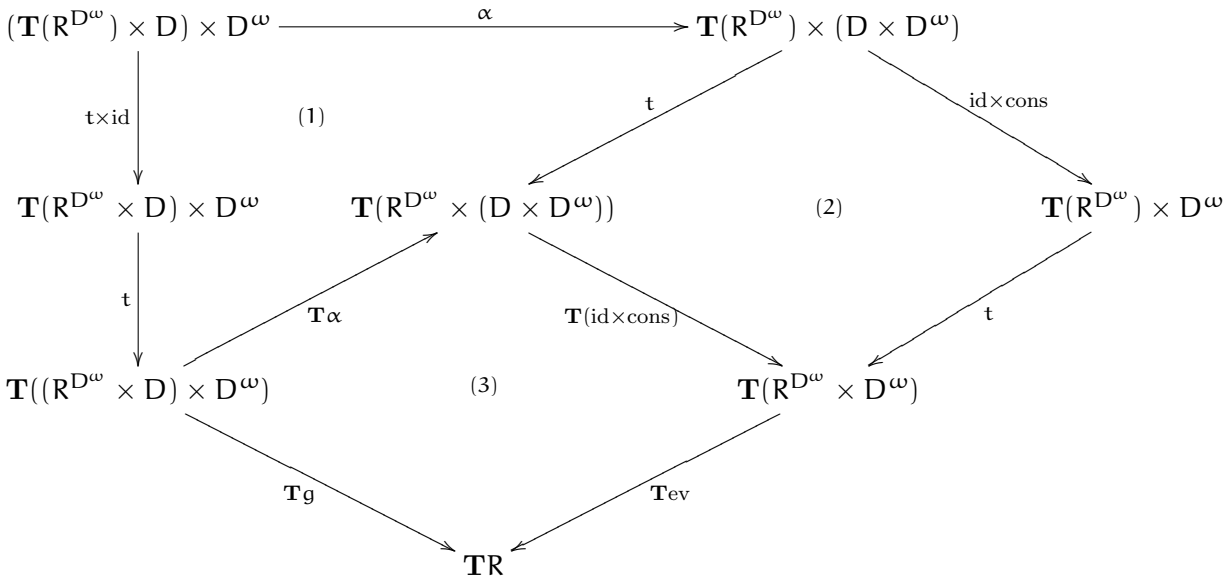


Figure 10.4: Auxiliary diagram for the distributivity

tativity and idempotence.

To prove completeness, we have introduced a category of partial equivalence relations adapted to parallelism, in which any extension of the basic equational theory of the calculus is induced by some model.

We have also presented abstract methods to construct models of the parallel λ-calculus in categories where particular equations have solutions, such as the category of Scott domains and its variants, and check that G. Boudol's original semantics is a particular case of ours.

This work on the denotational semantics of the parallel λ-calculus is initially motivated by the semantical study of the ρ-calculus [CK01], a formalism which combines ML-style pattern-matching with parallel aggregation. The next step is thus to find a satisfying way to integrate constructors and pattern-matching in our setting. However, combining pattern-matching with parallel aggregation naturally raises new problems related to additivity. To understand this point, let us consider the following example.

Assume that the parallel λ-calculus is enriched with two constant constructors a, b and a unary constructor $c(\_)$, plus a syntactic construct $[c(x) \ll N]M$ that matches the term N against the pattern $c(x)$, and binds all free occurrences of x in M to the argument of the destructed value. (We do not give any special meaning to this construction when N is not a constructed value.)

Now consider the term $M = [c(x) \ll c(a \mathbin{/\!/} b)]\, F\, x\, x$, where F is an arbitrary function. The naive way to reduce M is to substitute the term $(a \mathbin{/\!/} b)$ to x in the r.h.s. $F\, x\, x$, hence:

$$[c(x) \ll c(a \mathbin{/\!/} b)]\, F\, x\, x \quad \to \quad F\,(a \mathbin{/\!/} b)\,(a \mathbin{/\!/} b)\,.$$

(This strategy is the one which is actually implemented by the standard encodings of constructed values and pattern-matching in the λ-calculus.)

However, it is also legitimate to consider that a and b represent two possible choices for the argument of the constructed value $c(a \mathbin{/\!/} b)$. Following this intuition, a completely different reduction strategy is to distribute a and b w.r.t. the matching construct, which yields:

$$[c(x) \ll c(a \mathbin{/\!/} b)]F\, x\, x \quad \to \quad F\, a\, a \;\mathbin{/\!/}\; F\, b\, b\,.$$

Of course, both design choices are clearly incompatible, which is easy to see by taking $F = \lambda x y\,.\,x y$. This second strategy—which seems to be impossible to simulate in the core parallel λ-calculus—is much more interesting, since it suggests that both operations of construction and destruction are linear:

$$c(N_1 \mathbin{/\!/} N_2) \;=\; c(N_1) \mathbin{/\!/} c(N_2)$$

$$[c(x) \ll (N_1 \mathbin{/\!/} N_2)]M \;=\; [c(x) \ll N_1]M \mathbin{/\!/} [c(x) \ll N_2]M$$

This example naturally raises the exciting challenge of constructing a model of the ρ-calculus that implements the second reduction strategy, while being rich enough to reflect all the expressivity of ML-style pattern-matching, such as the existence of infinitely many constructors of all arities (with pairwise disjoint images), the existence of variadic constructors, etc.

Quatrième partie

Épilogue

Conclusion et travaux futurs

Dans la première partie de cette thèse, nous avons introduit une nouvelle approche pour résoudre des équations de filtrage dans le λ-calcul. La méthode utilisée n'est pas dépendante d'un système de types particulier et possède de plus l'avantage, contrairement aux autres approches [Hue76, HL78, CQS96], de ne pas introduire de nouvelles variables de filtrage durant le processus de résolution. Bien que l'ensemble des équations résolues soit particulièrement riche (équations du second-ordre et équations sur des motifs à la Miller entre autres), l'utilisation du λ-calcul comme langage support est restrictif.

Tout d'abord, dans le cadre de la transformation de programmes et dans le cadre de la preuve (calcul des constructions avec filtrage par exemple), il est fondamental de disposer d'algorithmes pour le filtrage modulo βη en présence de symboles AC. Une telle extension de l'algorithme du second-ordre de G. Huet et de B. Lang a été proposée [Cur93] mais il serait particulièrement intéressant d'étudier le filtrage modulo super-développements en présence de symboles AC et de le comparer avec l'approche précédente. Un tel algorithme utilisant les super-développements semble a priori plus simple à mettre en pratique.

D'autre part, lorsque l'on souhaite transformer des programmes fonctionnels non stricts[1] (programmes comme ρ-termes), ou lorsque l'on considère [Wac05] des termes de preuve riches pour la déduction modulo (preuves comme ρ-termes), la β-équation du λ-calcul doit être remplacée par la théorie équationnelle du ρ-calcul. Ainsi défini, un algorithme pour le filtrage d'ordre supérieur modulo super-développements permettrait de faire l'économiede l'encodage, souvent coûteux [MS01], du filtrage.

Les outils pour le filtrage d'ordre supérieur que nous avons introduits dans cette thèse ont des applications nombreuses et plus particulièrement dans le cadre de la preuve de propriétés de programme. En se basant sur un ensemble de règles de transformations préservant la sémantique des programmes et en utilisant, pour appliquer ces règles, un algorithme de filtrage certifié, on peut ainsi prouver des propriétés sur des programmes sémantiquement équivalents mais plus simples. Considérons par exemple les deux algorithmes classiques pour renverser les listes : l'algorithme naïf mais défini simplement par filtrage sur la structure de données `liste` et l'algorithme efficace utilisant une pile auxiliaire. L'approche que nous avons décrite consiste à prouver d'abord les propriétés souhaitées sur l'algorithme naïf (ce qui est toujours plus simple) puis, par transformation de programmes prouvée correcte et complète, on peut « hériter » de ces propriétés sur la version efficace.

Dans la deuxième partie de cette thèse, nous avons étudié le filtrage et les substitutions, mécanismes de base des calculs d'ordre supérieur avec motifs comme le ρ-calcul, le

[1] c'est-à-dire utilisant primitivement un mécanisme de filtrage

calcul pur de motifs [JK06] et le λ-calcul avec motifs [Oos90]. Nous avons tout d'abord introduit un calcul avec filtrage et substitutions explicites, donnant un cadre permettant de souligner les interactions possibles et, offrant un support pour l'implémentation de tels calculs.

Ce calcul ne traite pas explicitement l'α-conversion. Nous avons esquissé une version généralisée des substitutions explicites avec indices de de Bruijn adaptée aux calculs ayant des constructions pouvant lier plusieurs variables simultanément. D'autres approches peuvent être adaptées pour ces langages : le Λ-calcul avec opérateur explicite de portée [HO03], les directeurs qui sont utilisés pour la réécriture d'ordre supérieur dans [Sin05] ou encore les réseaux d'interactions qui ont été étudiés dans le cadre du ρ-calcul dans [FMS05, CFF+06].

Nous avons mis en avant l'importance de l'étude modulaire des calculs avec motifs et nous avons isolé les propriétés clés des algorithmes de filtrage permettant d'obtenir des calculs cohérents (confluents). Ces calculs avec motifs peuvent manipuler des motifs dynamiques, c'est-à-dire qui peuvent être instanciés et réduits. Ils conduisent à des langages fonctionnels avec de nouvelles formes de polymorphisme [Jay04].

Notre étude s'est limitée au cas du filtrage unitaire (au plus une solution). Pourtant en pratique, l'utilisation du filtrage modulo une théorie équationnelle permet d'écrire des programmes hautement déclaratifs, comme c'est la cas dans le langage Tom [BBK+06] implémentant le filtrage sur les listes (filtrage associatif avec élément neutre). Néanmoins, l'extension de l'étude réalisée ici au cas du filtrage non-unitaire nécessite une compréhension nouvelle du mécanisme de filtrage et plus particulièrement dans son interaction avec les collections de résultats (indispensables dans le cas du filtrage non-unitaire).

Dans la troisième partie de cette thèse, nous avons mis en évidence que l'ajout de collections de termes distributifs par rapport à l'application est non triviale même dans le cadre du λ-calcul. Nous avons montré que les collections de termes, la structure du ρ-calcul ou encore l'opérateur de parallélisme du λ-calcul parallèle [Bou94] conduisent à la notion plus générale d'agrégation.

Nous avons donc introduit une sémantique catégorique correcte et complète du λ-calcul avec opérateur de parallélisme. Cette sémantique repose fortement sur la notion d'agrégation et de termes linéaires. Les exemples de modèles que nous avons donnés sont essentiellement des domaines de Scott mais les modèles de la logique linéaire qui valident la règle Mix et les modèles du λ-calcul différentiel [ER03, Ehr05, Ehr02] sont clairement de bons candidats.

L'étape suivante est donc d'intégrer les constructeurs et le filtrage de manière satisfaisante dans notre cadre. Nous retrouvons naturellement les problèmes liés à la combinaison du filtrage avec les collections de résultats et que nous avons mentionnés dans la deuxième partie de cette thèse. Cette combinaison se traduit dans notre cadre sémantique par de nouveaux problèmes liés à la linéarité des constructeurs et du filtrage.

Dans cette thèse, nous avons, en résumé, cherché à expliciter les mécanismes fondamentaux des calculs avec motifs en utilisant une approche opérationnelle et dénotationnelle. L'étude du calcul de réécriture qui est à l'origine des travaux présentés ici

avait été réalisée pour donner une sémantique aux langages basés sur la réécriture, les stratégies [BKK96, Vis01b, KK04, Kir05, MOMV06] jouant un rôle particulièrement important dans tous ces langages (Tom, Elan, Maude, Stratego etc).

Intégrées en effet aux langages généralistes comme Java, les notions de base de la réécriture comme le filtrage et les stratégies, permettent d'introduire dans ces langages des îlots formels [BKM06]. Ceci est particulièrement important pour pouvoir développer des parties de code fortement sensibles, c'est-à-dire dont la sécurité (certification et sûreté) est cruciale. C'est par exemple le cas pour l'implémentation des politiques de sécurité [DKKS07]. La réécriture et les stratégies ainsi que la notion d'îlots formels donnent des outils particulièrement novateurs et efficaces.

Dans ce contexte particulièrement riche, il nous apparaît maintenant indispensable de revenir aux motivations initiales du ρ-calcul pour approfondir la sémantique des stratégies. Plusieurs pistes sont possibles. L'étude de la sémantique dénotationnelle des calculs avec motifs, initialisée dans cette thèse, en est une. Une autre approche est l'utilisation de la réécriture de dimension supérieure [Bur93, Laf97, Gui04, Gui06b, Gui06a] offrant un langage à la fois algébrique, graphique et topologique commun aux structures, aux calculs et aux démonstrations. Les règles de réécriture étant, dans ce formalisme, des objets de dimension 3, on peut envisager de considérer [Gui04] les stratégies comme des objets de dimension 4. Bien que très prospective, cette approche paraît particulièrement prometteuse.

Bibliographie

[Abr91] Samson Abramsky – « Domain theory in logical form », *Annals of Pure and Applied Logic* **51** (1991), no. 1–2, p. 1–77. 173

[AC96] Martín Abadi et Luca Cardelli – *A theory of objects*, Springer-Verlag, New York, 1996. 78

[AC98] Roberto Amadio et Pierre-Louis Curien – *Domains and lambda-calculi*, Cambridge Tracts in Theoretical Computer Science, vol. 46, Cambridge University Press, Cambridge, 1998. 147, 148, 151, 171

[ACCL91] Martín Abadi, Luca Cardelli, Pierre-Louis Curien et Jean-Jacques Lévy – « Explicit substitutions », *Journal of Functional Programming* **1** (1991), no. 4, p. 375–416. 7, 87, 120

[Acz78] Peter Aczel – « A general Church-Rosser theorem », Tech. report, University of Manchester, July 1978. 7, 29

[ADGR07] Jeremy Avigad, Kevin Donnelly, David Gray et Paul Raff – « A formally verified proof of the prime number theorem. », *ACM Transactions on Computational Logic, To appear.* (2007). 11

[AL91] Andrea Asperti et Guiseppe Longo – *Categories, types and structures*, MIT Press, 1991. 147

[All78] John Allen – *Anatomy of LISP*, McGraw-Hill, Inc., New York, USA, 1978. 5

[AMR06] Ariel Arbiser, Alexandre Miquel et Alejandro Ríos – « A lambda-calculus with constructors », *Term Rewriting and Applications – RTA'06* (Seattle, USA), Lecture Notes in Computer Science, vol. 4098, Springer, August 2006, p. 181–196. 8, 12, 125, 142, 143

[AO93] Samson Abramsky et Luke Ong – « Full abstraction in the lazy lambda calculus », *Information and Computation* **105** (1993), no. 2, p. 159–267. 173

[Bal06] Vincent Balat – « Ocsigen : Typing web interaction with objective caml », *ACM Sigplan Workshop on ML*, 2006. 12

[Bar84] Henk Barendregt – *The Lambda-Calculus, its syntax and semantics*, Studies in Logic and the Foundation of Mathematics, North Holland, 1984, Second edition. 17, 20, 22, 23, 69, 147

[Bar91] Franco Barbanera – « Adding algebraic rewriting to the calculus of constructions : Strong normalization preserved », *Conditional and Typed Rewriting Systems -CTRS'90* (S. Kaplan et M. Okada, éds.), Lecture Notes in Computer Science, vol. 516, Springer, 1991, p. 262–271. 11, 12

[Bar92] Henk Barendregt – « Handbook of logic in computer science », ch. Lambda Calculi with Types, p. 117–309, Clarendon Press, 1992. 17

[Bax77] Lewis Denver Baxter – « The complexity of unification. », Thèse de doctorat, University of Waterloo, 1977. 62

[BBCK04] Clara Bertolissi, Paolo Baldan, Horatiu Cirstea et Claude Kirchner – « A rewriting calculus for cyclic higher-order term graphs », *Proceedings of the 2nd International Workshop on Term Graph Rewriting* (Roma (Italy)) (M. Fernandez, éd.), Electronic Notes in Theoretical Computer Science, September 2004, to appear. 67

[BBCK06] Paolo Baldan, Clara Bertolissi, Horatiu Cirstea et Claude Kirchner – « A rewriting calculus for cyclic higher-order term graphs », *Mathematical Structures in Computer Science* (2006). 5

[BBK+06] Émilie Balland, Paul Brauner, Radu Kopetz, Pierre-Étienne Moreau et Antoine Reilles – « The Tom manual », 2006, `http://tom.loria.fr/soft/release-2.4/manual-2.4/index.html`. 7, 88, 118, 144, 206

[BCK06] Clara Bertolissi, Horatiu Cirstea et Claude Kirchner – « Expressing combinatory reduction systems derivations in the rewriting calculus », *Higher-Order and Symbolic Computation* **19** (2006), no. 4, p. 345–376. 31, 64, 67

[BCKL03] Gilles Barthe, Horatiu Cirstea, Claude Kirchner et Luigi Liquori – « Pure patterns type systems », *Principles of Programming Languages - POPL'03*, vol. 38, ACM, January 2003. 5, 8, 67, 125, 137, 140

[BCL99] Gérard Boudol, Pierre-Louis Curien et Carolina Lavatelli – « A semantics for lambda calculi with resources », *Mathematical Structures in Computer Science* **9** (1999), no. 4, p. 437–482. 9, 155

[BDL06] Sandrine Blazy, Zaynah Dargaye et Xavier Leroy – « Formal verification of a C compiler front-end », *Formal Methods, 14th International Symposium on Formal Methods*, Lecture Notes in Computer Science, vol. 4085, 2006, p. 460–475. 11

[Ber05] Clara Bertolissi – « The graph rewriting calculus : properties and expressive capabilities », Thèse de doctorat, Institut National Polytechnique de Lorraine - INPL, Oct 2005. 5, 67

[BF82] Klaus Berkling et Elfriede Fehr – « A consistent extension of the lambda-calculus as a base for functional programming languages », *Information and Control* **55** (1982), no. 1-3, p. 89–101. 122

[BHK07] Paul Brauner, Clément Houtmann et Claude Kirchner – « Principles of superdeduction », *Logics in Computer Science - LICS'07*, 2007, To appear. 117

[BJKO00] Mark van den Brand, Hayco Jong, Paul Klint et Pieter Olivier – « Efficient Annotated Terms », *Software, Practice & Experience* **30** (2000), p. 259–291. 118

[BK07] Clara Bertolissi et Claude Kirchner – « The rewriting calculus as a combinatory reduction system », *Foundations of Software Science and Computation Structures - FoSSaCS'07* (Braga, Portugal), Lecture Notes in Computer Science, Springer, March 2007. 8, 67, 125

[BKK96] Peter Borovansky, Claude Kirchner et Hélène Kirchner – « Controlling Rewriting by Rewriting », *1st International Workshop on Rewriting Logic – WRLA'96* (Asilomar (CA, USA)) (J. Meseguer, éd.), vol. 4, Electronic Notes in Theoretical Computer Science, September 1996. 11, 67, 99, 207

[BKK98a] Peter Borovanský, Claude Kirchner et Hélène Kirchner – « A functional view of rewriting and strategies for a semantics of ELAN », *The Third Fuji International Symposium on Functional and Logic Programming* (Kyoto) (M. Sato et Y. Toyama, éds.), World Scientific, April 1998, Also report LORIA 98-R-165, p. 143–167. 67, 122

[BKK+98b] Peter Borovanský, Claude Kirchner, Hélène Kirchner, Pierre-Etienne Moreau et Christophe Ringeissen – « An overview of ELAN », *Workshop on Rewriting Logic and its Applications - WRLA'98*, vol. 15, Electronic Notes in Theoretical Computer Science, September 1998. 118

[BKM06] Emilie Balland, Claude Kirchner et Pierre-Etienne Moreau – « Formal islands », *Proceedings of Algebraic Methodology and Software Technology - AMAST'06* (M. Johnson et V. Vene, éds.), LNCS, sv, july 2006. 207

[Bla05] Frédéric Blanqui – « Definitions by rewriting in the calculus of constructions », *Mathematical Structures in Computer Science* **15** (2005), no. 1, p. 37–92. 11

[BMV05] Mark van den Brand, Pierre-Etienne Moreau et Jurgen Vinju – « A generator of efficient strongly typed abstract syntax trees in java », *IEE Proceedings - Software Engineering* **152** (2005), no. 2, p. 70–78. 118

[Bou94] Gérard Boudol – « Lambda-calculi for (strict) parallel functions », *Information and Computation* **108** (1994), no. 1, p. 51–127. 9, 155, 173, 206

[Bru72] Nicolas G. de Bruijn – « A lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem », *Indagationes Mathematicae* **34** (1972), p. 381–392. 122

[Bru78] — , « Lambda calculus with namefree formulas involving symbols that represent reference transforming mappings », *Indagationes Mathematicae* **40** (1978), p. 348–356. 119

[Bür90] Hans-Jürgen J. Bürckert – « Matching — A special case of unification ? », Unification, Academic Press, 1990, p. 125–138. 46

[Bur93] Albert Burroni – « Higher-dimensional word problems with applications to equational logic », *Theoretical Computer Science* **115** (1993), no. 1, p. 43–62. 207

[CDE+02] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer et Jose F. Quesada – « Maude : specification and programming in rewriting logic », *Theoretical Computer Science* **285** (2002), no. 2, p. 187–243. 144

[CF07] Horatiu Cirstea et Germain Faure – « Confluence of pattern-based lambda-calculi », *Rewriting Techniques and Applications - RTA'07*, vol. 4533, 2007, p. 78–92. 8

[CFF+06] Horatiu Cirstea, Germain Faure, Maribel Fernandez, Ian Mackie et Francois-Regis Sinot – « From functional programs to interaction nets via the rewriting calculus », *Workshop on Reduction Strategies in Rewriting and Programming - WRS'06*, Electronic Notes in Theoretical Computer Science, 2006. 206

[CFK04] Horatiu Cirstea, Germain Faure et Claude Kirchner – « A rho-calculus of explicit constraint application », *Proceedings of the Workshop on Rewriting Logic and Applications - WRLA'04* (Barcelona (Spain)), Electronic Notes in Theoretical Computer Science, Mars 2004. 88, 95, 99

[CFK07] — , « A rho-calculus of explicit constraint application », *Higher-Order and Symbolic Computation* **20** (2007). 88

[CHL96] Pierre-Louis Curien, Thérèse Hardin et Jean-Jacques Lévy – « Confluence properties of weak and strong calculi of explicit substitutions », *Journal of the ACM (JACM)* **43** (1996), no. 2, p. 362–397. 100, 101, 117, 120, 134

[Chu36] Alonzo Church – « A note on the entscheidungsproblem. », *Journal Symbolic Logics* **1** (1936), no. 1, p. 40–41. 5

[Chu41] — , *The Calculi of Lambda Conversion*, Princeton University Press, Princeton, 1941. 5

[CHW06] Horatiu Cirstea, Clement Houtmann et Benjamin Wack – « Distributive rewriting calculus », *Sixth International Workshop on Reduction Strategies in Rewriting and Programming – WRLA'06*, Electronic Notes in Theoretical Computer Science, 2006. 5, 76, 82

[Cir00] Horatiu Cirstea – « Calcul de réécriture : fondements et applications », Thèse de Doctorat, Université Henri Poincaré - Nancy I, 2000. 5, 12, 67, 76, 120, 122

[CK98a] Horatiu Cirstea et Claude Kirchner – « The Rewriting Calculus as a Semantics of ELAN », *4th Asian Computing Science Conference* (Manila, The Philippines) (J. Hsiang et A. Ohori, éds.), Lecture Notes in Computer Science, vol. 1538, Springer-Verlag, December 1998. 12, 67

[CK98b] — , « ρ-calculus. Its Syntax and Basic Properties », *Workshop Construction of Computational Logics - CCL'98* (Jerusalem, Israel), September 1998. 5

[CK01] — , « The rewriting calculus — Part I *and* II », *Logic Journal of the Interest Group in Pure and Applied Logics* **9** (2001), p. 427–498. 5, 8, 12, 67, 76, 78, 82, 125, 126, 202

[CKL01] Horatiu Cirstea, Claude Kirchner et Luigi Liquori – « Matching Power », *Rewriting Techniques and Applications - RTA'01* (Utrecht (The Netherlands)), Lecture Notes in Computer Science, Springer-Verlag, May 2001. 5, 67, 78

[CKL02] — , « Rewriting Calculus with(out) Types », *Workshop on Rewriting Logic and its Applications - WRLA'2002, Pisa, Italy*, ENTCS, September 2002. 7, 67, 87

[CKLW03] Horatiu Cirstea, Claude Kirchner, Luigi Liquori et Benjamin Wack – « Rewrite strategies in the rewriting calculus », *Third International Workshop on Reduction Strategies in Rewriting and Programming* (Valencia, Spain) (B. Gramlich et S. Lucas, éds.), Electronic Notes in Theoretical Computer Science, June 2003. 74, 79

[CLW03] Horatiu Cirstea, Luigi Liquori et Benjamin Wack – « Rewriting calculus with fixpoints : Untyped and first-order systems », *Types for Proofs and Programs - TYPES'03* (Torino, Italy), Lecture Notes in Computer Science, vol. 3085, Springer, May 2003, p. 147–161. 8, 79, 80, 82, 125, 128, 135

[CMV+06] Francois Clément, Vincent Martin, Arnaud Vodicka, Roberto Di Cosmo et Pierre Weis – « Domain decomposition and skeleton programming with ocamlp3l. », *Parallel Computing* **32** (2006), no. 7-8, p. 539–550. 12

[Coq07] The Coq Development Team LogiCal Project – « Coq proof assistant reference manual », 2007. 118

[CP88] Thierry Coquand et Christine Paulin – « Inductively defined types. », *Conference on Computer Logic – CCL'88*, Lecture Notes in Computer Science, vol. 417, Springer, 1988, p. 50–66. 12

[CQS96] Régis Curien, Zhenyu Qian et Hui Shi – « Efficient second-order matching », *Rewriting Techniques and Applications - RTA'96*, Lecture Notes in Computer Science, vol. 1103, 1996. 59, 205

[CR36] Alonzo Church et John Barkley Rosser – « Some properties of conversion », *Trans. Amer. Math. Soc.* **40** (1936). 17

[Cur93] Régis Curien – « Second order E-matching as a tool for automated theorem proving », *Progress in Artificial Intelligence - EPIA'93)* (Porto, Portugal), Lecture Notes in Artificial Intelligence, vol. 727, Springer, 1993. 205

[Des98] Joëlle Despeyroux – « Natural semantics : specifications and proofs », Tech. report, INRIA, Février 1998. 23

[DG01] René David et Bruno Guillaume – « A λ-calculus with explicit weakening and explicit substitution », *Mathematical Structures for Computer Science* **11** (2001), p. 169–206. 87, 117

[DHK00] Gilles Dowek, Thérèse Hardin et Claude Kirchner – « Higher order unification via explicit substitutions », *Information and Computation* **157** (2000), no. 1/2, p. 183–235. 46, 87, 117

[DKKS07] Daniel J. Dougherty, Claude Kirchner, Hélène Kirchner et Anderson Santana de Oliveira – « Modular access control via strategic rewriting », *Proceedings of 12th European Symposium On Research In Computer Security (ESORICS'07)* (Dresden), Sep 2007, to appear. 207

[Dow94] Gilles Dowek – « Third order matching is decidable », *Annals of Pure and Applied Logic* **69** (1994), p. 135–155. 35

[Dow01] — , « Higher-order unification and matching », Handbook of Automated Reasoning, Elsevier, 2001. 40

[Ehr02] Thomas Ehrhard – « On köthe sequence spaces and linear logic. », *Mathematical Structures in Computer Science* **12** (2002), no. 5, p. 579–623. 206

[Ehr05] — , « Finiteness spaces. », *Mathematical Structures in Computer Science* **15** (2005), no. 4, p. 615–646. 206

[Eke95] Steven Eker – « Associative-commutative matching via bipartite graph matching », *Computer Journal* **38** (1995), no. 5, p. 381–399. 7, 87

[ER03] Thomas Ehrhard et Laurent Regnier – « The differential lambda-calculus. », *Theoretical Computer Science* **309** (2003), no. 1-3, p. 1–41. 206

[Fau06] Germain Faure – « Matching modulo superdevelopments application to second-order matching », *13th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning - LPAR'06*, Lecture Notes in Computer Science, vol. 4246, Springer, 2006, p. 60–74. 32

[Fau07] — , « Matching modulo superdevelopments application to second-order matching », *Journal of Automated Reasonning* **Submited** (2007). 32

[Fer93] Maribel Fernández – « Modèles de calculs multiparadigmes fondés sur la réécriture », Thèse de doctorat, Université Paris-Sud, Orsay, France, December 1993. 11

[Fil03] Jean-Christophe Filliâtre – « Why : a multi-language multi-prover verification tool », Tech. Report 1366, Université Paris Sud, 2003. 11

[FK02] Germain Faure et Claude Kirchner – « Exceptions in the rewriting calculus », *Proceedings of the Rewriting Techniques and Applications - RTA'02* (Copenhagen) (S. Tison, éd.), LNCS, Springer-Verlag, July 2002, p. pp. 67, 122

[Fly07] Project Flyspeck – « `http://www.math.pitt.edu/~thales/flyspeck/` », Tech. report, 2007. 11

[FM07] Germain Faure et Alexandre Miquel – « A categorical semantics of the parallel lambda-calculus », 2007, Submited. 9, 147

[FMS05] Maribel Fernández, Ian Mackie et François-Régis Sinot – « Interaction nets vs. the rho-calculus : Introducing bigraphical nets », *12th International Workshop on Expressiveness in Concurrency - EXPRESS'05*, Electronic Notes in Theoretical Computer Science, Elsevier, 2005. 206

[Fri06] Alain Frisch – « Ocaml + xduce. », *11th ACM SIGPLAN International Conference on Functional Programming - ICFP'06*, 2006, p. 192–200. 12

[Gan80] Robin Gandy – « Proof of strong normalisation », Essays on Combinatory Logic, Lambda Calculus, and Formalism (J. P. Seldin et J. R. Hindley, éds.), Academic Press inc., New York (NY, USA), 1980. 20

[Gol81] Donald Goldfarb – « The undecidability of the second order unification », *Journal of Theoretical Computer Science* **13** (1981), p. 225–230. 35

[Gon05] Georges Gonthier – « A computer-checked proof of the four colour theorem », Tech. report, Microsoft Research Cambridge, 2005. 11

[Gui04] Yves Guiraud – « Présentations d'opérades et systèmes de réécriture », Thèse, Université Montpellier 2, June 2004. 207

[Gui06a] — , « Termination orders for 3-polygraphs », *Comptes-Rendus de l'Académie des Sciences - Série I* **342** (2006), no. 4, p. 219–222. 207

[Gui06b] — , « Termination orders for three-dimensional rewriting », *Journal of Pure and Applied Algebra* **207** (2006), no. 2, p. 341–371. 207

[GWZ02] Herman Geuvers, Freek Wiedijk et Jan Zwanenburg – « A constructive proof of the fundamental theorem of algebra without using the rationals. », *Types for Proofs and Programs - TYPES'00*, Lectures Notes in Computer Science, vol. 2277, 2002. 11

[Hag90] Masami Hagiya – « Programming by example and proving by example using higher-order unification », *10th International Conference on Automated Deduction - CADE'90* (M. E. Stickel, éd.), Springer-Verlag, July 1990, p. 588–602. 31

[Hin78] J. Roger Hindley – « Standard and normal reductions », *Transactions of the American Mathematical Society* **241** (1978), p. 253–271. 17

[HJM06] Thérèse Hardin, Mathieu Jaume et Charles Morisset – « Access control and rewrite systems », *Proceedings of the 1st International Workshop on Security and Rewriting Techniques (SecReT'06)* (Venice, Italy), July 2006. 11

[HL78] Gérard Huet et Bernard Lang – « Proving and applying program transformations expressed with second-order patterns », *Acta Informatica* **11** (1978). 12, 31, 35, 46, 59, 205

[HM88] John Hannan et Dale Miller – « Uses of higher-order unification for implementing program transformers », *5th International Conference and Symposium on Logic Programming* (R. A. Kowalski et K. A. Bowen, éds.), The MIT Press, 1988, p. 942–959. 31

[HO03] Dimitri Hendriks et Vincent van Oostrom – « The adbmal calculus », *International Conference on Automated Deduction - CADE'03*, Lecture Notes in Artificial Intelligence, vol. 2741, 2003, p. 136–150. 119, 206

[HS86] J. Roger Hindley et Jonathan P. Seldin – *An introduction to combinators and the λ-calculus*, Cambrigde University Press, 1986. 20

[Hue73] Gérard Huet – « A mechanization of type theory », *Proceedings of the 3rd International Joint Conference on Artificial Intelligence* (Standford, CA) (N. J. Nilsson, éd.), 1973, p. 139–146. 35

[Hue75] — , « A unification algorithm for typed lambda-calculus. », *Theoretical Computer Science* **1** (1975), no. 1, p. 27–57. 35

[Hue76] — , « Résolution d'équations dans les langages d'ordre 1, 2, . . . , ω », Thèse de doctorat d'état, Université Paris, 7, 1976. 12, 31, 35, 59, 205

[Hue80] — , « Confluent reductions : Abstract properties and applications to term rewriting systems », *Journal of the ACM* **27** (1980), no. 4, p. 797–821. 99

[Jay04] Barry Jay – « The pattern calculus. », *ACM Trans. Program. Lang. Syst.* **26** (2004), no. 6, p. 911–937. 12, 206

[JK86] Jean-Pierre Jouannaud et Hélène Kirchner – « Completion of a set of rules modulo a set of equations », *SIAM Journal of Computing* **15** (1986), no. 4, p. 1155–1194. 99

[JK91] Jean-Pierre Jouannaud et Claude Kirchner – « Solving equations in abstract algebras : A rule-based survey of unification. », *Computational Logic - Essays in Honor of Alan Robinson*, 1991, p. 257–321. 43

[JK06] Barry Jay et Delia Kesner – « Pure pattern calculus », *European Symposium on Programming – ESOP'06* (Vienna, Austria), Lecture Notes in Computer Science, vol. 3924, Springer, March 2006, p. 100–114. 5, 8, 12, 125, 126, 128, 140, 141, 142, 206

[JO95] Jean-Pierre Jouannaud et Mitsuhiro Okada – « Abstract data type systems », Tech. report, LRI and Keio University, April 1995. 11

[Kah03] Wolfram Kahl – « Basic pattern matching calculi : Syntax, reduction, confluence, and normalisation », SQRL Report 16, Software Quality Research Laboratory, McMaster Univ., October 2003, available from `http://www.cas.mcmaster.ca/sqrl/sqrl_reports.html`. 5, 12

[Kes00] Delia Kesner – « Confluence of extensional and non-extensional lambda-calculi with explicit substitutions. », *Theoretical Computer Science* **238** (2000), no. 1-2, p. 183–220. 125

[Kes07] — , « The theory of explicit substitutions revisited », (2007). 87, 120, 123

[Kir84] Claude Kirchner – « A new equational unification method : A generalization of martelli-montanari's algorithm. », 1984, p. 224–247. 37, 43

[Kir05] — , « Strategic rewriting », *Electr. Notes Theor. Comput. Sci. Proceedings of the 4th International Workshop on Reduction Strategies in Rewriting and Programming - WRS'2004, Aachen, Germany* **124** (2005), no. 2, p. 3–9. 11, 207

[KK99] Claude Kirchner et Hélène Kirchner – « Rewriting, solving, proving », A preliminary version of a book available at `www.loria.fr/~ckirchne/rsp.ps.gz`, 1999. 99

[KK04] — , « Rule-based programming and proving : The ELAN experience outcomes », *Advances in Computer Science - ASIAN 2004, Higher-Level Decision Making, 9th Asian Computing Science Conference, Dedicated to Jean-Louis Lassez on the Occasion of His 5th Cycle Birthday* (Chiang Mai, Thailand) (M. J. Maher, éd.), Lecture Notes in Computer Science, vol. 3321, Springer, December 8-10 2004, p. 363–379. 11, 118, 207

[KKM07] Claude Kirchner, Radu Kopetz et Pierre-Etienne Moreau – « Anti-pattern matching », *European Symposium on Programming - ESOP'07* (Braga, Portugal), Lecture Notes in Computer Science, Springer, March 2007. 8, 126

[KL05] Delia Kesner et Stephane Lengrand – « Extending the explicit substitution paradigm », *16th International Conference on Term Rewriting and Applications - RTA'05*, Lecture Notes in Computer Science, vol. 3467, Springer, 2005, p. 407–422. 123

[Klo80] Jan Willem Klop – « Combinatory reduction systems », Ph.D. thesis, Mathematisch Centrum, Amsterdam, 1980. 33, 67, 76, 135

[KOR93] Jan Willem Klop, Vincent van Oostrom et Femke van Raamsdonk – « Combinatory reduction systems : introduction and survey », *Theoretical Computer Science* **121** (1993), p. 279–308. 8, 31, 33, 67, 125

[KOV07] Jan Willem Klop, Vincent van Oostrom et Roel de Vrijer – « Lambda calculus with patterns », Draft 1.2, 21pp, January-June 8, 2007. Accepted for publication, 2007. 5, 74, 125, 135, 138

[Kri90] Jean-Louis Krivine – *Lambda-calcul : Types et modèles*, Etudes et Recherches en Informatique, Masson, 1990. 17

[Laf97] Yves Lafont – « Two-dimensional rewriting. », *8th International Conference on Rewriting Techniques and Applications, - RTA'97*, Lecture Notes in Computer Science, vol. 1232, Springer, 1997, p. 228–229. 207

[LDG+04] Xavier Leroy, Damien Doligez, Jacques Guarrigue, Didier Rémy et Jérôme Vouillon – « The Objective Caml system », 2004, `http://caml.inria.fr/pub/docs/manual-ocaml/`. 5

[Ler02] Xavier Leroy – « Compiling functional languages », *Summer School on Semantics of Programming Languages* (2002), Slides. 117

[Ler06] — , « Formal certification of a compiler back-end or : programming a compiler with a proof assistant », *Principles of programming languages - POPL'06*, 2006, p. 42–54. 11

[Les94] Pierre Lescanne – « From λσ to λν a journey through calculi of explicit substitutions », *Conference on Principles of Programming Languages - POPL'94*, ACM, January 1994, p. 60–69. 7, 87, 117

[Lév78] Jean-Jacques Lévy – « Reductions correctes et optimales dans le lambda-calcul », Ph.D. thesis, Université de Paris, 1978. 24

[LHL07] Luigi Liquori, Furio Honsell et Marina Lenisa – « A framework for defining logical frameworks », *Electronic Notes in Theoretical Computer Science.*, vol. 172, 2007. 8, 126

[LN04] Chuck Liang et Gopalan Nadathur – « Choices in representation and reduction strategies for lambda terms in intensional contexts », *Journal of Automated Reasoning* **33** (2004), no. 2, p. 89–132. 87, 117

[Loa03] Ralph Loader – « Higher order beta matching is undecidable », *Logic Journal of the IGPL* **11** (2003), no. 1, p. 51–68. 34, 41

[LS04] Luigi Liquori et Bernard P. Serpette – « irho : an imperative rewriting calculus », *6th Conference on Principles and Practice of Declarative Programming – PPDP'04*, ACM, 2004, p. 167–178. 67, 118, 122

[LW04] Luigi Liquori et Benjamin Wack – « The polymorphic rewriting calculus : Type checking *vs.* type inference », *International Workshop on Rewriting Logics and its Applications - WRLA'04* (Barcelona) (N. Marti-Oliet, M. Clavel et A. Verdejo, éds.), Electronic Notes in Theoretical Computer Science, vol. 117, 2004, To be published, p. 89–111. 67

[Mac98] Saunders Mac Lane – *Categories for the working mathematician*, 2nd. edition éd., Graduate Texts in Mathematics, Springer, New York / Berlin, 1998. 147

[Mel95] Paul-André Melliès – « Typed lambda-calculi with explicit substitutions may not terminate », *Second International Conference on Typed Lambda Calculi and Applications - TLCA'95* (M. Dezani-Ciancaglini et G. D. Plotkin, éds.), Lecture Notes in Computer Science, vol. 902, Springer, 1995, p. 328–334. 87

[Mel96] — , « Description abstraite des systèmes de réécriture », Thèse, Université Paris 7, 1996. 87

[Mel02] — , « Axiomatic rewriting theory VI residual theory revisited. », *Rewriting Techniques and Applications – RTA'02* (Copenhagen, Denmark), Lecture Notes in Computer Science, vol. 2378, Springer, July 2002, p. 24–50. 125

[MHF94] John C. Mitchell, Furio Honsell et Kathleen Fisher – « A lambda calculus of objects and method specialization », **1** (1994). 78

[Mil91] Dale Miller – « A logic programming language with lambda-abstraction, function variables, and simple unification », *Journal of Logic and Computation* (1991). 7, 31, 41

[MM82] Alberto Martelli et Ugo Montanari – « An efficient unification algorithm », *ACM Transactions on Programming Languages and Systems* **4** (1982), no. 2, p. 258–282. 7, 37, 43

[MN98] Richard Mayr et Tobias Nipkow – « Higher-order rewrite systems and their confluence », *Theoretical Computer Science* **192** (1998), p. 3–29. 8, 31, 125

[Mog89] Eugenio Moggi – « Computational lambda-calculus and monads », Logic in Compute Science, LICS'89, IEEE Computer Society Press, Washington, DC, 1989, p. 14–23. 159

[Mog91] — , « Notions of computation and monads », *Information and Computation* **93** (1991), p. 55–92. 159

[MOMV06] Narciso Martí-Oliet, José Meseguer et Alberto Verdejo – « Towards a strategy language for maude », *P6th International Workshop on Rewriting Logic and its Applications - WRLA'06*, vol. 117, Electronic Notes in Theoretical Computer Science, 2006. 11, 207

[MPU07] Claude Marché, Christine Paulin et Xavier Urbain – « The krakatoa tool for certification of java/javacards programs annotated in jml », *Journal of Logic and Algebraic Programming* (2007), To appear. 11

[MS01] Ooge de Moor et Ganesh Sittampalam – « Higher-order matching for program transformation », *Theoretical Computer Science* **269** (2001), p. 218–237. 6, 12, 31, 32, 205

[Muñ97] César 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. 87, 117

[Nad02] Gopalan Nadathur – « The suspension notation for lambda terms and its use in metalanguage implementations », *9th Workshop on Logic, Language, Information and Computation - WoLLIC'02* **67** (2002). 117

[NKK02] Quang-Huy Nguyen, Claude Kirchner et Hélène Kirchner – « External rewriting for skeptical proof assistants », *Journal of Automated Reasoning* **29** (2002), no. 3-4, p. 309–336. 122

[Ohl98] Enno Ohlebusch – « Church-Rosser theorems for abstract reduction modulo an equivalence relation. », *Rewriting Techniques and Applications (RTA-98)*, Lecture Notes in Computer Science, vol. 1379, 1998. 99, 114, 117

[Oos90] Vincent van Oostrom – « Lambda calculus with patterns », Technical report, Vrije Universiteit, Amsterdam, November 1990. 5, 12, 74, 125, 135, 138, 206

[Pad00] Vincent Padovani – « Decidability of fourth-order matching », *Mathematical Structures in Computer Science* **3** (2000), no. 10, p. 361–372. 35

[Pag98] Bruno Pagano – « X.R.S : Explicit reduction systems - A first-order calculus for higher-order calculi », *International Conference on Automated Deduction - CADE'98*, 1998, p. 72–87. 122

[Pey87] Simon Peyton-Jones – *Implementation of functional programming languages*, Prentice-Hall, 1987. 5, 12

[Pey03] Simon Peyton Jones – « Special issue : Haskell 98 language and libraries », *Journal of Functional Programming* **13** (2003). 5

[Pic07] Robert Pickering – *Foundations of f#*, (To Appear), 2007. 5

[Plo75] Gordon D. Plotkin – « Call-by-name, call-by-value and the λ-calculus », *Theoretical Computer Science* **1** (1975), p. 125–159. 75

[Plo76] — , « A powerdomain construction », *SIAM Journal on Computing* **5** (1976), no. 3, p. 452–487. 171

[Qia96] Zhenyu Qian – « Unification of higher-order patterns in linear time and space », *J. Log. Comput* (1996). 31, 35

[Raa93] Femke van Raamsdonk – « Confluence and superdevelopments », *5th International Conference on Rewriting Techniques and Applications - RTA'93* (C. Kirchner, éd.), Lecture Notes in Computer Science, vol. 690, Springer-Verlag, 1993, p. 168–182. 6, 17, 27

[Raa96] — , « Confluence and normalisation for higher-order rewriting », Thèse de doctorat, Vrije Universiteit, 1996. 17, 29, 30

[Rei06] Antoine Reilles – « Canonical abstract syntax trees », *6th International Workshop on Rewriting Logic and its Applications - WRLA'06*, Electronic Notes in Theoretical Computer Science, 2006, to appear. 11, 118

[Ros96] Kristoffer Høgsbro Rose – « Operational reduction models for functional programming languages », Thèse, DIKU, University of Copenhagen, Denmark, February 1996. 7, 87, 88, 117, 122

[Rus06] Michaël Rusinowitch – « Deciding protocol insecurity with rewriting techniques », *Proceedings of the 1st International Workshop on Security and Rewriting Techniques (SecReT'06)*, Invited Talk, 2006. 11

[Sch65] David Schroer – « The Church-Rosser theorem », Thèse, Cornell, June 1965. 23

[Sco80a] Dana Scott – « Lambda calculus : Some models, some philosophy », *The Kleene Symposium* (1980), p. 223–265. 147, 150

[Sco80b] — , « Relating theories of the λ-calculus », *To H. B. Curry : Essays on Combinatory Logic, Lambda Calculus and Formalism* (1980), p. 403–450. 147, 150

[Sco82] — , « Domains for denotational semantics », *International Conference on Algebraic and Logic Programming - ICALP'82*, Lecture Notes in Computer Science, vol. 140, Springer Verlag, 1982, p. 577–613. 171

[Sco96] — , « A new category ? domains, spaces and equivalence relations. », Tech. report, 1996, Manuscript available on the web page of the author. 147

[SDK+03] Aaron Stump, Arun Deivanayagam, Spencer Kathol, Dylan Lingelbach et Daniel Schobel – « Rogue decision procedures », *1st International Workshop on Pragmatics of Decision Procedures in Automated Reasoning - PDPAR'03*, 2003. 118

[SG89] Wayne Snyder et Jean Gallier – « Higher-order unification revisited : Complete sets of transformations », *JSCOMP : Journal of Symbolic Computation* **8** (1989). 43, 59

[Sin05] Francois-Régis Sinot – « Director strings revisited : A generic approach to the efficient representation of free variables in higher-order rewriting », *Journal of Logic and Computation* **15** (2005), no. 2, p. 201–218. 119, 206

[Sit01] Ganesh Sittampalam – « Higher-order matching for program transformation », Thèse, Magdalen College, 2001. 12, 31, 32

[Smy78] Peter M. B. Smyth – « Power domains », *Journal of Computer and System Sciences* **16** (1978), no. 1, p. 23–36. 171

[SS04] Aaron Stump et Carsten Schürmann – « Logical Semantics for the Rewriting Calculus », *5th International Workshop on Strategies in Automated Deduction* (M. Bonacina and T. Boy de la Tour, éd.), 2004. 118

[Ste00] Mark-Oliver Stehr – « CINNI — A generic calculus of explicit substitutions and its application to λ-, σ- and π-calculi », *3rd International Workshop on Rewriting Logic and its Applications - WRLA'00* (K. Futatsugi, éd.), Electronic Notes in Theoretical Computer Science, vol. 36, Elsevier, 2000, p. 71–92. 122

[Sti06] Colin Stirling – « A game-theoretic approach to deciding higher-order matching. », *International Conference on Algebraic and Logic Programming - ICALP'06*, 2006, p. 348–359. 35, 41

[Tai67] William Tait – « Intensional interpretation of functionals of finite type I », *The Journal of Symbolic Logic* **32** (1967). 20

[Tak95] Masako Takahashi – « Parallel reductions in λ-calculus », *Information and Compututation* **118** (1995), p. 120–127. 144

[Ter03] Terese – *Term rewriting systems*, Cambridge University Press, 2003, M. Bezem, J. W. Klop and R. de Vrijer, eds. 8, 23, 31, 125

[TG89] Val Tannen et Jean H. Gallier – « Polymorphic rewriting conserves algebraic strong normalization and confluence », *Automata, Languages and Programming, 16th International Colloquium*, Lecture Notes in Computer Science, vol. 372, Springer-Verlag, 1989, p. 137–150. 11

[Vis01a] Eelco Visser – « Stratego : A language for program transformation based on rewriting strategies », *Rewriting Techniques and Applications - RTA'01*, Lecture Notes in Computer Science, vol. 2051, 2001. 118

[Vis01b] — , « Stratego : A language for program transformation based on rewriting strategies. System description of Stratego 0.5 », *Rewriting Techniques and Applications (RTA'01)* (A. Middeldorp, éd.), Lecture Notes in Computer Science, vol. 2051, Springer-Verlag, May 2001, p. 357–361. 11, 207

[Vis05] — , « A survey of strategies in rule-based program transformation systems », *Journal of Symbolic Computation* **40** (2005), no. 1. 11

[Wac05] Benjamin Wack – « Typage et déduction dans le calcul de réécriture », Thèse de doctorat, Université Henri Poincaré Nancy 1, 2005. 5, 6, 67, 117, 205

[Wad93] Philip WADLER – « Monads and functional programming », Marktoberdorf International Summer School on Program Design Calculi (M. BROY, éd.), Springer-Verlag, New York, NY, 1993. 159

[YH90] Hirofumi YOKOUCHI et Teruo HIKITA – « A rewriting system for categorical combinators with multiple arguments », *SIAM Journal on Computing* **19** (1990), no. 1, p. 78–97. 100, 134

Résumé

Le calcul de réécriture ou rho-calcul est une généralisation du lambda-calcul avec filtrage et agrégation de termes. L'abstraction sur les variables est étendue à une abstraction sur les motifs et le filtrage correspondant peut être effectué modulo une théorie équationnelle a priori arbitraire. L'agrégation est utilisée pour collecter les différents résultats possibles.

Dans cette thèse, nous étudions différentes combinaisons des ingrédients fondamentaux du rho-calcul : le filtrage, l'agrégation et les mécanismes d'ordre supérieur.

Nous étudions le filtrage d'ordre supérieur dans le lambda-calcul pur modulo une restriction de la beta-conversion appelée super-développements. Cette nouvelle approche est suffisamment expressive pour traiter les problèmes de filtrage du second-ordre et ceux avec des motifs d'ordre supérieur à la Miller.

Nous examinons ensuite les modèles catégoriques du lambda-calcul parallèle qui peut être vu comme un enrichissement du lambda-calcul avec l'agrégation de termes. Nous montrons que ceci est une étape significative vers la sémantique dénotationnelle du calcul de réécriture.

Nous proposons également une étude et une comparaison des calculs avec motifs éventuellement dynamiques, c'est-à-dire qui peuvent être instanciés et réduits. Nous montrons que cette étude, et plus particulièrement la preuve de confluence, est suffisamment générale pour s'appliquer à l'ensemble des calculs connus. Nous étudions ensuite l'implémentation de tels calculs en proposant un calcul de réécriture avec filtrage et substitutions explicites.

Mots clefs : Filtrage, lambda-calcul, calculs de réécriture, calculs de motifs, filtrage d'ordre supérieur, super-développements, calculs explicites, confluence, sémantique catégorique, lambda-calcul parallèle, modèles syntaxiques.

Abstract

The rewriting calculus, also called the rho-calculus, is a generalisation of the lambda-calculus with matching capabilities and term aggregation. The abstraction on variables is replaced by the abstraction on patterns and the corresponding matching theory can be a priori arbitrary. The term aggregation is used to collect all possible results.

This thesis is devoted to the study of different combinations of the fundamental ingredients of the rho-calculus: matching, term aggregation and higher-order mechanisms.

We study higher-order matching in the pure lambda-calculus modulo a restriction of beta-conversion known as superdevelopments. This new approach is powerful enough to deal with second-order and higher-order Miller pattern-matching problems.

We next propose a categorical semantics for the parallel lambda-calculus that is nothing but an extension of the lambda-calculus with term aggregation. We show that it is a significant step towards a denotational semantics of the rewriting calculus.

We also study and compare pattern-based calculi where patterns can be dynamic in the sense that they can be instantiated and reduced. We show that this study, and particularly the confluence proof, is general enough so that it can be instantiated to recover all the already existing pattern-based calculi. We then study implementation of such calculi by proposing a rewriting calculus with explicit matching and explicit substitution application.

Keywords: Pattern-matching, lambda-calculus, pattern calculi, rewriting calculi, higher-order matching, superdevelopments, explicit calculi, confluence, categorical semantics, parallel lambda-calculus, per models.