Le calcul de réécriture de graphes : propriétés et capacités d'expression

Clara Bertolissi

Institut National
Polytechnique de Lorraine

Département de formation doctorale en informatique

École doctorale IAEM Lorraine

The graph rewriting calculus: properties and expressive capabilities

THÈSE

présentée et soutenue publiquement le 28 octobre 2005

pour l'obtention du

Doctorat de l'Institut National Polytechnique de Lorraine
(spécialité informatique)

par

Clara Bertolissi

Composition du jury

Rapporteurs :	Delia Kesner, Professeur, PPS, Université Paris 7, France Andrea Corradini, Professore, Universitá di Pisa, Italie
Examinateurs :	Paolo Baldan, Ricercatore, Universitá di Venezia, Italy Furio Honsell, Professore , Universitá di Udine, Italy Claude Kirchner, Directeur de Recherche, LORIA, INRIA, France Jean-Yves Marion, Professeur, LORIA, INPL, France
Invité :	Horatiu Cirstea, Maître de conference, LORIA, Université Nancy 2, France

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

Mis en page avec la classe thloria.

Alla mia famiglia

Remerciements

I would like to thank first of all my supervisor Claude Kirchner who followed me with high competence during my PhD studies. He had great influence on the direction of this thesis and guided me in my first steps in the world of research since my Master thesis.

I am grateful also to Horatiu Cirstea which has been always ready to listen to my ideas and provided to me useful comments which played a fundamental role in the development of the material in this thesis.

I would like also to acknowledge the members of the PhD examining committee.

Furio Honsell, who accepted to be the Italian supervisor of this joint PhD with the University of Udine. I am very grateful for his interest in my work, for his sensible advice and for the time he gave me during my periods in Udine, in spite of his overfull agenda.

Andrea Corradini and Delia Kesner, who kindly accepted to referee this thesis, for their careful reading of the document. Their appropriate and useful remarks allowed me to improve the quality of the final version of this thesis.

Paolo Baldan, who stimulated repeatedly the research on the topics presented in this thesis. I have learned a lot working with him, his remarks and suggestions have been a source of inspiration for my work. I would like to thank him also for his comforting words and his hospitality in Mestre.

Jean-Yves Marion, who accepted to read this thesis as internal reviewer. I thank him for taking part at the committee and for the interest he showed for my work.

I cannot forget Luigi Liquori who has been the promoter of my coming to Nancy and has received me in the best way. He helped me to feel familiar with the new country and the new colleagues.

I am grateful to all the PROTHEO team and the LORIA who allowed me to accomplish my studies in very comfortable conditions. Many thanks in particular to my colleagues Benjamin, Liliana and the others PhD students who shared the office with me during these years.

J'aimerais aussi remercier tous ceux que j'ai rencontré pendant ces trois années et qui sont devenus mes amis. Tout d'abord Laika, pour sa gentillesse, sa générosité et sa disponibilité à l'écoute. Merci à Jérôme pour beaucoup de choses, surtout pour sa vision optimiste de ma vie qu'il n'a pas hésité à me répéter dès que j'en avais besoin. Merci à Hacène, pour avoir été prêt à m'aider dans plusieurs occasions. Merci à tous les trois pour les bons moments partagés.

Merci à Fr et Nicolas, pour l'accueil et les bons week-ends à Paris, les balades en roller la nuit, les gaufres, le champagne et les blagues sur les italiens (du sud).

Je tiens à remercier le groupe de la chorale CIES qui m'a accompagné pour trois ans tous les jeudi soirs, en particulier Bruno, responsable adoré, et Fabrice.

Je remercie les copains que j'ai rencontré, pour des raisons différentes, depuis le début de la thèse: Philippe, Peter, Julian, Ben, Laure, Ando. Je remercie particulièrement Françoise, toujours prévenante vers les autres et de bonne humeur.

Merci à Bertrand pour son aide à plusieurs reprises et pour sa patience dans cette dernière période de la thèse.

Voglio inoltre ringraziare chi ha saputo starmi vicino anche se, dopo il primo anno a Nancy, ha scelto di tornare a casa. Manuela, per l'amicizia, l'incoraggiamento e l'energia che mi ha sempre trasmesso, e Matteo per le sue mail quasi quotidiane durante questi tre anni. Vorrei ringraziare anche Luca, per le belle estati passate insieme in spiaggia e l'affetto che ci lega, e gli altri amici di Lignano, Valeria, Giorgio e Ale.

Infine e soprattutto, grazie alla mia famiglia, per l'incoraggiamento e l'amore di sempre.

Contents

List of Figures **ix**

Introduction **1**

Introduzione **7**

Introduction **15**

La réécriture . . . 15
Le λ-calcul . . . 16
La réécriture d'ordre supérieur . . . 17
Les CRSs . . . 17
Le calcul de réécriture . . . 18
Des termes aux terme graphes . . . 19
Le λ-calcul avec récursion explicite . . . 20
Brève description du contenu . . . 20
Filtrage CRS . . . 21
Encodage des CRS . . . 22
Le calcul de réécriture de graphes . . . 23
La confluence du calcul de réécriture de graphes . . . 24
Survol des chapitres . . . 25
Perspectives . . . 26

1 First-order term rewriting **29**

1.1 Term algebra . . . 29
1.2 Binary relations and their properties . . . 31
1.3 Term rewrite systems . . . 33
1.3.1 Confluence . . . 35
1.3.2 Termination . . . 36
1.4 Rewriting modulo equational theories . . . 38
1.4.1 Equational theories . . . 38

1.4.2 Class rewriting and rewriting modulo E . . . 40
1.4.3 Properties . . . 40

2 Higher-order term rewriting 45
2.1 The λ-calculus . . . 45
2.1.1 Properties . . . 48
2.1.2 Simply typed λ-calculus . . . 50
2.2 Higher-order rewrite systems . . . 51
2.3 Combinatory Reduction Systems . . . 53
2.3.1 Terms and metaterms . . . 53
2.3.2 Substitution . . . 53
2.3.3 Rules and rewritings . . . 54

3 Matching in the CRS 57
3.1 Higher-order and CRS matching . . . 57
3.2 From CRS to λ-calculus . . . 58
3.3 Back from λ-calculus to CRS . . . 62
3.4 Properties . . . 64

4 The rewriting calculus 69
4.1 Syntax . . . 70
4.2 Semantics . . . 73
4.3 Properties and expressiveness . . . 75
4.4 The $\rho_{\mathtt{x}}$-calculus . . . 77

5 Expressing CRS in the ρ-calculus 81
5.1 The translation . . . 81
5.2 Examples . . . 83
5.3 Properties . . . 87
5.4 Comparison of the two systems . . . 91
5.5 Application to Higher-order Rewrite Systems . . . 92

6 Term graph rewriting 95
6.1 Operational approach . . . 96
6.2 Categorical approach . . . 99
6.3 Equational approach . . . 102
6.4 The cyclic lambda calculus . . . 104

7 The graph rewriting calculus **109**
7.1 General overview . . . 109
7.2 The syntax . . . 111
7.3 The graphical representation . . . 116
7.4 The semantics . . . 118
7.4.1 Basic rules . . . 120
7.4.2 Matching rules . . . 120
7.4.3 Graph rules . . . 121
7.5 Examples of reductions . . . 121
7.6 A strategy to maintain sharing . . . 125
7.7 Potential alternatives for the MATCHING RULES . . . 126

8 Confluence of the graph rewriting calculus **129**
8.1 General presentation . . . 129
8.2 Confluence modulo AC_ϵ for the τ-rules . . . 133
8.3 Confluence modulo AC_ϵ for the Σ-rules . . . 136
8.4 General confluence . . . 141
8.5 Summary: outline of the proof . . . 143

9 Expressivity of the ρ_{g}-calculus **145**
9.1 ρ_{g}-calculus versus ρ-calculus . . . 145
9.2 ρ_{g}-calculus versus cyclic λ-calculus . . . 148
9.3 Simulation of term graph rewriting into the ρ_{g}-calculus . . . 152

Conclusions et perspectives **161**

Bibliography **165**

List of Figures

1 Examples of term graphs . . . 3

1 Esempi di termini-grafi . . . 9

1 Exemples de terme graphes . . . 19
2 Traduction de problemes de filtrage CRS . . . 22

1.1 Properties of term rewriting . . . 33
1.2 Properties of rewriting modulo $\sim_E$ . . . 42
1.3 Proof diagrams . . . 43

2.1 Typing rules of $\lambda_{\rightarrow}$ . . . 51

3.1 Translation of CRS matching problems . . . 58

4.1 Small-step reduction semantics of the ρ-calculus . . . 73
4.2 Syntax of $\rho_{\mathtt{x}}$ . . . 77
4.3 Small-step semantics of the $\rho_{\mathtt{x}}$ . . . 78

6.1 Some term graphs . . . 97
6.2 Examples of term graphs . . . 99
6.3 Rewriting in the algebraic approach . . . 100
6.4 Evaluation rules of the $\lambda\phi$-calculus . . . 106

7.1 Syntax of the $\rho_{\mathtt{g}}$-calculus . . . 111
7.2 Some $\rho_{\mathtt{g}}$-terms . . . 113
7.3 $\rho_{\mathtt{g}}$-terms with matching constraints . . . 117
7.4 Graphs with cycles . . . 117
7.5 Rewrite rules with sharing . . . 118
7.6 Small-step semantics of the $\rho_{\mathtt{g}}$-calculus . . . 119
7.7 Examples of reductions . . . 122
7.8 Example of reductions . . . 124
7.9 Reduction to a cyclic term . . . 125
7.10 Reduction to black hole . . . 125

8.1 Confluence proof scheme . . . 133
8.2 Church-Rosser property for ρ_g/AC_ϵ . . . 143

9.1 Example of term graph rewriting . . . 154
9.2 Soundness . . . 158

9.3 Counterexample for cyclic terms 159
9.4 Counterexample to general soundness 160

Introduction

Computer systems play nowadays an important role in the society: business, economy, transport, health, travel and leisure take more and more advantage of the technological progress of software engineering. From this consideration, the need of correct and reliable systems comes as an evidence. Correctness of computer programs by a mathematical theory of computation is a fundamental concern of the theory of programming. To help in this task, *formal models* provide a suitable scientific and technological framework. A formal model is meant to represent a given system in a setting that is at the same time more abstract but sufficiently refined to describe the aspects of the system needed for testing its properties. The use of declarative rewrite-based languages during the development of a program can certify that formal and correct methodologies are followed during the production process. Such programs, written in rewrite-based languages, can be modeled as *term rewrite systems*.

Term rewriting

Term rewriting is a general model of computation which has been successfully applied in many areas of computer science, like functional and logic programming, automated theorem proving, transition systems, constraint resolution, to name just a few.

Even if term rewriting has been formally studied as a branch of theoretical computer science only since the last century, it is known and unconsciously used by everyone since childhood. Indeed, children games like the puzzle, consisting of different geometrical shapes to be put into holes of the corresponding shape, gives a first intuition of the concept of matching which is a key ingredient of rewriting. Moreover we start practicing rewriting since the primary school. The transformation of the mathematical expression $1 + 1$ into the elementary one 2 can be seen as a reduction, or in other words a step of rewriting, obtained by applying the rewrite rule $1 + 1 \to 2$ to the initial expression. The application of the rule is possible since the expression to rewrite is equal to the left-hand side of the rule. A more general rule would be the one of the form $x + x \to 2x$ where the variable x (representing a "hole" in the rule) can be replaced by any natural number. In this case any of the expressions $1 + 1$, $2 + 2$, *etc* can be reduced by the rule using an appropriate instantiation (matching) of the variable x.

Generally speaking, functional program execution can be seen as a reduction from input terms to output terms by applying a sequence of rules. In the associated abstract model, the terms are the subjects of the rewriting and are expressed in a certain formal language. The set of the applied rules forms the term rewrite system associated to the program. If a certain term contains a subterm that matches the left-hand side of one of these rules, *i.e.* it is a *redex*, then the subterm can be rewritten, that is replaced by the corresponding instance of the right-hand side. The interest of this formalisation relies in the fact that the rewrite semantics can be used to reason about the reduction behavior of the program. Typically, the properties analysed for term rewrite systems are termination and confluence. The termination property ensures that

a final output term exists for any given input term, while the confluence property ensures the unicity of this output.

Term rewrite systems as presented so far are more precisely called *first-order* rewrite systems [DJ90, Klo90, BN98], since variables can be instantiated only by terms and not by *e.g.* function symbols. First-order rewrite systems are useful to describe rewrite-based programs, but they are not powerful enough to manipulate programs as objects of the rewriting. Think for example at the derivative of a function. We all have learnt at the high school that the computation of the derivative of the sum of two functions can be simplified into the computation of the sum of the two derivative, *i.e.* $\mathsf{d}(f+g)(x) = \mathsf{d}(f)(x)+\mathsf{d}(g)(x)$. If we want to model this expression using term rewriting, we need to be able to express functions taking as arguments other functions and giving back functions as results. Another limitation of first-order rewriting is the impossibility to express a feature existing in most programming languages, that is to support a construct for local (or bound) variables.

Higher-order systems

The λ-*calculus*, introduced by Church in the 40's [Chu41], is a higher-order system that overcomes these limitations, in the sense that it naturally deals with functionality and has a notion of bound variables. This system has simple syntax and semantics, but it is expressive enough to represent all computable functions. For example, the identity function $id : x \mapsto x$ is written in the λ-calculus notation as the abstraction $\lambda x.x$ and the successor function $s : x \mapsto s(x)$ is written as $\lambda x.(s\ x)$. However, the simplicity of the λ-calculus turns out to be also a drawback, in the sense that the encoding of more complex functions is often not trivial and not easily readable. An integer n can be represented by the λ-term $\lambda f.\lambda x.f(f(\dots\ x))$ where f appears n times in the right-hand side of the "λ". It is clear that it would be more convenient to add to the λ-calculus a specification for natural numbers explicitly. Moreover, recursive functions are difficult to define and to manipulate in the λ-calculus.

This has motivated the study of *higher-order rewrite systems* [BT88, Oka89, Klo80, vR96] which combine the capability of dealing with functions as first-class citizens of the λ-calculus together with the representation of data structures with the more adapted syntax of term rewrite systems. For example, the rule for the derivative of the sum of two functions we mentioned before can be expressed in a higher-order rewrite formalism using the rewrite rule $\mathsf{d}(\lambda x.f\ x + \lambda x.g\ x)(y) = \mathsf{d}(\lambda x.f\ x)(y) + \mathsf{d}(\lambda x.g\ x)(y)$ where we can observe that the derivative $\mathsf{d}(\_)(\_)$ is represented by a higher-order function symbol. The first work on higher-order rewriting dates to 1980 and is due to J.W. Klop who presented the *Combinatory Reduction Systems* (CRS) in its P.h.D. thesis [Klo80]. Many other formalisms have been proposed since, like the Higher-order Rewrite Systems of T. Nipkow [Nip91] or the Expression Reduction Systems of Z. Khasidashvili [Kha90]. In these systems, function definition uses parameter passing inherited from the λ-calculus and rule application uses pattern matching inherited from term rewriting.

In the same line, the rewriting calculus (also called ρ-calculus) introduced by H. Cirstea and C. Kirchner in the late nineties [CK01], extends first-order term rewriting and λ-calculus. This calculus will play a central role in this thesis. The main design concept of the ρ-calculus is to make all the basic ingredients of rewriting such as the notions of rule *formation*, *application* and *result* explicit objects. With respect to the mentioned higher-order rewrite systems, the innovative feature of the ρ-calculus, is the possibility of parametrising the calculus choosing the way in which results of a rule application are returned, for examples using a multi-set, if the order of the result is irrelevant, or a set, if, in addition, the number of identical results is not important. Furthermore, matching can be performed using theories different from the usual empty theory, as

(a) A graph rewrite rule with sharing (b) Infinite list of ones

Figure 1: Examples of term graphs

equationally defined theories (like the associative-commutative theory) or higher-order theories. Adjusting these parameters to various situations permits to conveniently describe in a uniform but still appropriate manner different calculi. In particular it was shown that the ρ-calculus can be used to describe rewrite-based languages and object oriented calculi in a natural and simple way [CK98, CKL01].

From terms to term graphs

All the formalisms previously cited can be seen as models of computations that provide a theoretical basis for functional programming. Anyway, a certain gap exists between these frameworks and the implementation of functional languages. In real world applications, programmers have to address practical aspects like efficiency, sharing, cycles, data structures. In term rewriting, terms are represented by trees and computation is performed via subtree replacement. This way of rewriting turns out to be space and time consuming, and thus quite expensive. For example, in term rewrite rules multiple occurrences of variables in the right-hand side of rules lead to duplication of actual instances, as it happens for the variable y in the rule $\texttt{Twice} : \mathit{twice}(y) \to y+y$. Moreover, if y is instantiated by a reducible term $\mathit{twice}(1)$ then this redex will be duplicated in the right-hand side and each copy will need to be reduced separately.

We may want to reason not only about the outcome of computation, but also about the efficiency with which the result is obtained. A natural measure for this is the number of reduction steps, for what concerns efficiency with respect to time, and the number of nodes we need to allocate during the reduction, for efficiency with respect to space. Many problems and solutions concerning efficiency can be understood better when the semantics of the considered formalisms are extended by incorporating graph structures, leading to the so-called *term graph rewrite systems*.

In term graph rewriting, duplication of terms is avoided by copying the references to the term rather than the term itself. As a consequence, reductions of shared redexes are performed only once making the computation more efficient. Intuitively, if terms can be though of as trees, term graphs can be represented by trees whose nodes can be reached by several different edges. For example, the `Twice` rule mentioned above can be informally depicted as the term graph rewrite rule of Figure 1 (a). Beside their sharing features, term graphs can also be endowed with cyclic characteristics which increase their expressiveness. Indeed, the possibility of defining cycles allows one to represent easily regular infinite data structures. For example, the circular list $\mathit{ones} = 1 : \mathit{ones}$, where ":" denotes the concatenation operator, can be represented by the cyclic graph of Figure 1 (b).

Cyclic term graph rewriting has been widely studied, both from an operational point of view [BvEG$^+$87, Ken87, HP91, BS96] and from a categorical/logical point of view [SEP73, Löw93, EHK$^+$97, Kah97]. As in the case of term rewriting, higher order systems can be considered also in a term graph setting. For example, different versions of λ-calculus, involving sharing of functions and their arguments, have been proposed by Wadsworth in his Ph.D. thesis [Wad71]

and in the early nineties by Lamping [Lam90]. A few years later, an extension of the λ-calculus with explicit recursion, has been proposed by Ariola and Klop in an equational setting [AK96b]. More recently, a relational treatment of term graphs with bound variables have been proposed by Kahl in [Kah98].

As a conclusion, we can say that term graph rewriting provides a means for generating, manipulating and reducing term graphs, yielding to a framework which is a useful tool for reasoning about the compilation and execution of functional programming languages.

General approach of the thesis

This thesis aims at investigating the expressive capabilities of the rewriting calculus, with special interest for higher-order rewriting and the possibility of dealing with sharing and cycles.

Higher-order terms and their rewriting can be naturally represented in the current version of the rewriting calculus by instantiating appropriately the parameters of the calculus. On the other hand, the expression of graph-like structures in the rewriting calculus requires an extension of the classical syntax and consequently the definition of new suitable evaluation rules.

The work we present in this manuscript is divided into two parts.

The First part is dedicated to the analysis of the relationship between the ρ-calculus and other existing higher-order rewrite systems, most particularly with the *Combinatory Reduction Systems*.

Since the different higher-order rewrite systems and the rewriting calculus share similar concepts and have similar applications, it is important to compare these formalisms in order to better understand their respective strengths and differences. Such a comparison between different higher-order formalisms has already been conducted between *Combinatory Reduction Systems* and *Higher-order Rewrite Systems* by Van Oostrom and Van Raamsdonk in the nineties [VOVR93]. In this thesis, we concentrate on the relationship between *Combinatory Reduction Systems* and rewriting calculus analysing how CRS components and reductions can be expressed in the ρ-calculus. Like in functional languages, in CRSs there is a separation between specifications (rewrite rules) and applications (terms) of operations. The selection of an appropriate rewrite rule is realised by pattern matching. A clear understanding of the matching mechanism in the CRSs becomes therefore essential to be able to define a correct encoding of CRS derivations in the rewriting calculus. To our knowledge, an algorithm specifying from an operational point of view how a solution of a CRS matching problem can be provided has not been presented so far. Consequently, a study on CRS matching, which aimed at make more clear the higher-order nature of the matching and formalise it, has been carried out. The analysis of matching in CRSs is done with respect to higher-order matching in the λ-calculus, *i.e.* modulo $\beta\eta$. We show that the solutions of a CRS matching problem can be completely characterised by solving the corresponding higher-order matching problem on λ-terms.

Once CRS pattern matching has been completely specified, we turn our attention to the simulation of CRSs in the ρ-calculus. Given a CRS-term and a set of CRS-rules and the reduction of this term *w.r.t.* this set of rules, a method to automatically build a ρ-term having a corresponding reduction in the rewriting calculus can be defined. This study enlightened some differences in the way the two systems generate a rewrite step and treat substitutions and their application. In a more general context, this provides a contribution on the work of comparison of different higher-order rewrite systems.

In the Second part of the thesis we study an extension of the ρ-calculus enriched with features typical of term graph rewriting and we believe that this new formalism can provide a

support for future implementations of programming languages whose semantics can be described by the evaluation rules of the rewriting calculus.

If we consider other existing works which allow for the combination of graphical structures with higher-order capabilities, as the studies on optimal reduction strategies in the λ-calculus, we remark that these calculi gain in efficiency with respect to the systems computing on simple terms. Nevertheless, a last important ingredient is still missing in these formalisms: pattern matching. The possibility of discriminating using pattern matching could be encoded, in particular in the λ-calculus, but it is much more attractive to directly discriminate and to use indeed rewriting. Programs become quite compact and the encoding of data type structures is no longer necessary. On the other hand, matching is a feature provided by the ρ-calculus, where the computation of matching can be regulated using arbitrary theories increasing thus the expressiveness of the calculus.

In this thesis we combine the term graph techniques and the matching power of the ρ-calculus in an extension of the rewriting calculus towards the design of more efficient implementations.

Following an equational approach, this new system, called graph rewriting calculus or simply ρ_{g}-calculus, enriches the terms of the ρ-calculus with lists of constraints modeling sharing and cycles and allowing one to represent and compute over regular infinite entities. In addition, the computations related to the matching are made explicit and performed at the object-level. The calculus is shown to enjoy the confluence property under some linearity restrictions on patterns. Since in an equational setting different syntactical representations of a term can denote semantically the same graph, the ρ_{g}-calculus terms are grouped into equivalence classes and computations are performed over these classes. The calculus is thus considered in the context of rewriting modulo an equational theory, defined by the axioms characterising the equivalence classes, and the confluence property is proved modulo this theory.

The graph rewriting calculus is shown to be a common generalisation of the rewriting calculus and the λ-calculus with explicit recursion. Moreover, the calculus is shown to be expressive enough to simulate term graph rewriting. We obtain thus a framework where matching, graphical structures and higher-order capabilities are primitive.

Overview

The First part of the thesis consists of five chapters.

In Chapter 1, we introduce some background material. After fixing the basic mathematical notation, we present term rewriting with its terminology and its fundamental properties. In Chapter 2 we generalise the first chapter to higher-order systems, insisting on the definition of the λ-calculus and the *Combinatory Reduction Systems*. Chapter 4 uses notions taken from both the previous chapters in order to define the central calculus of this thesis, the ρ-calculus, which will be referred to in the following of the thesis.

Chapters 3 and 5 contain the original contribution of the First part.

In Chapter 3 we study a particular component of CRSs, namely CRS matching. The definition of a CRS matching problem is strongly related to higher-order pattern matching in the λ-calculus and thus we specify a method for providing a solution to a CRS matching problem using one of the already known algorithms on λ-terms. We present the transformations of CRS-terms into the simply typed λ-calculus, where λ-terms are built starting from only one base type, and from the λ-calculus back to CRSs. This allow us to correctly and completely define matching in CRSs exactly as the matching of the corresponding λ-terms. Consequently, we obtain as a result the proof of the decidability and the uniqueness of the solution for CRS matching.

In Chapter 5 we analyse the relation between the derivations performed in CRSs and in the

rewriting calculus, respectively. We define a translation of the various CRS concepts to the corresponding notions of the ρ-calculus, and, using this translation, we propose a method that associates to every derivation of a CRS-term a term with a corresponding derivation in ρ-calculus. More precisely, the target calculus is a particular instance of the ρ-calculus where the congruence modulo which matching is performed is chosen according to the results obtained in Chapter 3.

The SECOND PART of the thesis is divided into four chapters.

Chapter 6 provides an introduction to term graph rewriting, surveying briefly the different existing approaches, operational, categorical and equational. A section is dedicated to the presentation of the cyclic λ-calculus, an extension of the λ-calculus with sharing and cycles which will be referred to in the following chapters.

The original contributions of the SECOND PART are developed in Chapters 7 to 9.

Chapter 7 introduces the ρ_{g}-calculus, an extension of the ρ-calculus able to deal with graph like structures, where sharing of subterms and cycles can be naturally expressed by means of unification constraints. We present the components of the ρ_{g}-calculus syntax and the rules used for their transformation. In addition, a strategy meant to restrict the application of these rules in order to maintain the sharing information in the terms during the reduction is defined. A graphical representation of terms is proposed and several examples of terms and reductions are given. Several possibilities of a different ρ_{g}-calculus semantics, in particular *w.r.t.* the rules dealing with matching, are discussed at the end of the chapter.

Chapter 8 is devoted to the proof of confluence of the ρ_{g}-calculus, which can be achieved under some linearity assumptions. In a list of constraints associated to a ρ_{g}-calculus term the order of constraints and the empty constraint are not relevant, therefore terms are grouped into equivalence classes defined modulo the underlying theory associated to the constraint conjunction operator and rewriting is performed over these classes. The confluence proof is hence done in the setting of rewriting modulo a congruence relation using a technique that generalises the method of finite developments defined for the classic λ-calculus.

Chapter 9 studies the expressiveness of the ρ_{g}-calculus. We formally show that the calculus is a generalization of both the plain ρ-calculus and the cyclic λ-calculus. The representation of cyclic λ-terms and their derivations in the ρ_{g}-calculus is realised by defining a translation function between the terms of the two formalisms and by showing that reductions in the two calculi are equivalent modulo this translation. Moreover, we prove that the ρ_{g}-calculus can be naturally seen as an extension of ordinary term graph rewriting. We describe how ρ_{g}-calculus terms corresponding to reductions in term graph rewrite systems can be defined and we exploit the confluence result of the previous chapter to prove that the conservativity property holds for the ρ_{g}-calculus with respect to term graph rewriting.

The final chapter contains a summary of the presented results and sketches possible directions for future research.

Introduzione

I sistemi informatici giocano un ruolo importante nella società odierna: economia, trasporti, energia, salute, viaggi e divertimenti ricavano grandi vantaggi dal progresso tecnologico dell'ingegneria del software. Da questa considerazione, appare evidente la necessità di sistemi corretti che rispondano alle esigenze degli utilizzatori. La correttezza dei programmi informatici, studiata per esempio utilizzando una teoria matematica di calcolo, è quindi una preoccupazione fondamentale della teoria della programmazione. Uno dei contesti tecnologici e scientifici adeguati per la soluzione di questo problema è dato dai *metodi formali*. Un metodo formale è inteso a rappresentare un certo sistema in un contesto che è piú astratto e allo stesso tempo sufficientemente raffinato per descrivere gli aspetti del sistema necessari per provare le sue proprietá. L'uso di linguaggi dichiarativi basati sulla riscrittura durante lo sviluppo di un programma puó certificare che metodologie corrette e formali sono seguite durante il processo di produzione. Questi programmi, scritti in linguaggi basati sulla riscrittura, possono essere interpretati come *sistemi di riscrittura*.

Riscrittura di termini

La riscrittura di termini è un modello generale di calcolo che è stato applicato con successo in molte aree dell'informatica, come la programmazione funzionale e logica, dimostratori di teoremi automatizzati, sistemi di transizione, risoluzione di vincoli, per nominarne solo qualcuno.

Anche se la riscrittura di termini è stata studiata formalmente, come ramo dell'informatica teorica, solamente dal secolo scorso, essa è inconsciamente usata da tutti fin dall'infanzia. Infatti, i giochi per bambini come il puzzle in cui figure geometriche diverse devono essere introdotte in buchi aventi la forma corrispondente, danno una prima intuizione del concetto di "filtraggio" che è uno degli ingredienti principali della riscrittura. Inoltre, noi tutti cominciamo a usare nella pratica la riscrittura fin dalla scuola elementare. La trasformazione dell'espressione matematica $1 + 1$ nella sua forma elementare 2 puó essere vista come una riduzione, o in altre parole un passo di riscrittura, ottenuto applicando la regola di riscrittura $1 + 1 \to 2$ all'espressione iniziale. L'applicazione della regola è possibile dato che l'espressione da riscrivere è uguale al membro di sinistra della regola. Una regola piú generale è per esempio la regola del tipo $x + x \to 2x$, dove la variabile x (che rappresenta un "buco" nella regola) puó essere sostituita da qualsiasi numero naturale. In questo caso, ognuna delle espressioni $1 + 1$, $2 + 2$, *etc* puó essere ridotta dalla regola usando una instanziazione appropriata (data dal filtraggio) della variabile x.

In generale, l'esecuzione di un programma funzionale puó essere vista come una riduzione, partendo dai termini in entrata fino ai termini in uscita, data dall'appicazione di una sequenza di regole. Nel modello astratto associato, i termini sono soggetti alla riscrittura e sono espressi in un certo linguaggio formale. L'insieme delle regole che vengono applicate forma il sistema di riscrittura associato al programma. Se un certo termine contiene un sotto-termine che che filtra il membro di sinistra di una di queste regole, *i.e.* è un *redesso*, allora il sotto-termine puó essere

riscritto, cioé sostituito dalla corrispondenze instanza del membro destro della regola. L'interesse di questa formalizzazione sta nel fatto che la semantica della riscrittura puó essere usata per ragionare sul comportamento del programma durante le riduzioni. Di solito, le proprietá analizzate per un sistema di riscrittura sono la terminazione e la confluenza. La proprietá di terminazione assicura che un termine finale di uscita esiste per ogni termine di entrata fornito, mentre la proprietá di confluenza assicura l'unicitá di tale termine di uscita.

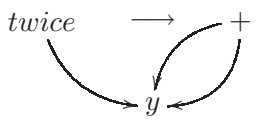
I sistemi di riscrittura di termini presentati fino a qui sono piú precisamente chiamati sistemi di riscrittura di *primo ordine* [DJ90, Klo90, BN98], poiché le variabili possono essere instanziate solo da termini e non *e.g.* da simboli funzionali. I sistemi di riscrittura di primo ordine sono utili per descrivere programmi basati sulla riscrittura, ma non sono abbastanza potenti per manipolare programmi come oggeti della riscrittura. Basti pensare per esempio alla derivata di una funzione. Sappiamo tutti dalla scuola superiore che la derivata di una somma di due funzioni si puó semplificare nel calcolo della somma delle due derivate, *i.e.* $\mathsf{d}(f+g)(x) = \mathsf{d}(f)(x)+\mathsf{d}(g)(x)$. Se vogliamo modellizzare questa espressione usando la riscrittura, dobbiamo essere in grado di esprimere funzioni che prendono come argomento altre funzioni e ritornano come risultato funzioni. Un'altro limite della riscrittura di primo ordine è l'impossibilitá di esprimere una caratteristica esistente in molti linguaggi di programmazione, ossia un costrutto per variabili locali (o legate).

Sistemi di ordine superiore

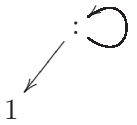
Il λ-calcolo, introdotto da Church negli anni 40 [Chu41], è un sistema di ordine superiore che supera queste limitazioni, nel senso che tratta naturalmente la funzionalitá e ha una nozione di variabili legate. Questo sistema ha una sintassi e una semantica semplici, ma è abbastanza espressivo per rappresentare tutte le funzioni calcolabili. Per esempio, la funzione identitá $id : x \mapsto x$ è scritta nel λ-calcolo come l'astrazione $\lambda x.x$ e la funzione successore $s : x \mapsto s(x)$ viene scritta $\lambda x.(s\ x)$. Tuttavia, la semplicitá del λ-calcolo diventa anche uno svantaggio, poiché spesso la rappresentazione di funzioni piú complesse non è triviale ne facilmente leggibile. Un numero intero n è tradotto nel λ-termine $\lambda f.\lambda x.f(f(\ldots\ x))$, dove f appare n volte nella parte destra del "λ". É chiaro che sarebbe piú conveniente aggiungere esplicitamente al λ-calcolo una specifica per i numeri naturali. Inoltre, le funzioni ricorsive sono difficili da definire e manipolare nel λ-calcolo.

Questo ha spinto allo studio dei *sistemi di riscrittura di ordine superiore* [BT88, Oka89, Klo80, vR96] che combinano la capicitá di trattare le funzioni come cittadini di prima classe del λ-calcolo insieme con la rappresentazione delle strutture dati attraverso la sintassi piú adatta dei sistemi di riscrittura di termini. Per esempio, la regola per la derivata della somma di due funzioni citata prima puó essere espressa in un formalismo di riscrittura di ordine superiore usando la regola di riscrittura $\mathsf{d}(\lambda x.f\ x+\lambda x.g\ x)(y) = \mathsf{d}(\lambda x.f\ x)(y)+\mathsf{d}(\lambda x.g\ x)(y)$ dove possiamo osservare che la derivata $\mathsf{d}(\_)(\_)$ è rappresentata da un simbolo funzionale di ordine superiore. Il primo lavoro sulla riscrittura di ordine superiore è dovuto a J.W. Klop che nel 1980 presentó i *Sistemi di Riduzione Combinatoria* (CRS) nella sua tesi di dottorato [Klo80]. Molti altri formalismi sono stati proposti da allora, come i Sistemi di Riscrittura di Ordine Superiore di T. Nipkow [Nip91] o i Sistemi di Riduzione di Espressioni di Z. Khasidashvili [Kha90]. In questi sistemi, la definizione di funzioni usa il meccanismo per il passaggio di parametri ereditato dal λ-calcolo e l'applicazione delle regole di riscrittura usa il metodo del filtraggio ereditato dalla riscrittura di primo ordine.

In linea con questi sistemi, il calcolo di riscrittura (anche chiamato ρ-calcolo) introdotto da H. Cirstea and C. Kirchner alla fine degli anni 90 [CK01], estende la riscrittura di primo ordine e il λ-calcolo. Questo calcolo ha un ruolo fondamentale in questa tesi. Il concetto principale del



(a) Una regola di riscrittura di grafi con condivisione



(b) Lista infinita di uno

Figure 1: Esempi di termini-grafi

ρ-calcolo è fare degli ingredienti di base della riscrittura, come la nozione di formazione, applicazione e risultato di una regola, degli oggetti espliciti del calcolo. La caratteristica innovativa del ρ-calcolo rispetto ai giá citati sistemi di riscrittura di ordine superiore è la possibilitá di parametrare il calcolo scegliendo il modo in cui i risultati dell'applicazione di una regola sono ritornati, per esempio usando un multi-insieme, se l'ordine dei risultati non è importante, o un insieme, se oltre all'ordine nemmeno il numero di risultati identici è importante. Inoltre, il filtraggio puó essere effettuato usando teorie diverse dalla solita teoria vuota, come teorie definite tramite equazioni (come la teoria associativa-commutativa) o teorie piú elaborate di ordine superiore. Aggiustando questi parametri a varie situazioni, si possono descrivere convenientemente calcoli diversi in maniera uniforme ma comunque appropriata. In particolare, è stato mostrato che il ρ-calcolo puó essere usato per descrivere linguaggi basati sulla riscrittura e calcoli orientati a oggetti in modo semplice e naturale [CK98, CKL01].

Dai termini ai termini-grafi

Tutti i formalismi citati precedentemente possono essere considerati come modelli di calcolo che forniscono una base teorica per la programmazione funzionale. Tuttavia, un divario esiste tra questi sistemi e le implementazioni di linguaggi funzionali. Nelle applicazioni reali, i programmatori devono gestire aspetti pratici come l'efficienza, la condivisione, i cicli, le strutture dati. Nella riscrittura di termini, i termini sono rappresentati da alberi e il calcolo viene fatto per mezzo della sostituzione di sotto-alberi. Questo modo di riscrivere si rivela dispendioso in termini di tempo e spazio, e quindi piuttosto costoso. Per esempio, in regole di riscrittura di termini, occorrenze multiple di una variabile nel membro di destra di una regola portano alla duplicazione dele istanze attuali della variabile, come succede per la variabile y nella regola `Twice` : $twice(y) \to y + y$. Inoltre, se y è istanziata con un termine riducibile $twice(1)$, questo redesso sará duplicato nel membro di destra della regola e ogni sua copia sará poi ridotta separatamente.

In effetti, siamo interessati non solo al risultato del calcolo, ma anche all'efficienza con la quale tale risultato è ottenuto. Un metodo di misura naturale per questo è il numero dei passi di riduzione, per quanto riguarda l'efficienza rispetto al tempo, e il numero di nodi che abbiamo bisogno di allocare in memoria durante la riduzione, per quanto riguarda l'efficienza rispetto allo spazio. Molti problemi e soluzioni sull'efficienza possono essere capiti meglio quando la semantica dei formalismi considerati è estesa incorporando strutture a forma di grafi, creando i cosí detti *sistemi di riscrittura di termini-grafi*.

Nella riscrittura di termini-grafi, la duplicazione di termini è evitata copiando la referenza al termine invece che il termine stesso. Di conseguenza, le riduzioni di termini condivisi sono effettuate solo una volta e questo rende il calcolo piú efficace. Intuitivamente, se i termini sono pensati come alberi, i termini-grafi possono essere rappresentati da alberi i cui nodi possono essere raggiunti da diversi archi. Per esempio, la regola `Twice` menzionata qui sopra puó essere informalmente disegnata come la regola di riscrittura di termini-grafi della Figura 1 (a). In aggiunta alla possibilitá di condivisione, i termini-grafi possono essere dotati anche di caratteris-

tiche cicliche che aumentano la loro espressivitá. Infatti, la possibilitá di definire cicli permette di rappresentare facilmente strutture dati infinite e regolari. Per esempio, la lista circolare $ones = 1 : ones$, dove ":" denota l'operatore di concatenazione, puó essere rappresentata dal grafo ciclico della Figura 1 (b).

La riscrittura di termini-grafi ciclica è stata ampiamente studiata, sia da un punto di vista operazionale [BvEG$^+$87, Ken87, HP91, BS96] che da un punto di vista logico/categoriale [SEP73, Löw93, EHK$^+$97, Kah97]. Come nel caso della riscrittura di termini, sistemi di ordine superiore possono essere considerati anche nel contesto dei termini-grafi. Per esempio, diverse versioni del λ-calcolo, con condivisione di funzioni e dei loro argomenti, sono state proposte da Wadsworth nella sua tesi di dottorato [Wad71] e nei primi anni novanta da Lamping [Lam90]. Qualche anno dopo, un'estensione del λ-calcolo con ricorsione esplicita è stata proposta da Ariola e Klop in un contesto equazionale [AK96b]. Piú recentemente, un trattamento relazionale di termini-grafi con variabili legate è stata studiato da Kahl in [Kah98].

In conclusione, possiamo dire che la riscrittura di termini-grafi fornisce un metodo per generare, manipolare e ridurre termini-grafi e costituisce un utile strumento per ragionare sulla compilazione e l'esecuzione di linguaggi di programmazione funzionali.

Approccio generale della tesi

Questa tesi vuole investigare le capacitá d'espressione del calcolo di riscrittura, con interesse particolare per la riscrittura d'ordine superiore e la possibilitá di trattare condivisione e cicli nei termini.

I termini di ordine superiore e la loro riscrittura possono essere rappresentati in maniera naturale nella versione attuale del calcolo di riscrittura, instanziando opportunamente i parametri del calcolo. D'altro canto, l'espressione di strutture di grafi nel calcolo di riscrittura necessita un'estensione della sintassi classica e di conseguenza la definizione di nuove regole di evaluzione appropriate.

Il lavoro presentato in questo documento si divide in due parti. La PRIMA PARTE è dedicata all'analisi della relazione tra il ρ-calcolo e altri sistemi di riscrittura di ordine superiore, piú in particolare con i *Sistemi di riduzione combinatoria*.

Poiché i diversi sistemi di riscrittura di ordine superiore e il ρ-calcolo hanno concetti simili e simili applicazioni, è importante confrontare questi formalismi per capire meglio le loro forze e debolezze rispettive. Un confronto tra due diversi sistemi di riscrittura di ordine superiore è giá stato studiato tra i "Sistemi di riduzione combinatoria" e i "Sistemi di riscrittura di ordine superiore" da Van Oostrom e Van Raamsdonk negli anni 90 [VOVR93]. In questa tesi, ci concentriamo sulla relazione tra i sistemi di riduzione combinatoria CRS$_s$ e il calcolo di riscrittura, analizzando come le componenti e le riduzioni dei CRS$_s$ possono essere espresse nel ρ-calcolo.

Nei CRS$_s$, come nei linguaggi funzionali, c' è una separazione tra le specifiche (le regole di riscrittura) e le applicazioni delle operazioni (i termini). La selezione di una regola di riscrittura appropriata è realizzata via filtraggio di motivi. Una comprensione chiara del meccanismo di filtraggio nei CRS$_s$ diviene quindi essenziale per definire una traduzione corretta delle derivazioni dei CRS$_s$ nel ρ-calcolo. A nostra conoscenza, nessun algoritmo che specifichi da un punto di vista operazionale come una soluzione di un problema di filtraggio CRS puó essere trovata è stato finora presentato. Di conseguenza, abbiamo sviluppato uno studio sul filtraggio CRS, avendo come scopo la chiarificazione della natura di ordine superiore del filtraggio e la sua formalizzazione. L'analisi del filtraggio nei CRS$_s$ è fatta rispetto al filtraggio di ordine superiore nel λ-calcolo, *i.e.* modulo $\beta\eta$. Mostriamo che le soluzioni di un problema di filtraggio CRS possono essere completamente caratterizzate risolvendo il corrispondente problema di filtraggio d'ordine superiore sui λ-termini.

Una volta terminato lo studio del filtraggio CRS, ci siamo interessati alla simulazione dei CRSs nel ρ-calcolo. Dati un termine CRS, un insieme di regole CRS e la riduzione di questo termine rispetto a questo insieme di regole, abbiamo definito un metodo per costruire in maniera automatica un ρ-termine avente una riduzione corrispondente nel ρ -calcolo. Questo lavoro ha evidenziato alcune differenze nel modo in cui i due sistemi generano un passo di riscrittura e trattano le sostituzioni e le loro applicazioni. In un contesto piú generale, questo nostro studio fornisce un contributo ai lavori sul confronto di diversi sistemi di riscrittura di ordine superiore.

Nella SECONDA PARTE della tesi studiamo un'estensione del ρ-calcolo arricchito con caratteristiche tipiche della riscrittura di termini-grafi. Siamo convinti che questo nuovo formalismo possa costituire un supporto per le future implementazioni di linguaggi di programmazione la cui semantica è descritta dalle regole di evaluazione del ρ-calcolo.

Se consideriamo altri lavori esistenti che combinano le strutture di grafi con capacitá di ordine superiore, come gli studi sulle strategie di riduzione ottimali nel λ-calcolo, possiamo notare che questi calcoli sono piú efficienti rispetto ai sistemi che fanno calcoli sui semplici termini. Nonostante ció, un ingrediente importante è ancora mancante in questi formalismi: il filtraggio di motivi. La possibilitá di discriminazione data dal filtraggio puó essere simulata, in particolare nel λ-calcolo, ma è molto piú attrattivo discriminare direttamente usando la riscrittura. In questo modo, i programmi diventano piuttosto compatti e la codifica di strutture dati non è piú necessaria. Dall'atro lato, il filtraggio è una caratteristica fornita dal ρ-calcolo, dove il calcolo del filtro puó essere regolato usando teorie arbitrarie, accrescendo cosí l'espressivitá del calcolo.

In questa tesi uniamo le tecniche della riscrittura di termini-grafi con il potere del filtraggio del ρ-calcolo, definendo cosí una estensione del calcolo di riscrittura in vista di implementazoni piú efficaci.

Seguendo un approccio equazionale, questo nuovo sistema, chiamato calcolo di riscrittura di grafi o semplicemente $\rho_{\mathtt{g}}$-calcolo, arricchisce i termini del ρ-calcolo con liste di vincoli sui termini che sono usate per rappresentare la condivisione e i cicli e permettono di descrivere e calcolare su entitá infinite regolari. Inoltre, i calcoli relativi al filtraggio sono resi espliciti ed effettuati a livello oggetto del calcolo. Abbiamo dimostrato che il $\rho_{\mathtt{g}}$-calcolo gode della proprietá di confluenza, sotto alcune restrizioni di linearitá sui motivi. Poiché in un contesto equazionale, diverse rappresentazioni sintattiche di un termine possono denotare semanticamente lo stesso grafo, i termini del $\rho_{\mathtt{g}}$-calcolo sono raggruppati in classi di equivalenza e i calcoli sono effettuati su queste classi. Il calcolo è quindi considerato nel quadro della riscrittura modulo una teoria equazionale, dove quest'ultima è definita dagli assiomi caratterizzanti le classi di equivalenza, e la proprietá di confluenza è dimostrata modulo questa teoria. Abbiamo mostrato che il calcolo di riscrittura di grafi è una generalizzazione comune del calcolo di riscrittura e del λ-calcolo con ricorsione esplicita. Inoltre, il calcolo è abbastanza espressivo per simulare la riscrittura di termini-grafi.

Abbiamo quindi cosí ottenuto un formalismo dove il filtraggio, le strutture di grafi e le capacitá di ordine superiore sono oggetti primitivi.

Sommario dei capitoli

La PRIMA PARTE della tesi è composta da cinque capitoli. Nel Capitolo 1, introduciamo i prerequisiti necessari. Dopo aver fissato le notazioni matematiche di base, presentiamo la riscrittura di termini con la sua terminologia e le sue proprietá fondamentali. Nel Capitolo 2 generalizziamo il primo capitolo alla riscrittura di ordine superiore, insistendo sulla definizione del λ-calcolo e dei sistemi di riduzione combinatoria. Il Capitolo 4 utilizza nozioni di entrambi i precedenti capitoli per definire il calcolo centrale di questa tesi, il ρ-calcolo, a cui sará fatto riferimento nel seguito

del documento.

I Capitoli 3 e 5 contengono le contribuzioni originali della PRIMA PARTE .

Nel Capitolo 3 studiamo una particolare componente dei CRSs, ovvero il filtraggio CRS. La definizione di un problema di filtraggio CRS è strettamente correlata con il filtraggio di ordine superiore, nel λ-calcolo. Per questo motivo specifichiamo un metodo per fornire una soluzione di un problema di filtraggio CRS usando uno degli algoritmi giá conosciuti per i λ-termini. Presentiamo la trasformazione dei termini CRS nel λ-calcolo con tipi semplici, dove tutti i λ-termini sono costruiti partendo da un solo tipo di base, e la trasformazione inversa dal λ-calcolo ai CRSs. Questo ci permette di definire correttamente e completamente il filtraggio nei CRSs esattamente come il filtraggio sui λ-termini corrispondenti. Come conseguenza, otteniamo la prova di decidibilitá e di unicitá della soluzione di un problema di filtraggio CRS.

Nel Capitolo 5 analizziamo la relazione tra le derivazioni effettuate nei CRSs e nel calcolo di riscrittura, rispettivamente. Definiamo una traduzione dei vari elementi di un CRS nelle nozioni corrispondenti del ρ-calcolo e, usando questa traduzione, proponiamo un metodo che associa a ogni derivazione di un termine CRS un termine con una derivazione corrispondente nel ρ-calcolo. Piú precisamente, il ρ-calcolo considerato è un'istanza particolare del ρ-calcolo dove la congruenza modulo cui il filtraggio viene effettuato è scelta seguendo i risultati ottenuti nel Capitolo 3.

La SECONDA PARTE della tesi è divisa in quattro capitoli. Il Capitolo 6 dún'introduzione alla riscrittura di termini-grafi, sorvolando brevemente i diversi approcci esistenti, operazionale, categoriale ed equazionale. Una sezione è dedicata alla presentazione del λ-calcolo ciclico, un'estensione del λ-calcolo con condivisione e cicli a cui verrá fatto riferimento nei capitoli seguenti.

Le contribuzini originali della SECONDA PARTE sono sviluppate nei Capitoli 7, 8 e9.

Il Capitolo 7 introduce il ρ_{g}-calcolo, un'estensione del ρ-calcolo capace di manipolare strutture a forma di grafo, dove la condivisione di sotto-termini e i cicli possono essere naturalmente espressi usando vincoli di unificazione. Presentiamo le componenti della sintassi del ρ_{g}-calcolo e le regole usate per la loro trasformazione. Inoltre, definiamo una strategia che vuole guidare l'applicazione di queste regole per mantenere le informazioni sulla condivisione dei termini durante la riduzione. Proponiamo una rappresentazione grafica dei termini e diamo diversi esempi di termini e loro riduzioni. Alla fine del capitolo, varie possibilitá per una semantica del ρ_{g}-calcolo diversa da quella scelta sono analizzate, in particolare per quanto riguarda le regole che effettuano il filtraggio.

Il Capitolo 8 è consacrato alla prova di confluenza del calcolo, che puó essere ottenuta sotto alcune assunzioni di linearitá. In una lista di vincoli associati a un termine del ρ_{g}-calcolo, l'ordine dei vincoli o il vincolo vuoto non hanno importanza. Per questo motivo, i termini sono raggruppati in classi di equivalenza definite modulo la teoria soggiacente all'operatore di congiunzione dei vincoli, e la riscrittura è effettuata su queste classi. La prova di confluenza è quindi fatta nell'ambito della riscrittura modulo una relazione di congruenza usando una tecnica che generalizza il metodo degli sviluppi finiti definito per il λ-calcolo.

Il Capitolo 9 studia l'espressivitá del ρ_{g}-calcolo. Mostriamo formalmente che il ρ_{g}-calcolo è una generalizzazione sia del ρ-calcolo classico che del λ-calcolo ciclico. La rappresentazione di λ-termini ciclici e delle loro derivazioni è realizzata nel ρ_{g}-calcolo definendo una funzione di traduzione tra i termini dei due formalismi e mostrando che le riduzioni nei due calcoli sono equivalenti modulo questa traduzione. Inoltre, proviamo che il ρ_{g}-calcolo puó essere visto in maniera naturale come un'estensione dell'ordinaria riscrittura di termini-grafi. Descriviamo come possiamo definire termini del ρ_{g}-calcolo che danno luogo a riduzioni corrispondenti a quelle del sistema di riscrittura di termini-grafi e usiamo il risultato di confluenza del capitolo precedente

per dimostrare che la proprietá di conservativitá del ρ_{g}-calcolo rispetto alla riscrittura di terminigrafi vale per la maggior parte delle riduzioni.

Il capitolo finale contiene un sommario dei risultati presentati e propone alcune direzioni possibili per futuri sviluppi di questa ricerca.

Introduction

De nos jours les systèmes informatiques jouent un rôle important dans notre société : commerce, économie, transports, santé, énergie, voyages et loisirs profitent de plus en plus de projets technologiques de l'ingénierie logiciel. Ainsi, le besoin de systèmes sûrs et fiables apparaît comme une évidence. La sûreté des programmes informatiques assurée par une théorie de calcul mathématique est un point fondamental de la théorie de la programmation. Pour aider à cette tâche, des modèles formels fournissent un cadre scientifique et technologique adequate. Un modèle formel a pour but de représenter un système donné dans des conditions à la fois plus abstraites mais suffisamment raffinées pour décrire les aspects du système nécessaires aux tests de ses propriétés. L'utilisation de langages déclaratifs et basés sur des règles lors du développement d'un programme peut certifier que des méthodes formelles fiables sont suivies durant le processus de production. De tels programmes, écrits en langages basés sur des règles, sont la plupart du temps modèlises comme des systèmes de réécriture de termes.

La réécriture de termes

La réécriture de termes est un modèle de calcul général qui a été appliqué avec succès dans de nombreux domaines de l'informatique, comme les langages de programmation basés sur la logique équationelle en particulier pour les preuves automatisées de théorèmes [JK84] et la résolution de contraintes [JK91]. Elle est aussi utilisée pour définir la sémantique opérationnelle de langages de programmation [Kah87], pour décrire des transformation de programmes [Deu96] et des systèmes de transition [DDHY92].

Si la réécriture de termes a été étudiée formellement comme une branche de l'informatique théorique depuis le dernier siècle seulement, elle est connue et utilisée implicitement dès notre plus jeune âge. En effet, les jeux d'enfance comme les puzzles consistant en différentes formes géométriques à placer dans des trous correspondant à chaque forme, donne une première intuition du concept de filtrage qui est un ingrédient clef de la réécriture. Bien encore, nous commençons à pratiquer la réécriture dès l'école primaire. La transformation de l'expression mathématique $1+1$ en l'expression élémentaire 2 peut être vue comme une réduction ou en d'autre termes un pas de réécriture obtenu en appliquant la règle $1+1 \to 2$ à l'expression initiale. L'application de la règle est possible puisque l'expression à réécrire est égale au membre gauche de la règle. Une règle plus générale pourrait être celle de la forme $x+x \to 2x$ où la variable x (représentant un "trou" dans la règle) peut être remplacée par n'importe quel entier naturel. Dans ce cas, n'importe quelle expression $1+1$, $2+2$, *etc* peut être réduite par la règle en utilisant une instance appropriée (filtrage) de la variable x.

En général, l'exécution de programmes fonctionnels peut être vue comme une réduction des termes d'entrée vers les termes de sortie par application d'une suite de règles. Dans le modèle abstrait associé, les termes sont les objets à réécrire et ils sont exprimés dans un certain langage

formel. L'ensemble des règles appliquées forme le système de réécriture de termes associé au programme. Si un certain terme contient un sous-terme qui filtre le membre gauche d'une de ces règles on dit qu'il est un *redex*, alors le sous-terme peut être réécrit, c'est-à-dire remplacé, par l'instance correspondante du membre droit de la règle. L'intérêt de cette formalisation tient dans le fait que la sémantique de la réécriture peut être utilisée pour raisonner sur le comportement de réduction du programme. Typiquement, les propriétés analysées pour les systèmes de réécriture de termes sont la terminaison et la confluence. La propriété de terminaison assure qu'un terme de sortie existe bien quel que soit le terme d'entrée, tandis que la propriété de confluence assure que ce terme de sortie est unique.

Les systèmes de réécriture de termes comme présentés ci-dessus sont plus précisément appelés systèmes de réécriture *du premier ordre*, ce qui peut être expliqué par le fait que les variable peuvent être seulement instantiées par des termes et non, par exemple, par des symboles de fonction. Les systèmes de réécriture du premier ordre sont utiles pour décrire des programmes basés sur la réécriture mais ils ne sont pas assez puissants pour manipuler les programmes comme des objets de la réécriture. Prenons par exemple la fonction de dérivée. Nous avons tous appris au lycée que le calcul de la dérivée de la somme de deux fonctions peut être simplifié par le calcul de la somme des dérivées de ces fonctions, *i.e.* $\mathsf{d}(f + g)(x) = \mathsf{d}(f)(x) + \mathsf{d}(g)(x)$. Si nous voulons modéliser cette expression en utilisant la réécriture de termes nous devons être capable d'exprimer des fonctions prenant pour arguments d'autres fonctions et dont le résultat est encore une fonction. Une autre limitation de la réécriture du premier ordre est l'impossibilité de définir des variables locales (ou *liées*), une une capacité existant dans la plupart des langages de programmation.

Le λ-calcul

Le λ-calcul, inventé et développé par Church au début des années 40, est un système de réécriture d'ordre supérieur qui permet de dépasser ces limitations, car il intègre l'aspect fonctionnel et possede une notion de variable liée. Ce système a une syntaxe et une sémantique simples, mais il est suffisamment expressif pour représenter toutes les fonctions calculables. Une fonction est représentée en utilisant une abstraction par rapport à ses arguments et l'application d'une fonction à un terme est réalisée en substituant le terme à la variable abstraite correspondante. Par exemple, la fonction identité $id : x \mapsto x$ est écrite dans la notation du λ-calcul comme l'abstraction $\lambda x.x$ et la fonction successeur $s : x \mapsto s(x)$ est écrite $\lambda x.(s\ x)$. Cependant la simplicité du λ-calcul se révèle aussi être un désavantage dans le sens que l'encodage de fonctions plus complexes est souvent non trivial et difficilement lisible. Un entier n correspond au λ-terme $\lambda f.\lambda x.f(f(\ldots\ x))$ ou f apparaît n fois dans le membre droit du "λ". Il est clair qu'il serait plus pratique d'ajouter au λ-calcul une spécification explicite pour les entiers naturels. De plus, les fonctions récursives sont difficiles à définir et à manipuler dans le λ-calcul.

Chacun de ces deux formalismes, à savoir les systèmes de réécriture du premier ordre et le λ-calcul, présentent des caractéristiques complémentaires. D'un part, le λ-calcul fournit un modèle de la fonctionnalité, mais il est souvent inadapté pour gérer de façon efficace certaines structures de données. Par exemple les entiers naturels munis de l'opération d'addition peuvent être codés dans le λ-calcul, mais ce codage est complexe et coûteux. En pratique donc, même si on sait coder l'algèbre dans le λ-calcul, on préfère l'utilisation des vraies structures algébriques à la place de leurs codages en λ-calcul. En outre il y a des égalités qui ne peuvent pas être définies dans le λ-calcul lui-même, comme par exemple l'axiome de *surjective pairing* [Bar84]. Ceci montre l'intérêt d'enrichir le pouvoir d'expression d'un système comme le λ-calcul.

D'autre part, la réécriture du premier ordre n'est pas bien adaptée pour traiter les fonctions comme paramètres et comme résultats d'autres programmes. Son pouvoir d'expression n'est pas suffisant pour décrire directement les fonctions agissant sur des fonctions comme par exemple la composition de fonctions.

Ceci justifie l'introduction d'un nouveau formalisme appelé réécriture d'ordre supérieur.

La réécriture d'ordre supérieur

La réécriture d'ordre supérieur est le moyen de traiter, par la réécriture, le calcul lié à la fonctionnalité : l'objectif est de pouvoir représenter une fonction comme un terme. Ainsi nous pouvons conserver la capacité de gérer les fonctions comme des citoyens de première classe grâce au mécanisme d'abstraction du λ-calcul et de la combiner avec la représentation de structures de données via la syntaxe agréable des systèmes de réécriture du premier ordre.

La réécriture d'ordre supérieur peut être vue comme une "réécriture avec fonctions" par rapport à la réécriture avec objets du premier ordre. Une fonction est définie syntaxiquement par un symbole que l'on nomme abstracteur associant un terme à une variable. L'application d'une fonction à une valeur consiste à instancier la variable dans le terme par la valeur. Les systèmes de réécriture d'ordre supérieur incorporent un tel abstracteur et un tel mécanisme d'instanciation que l'on appelle substitution. Par exemple, la règle pour la dérivée de la somme de deux fonctions mentionnée plus haut peut être exprimée dans un formalisme de réécriture d'ordre supérieur en utilisant la règle $\mathsf{d}(\lambda x.f\ x + \lambda x.g\ x)(y) = \mathsf{d}(\lambda x.f\ x)(y) + \mathsf{d}(\lambda x.g\ x)(y)$ où l'on voit que la dérivée $\mathsf{d}(\_)(\_)$ est représentée par un symbole de fonction d'ordre supérieur.

Ce formalisme est capable de modéliser en même temps les langages fonctionnels, équationnels ou orientés objets, aussi bien que des assistants à la démonstration.

Il y a deux façons distinctes de définir les systèmes de réécriture d'ordre supérieur :

- En étendant le λ-calcul avec des symboles fonctionnels et des règles de réécriture du premier ordre [BT88, Oka89, GBT89, JO97].
- En dotant les systèmes de réécriture du premier ordre d'un mécanisme comparable à la β-réduction du λ-calcul [KvOvR93, Wol93, NP98].

Le premier travail sur la réécriture d'ordre supérieur date de 1980 et a été accompli par J.W. Klop qui a présenté dans sa thèse les systèmes de réduction combinatoire (en anglais *Combinatory Reduction Systems*, abrégé CRSs par la suite). Depuis, beaucoup d'autres formalismes ont été proposés, comme les systèmes de réécriture d'ordre supérieur (Higher-order Rewrite Systems, abrégé HRSs) de T. Nipkow ou les systèmes de réduction d'expression (Expression Reduction Systems) de Z. Khasidashvili. Dans ces systèmes, la définition des fonctions utilise le passage de paramètre hérité du λ-calcul et l'application des règles utilise le filtrage des motifs hérité de la réécriture de termes.

Les CRSs

L'idée originale des CRSs est due à Aczel [Acz78], qui a présenté une classe restreinte des CRSs. Jan Willem Klop a ensuite étudié ces systèmes et en a proposé une généralisation dans sa thèse [Klo80]. Les CRSs sont des extensions de systèmes de réécriture du premier ordre par un mécanisme de liaison de variables. Un CRS est composé d'un ensemble de termes et d'un ensemble de règles de réécriture. Les termes, comme les règles de réécriture, peuvent contenir des variables liées par un abstracteur similaire à celui du λ-calcul. Les règles de réécriture

CRS, à la différence des règles de réécriture des systèmes de premier ordre, peuvent contenir des structures de liaison et un nouveau type de variables, appelées *métavariables*, d'arité fixée. Ces métavariables se comportent comme les variables libres dans les règles de réécriture du premier ordre. Le mécanisme d'évaluation est basé sur le filtrage appelé d'ordre supérieur, qui, en cas de succès, fournit une substitution appelée *assignation*. Les termes qui apparaissent dans une assignation sont des termes CRSs "impures" car ils contiennent en tête des λ-abstractions dont le nombre dépend de l'arité de la metavariable que le terme instancie. La réduction de ces λ-abstractions se fait au niveau méta du calcul au moment où l'assignation est appliquée à un terme CRS. Dans le langage du λ-calcul, la réduction d'une λ-abstraction correspond à un *développement*, et donc elle est toujours finie. Le résultat de l'application d'une assignation à un terme CRS est donc toujours un terme CRS.

Si toutes les règles de réécriture d'un CRS ne se superposent pas et sont linéaires gauche (c'est-à-dire qu'aucune métavariable n'apparait plus d'une fois dans les membres gauches) alors le CRS est dit orthogonal, ou régulier. Dans sa thèse, Klop a montré que cette classe de CRSs est confluente et la preuve a été reprise dans [KvOvR93] en utilisant une forme généralisée de développement appelée super-développement.

Le calcul de réécriture

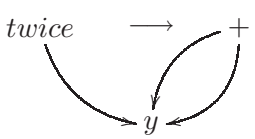
La réécriture et le λ-calcul ont été également généralisés dans le calcul de réécriture (appelé aussi ρ-calcul) introduit par H. Cirstea et C. Kirchner à la fin des années 90. Ce calcul jouera un rôle central dans cette thèse.

Les objets manipulés dans la réécriture du premier ordre sont les termes du premier ordre et dans une présentation simpliste nous pouvons dire qu'à chaque pas de réécriture on applique une *règle de réécriture* à une position quelconque d'un *terme (initial)* pour obtenir un autre *terme (résultat)*. On peut remarquer que les termes ne sont pas décrits au même niveau que la description des règles de réécriture qui les transforment et que la façon dont cette application est effectuée est définie au méta-niveau. On n'a donc aucun contrôle ni sur la sélection de la règle de réécriture appliquée, ni sur la position où cette règle est appliquée dans le terme à réduire.

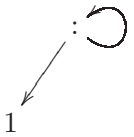
L'idée principale qui a inspiré le ρ-calcul est de rendre explicites tous les ingrédients basiques de la réécriture, à savoir les notions de formation, application et résultat de règles. Les règles et l'application des règles (ou des ρ-termes plus compliqués) sont des objets du ρ-calcul et les résultats des applications sont représentés par des ensembles qui sont également des ρ-termes.

Ainsi, les objets du ρ-calcul sont construits en utilisant une signature, un ensemble de variables, l'opérateur d'abstraction $\twoheadrightarrow$, l'opérateur d'application et nous considérons des ensembles de tels objets. Cela donne au ρ-calcul la capacité d'expliciter le non-déterminisme de la réécriture au moyen des ensembles de résultats. Naturellement, des variables peuvent être utilisées dans les règles de réécriture et nous pouvons dire qu'une ρ-règle de réécriture construite en utilisant l'opérateur $\twoheadrightarrow$ est une *abstraction* dont le rapport avec la λ-abstraction pourra fournir une intuition utile : une λ-expression $\lambda x.t$ est représentée dans le ρ-calcul par la règle $x \twoheadrightarrow t$. Le membre gauche des règles de réécriture peut évidemment être plus élaboré qu'une constante ou une variable. Dans ce cas, nous retrouvons dans l'évaluation le même comportement que dans le cas de la réécriture.

Par rapport aux systèmes de réécriture d'ordre supérieur déjà mentionnés, l'apport innovant du ρ-calcul est la possibilité de paramétrer le calcul en choisissant la manière dont les résultats de l'application d'une règle sont représentés, par exemple en utilisant un multi-ensemble si l'ordre du résultat n'est pas important, ou un ensemble si en plus le nombre des résultats identiques ne



(a) Une règle de réécriture de graphes avec partage



(b) Liste infinie de un

Figure 1: Exemples de terme graphes

compte pas. Aussi, le filtrage peut être effectué en utilisant différentes théories plus complexes que la théorie vide habituelle, comme les théories équationelles (par exemple la théorie contenant les axiomes d'associativité et commutativité) ou encore des théories d'ordre supérieur. Ajuster ces paramètres aux situations variées permet de décrire différents calculs de manière uniforme mais toujours appropriée.

Le ρ-calcul a de nombreuses applications tant comme cadre sémantique des langages basés sur la réécriture et le λ-calcul (ELAN, ML, XML, ...), que comme cadre logique pour la preuve et la vérification. Il est donc un formalisme très général et particulièrement puissant puisqu'il permet en particulier de représenter simplement les calculs à objets [CKL01] et la notion de contrôle des règles par des stratégies. Le calcul de base a été étudié en détail dans la thèse d'Horatiu Cirstea [Cir00] où il est montré que le pouvoir d'expression du ρ-calcul est suffisant pour exprimer les réductions du λ-calcul et de la réécriture du premier ordre.

Des termes aux terme graphes

Tous les formalismes cités précédemment peuvent être vus comme des modèles de calcul qui fournissent une base théorique pour la programmation fonctionnelle. Néanmoins, la différence entre ces cadres formels et l'implantation concrète des langages fonctionnels est non négligeable. Dans les applications réelles, les programmeurs doivent prendre en compte des aspects pratiques comme l'efficacité, le partage, les cycles, les structures de données, *etc*. En réécriture de termes, les termes sont représentés par des arbres et le calcul est effectué via remplacement de sous-arbres. Cette manière de réécrire se révèle coûteuse en terme d'espace et de temps. Par exemple, dans les règles de réécriture de termes les occurrences multiples de variables mènent à la duplication des instances, comme pour la variable y dans le membre droit de la règle `Twice` : $twice(y) \to y + y$. De plus, si y est instancie par un terme $twice(1)$ réductible, alors ce redex va être dupliqué dans le membre droit et il sera nécessaire de réduire chaque copie séparément.

On peut s'intéresser à raisonner non seulement sur le résultat des calculs, mais aussi sur l'efficacité avec laquelle il est obtenu. Pour cela, une mesure naturelle est le nombre de pas de réduction en ce qui concerne l'efficacité par rapport au temps, et le nombre de nœuds mis en mémoire durant la réduction, pour l'efficacité du point de vu de l'espace. Beaucoup de problèmes et solutions concernant l'efficacité peuvent être mieux compris quand la sémantique des formalismes considérés est étendue en y incorporant des structures de graphes, menant ainsi aux *systèmes de réécriture de terme graphes*.

Dans la réécriture de terme graphes la duplication des termes est évitée en copiant les références à un termes plutôt que le terme lui-même. En conséquence, les réductions des redexes partagés ne sont effectuées qu'une seule fois, rendant ainsi le calcul plus efficace. Intuitivement, si l'on pense aux termes comme à des arbres, les terme graphes peuvent être représentés par des arbres dont les nœuds peuvent être atteints par plusieurs arcs différents. Par exemple, la règle

`Twice` mentionnée auparavant peut être informellement décrite comme la règle de réécriture de terme graphes de la Figure 1 (a).

En plus de leur capacité de partage, les terme graphes peuvent être aussi munis de caractéristiques cycliques qui augmentent leur capacité d'expression. En effet, la possibilité de définir des cycles permet de représenter facilement des structures de données régulières infinies. Par exemple, la liste circulaire $ones = 1 : ones$, ou ":" symbolise l'opérateur de concaténation, peut être représentée par le graphe cyclique de la Figure 1 (b).

La réécriture de terme graphes fourni un moyen pour créer, manipuler et réduire des terme graphes, elle constitue donc un formalisme utile pour le raisonnement sur la compilation et l'exécution des langages de programmation fonctionnels. La réécriture de terme graphes cycliques a été largement étudiée, aussi bien du point de vue opérationnel [BvEG$^+$87, Ken87, HP91, BS96] que du point de vue de la théorie des catégories et de la logique [SEP73, Löw93, EHK$^+$97, Kah97]. Comme dans le cas de la réécriture de termes, des systèmes d'ordre supérieur peuvent être considérés pour les terme graphes. Par exemple, différentes versions du λ-calcul, mettant en œuvre le partage de fonctions et de leurs arguments, ont été proposées par Wadsworth dans sa thèse [Wad71] et, au début des années 90 par Lamping [Lam90]. Quelques années après, une extension du λ-calcul avec récursion explicite a été proposée par Ariola et Klop, dans un cadre équationel [AK96b].

Le λ-calcul avec récursion explicite

Le λ-calcul cyclique introduit par Ariola et Klop in [AK96b] généralise le λ-calcul ordinaire avec des cycles et du partage dans les termes et fourni ainsi un système de réécriture de termes graphes dans un cadre équationel. Les termes, appelés λ-graphes, sont représentés par un système d'équations de récursion qui peut être lu comme une construction `letrec` des langages fonctionnels. Dans le cas plus général, un λ-graphe peut être composé de plusieurs systèmes d'équations imbriquées. L'évaluation de ces termes se fait en utilisant un ensemble de règles qui rajoute à la règle β du λ-calcul des règles de transformation qui prennent en compte la nouvelle structure graphique des termes, comme par exemple des règles de dépliement, qui copient une partie du graphe, ou des règles de "garbage collection". Si ces règles sont appliquées sans aucune restriction, la confluence du calcul est perdue, en particulier à cause des configurations cycliques de redex mutuellement dépendants qui peuvent donner lieu à des réductions divergentes. La confluence peut être récupérée en introduisant un mécanisme de restriction sur les opérations de copie. Ce formalisme a été le point de départ pour le développement du calcul de réécriture de graphes qui est une des contributions principales de cette thèse, comme détaillé par la suite.

Description du contenu et résultats obtenus

Le sujet de cette thèse porte sur les capacités expressives du calcul de réécriture, en particulier par rapport à la réécriture d'ordre supérieur et la réécriture de terme graphes.

Les termes d'ordre supérieur et leur réécriture peuvent être naturellement représentés dans la version actuelle du ρ-calcul en instanciant de façon appropriée les paramètres du calcul. Par ailleurs, l'expression de structures de graphes dans le calcul de réécriture nécessite une extension de la syntaxe et par conséquent la définition de règles d'évaluation adaptées à la manipulation des nouveaux termes.

La thèse est constituée de deux parties principales.

La première partie est dédiée à l'analyse de la relation entre le ρ-calcul et d'autres systèmes de réécriture d'ordre supérieur déjà existants.

De façon générale, une fois un nouveau calcul défini, il est utile de le placer dans le panorama des calculs déjà existants ayant des capacités et des applications similaires, dans le but de mieux comprendre son pouvoir d'expression et son efficacité par rapport à ces autres calculs. Il a déjà été montré que le calcul de réécriture est une généralisation du λ-calcul et de la réécriture de termes du premier ordre, et plus généralement qu'il a un potentiel intéressant pour exprimer facilement les formalismes de calcul habituels. Sa nature et ses capacités d'ordre supérieur mènent à le comparer naturellement avec la réécriture d'ordre supérieur.

Une telle comparaison entre des formalismes d'ordre supérieur a été déjà étudiée pour les CRSs et les HRSs par Van Oostrom et Van Raamsdonk dans les années 90 [VOVR93]. Dans cette thèse, nous nous concentrons sur la relation entre les CRSs et le calcul de réécriture en analysant comment les composants des CRSs et leurs réductions peuvent être exprimés dans le ρ-calcul. Nous avons montré que la traduction d'un CRS dans le calcul de réécriture est complète et correcte (pour toute réduction dans un CRS on a une réduction correspondante dans le calcul de réécriture) [BCK03].

Comme dans les langages fonctionnels, il existe dans les systèmes de réécriture d'ordre supérieur tel-que les CRSs une séparation entre les spécifications (les règles de réécriture) et les applications (les termes) des opérations. La sélection d'une règle de réécriture appropriée est faite en utilisant le filtrage de motifs. Une compréhension claire du mécanisme de filtrage dans les CRSs est donc essentielle pour être capable de définir un encodage correct des réductions des CRSs dans le ρ-calcul. Une étude sur le filtrage CRS a amené à une formalisation précise des ses solutions et de la façon dont elles peuvent être produites.

Filtrage CRS

Dans les systèmes de réécriture d'ordre supérieur, tout comme dans ceux de premier ordre, le mécanisme de sélection d'une règle de réécriture est basé sur le filtrage avec motifs. D'un point de vue opérationnel, nous sommes intéressés à trouver une méthode de résolution d'un problème de filtrage qui nous fournisse la solution du problème. Autrement dit, nous voudrions définir un algorithme qui, en partant d'un motif et d'un terme soit capable de nous rendre une substitution qui appliquée au motif le rend égal au terme en entrée. Ce type d'algorithmes sont en général appelé algorithme de filtrage. Puisque jusqu'à maintenant à notre connaissance aucun algorithme de filtrage pour les CRSs n'a été spécifié, on montre dans cette thèse comment on peut produire une solution d'un problème de filtrage CRS donné. En conséquence, une étude du filtrage CRS a été menée dans le but de clarifier la nature d'ordre supérieur du filtrage et de le formaliser. Puisque le méta-langage des CRSs, c'est-à-dire le langage dans lequel les notions de substitution et de pas de réécriture sont exprimés, est basé sur le λ-calcul, nous avons une définition du filtrage CRS qui suggère naturellement la comparaison avec le filtrage d'ordre supérieur du λ-calcul, *i.e.* le filtrage modulo les règles β et η sur le λ-termes. De plus, une notion de motif semblable à celle du λ-calcul est utilisée dans les CRSs. En général, le filtrage d'ordre supérieur est indécidable, mais le filtrage de motifs d'ordre supérieur est décidable et unitaire comme conséquence de la décidabilité de l'unification de motifs [Mil91]. On décrit dans cette thèse comment une solution d'un problème de filtrage CRS peut être exhibée en transformant ce problème en un problème de filtrage d'ordre supérieur sur des λ-termes et en appliquant la transformation inverse à la substitution obtenue.

L'approche que nous proposons est résumée par le chemin de transformation décrit dans la Figure 2. La fonction de projection Π_λ traduit les termes CRS en λ-termes simplement typés et donc les contraintes de filtrage CRS en contraintes de filtrage d'ordre supérieur sur les λ-termes. On peut alors utiliser un des algorithmes déjà connu pour le filtrage d'ordre supérieur avec motifs

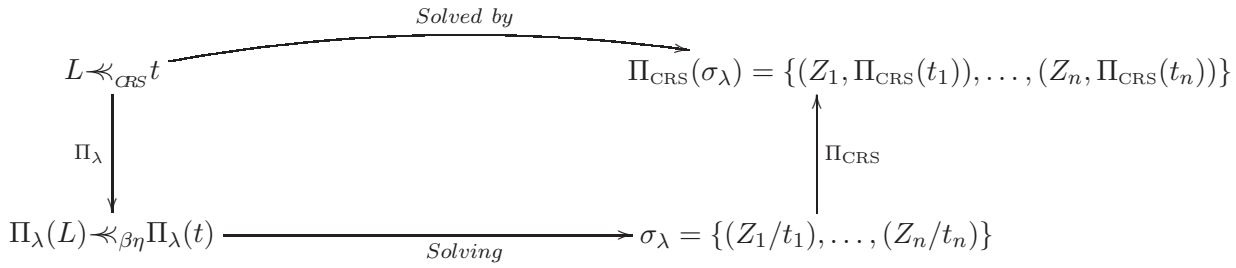


Figure 2: Traduction de problemes de filtrage CRS

pour trouver la substitution σ_λ qui résout la contrainte de filtrage ainsi traduite. Finalement, la fonction de projection Π_{CRS} traduit les λ-termes en termes CRS et donc la substitution σ_λ en une substitution CRS bien définie. Nous montrons que notre approche est correcte et complète. Le résultat final ainsi obtenu est la preuve de décidabilité du filtrage CRS et de l'unicité de sa solution.

Encodage des CRS

Puisque les différents systèmes de réécriture d'ordre supérieur et le calcul de réécriture partagent des notions similaires et ont des applications similaires, il est important de comparer ces formalismes pour mieux comprendre leurs points forts respectifs et leurs différences.

La relation entre le calcul de réécriture et les systèmes de la réécriture d'ordre supérieur est donc étudiée, en particulier avec les CRSs qui ont été historiquement le premier formalisme de réécriture d'ordre supérieur et ont inspiré des nombreux travaux ultérieurs dans ce domaine. Cette comparaison entre le ρ-calcul et les CRSs permet d'analyser le comportement de ces systèmes et de mieux comprendre comment les règles de réécriture sont appliquées aux termes et comment les réductions des termes sont effectuées dans les CRSs. Pour ce faire, nous définissons une traduction des différentes notions relatives aux CRSs (termes, substitutions et leur application) en notions correspondantes dans le ρ-calcul. En utilisant cette traduction, nous avons montré que chaque dérivation d'un terme par rapport à un CRS donné peut être représentée par un ρ-terme correspondant, construit automatiquement en partant du terme CRS initial et de sa réduction. Nous avons prouvé la complétude de notre approche, c'est-à-dire que chaque dérivation issue d'un ρ-terme du type décrit converge vers un terme équivalent au terme final CRS, modulo la traduction syntaxique. Cette étude a montré une certaine différence entre les deux systèmes. Dans les CRSs on peut distinguer deux niveaux: le niveau méta où les substitutions, leurs applications et les pas de réécriture sont exprimés, et le niveau objet, où les termes et les règles de réécriture sont définis. Le ρ-calcul est un calcul plus explicite où les notions considérées ci-dessus sont essentiellement traitées au niveau objet. Par exemple, dans les deux calculs, une fois un redex identifié, un pas de réécriture est composé de deux phases. La première phase consiste a résoudre le problème de filtrage entre le membre gauche de la règle et le terme auquel elle est appliquée. Si une solution existe, dans la deuxième phase cette solution est appliquée au membre droit de la règle. Dans les CRSs les deux phases sont réalisées au niveau méta, alors que dans le ρ-calcul l'application de la substitution peut engendrer des pas de réduction explicites. Pour cette raison, les réductions dans le ρ-calcul peuvent être en général plus longues que celles des CRSs. Si on considère la relation de réécriture dans sa généralité, l'encodage que nous avons proposé met en évidence des différences entre les deux calculs. "Marcher à travers le contexte" est fait implicitement dans

les CRSs, alors que pour diriger la réduction dans le ρ-calcul des termes supplémentaires sont nécessaires. Ceci est une conséquence du fait que les règles de réécriture sont définies au niveau objet dans le ρ-calcul et que leur application aux termes est explicite. La réduction est ensuite effectuée par les règles de la sémantique du calcul. Dans les CRSs, au contraire, on a un ensemble de règles de réécriture séparé de l'ensemble des termes. Ces règles sont particulières au CRS considéré et la stratégie utilisée pour leur application est laissée implicite. Ceci est probablement la différence principale entre le calcul de réécriture et les CRSs. Les règles de réécriture et par conséquent leur contrôle (la position d'application) sont définies au niveau objet dans le ρ-calcul, alors que pour le CRSs la stratégie de réduction n'est pas donnée explicitement.

Dans la deuxième partie nous nous sommes intéressés à la réécriture d'ordre supérieur sur les graphes. Après avoir analysé le rapport entre le ρ-calcul et plusieurs systèmes pour la transformation de graphes [AK96a, KKSd94, Cor93], nous nous sommes concentrés sur la définition d'une extension du ρ-calcul qui généralise le λ-calcul cyclique. Nous sommes intéressés d'un côté à étendre cette version du calcul pour pouvoir exprimer des entités régulières plus générales que les termes, notamment des termes avec partage (qui permettent une optimisation au niveau implémentation) et des termes cycliques (qui permettent la représentation de structures infinies), et d'un autre côté à analyser les systèmes de réécriture d'ordre supérieur et leur généralisation à la réécriture de graphes.

Si nous considérons d'autres travaux existants permettant la combinaison de structures graphiques et de capacités d'ordre supérieur, comme les études sur les optimisations du λ-calcul, nous remarquons que ces calculs gagnent en efficacité par rapport aux systèmes calculant sur de simples termes. Néanmoins, un dernier ingrédient important est toujours manquant dans ces formalismes: le filtrage avec motifs. La possibilité de faire la distinction utilisant le filtrage de motifs pourrait être encodée, en particulier dans le λ-calcul, mais il est bien plus attrayant de discriminer directement et d'utiliser effectivement la réécriture. Les programmes deviennent plutôt compacts et l'encodage de structures de type donné n'est plus nécessaire. D'autre part, le filtrage est une caractéristique intrinsèque du ρ-calcul, ou le calcul du filtrage peut être paramètré en utilisant des théories arbitraires augmentant ainsi la faculté d'expression du calcul.

Dans cette thèse, nous combinons les techniques des terme graphes et la puissance du filtrage du ρ-calcul dans une extension du calcul de réécriture en vue d'une implémentation plus efficace [BBCK05].

Le résultat est un calcul plutôt simple qui est montré confluent et assez puissant pour simuler le λ-calcul cyclique et la réécriture de terme graphes classique [Ber05].

Le calcul de réécriture de graphes

Le ρ-calcul et le λ-calcul avec récursion explicite ont tous deux été pensés pour fournir un modèle abstrait de calcul pour les langages de programmation fonctionnels et ils partagent donc des capacités communes, comme la représentation de fonctions au moyen d'un opérateur d'abstraction et un contrôle sur la relation de réécriture grâce à un opérateur d'application explicite. Ils ont aussi quelques capacités complémentaires très utiles, comme la possibilité de discriminer les termes via le filtrage du ρ-calcul, et l'efficacité en terme d'espace et de temps pour les réductions dans le λ-calcul avec récursion explicite, dûe à l'utilisation des structures de graphes.

En s'inspirant de ces deux calculs, nous introduisons le $\rho_{\mathtt{g}}$-calcul, un cadre formel unique intégrant de manière naturelle des caractéristiques d'ordre supérieur, les structures de graphes et la capacité de filtrage.

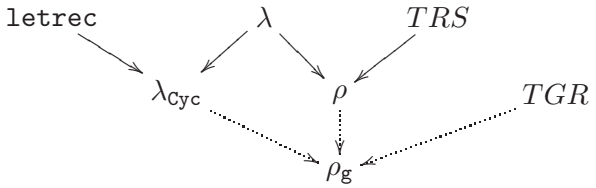
Dans la réécriture de termes, les termes ne sont pas décrits au même niveau que les règles de

réécriture et en particulier l'application d'une règle à un terme est définie au niveau méta. La gestion uniforme des notions de règle, d'application et de résultat au niveau objet est une des principales caractéristique du ρ-calcul, ainsi que la notion d'abstraction généralisée qui permet d'avoir dans le membre gauche du terme "$\_ \twoheadrightarrow \_$" des termes plus élaborés que des simples variables. Toutes ces capacités très utiles appartiennent aussi au $\rho_{\mathtt{g}}$-calculus.

Le $\rho_{\mathtt{g}}$-calcul permet en plus la représentation de graphes d'ordre supérieur éventuellement cycliques. Un terme du $\rho_{\mathtt{g}}$-calcul peut être vu comme une représentation compacte d'un terme du ρ-calcul éventuellement infini, obtenue par "dépliement" du graphe original. Plus formellement, par rapport au ρ-calcul, le $\rho_{\mathtt{g}}$-calcul étend la syntaxe avec des listes de contraintes. Ces contraintes peuvent être soit des contraintes de filtrage, soit des équations de récursion qui sont utilisées pour modéliser le partage et la présence de cycles dans les termes. Comme dans le ρ-calcul explicite, l'algorithme de filtrage est décrit directement au niveau du calcul, tout en tenant compte des nouvelles constructions. Nous avons montré la correction de cet algorithme dans le cas de termes non cycliques. Les termes du $\rho_{\mathtt{g}}$-calcul peuvent être manipulés en utilisant un ensemble de règles de transformation appropriées. Comparées au règles d'évaluation du ρ-calcul, celles du $\rho_{\mathtt{g}}$-calcul sont enrichies par un ensemble de règles traitant la structure de graphe de termes du $\rho_{\mathtt{g}}$-calcul et un ensemble de règles calculant la solution des contraintes de filtrage.

Puisque dans le cadre équationel différentes représentations syntaxiques d'un terme peuvent identifier sémantiquement le même graphe, les termes du $\rho_{\mathtt{g}}$-calcul sont groupés dans des classes d'équivalence et les calculs sont effectués sur ces classes. Le calcul est donc considéré dans le contexte de la réécriture modulo une théorie équationelle, définie par les axiomes caractérisant les classes d'équivalence, et la propriété de confluence a été prouvée modulo cette théorie.

Nous montrons aussi que le calcul de réécriture de graphes est une généralisation à la fois du calcul de réécriture et du λ-calcul avec récursion explicite ($\lambda_{\mathtt{Cyc}}$). Plus encore, le $\rho_{\mathtt{g}}$-calcul est assez expressif pour simuler la réécriture de terme graphes (TGR). On peut organiser informellement les formalismes mentionnés ci-dessus dans le diagramme suivant:



Le $\rho_{\mathtt{g}}$-calcul est donc un formalisme qui compte des contributions multiples de la part de différents calculs et permet à toutes leurs caractéristiques de cohabiter ensemble.

La confluence du calcul de réécriture de graphes

Un résultat important de cette thèse est la preuve de la confluence du calcul de réécriture de graphes, qui est obtenue sous des restrictions de linéarité sur les motifs. La confluence pour les systèmes d'ordre supérieur utilisant un filtrage non-linéaire est difficile puisque nous obtenons usuellement des paires critiques non-joignable, comme montré par Klop dans le cas du λ-calcul. Le contre-exemple de Klop peut être encodé dans le ρ-calcul, montrant donc que le ρ-calcul non-linéaire n'est pas confluent si aucune stratégie d'évaluation n'est imposée sur les réductions. Le contre-exemple est toujours valide en généralisant le ρ-calcul en $\rho_{\mathtt{g}}$-calcul. Nous nous restreindrons donc au $\rho_{\mathtt{g}}$-calcul linéaire pour l'étude de la confluence.

La preuve est plutôt élaborée, cela étant dû au fait que le calcul n'est pas normalisant. Les techniques usuelles pour démontrer la confluence des systèmes de réécriture de termes ont du

être adaptées aux nouvelles structures du calcul, *i.e.* les termes contenant des contraintes.

La démarche pour la preuve généralise celle du λ-calcul cyclique et utilise la notion de "développement" et la propriété de "finitude des développements" comme définies dans la théorie du λ-calcul. Puisque les termes sont groupés en classes d'équivalence définies modulo la théorie sous-jacente spécifiée pour l'opérateur de conjonction de contraintes, la preuve de la confluence est faite pour la relation de réécriture agissant sur ces classes d'équivalence. Plus précisément, deux termes linéaires du ρ_{g}-calcul sont considérés équivalents modulo l'ordre de leurs contraintes dans leur liste et la présence de la contrainte vide. Ils sont donc groupés dans des classes d'équivalence définie modulo l'associativite (A), la commutativité (C) et l'élément neutre (ϵ) de l'opérateur pour les listes. Ainsi, nous montrons la confluence modulo AC_ϵ de la relation de réécriture sur les classes d'équivalence AC_ϵ, noté $\rho_{\mathsf{g}}/AC_\epsilon$. Pour arriver à ce résultat, la relation plus opérationnelle $(\rho_{\mathsf{g}}, AC_\epsilon)$ *à la* Peterson and Stickel [PS81], qui effectue du filtrage modulo AC_ϵ sur un terme spécifique du ρ_{g}-calcul est utilisée tout au long de la preuve. D'une part cette notion de réécriture est plus pratique d'un point de vue calculatoire que la réécriture de classes AC_ϵ, d'autre part, sous quelques hypothèses convenables satisfaites par notre calcul, la confluence de la relation $(\rho_{\mathsf{g}}, AC_\epsilon)$ implique la confluence de la relation $\rho_{\mathsf{g}}/AC_\epsilon$. La confluence est obtenue par le biais de plusieurs lemmes qui utilisent des propriétés classiques de la réécriture de termes adaptées au cadre de la réécriture modulo un ensemble d'équations. Par exemple, la confluence de l'union de deux relations confluentes et qui commutent ou la confluence d'une relation terminante et localement confluente sont vérifiées si de plus une propriété supplémentaire, assurant le bon comportement de la relation de réécriture par rapport à la relation de congruence sur les termes, est satisfaite. Cette propriété, appelée compatibilité, assure que s'il existe un pas de réécriture à partir d'un ρ_{g}-terme G, alors le "même" pas peut être appliqué à partir de n'importe quel terme équivalent modulo AC_ϵ à G. Elle est valable pour toutes les règles du ρ_{g}-calcul et nous permet de conclure la confluence du ρ_{g}-calcul linéaire en partant de celle de la relation $(\rho_{\mathsf{g}}, AC_\epsilon)$.

Survol des chapitres

La première partie de la thèse est constituée de cinq chapitres.

Dans le Chapitre 1 nous introduisons quelques prérequis. Après avoir fixé les notations de base, nous présentons la réécriture de termes avec sa terminologie et ses propriétés fondamentales. Dans le Chapitre 2 nous généralisons les notions vues au premier chapitre aux systèmes d'ordre supérieur, en insistant sur la définition du λ-calcul et des Systèmes de Réduction Combinatoire (CRSs). Le Chapitre 4 utilise les concepts de deux chapitres précédents dans le but de définir le calcul central de cette thèse, le ρ-calcul, auquel fera référence la suite de la thèse.

Les Chapitres 3 et 5 contiennent les contributions originales de la première partie. Dans le Chapitre 3 nous étudions une composante particulière des CRSs, à savoir le filtrage CRS. La définition d'un problème de filtrage CRS est fortement reliée au filtrage de motifs d'ordre supérieur dans le λ-calcul et nous spécifions donc une méthode pour obtenir une solution d'un problème de filtrage CRS en utilisant un des algorithmes déjà connus pour le cas de λ-termes.

Dans le Chapitre 5 nous analysons la relation entre les dérivations effectuées respectivement dans les CRSs et dans le calcul de réécriture. Nous définissons une traduction de différents concepts des CRSs en notions correspondantes du ρ-calcul, et, en utilisant cette traduction, nous proposons une méthode qui associe à chaque dérivations d'un terme CRS un terme avec une dérivation correspondante dans le ρ-calcul. Plus précisément, le calcul cible est une instance particulière du ρ-calcul ou la congruence modulo laquelle le filtrage est effectué est choisie en

fonction des résultats obtenus dans le Chapitre 3.

La seconde partie de cette thèse est divisée en quatre chapitres. Le Chapitre 6 fournit une introduction à la réécriture de terme graphes survolant brièvement les différentes approches existantes: opérationnelle, catégorielle, équationelle. Une section est dédiée à la présentation du λ-calcul cyclique, une extension du λ-calcul avec partage et cycles à laquelle on se référera dans les chapitres suivants.

Les contributions originales de la seconde partie sont développées dans les chapitres 7 à 9. Le Chapitre 7 introduit le ρ_{g}-calcul, une extension du ρ-calcul capable de manipuler les structures de graphes, où le partage de sous-termes et les cycles peuvent être naturellement exprimés en terme de contraintes d'unification. Nous présentons les composants de la syntaxe du ρ_{g}-calcul et les règles utilisées pour leur transformation. En addition, nous définissons une stratégie pour restreindre l'application de ces règles dans le but de maintenir le partage d'information dans les termes. Une représentation graphique des termes est proposée et plusieurs exemples de termes et de réductions sont donnés. A la fin du chapitre, nous analysons différentes possibilités pour une sémantique du ρ_{g}-calcul différente de celle choisie, en particulier par rapport aux règles qui gèrent le filtrage.

Le chapitre 8 est consacré à la preuve de la confluence du ρ_{g}-calcul, qui peut être obtenue sous quelques hypothèses de linéarité. Les ρ_{g}-termes sont groupés dans des classes d'équivalence et la réécriture est effectuée sur ces classes. La preuve de la confluence est alors faite dans le cadre de la réécriture modulo une relation de congruence en utilisant une technique qui généralise la méthode des développements finis définie pour le λ-calcul classique.

Le chapitre 9 étudie la capacité d'expression du ρ_{g}-calcul. Nous montrons formellement que ce calcul est une généralisation à la fois du ρ-calcul de base et du λ-calcul cyclique. La représentation de λ-termes cycliques et de leurs dérivations dans le ρ_{g}-calcul est réalisée en définissant une fonction de traduction entre les termes de deux formalismes et en montrant que les réductions dans les deux calculs sont équivalentes modulo cette traduction. De plus, nous montrons que le ρ_{g}-calcul peut être vu comme une extension naturelle de la réécriture de terme graphes ordinaire. Nous décrivons comment les termes du ρ_{g}-calcul correspondant aux réductions dans les systèmes de réécriture de terme graphes peuvent être définis et nous utilisons le résultat de confluence du chapitre précèdent pour prouver que la propriété de conservativité est vérifiée par le ρ_{g}-calcul par rapport à la réécriture de terme graphes.

Le chapitre final contient un résumé des résultats présentés et décrit les directions possibles pour des futures recherches.

Perspectives

Nous nous sommes intéressés dans cette thèse au pouvoir expressif du calcul de réécriture. Dans ce contexte, nous avons proposé une traduction des CRSs dans le ρ-calcul, mais la traduction inverse n'a pas été explicitement définie ici. Nous pensons que cette traduction est possible, mais plutôt complexe car le contrôle explicite des règles de réécriture dans le ρ-calcul doit être simulé dans le système de réécriture CRS correspondant.

Les résultats présentés sont obtenus pour des CRSs satisfaisant la condition de motif. Il serait intéressant de comprendre si l'encodage proposé peut être étendu à des CRSs plus généraux et si la correspondance entre les réductions CRS et les réductions dans une version appropriée du ρ-calcul peuvent être définies de façon similaire. Ce n'est pas une généralisation triviale de notre résultat, puisque le filtrage d'ordre supérieur, utilisé dans les CRSs, est indécidable quand aucune restriction n'est imposée sur les motifs.

Le terme du ρ-calcul qui est utilisé pour l'encodage d'une dérivation CRS utilise des informations contenues dans le terme CRS initial, l'ensemble des règles CRS et la réduction de ce terme par rapport à cet ensemble de règles. Il serait certainement intéressant de construire un ρ-terme similaire en utilisant seulement le terme CRS initial et l'ensemble des règles CRS. Le développement d'une telle méthode a besoin de la définition de stratégies d'itération et de stratégies pour la traversée de termes. Il a été montré dans [CLW03] qu'on peut construire des ρ-termes qui décrivent l'application d'un système de réécriture du premier ordre par rapport à une stratégie donnée, comme par exemple "innermost" ou "outermost". Intuitivement, le ρ-terme qui encode le système de réécriture du premier ordre est modélisé en utilisant l'opérateur de structure du ρ-calcul et consiste en l'ensemble de règles de réécriture enveloppées dans un itérateur qui permet l'application répétitive des règles. Nous pensons que cette approche, basée sur des points fixes typés orientés objets, peut être appliquée ici pour la construction d'un ρ-terme approprié, en partant seulement de l'ensemble des règles CRS. Dans ce cas, l'application du ρ-terme ainsi obtenu à un terme donné encode la réduction de ce terme par rapport à l'ensemble de règle d'un CRS guidé par une stratégie de réduction donnée. Néanmoins, l'application de cette technique pour l'encodage d'un système de réécriture du premier ordre n'est pas directement applicable dans le cas d'un système de réécriture d'ordre supérieur. Dans l'itération, les règles de réécriture sont appliquées à un terme en partant de sa racine jusqu'à ses feuilles, en descendant sous les lieurs éventuellement présents dans le terme. Pour cette raison, si on veut que la procédure soit correcte, la méthode de déstructuration des termes nécessite la définition de manipulations sur les variables liées.

Le ρ_{g}-calcul est proche des implémentations concrètes de langages à base de règles, puisqu'il modélise le partage et les cycles. Une implémentation du calcul n'a pas encore été réalisée et représente sans doute un objectif intéressant pour un travail futur. On peut naturellement étendre l'implémentation de la version explicite du ρ-calcul, pour laquelle des études sont en cours et un interprète expérimental dans le langage TOM [MRV03] est déjà disponible, en y ajoutant les nouvelles caractéristiques du ρ_{g}-calcul.

Un sujet intéressant pour les travaux futurs est une version du ρ_{g}-calcul où les restrictions actuelles sur les motifs sont relâchées. Par exemple, on demande que les membres gauches des contraintes de filtrage soient acycliques, c'est-à-dire, intuitivement, qu'ils ne représentent que des termes finis. C'est une hypothèse commune lorsque la réécriture de terme graphes est pensée comme une implémentation de la réécriture de termes. D'autre part, un problème attrayant est la généralisation du ρ_{g}-calcul capable de traiter des théories de filtrage différentes, non syntaxiques. Le filtrage cyclique, c'est à dire le filtrage qui implique des membres gauches cycliques, peut être utile, par exemple, pour la modélisation des réactions sur des molécules cycliques, ou des transformations sur les réseaux de distributions. Le filtrage cyclique peut être formalisé comme une relation entre les graphes appelé *bisimulation*, qui intuitivement peut être vue comme l'équivalence des termes infinis obtenus par dépliement de deux graphes. Une sémantique du ρ_{g}-calcul adaptée pour gérer de façon appropriée ce type de calculs sur le filtrage est actuellement en cour d'étude.

De plus, puisqu'un terme du ρ_{g}-calcul peut être vu comme une représentation d'un terme du ρ-calcul éventuellement infini, il serait intéressant de définir une version infinie du ρ-calcul, en s'inspirant par exemple des travaux sur le λ-calcul infini [KKSd97] et sur la réécriture infinie [KKSdV91, Cor93]. Intuitivement, une réduction d'un terme cyclique du ρ_{g}-calcul peut correspondre à la réduction d'un nombre infini de redexes dans le terme infini correspondant. Donc, une relation précise entre les dérivations dans le ρ_{g}-calcul et dans la version infinie du ρ-calcul, respectivement, doit être étudiée pour avoir un résultat d'*adéquation* dans le style

de [KKSd94, CD97] et renforcer ainsi la vision du ρ_{g}-calcul comme une implémentation efficace des termes infinis et de leur réécriture.

L'expressivité du ρ_{g}-calcul offre un vaste spectre d'applications, comme les applications dans les réseaux de communications ou les applications dans le web.

Par exemple, les réseaux telecom peuvent être modélisés par un graphe dont la configuration change localement, selon les signaux reçus ou envoyés par ses nœuds. La notion de transformation locale est une caractéristique typique du ρ_{g}-calcul, ou le traitement explicite de l'application et du filtrage permet la description de la réécriture locale.

D'autre part, le ρ_{g}-calcul pourrait être utilisé pour décrire la sémantique et l'inter-opérabilité des services web. Les listes de contraintes associées aux termes et l'organisation hiérarchique de ces listes peuvent être utilisés pour représenter des informations sur les ressources dans le "World Wide Web", et donc le ρ_{g}-calcul peut être vu comme un point de départ pour un langage dans le style de RDF [MSB, HBEV04].

D'autres domaines intéressants sont la bio-informatique et l'informatique chimique. Les réactions entre molécules et les résultats de ces réactions peuvent être naturellement modélisés en utilisant un formalisme de réécriture de graphes comme le ρ_{g}-calcul, où les molécules peuvent être représentées par des graphes cycliques et les réactions chimiques peuvent être modélisées par des règles. Des chaînes de réductions sur une molécule deviennent alors des séquences de pas de réécriture sur le terme qui représente la molécule initiale. Un premier pas dans cette direction a été fait dans [Iba04]. Les molécules cycliques sont encodées comme des termes spéciaux équipés de labels sur les arrêts dans le but de garder la trace de cycles, et les réactions chimiques correspondent aux dérivations de réécriture sur les termes. Il est certainement intéressant de comprendre si, sur les mêmes lignes, une approche plus simple peut être fournie par le ρ_{g}-calcul, qui est capable de gérer les termes cycliques et leur évaluation.

Chapter 1

First-order term rewriting

In order to setup the necessary foundations for this work, in this chapter we first present some preliminary notions and we focus then on term rewriting, a general framework for specifying and reasoning about computation which has been successfully applied in many areas of computer science. After the definition of first-order rewrite systems, we take a closer look to their main properties, namely confluence and termination. We describe finally the extension of term rewriting to rewriting modulo a set of equations in which computations take into account non oriented equalities. Properties of term rewriting are generalised to this setting and adequate notions of coherence between the rewrite relation and the set of equalities are introduced for expressing abstract properties of such rewrite relations.

1.1 Term algebra

We first give the basic notions of first-order term algebra that will be used in several points through-out this thesis.

Definition 1 (Signature) *A signature Σ is composed of a finite set of symbols $\mathcal{F}$ and a countable set of variables $\mathcal{X}$. Every symbol in f in $\mathcal{F}$, called functional symbol, is associated to a natural number called its* arity *and denoted $arity(f)$. The subset of symbols of arity n is denoted $\mathcal{F}^n$. A symbol $f \in \mathcal{F}$ such that $arity(f) = 0$ is called a* constant.

Definition 2 (Term) *The set of terms generated from the signature $\Sigma = (\mathcal{F}, \mathcal{X})$ is denoted $\mathcal{T}(\mathcal{F}, \mathcal{X})$, or simply $\mathcal{T}$ when the signature is clear from the context, and it is defined as follows:*

$$\mathcal{T} ::= \mathcal{X} \mid \mathcal{F}^n(\mathcal{T}_1, \ldots, \mathcal{T}_n)$$

The set of terms $\mathcal{T}(\mathcal{F}, \mathcal{X})$ is also called the term algebra generated by the signature $\Sigma = (\mathcal{F}, \mathcal{X})$. We denote by $\equiv$ the syntactic identity of terms.

Example 1 (Peano integers) *Term algebras are very simple constructions and nevertheless quite expressive. Take for example the infinite set of integers. We can define it as a set of terms over a signature with only two symbols s of arity 1 (the successor function) and the constant 0. In this representation, the integer 1 is defined as $s(0)$ and an integer n is defined as $s(s(\ldots(0)\ldots))$ with n symbols s.*

Definition 3 (Variables of a term) *The set $\mathcal{V}ar(t)$ of variables of a term t is inductively defined by:*

- $\mathcal{V}ar(t) = \emptyset$ *if* $t \in \mathcal{F}^0$,
- $\mathcal{V}ar(t) = \{t\}$ *if* $t \in \mathcal{X}$,
- $\mathcal{V}ar(t) = \bigcup_{i=1}^{n} \mathcal{V}ar(t_i)$ *if* $t = f(t_1, \ldots, t_n)$ *with* $f \in \mathcal{F}^n$.

Definition 4 (Closed term, linear term) *A term t containing no variables, i.e. $\mathcal{V}ar(t) = \emptyset$, is called* closed. *A term in which every variable appears at most once is called* linear.

A term t may be viewed also as a *finite labelled tree*, the leaves of which are labelled with variables or constants, and the internal nodes of which are labelled with symbols of positive arity. The notion of *position* of a term, introduced formally in Definition 5, can then be seen as a sequence of naturals describing the path from the root of the term (seen as a tree) to the root of the sub-term at that position.

Definition 5 (Positions) *Let $\mathbb{N}_+$ be the set of positive naturals and $\mathbb{N}_+^*$ the corresponding monoid with neutral element ϵ and the concatenation operator ".". For all $\omega, \omega' \in \mathbb{N}_+^*$, ω is a prefix of ω', denoted $\omega \leq \omega'$, if it exists $\omega'' \in \mathbb{N}_+^*$ such that $\omega = \omega'.\omega''$. ω is a strict prefix of ω', denoted $\omega < \omega'$, if $\omega \leq \omega'$ and $\omega \neq \omega'$.*

A term (seen as a tree) built on a signature $\Sigma = (\mathcal{F}, \mathcal{X})$ is an application t from a non-empty part $\mathcal{P}os(t)$ of $\mathbb{N}_+^*$ to Σ such that

1. *$\mathcal{P}os(t)$ is closed under prefix (i.e. if $\omega \in \mathcal{P}os(t)$, then all prefixes of ω are belonging to $\mathcal{P}os(t)$) and*
2. *for all $\omega \in \mathcal{P}os(t)$ and any $i \in \mathbb{N}_+$, $\omega.i \in \mathcal{P}os(t)$ if and only if $t(\omega) = f \in \mathcal{F}$ and $1 \leq i \leq arity(f)$.*

$\mathcal{P}os(t)$ is called the set of positions (or occurrences) of t, and if t is finite then $\mathcal{P}os(t)$ is also finite. We call $\epsilon \in \mathcal{P}os(t)$ the empty sequence denoting the head position of t. $t(\omega)$ denotes the symbol at position ω in t; $t(\epsilon)$ is also called the head symbol of t. A sub-term of t at position $\omega \in \mathcal{P}os(t)$ is denoted $t_{|\omega}$ and defined by $\forall \omega.\omega' \in \mathcal{P}os(t), \omega' \in \mathcal{P}os(t_{|\omega}), t_{|\omega}(\omega') = t(\omega.\omega')$. We use the notation $t_{\lceil u \rceil_\omega}$ to precise that t has a sub-term u at position ω and the notation $t_{\lceil u \rceil_\omega}$ to signify that the sub-term $t_{|\omega}$ has been replaced by u in t. To simplify notations we write $t(\omega.i^n.\omega')$ for $t(\omega.\underbrace{(i \ldots i)}_{n}.\omega')$.

We introduce next the notion of *context*, that can be seen as a term with holes in it.

Definition 6 (Context) *Let $\Box$ be a new symbol that does not occur in Σ. A* context *is a term in $\mathcal{T}(\mathcal{F}, \mathcal{X} \cup \Box)$. We denote by $\mathsf{Ctx}\{\Box\}$ a context with exactly one hole and by $\mathsf{Ctx}\{t\}$ the term obtained by replacing the hole $\Box$ by t.*

Following the previous definition, $\mathsf{Ctx}\{\Box\} = f(g(\Box), h(a)))$ is a context and $\mathsf{Ctx}\{c\} = f(g(c), h(a)))$ is the term obtained by replacing the hole with the constant c. We define next an operation on terms called substitution, which consists of replacing a variable in a term by another term.

Definition 7 (First-order substitution) *A substitution σ is a mapping from the set of variables to the set of terms. A finite substitution has the form $\sigma = \{x_1/t_1 \ldots x_m/t_m\}$, for $m \geq 0$, where the set $Dom(\sigma) = \{x_1, \ldots, x_m\}$ is called the* domain *of the substitution. The application of a substitution σ to a term t, denoted by $\sigma(t)$ or $t\sigma$, is defined as follows:*

$$\sigma(f(t_1, \ldots, t_m)) = f(\sigma(t_1), \ldots, \sigma(t_m)) \qquad \sigma(x_i) = \begin{cases} t_i & \text{if } x_i \in Dom(\sigma) \\ x_i & \text{otherwise} \end{cases}$$

The composition of two substitutions τ and σ is denoted by concatenation, $\tau\sigma$, and defined as $(\tau\sigma)(t) = \tau(\sigma(t))$.

The substitution operation is often used with the aim of making two terms equal, solving thus a so-called (syntactic) *unification* problem. This kind of problem arises in many symbolic computation algorithms and for example in confluence tests based on critical pairs, defined in Section 1.3.1.

Definition 8 (Unification) *A substitution σ is solution of the* unification equation $s =^? t$ *if $\sigma(s) \equiv \sigma(t)$. If such substitution exists, the two terms s and t are said* unifiable *and the substitution σ is called a* unifier. *Moreover, σ is a* most general unifier *of s and t if, for any other unifier σ', there exists a substitution τ such that over $\mathcal{V}ar(s) \cup \mathcal{V}ar(t)$ we have $\sigma' = \tau\sigma$.*

When one of the terms is closed, the syntactic unification problem becomes simpler and is called a *matching* problem.

Definition 9 (Matching) *A substitution σ is solution of the* match equation $t \lessdot s$ *if $s \equiv \sigma(t)$. The term t is called the* pattern *against which s is matched.*

The matching (unification) problems defined above are called *syntactic* to distinguish them from matching (unification) problems modulo an equational theory, defined in Section 1.4.1, and from higher-order matching defined in Section 2.1.

Syntactic matching is decidable and linear in the size of a linear pattern. Otherwise, it is linear in the size of the term to be matched (the term s in the definition above).

Syntactic unification is decidable and linear if terms are represented with maximal sharing, *i.e.* using data structures as the *dags* (directed acyclic graphs) [Kir90]. In the general case, computing the most general unifier of a syntactic unification problem requires exponential space.

Example 2 (Unification and matching problems)

- *The unification equation $f(g(x)) =^? f(y)$ has as unifier $\sigma_1 = \{y/g(x)\}$ and $\sigma_2 = \{y/g(a), x/a\}$. σ_1 is the most general unifier, since $\sigma_2 = \{x/a\}\sigma_1$.*
- *The match equation $f(g(x)) \lessdot f(a)$ has no solution.*
- *The match equation $h(x, x) \lessdot h(a, a)$ has solution $\sigma = \{x/a\}$.*
- *The match equation $h(x, x) \lessdot h(a, b)$ has no solution.*

1.2 Binary relations and their properties

Terms are connected by means of relations or transformations. We give here some definitions for binary relations that will be used in the following.

Definition 10 (Binary relations) *Let $\rightarrow$ be a binary relation over a set $\mathcal{T}$. We denote by*

- *$\leftarrow$ the inverse relation of $\rightarrow$,*
- *$\twoheadrightarrow$ the reflexive and transitive closure relation of $\rightarrow$,*
- *$\leftrightarrow\!\!\!\!\!\rightarrow$ or $=$ the reflexive symmetric and transitive closure of $\rightarrow$. $\leftrightarrow\!\!\!\!\!\rightarrow$ $(=)$ is called an* equivalence *or* congruence *relation.*

Moreover, we define the context closure of $\mapsto$ *as the smallest relation containing* $\rightarrow$ *closed w.r.t. the formation rules of the terms of* $\mathcal{T}$*.*

The composition of two relations $\rightarrow_1$ and $\rightarrow_2$ is denoted $\rightarrow_1 \rightarrow_2$ or $\rightarrow_{12}$. A binary relation $\mapsto$ on terms is called *stable by context* if for any terms s, t, u and any position $\omega \in \mathcal{P}os(u)$ we have that $s \mapsto t$ implies $u_{[s]_\omega} \mapsto u_{[t]_\omega}$.

A pre-order over a set T is a binary, transitive, reflexive relation over the elements of T. An order over a set T is a binary, transitive, reflexive, antisymmetric relation over the elements of T. An order is *total* over a set T if for all $s, t \in T$, we have $s \geq t$ or $t \geq s$. An order is said *partial* otherwise. The associated *strict* order $>$ is defined by $s > t$ if $s \geq t$ and $s \neq t$.

Definition 11 (Well-founded ordering) *An ordering* $\geq$ *over a set* T *is* well-founded *if there exists no infinite chain* $x_1 > x_2 > \ldots > x_n > \ldots$ *where* $x_i \in T$ *(*$i \in \mathbb{N}_+$*).*

Binary relations can be applied to transform terms and this leads to the notion of term reducibility.

Definition 12 (Reducibility) *Let* R *be a relation on a set* $\mathcal{T}$*. An element* t *of* $\mathcal{T}$ *is* R-reducible *if there exists* t' *in* $\mathcal{T}$ *such that* $t \mapsto_R t'$*. Otherwise* t *is called* irreducible *for* R*. We call* R-normal form *of* t *any irreducible element* t' *such that* $t \mapsto\!\!\!\!\!\rightarrow_R t'$ *and we denote it by* $t\downarrow_R$*, if it is unique.*

We define next some properties of a relation R on a set $\mathcal{T}$. For a graphical representation see Figure 1.1 where solid arrows stand for universal and dotted arrow stand for existential quantification.

Definition 13 (Termination, Confluence) *Let* R *be a binary relation over a set* $\mathcal{T}$*.*

- *The relation* R *is* strongly normalising *(or* terminating*), denoted SN(R), if there exists no reduction with an infinite sequence of steps.*
- *The relation* R *is* weakly normalising*, denoted WN(R), if there exists at least one finite reduction of* t *for all* $t \in \mathcal{T}$*.*
- *The relation* R *has the* diamond property *if*

$$\leftarrow_R \cdot \mapsto_R \subseteq \mapsto_R \cdot \leftarrow_R$$

- *The relation* R *is* locally confluent *if*

$$\leftarrow_R \cdot \mapsto_R \subseteq \ \mapsto\!\!\!\!\!\rightarrow_R \cdot \leftarrow\!\!\!\!\!\!\rightarrow_R$$

- *The relation* R *is* confluent *if*

$$\leftarrow\!\!\!\!\!\!\rightarrow_R \cdot \mapsto\!\!\!\!\!\rightarrow_R \ \subseteq \ \mapsto\!\!\!\!\!\rightarrow_R \cdot \leftarrow\!\!\!\!\!\!\rightarrow_R$$

- *The relation* R *is Church-Rosser if*

$$\leftrightarrow\!\!\!\!\!\rightarrow_R \ \subseteq \ \mapsto\!\!\!\!\!\rightarrow_R \cdot \leftarrow\!\!\!\!\!\!\rightarrow_R$$

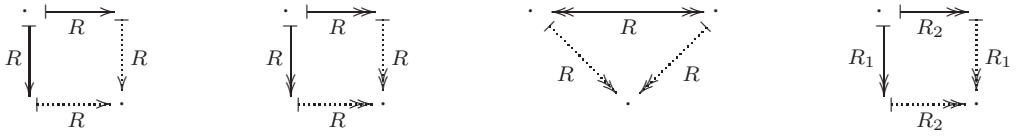


Figure 1.1: Properties of term rewriting

It is easy to see that the diamond property implies confluence and confluence implies local confluence. The Church-Rosser property and confluence are equivalent properties.

In general, proving confluence is a difficult task, since one has to consider reductions of arbitrary length. If the relation is strongly normalising, a local test of confluence is sufficient.

Theorem 1 (Newman [New42]) *A strongly normalising relation is confluent if it is locally confluent.*

Given two confluent relations, their composition is not confluent in general, but it is if the two relations commute.

Definition 14 (Commutation) *Let R_1 and R_2 be two binary relations over a set $\mathcal{T}$.*

- *The relations R_1 and R_2 commute if*

$$\twoheadleftarrow_{R_1} \cdot \twoheadrightarrow_{R_2} \subseteq \twoheadrightarrow_{R_2} \cdot \twoheadleftarrow_{R_1}$$

- *The relations R_1 and R_2 strongly commute if*

$$\leftarrow_{R_1} \cdot \rightarrow_{R_2} \subseteq \rightarrow_{R_2}^{0/1} \cdot \twoheadleftarrow_{R_1}$$

Lemma 1 (Hindley-Rosen [Ros73]) *Let R_1 and R_2 be two binary relations over a set $\mathcal{T}$. If R_1 and R_2 are confluent and commute, then $R_1 \cup R_2$ is confluent.*

In practice, proofs can be simplified using the following results.

Lemma 2 (Hindley [Hin64]) *Two strongly commuting relations commute.*

Lemma 3 (Staples [Sta75]) *Let R_1 and R_2 be two binary relations over a set $\mathcal{T}$. If R_1 and R_2 verify the following property $\leftarrow_{R_1} \cdot \twoheadrightarrow_{R_2} \subseteq \twoheadrightarrow_{R_2} \cdot \twoheadleftarrow_{R_1}$, then R_1 and R_2 commute.*

1.3 Term rewrite systems

Term rewrite systems are defined as transformation rules working over trees labelled by variables and function symbols. This simple idea is very powerful: term rewriting system can be regarded as an abstract model of computation that can express different programming paradigms (functional, logical, parallel, ...). Transformations are performed using the notion of rewrite rule.

Definition 15 (Rewrite rule) *Consider the term algebra $\mathcal{T}(\mathcal{F}, \mathcal{X})$. A* rewrite rule *for $\mathcal{T}(\mathcal{F}, \mathcal{X})$ is an oriented pair of terms, denoted $l \to r$, where l and r are terms in $\mathcal{T}(\mathcal{F}, \mathcal{X})$, called respectively* left-hand side *and* right-hand side *of the rule.*

The set of variables of a rule $l \to r$, denoted $\mathcal{V}ar(l,r)$, is defined by $\mathcal{V}ar(l) \cup \mathcal{V}ar(r)$. Usually some restrictions are imposed on a rewrite rule $l \to r$:

- $l \notin \mathcal{X}$ (the left-hand side is not a variable), and
- $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l)$ (the set of variables of the right-hand side is a subset of the set of variables of the left-hand side).

In this case, $\mathcal{V}ar(l,r)$ is equal to $\mathcal{V}ar(l)$. A rewrite rule is called *left-linear* if its left-hand side is linear.

Definition 16 (Term rewrite system) *Given a signature $\Sigma = (\mathcal{F}, \mathcal{X})$, a* term rewrite system *is a set R of rewrite rules for $\mathcal{T}(\mathcal{F}, \mathcal{X})$.*

A Term rewrite system is left-linear if all its rewrite rule are left-linear.

Example 3 *The following rewrite system defines the well-known Ackermann's function on Peano integers.*

$$\begin{aligned} ack(0,y) &\to succ(y) \\ ack(succ(x),0) &\to ack(x, succ(0)) \\ ack(succ(x),succ(y)) &\to ack(x, ack(succ(x), y)). \end{aligned}$$

Rewriting is performed by applying (at the meta level) a rewrite rule to a term.

Definition 17 (Rewrite step) *Given a rewrite system R over $\mathcal{T}(\mathcal{F}, \mathcal{X})$. A term t* rewrites *to a term t', denoted by $t \mapsto_R t'$, if there exists a rule $l \to r$ of R, a position ω in t, a substitution σ, satisfying $t_{|\omega} = \sigma(l)$, such that $t' = t_{\lceil \sigma(r) \rceil_\omega}$.*

Note that, by definition, the relation induced by the set of rewrite rules is stable by context.

When either the rule, the substitution and/or the position need to be precised, a rewriting step is denoted by $t \mapsto^{\omega,\sigma}_{l\to r} t'$.

The subterm $t_{|\omega}$ where the rewriting step is applied is called the *redex*. A term that has no redex is irreducible for R or in R-normal form.

Definition 18 (Rewrite reduction) *A* rewrite reduction*, or derivation, is any sequence of rewriting steps*

$$t_1 \mapsto_R t_2 \mapsto_R \dots$$

The rewrite relation *$\mapsto\!\!\!\!\!\rightarrow_R$ is defined on terms by $t \mapsto\!\!\!\!\!\rightarrow_R t'$ if there exists a rewrite reduction from t to t'. The number of steps can be precised writing $\mapsto\!\!\!\!\!\rightarrow_R^n$ for a derivation of length n, or $\mapsto_R^{0/1}$ for a derivation of zero or one step.*

Example 4 (Addition) *We show in this example how the operation of addition can be modeled on the set of Peano integers by means of a set of rewrite rules. Consider the set of symbols $\{s, 0\}$ of Example 1. We add to this set a binary symbol $+$ which denotes addition and we consider the signature $\Sigma = (\mathcal{F} = \{+, s, 0\}, \mathcal{X})$. The addition over the set of natural numbers $\mathcal{T}(\mathcal{F}, \mathcal{X})$ can then be defined by the rewrite system $R = \{s(x) + y \to s(x + y), 0 + x \to x\}$.*

Let us show a reduction in this rewrite system. We consider the addition $1 + 1$ on integers represented by the term $t_0 = s(0)+s(0)$ and we want to reduce it to its result 2. In order to do that,

we apply the first rule of R at the head position of t_0 using the substitution $\sigma = \{x/0, y/s(0)\}$, since we have $t_{0|\epsilon} = \sigma(s(x) + y)$. We obtain the term

$$t_0 \mapsto_R t_1 = t_{0\lceil\sigma(s(x+y))\rceil_\epsilon} = s(0 + s(0))$$

To reach the normal form of the term t_0, we apply then the second rule to the the subterm $0+s(0)$. We have the match $t_{1|1} = \sigma(0 + x)$ using $\sigma = \{x/s(0)\}$. We obtain thus the final result

$$t_1 \mapsto_R t_2 = t_{1\lceil\sigma(s(x))\rceil_1} = s(s(0))$$

The complete reduction of the term t_0 is therefore the following:

$$t_0 = s(0) + s(0) \mapsto_R s(0 + s(0)) \mapsto_R s(s(0)) = t_2$$

where the term t_2 is indeed the representation of the integer 2.

The more direct way to test joinability, that is to verify if t and t' reduce to the same term, is to reduce t and t' in normal form and then check their equality. Unfortunately this is not always possible, since the normal form may not exist, or may not be unique. In these cases, we talk formally about (non) termination and (non) confluence of the reduction relation.

Example 5 *Consider the rewrite system R given by the following rules*

$$\begin{aligned} f(x, g(x)) &\rightarrow c \\ f(x, x) &\rightarrow b \\ a &\rightarrow g(a) \end{aligned}$$

The rewrite relation R is not terminating, since $a \mapsto_R g(a) \mapsto_R g(g(a) \mapsto_R \ldots$ and it is not confluent, since $f(a, a) \mapsto_R b$ and $f(a, a) \mapsto_R f(a, g(a)) \mapsto_R c$.

1.3.1 Confluence

Proving the confluence of a rewrite relation is in general undecidable. In practice, if the relation is stronlgy normalising, confluence is checked locally. Local confluence can be proved by analysis of the *critical pairs* [KB70], that is the pairs obtained from the rewrite rules whose left-hand sides are overlapping.

Definition 19 (Critical pair) *Let $l_i \rightarrow r_i$, $i = 1, 2$, be two rules whose variables have been renamed such that $\mathcal{V}ar(l_1, r_1) \cap \mathcal{V}ar(l_2, r_2) = \emptyset$. Let $\omega \in \mathcal{P}os(l_1)$ be such that $l_1|_\omega$ is not a variable. We call a* critical pair *the pair $(\sigma(r_1), \sigma(l_1)_{\lceil\sigma(r_2)\rceil_\omega}$ where σ is the most general unifier of $l_1|_\omega =^? l_2$.*

If all critical pairs are joinable, then the rewrite system is locally confluent.

Lemma 4 (Critical pair lemma [Hue80]) *A term rewrite system is locally confluent if and only if all its critical pairs are joinable.*

Since the number of critical pairs in a finite rewrite system is finite, local confluence is decidable.

Example 6 *Consider a rewrite system over $\mathcal{T}(\mathcal{F},\mathcal{X})$ given by the oriented version of the group axioms*

$$\begin{array}{lrcl} i) & (x*y)*w & \to & x*(y*w) \\ ii) & e*z & \to & z \\ iii) & i(z)*z & \to & e \end{array}$$

where $*$ is in $\mathcal{F}^2$, i is in $\mathcal{F}^1$, e is in $\mathcal{F}^0$ and x,y,z,w are in $\mathcal{X}$. The critical pairs of the systems are the following:

1. *rule i) with itself: $\sigma = \{x/(x'*y), y/z'\}$ gives the pair $((x'*y')*(z'*z),((x'*(y'*(z'))*z)$*
2. *rule i) with rule ii): $\sigma = \{x/e, y/z\}$ gives the pair $((e*z)*w, z*w)$*
3. *rule i) with rule iii): $\sigma = \{x/i(z), y/z\}$ gives the pair $(i(z)*(z*w), e*w)$*

The first two pairs are joinable, but the third is not. Thus the system is not locally confluent.

1.3.2 Termination

Just as for confluence, the termination of a relation is in general undecidable. Some existing methods, two of which will be considered in this section, can help to pursue this issue. The first method is based on reduction orderings, while the second one is based on simplification orderings. The presentation of this section is based on the book by C. and H. Kirchner [KK99].

We start by defining the notion of reduction ordering on terms, that can be used for proving the strong normalisation of a rewrite system.

Definition 20 (Reduction ordering) *A* reduction ordering $>$ *over a set of terms $\mathcal{T}(\mathcal{F},\mathcal{X})$ is a well-founded ordering closed under context and substitution, that is such that for any context $\mathsf{Ctx}\{\Box\}$ and any substitution σ, if $t > s$ then $\mathsf{Ctx}\{t\} > \mathsf{Ctx}\{s\}$ and $\sigma(t) > \sigma(s)$.*

By well-foundness, in a reduction ordering $>$ a term is never smaller than one of its subterms. Using a reduction ordering, termination of rewriting can be proved by comparing left and right-hand sides of rules.

Theorem 2 *A rewrite system R over the set of terms $\mathcal{T}(\mathcal{F},\mathcal{X})$ is terminating iff there exists a reduction ordering $>$ such that each rule $l \to r \in R$ satisfies $l > r$.*

It is often convenient to build a reduction ordering by *interpretation* using an homomorphism τ from ground terms to an algebra $\mathcal{A}$ with a well-founded ordering $>$. To compare terms with variables, variables are added to $\mathcal{A}$ producing $\mathcal{A}(\mathcal{X})$ and variables in $\mathcal{X}$ are mapped to distinct variables in $\mathcal{A}(\mathcal{X})$. For a more detailed definition and some examples see [KK99]. In practice, the algebra of natural numbers is usually chosen, with the natural ordering and polynomial or exponential interpretations.

This method is quite powerful but in practice the main difficulty clearly comes from the choice of the interpretation. Some heuristics and methods can be found in [BCL87].

The following example uses as algebra $\mathcal{A}$ the natural numbers with the usual ordering $>$, together with exponential interpretations and polynomial interpretations [Lan75, Lan79].

Example 7 *Consider the one-rule system* $i(f(x,y)) \to f(f(i(x),y),y)$ *and the following interpretation* $\mathcal{I}$ *of* i *and* f *respectively by the square and the sum functions on positive natural numbers:*

$$\begin{aligned} \mathcal{I}(i(x)) &= \mathcal{I}(x)^2 \\ \mathcal{I}(f(x,y)) &= \mathcal{I}(x) + \mathcal{I}(y) \end{aligned}$$

In addition, let $\mathcal{I}(x) = x$ *and* $\mathcal{I}(y) = y$.

The monotonicity condition $a > b$ implies $\mathcal{I}(a) > \mathcal{I}(b)$ *is clearly satisfied, since each function is increasing on natural numbers. Now,*

$$\begin{aligned} \mathcal{I}(i(f(x,y))) &= (x+y)^2 = x^2 + y^2 + 2xy \\ \mathcal{I}(f(f(i(x),y),y)) &= x^2 + 2y. \end{aligned}$$

and for any assignment of positive natural numbers n *and* m *to the variables* x *and* y, $n^2 + m^2 + 2nm > n^2 + 2m$. *So this one-rule system is terminating.*

Another approach consists in using a *simplification ordering*, which is a reflexive, transitive and term-closed relation $\geq$ on terms, that contains the subterm strict ordering.

Simplification orderings can be built from a well-founded ordering on the function symbols $\mathcal{F}$ called a *precedence*. A powerful example is the following *multiset path ordering* also called *recursive path ordering*:

Definition 21 (Multiset path ordering) *Let* $>_{\mathcal{F}}$ *be a precedence on* $\mathcal{F}$. *The* multiset path ordering $>_{mpo}$ *is defined on ground terms by* $s = f(s_1,..,s_n) >_{mpo} t = g(t_1,\ldots,t_m)$ *if at least one of the following conditions holds:*

1. $f = g$ *and* $\{s_1,\ldots,s_n\} >_{mpo} \{t_1,\ldots,t_m\}$
2. $f >_{\mathcal{F}} g$ *and* $\forall j \in \{1,\ldots,m\}, s >_{mpo} t_j$
3. $\exists i \in \{1,\ldots,n\}$ *such that either* $s_i >_{mpo} t$ *or* $s_i \sim t$
   where $\sim$ *means equivalent up to permutation of subterms.*

Example 8 *Let* $>$ *be the natural ordering on* $\mathbb{N}$. *For example we have*

$$\{3,3,4,0\} >_{mpo} \{3,2,2,1,1,1,4,0\} >_{mpo} \{3,4\} >_{mpo} \{3,3,3,3,2,2\} >_{mpo} \{\ \}$$

Instead of comparing the multisets of subterms, we can compare terms lexicographically.

Definition 22 (Lexicographic path ordering) *Let* $>_{\mathcal{F}}$ *be a precedence on* $\mathcal{F}$. *The* lexicographic path ordering $>_{lpo}$ *is defined on terms by* $s = f(s_1,..,s_n) >_{lpo} t = g(t_1,\ldots,t_m)$ *if one at least of the following condition holds:*

1. $f = g$ *and* $(s_1,\ldots,s_n) >_{lpo} (t_1,\ldots,t_m)$ *and* $\forall j \in \{1,\ldots,m\}, s >_{lpo} t_j$
2. $f >_{\mathcal{F}} g$ *and* $\forall j \in \{1,\ldots,m\}, s >_{lpo} t_j$
3. $\exists i \in \{1,\ldots,n\}$ *such that either* $s_i >_{lpo} t$ *or* $s_i = t$.

Both the multiset path ordering and the lexicographic path ordering are simplification ordering, as proved in [Der82] and [KL82] respectively.

Example 9 *Consider the rewrite system defining the Ackermann's function on natural numbers as defined in Example 3. Assuming $ack >_{\mathcal{F}} succ$, we can show that:*

$$\begin{aligned} ack(0,y) &>_{lpo} succ(y) \\ ack(succ(x),0) &>_{lpo} ack(x,succ(0)) \\ ack(succ(x),succ(y)) &>_{lpo} ack(x,ack(succ(x),y)). \end{aligned}$$

We conclude therefore that the Ackermann's function is terminating.

1.4 Rewriting modulo equational theories

A rewrite system can be obtained starting from a set of equations $E = \{l_i = r_i, i = 1 \dots n\}$ and orienting each equation, for example $R = \{l_i \to r_i, i = 1 \dots n\}$. Choosing between one orientation or the converse is not automatic, since the resulting rewrite rules have to respect the condition of being well-formed, *i.e.* l_i can not be a variable, the variables of r_i must be a subset of those of l_i, *etc.* There exists equalities, like for example commutativity, that can not be added to a rewrite system without loosingx termination. A solution to this problem is to quotient the set of terms by the congruence generated by these equalities, called axioms, and to rewrite on equivalence classes.

1.4.1 Equational theories

Definition 23 (Axiom) *A pair of terms (l, r) is called* equality*,* axiom*, or* equation *according to the context, and it is denoted $(l = r)$.*

Definition 24 (Equational theory) *Given a signature $\Sigma = (\mathcal{F}, \mathcal{X})$ and a set of axioms E over $\mathcal{T}(\mathcal{F}, \mathcal{X})$. We define the* equational theory generated by *E, or E-equality, denoted $=_E$, the smallest congruence on $\mathcal{T}(\mathcal{F}, \mathcal{X})$ containing all the equalities $\sigma(l) = \sigma(r)$, where $(l = r)$ is an axiom in E and σ any substitution.*

The E-equality can be also obtained by the equal by equal replacing, as described next.

Definition 25 *Given a set of axioms E, we denote by $\sim_E^1$ the symmetric binary relation over $\mathcal{T}(\mathcal{F}, \mathcal{X})$ defined by $s \sim_E^1 t$ if there exists an axiom $(l = r)$ in E, a position $\omega \in \mathcal{P}os(s)$ and a substitution σ such that $s|_\omega = \sigma(l)$ and $t = s_{\lceil \sigma(r) \rceil_\omega}$. The reflexive and transitive closure of $\sim_E^1$ is denoted $\sim_E$.*

Theorem 3 (Birkhoff [Bir35]) *$s =_E t$ if and only if $s \sim_E t$*

By abuse of language and notation, we often do not make the difference between the equational theory $\sim_E$ and the set of axioms E.

The notions of matching and unification can be generalised to consider terms equivalent modulo an equational theory. Well-known examples are matching and unification modulo associativity and commutativity.

Definition 26 (Unification and matching modulo E)

- *A substitution σ is solution of the* E-unification equation $t_1 =^?_E t_2$ *if* $\sigma(t_1) =_E \sigma(t_2)$. *The substitution σ is called a* E-unifier. *The set of all the solutions is denoted* $\mathcal{S}ol(t_1 =^?_E t_2)$. *A* E-unifier σ *is more general modulo* E *than a* E-unifier σ' *over the set of variables* $X = \mathcal{V}ar(t_1) \cup \mathcal{V}ar(t_2)$, *denoted* $\sigma \leq_E \sigma'$ *if there is a substitution* τ *such that* $\sigma'(x) =_E \tau(\sigma(x))$ *for all* $x \in X$.

- *A substitution σ is solution of the* E-match equation $t_1 \ll_E t_2$ *if* $\sigma(t_1) =_E t_2$.

In general matching and unification modulo an equational theory are undecidable. To avoid the computation of redundant and irrelevant solutions, the set Σ of solutions should be as small as possible and complete, *i.e.* for each solution θ there should be a solution σ in Σ such that σ is equal to θ under E or can be instantiated to a substitution which is equal to θ under E.

Definition 27 (Complete set of solutions) *A set of substitutions Σ is a complete set of solutions of a E-unification equation $\mathcal{E} = t_1 =^?_E t_2$ when it is:*

correct $\Sigma \subseteq \mathcal{S}ol(t_1 =^?_E t_2)$,

complete $\forall \theta \in \mathcal{S}ol(t_1 =^?_E t_2), \exists \sigma \in \Sigma$ *such that* $\sigma \leq_E \theta$ *over* $\mathcal{V}ar(t_1) \cup \mathcal{V}ar(t_2)$.

It is called minimal is furthermore we have:

minimal $\forall \sigma_1, \sigma_2 \in \Sigma$, *such that* $\sigma_1 \leq_E \sigma_2$ *implies* $\sigma_1 = \sigma_2$ *over* $\mathcal{V}ar(t_1) \cup \mathcal{V}ar(t_2)$.

The substitution σ is a most general E-unifier *of $\mathcal{E}$ iff $\{\sigma\}$ is a minimal complete set of solutions of $\mathcal{E}$.*

Unfortunately, minimal complete sets of solutions do not always exists [FG86]. A unification or matching problem is said *unitary* (resp. finitary), when it has a complete set of solutions reduced to at most one (resp. a finite number of) element. Typical examples are matching in the empty theory which is unitary and matching in commutative theories which is finitary.

We will define in Section 2.1 a different notion of matching, called higher-order matching, performed modulo a congruence which is not generated by a set of axioms. For a more detailed presentation of general matching and unification see [BS01, JK91].

Example 10 (Equational matching)

- *The matching problem $f(x,y) \ll_C f(a,b)$ where C is the commutative axiom $f(x,y) = f(y,x)$ for all x, y has two solutions: $\sigma_1 = \{x/a, y/b\}$ and $\sigma_2 = \{x/b, y/a\}$.*

- *The matching problem $f(x,y) \ll_{AC} f(f(a,b),c)$ where AC is the set composed by the previous commutative axiom and the associative axiom A defined as $f(f(x,y),z) = f(x,f(y,z))$ for all x, y, z, has eight solutions, some of them being AC-equivalent.*

$$\begin{array}{lll} \sigma_1 = \{x/f(a,b), y/c\} & \sigma_3 = \{x/c, y/f(a,b)\} & \sigma_7 = \{x/f(b,c), y/a\} \\ \sigma_2 = \{x/f(b,a), y/c\} & \sigma_4 = \{x/c, y/f(b,a)\} & \sigma_8 = \{x/f(c,b), y/a\} \\ \sigma_5 = \{x/a, y/f(b,c)\} & \sigma_6 = \{x/a, y/f(c,b)\} & \end{array}$$

1.4.2 Class rewriting and rewriting modulo E

The rewrite relation defined below applies to a term if there exists a term in the same equivalence class that is reducible with a rewrite rule of the rewrite system R, in other words the rewrite rules perform computations on equivalence classes of terms over $\sim_E$.

Definition 28 (E-Class rewriting) *Given a set of rewrite rules R and a set of equations E over a set of terms $\mathcal{T}$. A term t_1 E-class rewrites to a term t_2, denoted $t_1 \mapsto_{R/E} t_2$ iff there exists a rewrite rule $l \to r \in R$, a context $\mathsf{Ctx}\{\Box\}$ and a substitution σ such that $t_1 \sim_E \mathsf{Ctx}\{\sigma(l)\}$ and $t_2 \sim_E \mathsf{Ctx}\{\sigma(r)\}$.*

This approach is rather general but not very practical from an operational point of view, since one needs to explore the entire (possibly infinite) class looking for reducible terms. Possible refinements of these reduction relation have been proposed, as for example Huet's approach [Hue80] which uses standard rewriting but is restricted to left-linear rules, Peterson and Stickel's approach [PS81] which uses *rewriting modulo E*, and Jouannaud and Kirchner's method [JK84] which mixes advantages of the two first methods.

Several abstract properties are common to these relations and are formalised in the next section for a general relation $\mapsto_S$ that may be any of these relations. We first define more precisely the Peterson and Stickel's relation, which is the most commonly used rewrite relation for class rewrite systems.

Definition 29 (Rewriting modulo E) *Given a set of rewrite rules R and a set of equations E over a set of terms $\mathcal{T}$. A term t_1 rewrites modulo E to a term t_2, denoted $t_1 \mapsto_{R,E} t_2$ iff there exists a rewrite rule $l \to r \in R$ and a substitution σ such that $t_1 = \mathsf{Ctx}\{t\}$ with $t \sim_E \sigma(l)$ and $t_2 = \mathsf{Ctx}\{\sigma(r)\}$.*

Using this notion of reduction, the rules apply on terms rather than on equivalence classes and matching modulo E is performed at each step of the reduction.

Note that in this relation t_1 is concerned with the E-equality steps only in its subterm t and thus $\to_{R,E}$ is clearly included in $\to_{R/E}$.

1.4.3 Properties

In all this section, given a rewrite relation R, following [JK84] we will simply write S for denoting any relation satisfying $\mapsto_R \subseteq \mapsto_S \subseteq \mapsto_{R/E}$. Moreover, we will write $\sim$ for denoting the congruence relation generated from a set of equations E. The definitions of the classical properties of term rewrite systems generalised to rewriting modulo an equational theory take then the following form:

Definition 30 (Normalisation, confluence) *Let R be a rewrite relation, E be a set of equations and S be any rewrite relation such that $\mapsto_R \subseteq \mapsto_S \subseteq \mapsto_{R/E}$.*

- *The rewrite relation R is* strongly normalising modulo E*, denoted $SN_\sim(R)$, if there exists no infinite $\sim \mapsto_R \sim$ reduction sequence, i.e. if the rewrite relation R/E is strongly normalising.*
- *The rewrite relation S has the* diamond property modulo E*, denoted $D_\sim(S)$, if*

$$\leftarrow_S \cdot \mapsto_S \subseteq \mapsto_S \cdot \sim_E \cdot \leftarrow_S$$

- *The rewrite relation S is* locally confluent modulo E*, denoted $LCON_{\sim}(S)$ if*

$$\leftarrow_S \cdot \mapsto_S \subseteq \ \mapsto\!\!\!\!\!\rightarrow_S \cdot \sim_E \cdot \leftarrow\!\!\!\!\!\!\!\!\rightarrow_S$$

- *The rewrite relation S is* confluent modulo E*, denoted $CON_{\sim}(S)$ if*

$$\leftarrow\!\!\!\!\!\!\!\!\rightarrow_S \cdot \mapsto\!\!\!\!\!\rightarrow_S \ \subseteq \ \mapsto\!\!\!\!\!\rightarrow_S \cdot \sim_E \cdot \leftarrow\!\!\!\!\!\!\!\!\rightarrow_S$$

- *The rewrite relation R is* Church-Rosser modulo E*, denoted $CR_{\sim}(R)$ if*

$$\leftrightarrow\!\!\!\!\!\!\!\!\rightarrow_{R \cup E} \ \subseteq \ \mapsto\!\!\!\!\!\rightarrow_S \cdot \sim_E \cdot \leftarrow\!\!\!\!\!\!\!\!\rightarrow_S$$

Some additional properties describe the well-behaving of a relation *w.r.t.* the set of equalities E and the commutativity property modulo E for two relations S_1 and S_2.

Definition 31 (Coherence, Commutativity) *Let R be a rewrite relation, E be a set of equations and S be any rewrite relation such that $\mapsto_R \subseteq \mapsto_S \subseteq \mapsto_{R/E}$.*

- *The rewrite relation S is* compatible with E*, denoted $CPB_{\sim}(S)$, if*

$$\leftarrow_S \cdot \sim_E \subseteq \sim_E \cdot \leftarrow_S$$

- *The rewrite relation S is* locally coherent with E*, denoted $LCH_{\sim}(S)$, if*

$$\leftarrow_S \cdot \sim_E \subseteq \ \mapsto\!\!\!\!\!\rightarrow_S \cdot \sim_E \cdot \leftarrow\!\!\!\!\!\!\!\!\rightarrow_S$$

- *The rewrite relation S is* coherent with E*, denoted $CH_{\sim}(S)$, if*

$$\leftarrow\!\!\!\!\!\!\!\!\rightarrow_S \cdot \sim_E \subseteq \ \mapsto\!\!\!\!\!\rightarrow_S \cdot \sim_E \cdot \leftarrow\!\!\!\!\!\!\!\!\rightarrow_S$$

- *The rewrite relations S_1 and S_2* commute modulo E*, denoted $COM_{\sim}(S_1, S_2)$, if*

$$\leftarrow\!\!\!\!\!\!\!\!\rightarrow_{S_1} \cdot \mapsto\!\!\!\!\!\rightarrow_{S_2} \subseteq \mapsto\!\!\!\!\!\rightarrow_{S_2} \cdot \sim_E \cdot \leftarrow\!\!\!\!\!\!\!\!\rightarrow_{S_1}$$

- *The rewrite relations S_1 and S_2* strongly commute modulo E*, denoted $SCOM_{\sim}(S_1, S_2)$, if*

$$\leftarrow_{S_1} \cdot \mapsto_{S_2} \subseteq \mapsto_{S_2}^{0/1} \cdot \sim_E \cdot \leftarrow\!\!\!\!\!\!\!\!\rightarrow_{S_1}$$

Diagrams in Figure 1.2 give a graphical representation of some of the properties defined above. Note that the compatibility property is stronger than the coherence and local coherence properties. The following propositions, proved in [Ohl98], generalises Lemma 1 and Lemma 1 to rewrite relations working modulo the E-equality.

Proposition 1 (Generalisation of the Newman's lemma) *([Ohl98])*
A strongly normalising relation S is confluent modulo E if it is locally confluent modulo E and locally coherent with E.

Proposition 2 (Generalisation of the Hyndley-Rosen's lemma) *([Ohl98])*
If S_1 and S_2 are confluent modulo E, commute modulo E and are both compatible with E, then the relation $S_1 \cup S_2$ is confluent modulo E.

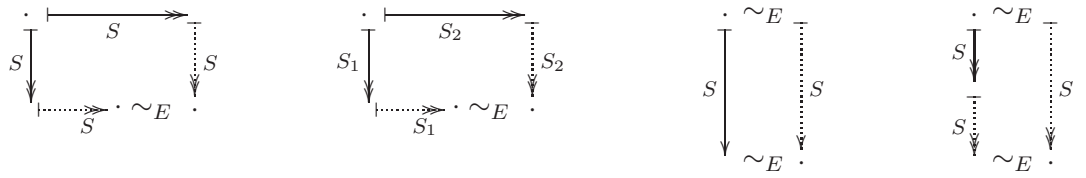


Figure 1.2: Properties of rewriting modulo $\sim_E$

Notice that, in order to have confluence modulo a set of equations E, a property of "well-behaviour" of the rewrite system *w.r.t.* the congruence relation generated by E is always needed.

Similarly, we have proved the generalisation of Lemmata 2 and 3 about commutation.

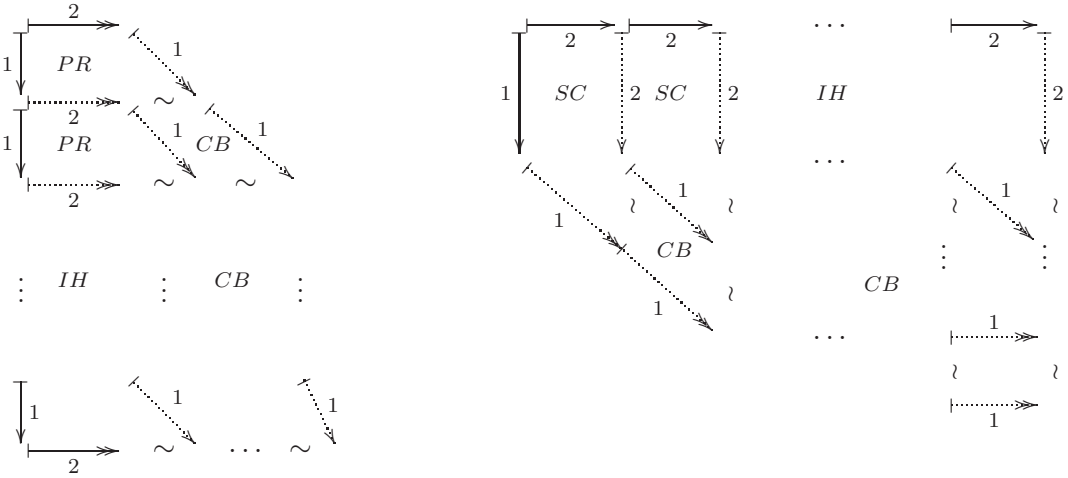
Proposition 3 *If S_1 and S_2 verify the property $\leftarrowtail_{S_1} \cdot \twoheadrightarrow_{S_2} \subseteq \twoheadrightarrow_{S_2} \cdot \sim_E \cdot \twoheadleftarrow_{S_1}$, called $PR_\sim(S_1, S_2)$ for short, and are compatible with E, then S_1 and S_2 commute modulo E.*

Proof : By induction on the number of steps of S_1, as shown in Figure 1.3 a) where 1 and 2 denote S_1 and S_2 respectively, *CB* stands for compatibility and *IH* for induction hypothesis. □

Proposition 4 *Two strongly commuting relation S_1 and S_2 compatible with E commute modulo E.*

Proof : We show that $SCOM_\sim(S_1, S_2)$ implies $PR_\sim(S_1, S_2)$ (defined in Proposition 3) and we conclude by Proposition 3. We proceed by induction on the number of steps of S_2. See the diagram *b*) in Figure 1.3. where *SC* stands for strong commutation, *CB* for compatibility and *IH* for induction hypothesis. Notice that the diagram can be closed by the lower 1 dashed arrows since $CPB_\sim(S_1)$ holds by hypothesis. □

These propositions, together with the properties defined in this section, will be extensively used in Chapter 8 where the proof of confluence for a rewriting relation modulo a certain set of equation is developed.



a) *Proposition* 3 *b*) *Proposition* 4

Figure 1.3: Proof diagrams

Chapter 2

Higher-order term rewriting

First-order term rewriting presented in Chapter 1 is not powerful enough to describe easily functionality. For this purpose, a mechanism of abstraction on variables is introduced in a higher-order system called λ-calculus. This calculus, which is expressive enough to represent all computable functions, had a deep influence on the development of programming and specification languages. After the definition of the classical λ-calculus in Section 2.1, we briefly present the simply typed version of the calculus. A more detailed presentation can be found in [Bar84] and [HS86].

The second part of the chapter is dedicated to the combination of first-order term rewriting and λ-calculus. Many frameworks have been designed with a view to integrate these two formalisms, since on one hand first-order term rewriting is not expressive enough to represent easily functions as "first-class" citizens and on the other hand in the λ-calculus data structures, like integers and lists, need to be encoded and are thus more difficult to manipulate. This integration has been handled either by adding to λ-calculus algebraic features or by enriching first-order rewriting with higher-order capabilities. In the first case, we find the works on combination of λ-calculus with term rewriting [BFG97, Bla01, JO97], in the second case the works on *Combinatory Reduction Systems* (CRS), introduced by J.W.Klop [Klo80], [KvOvR93] and other higher-order rewriting systems [Wol93, NP98].

We will make a brief survey of some of these systems in Section 2.2, insisting more on CRS in Section 2.3.

2.1 The λ-calculus

The λ-calculus was created by Church [Chu41] with the intent of developing a general theory of functions extended with logical notions. In spite of its very simple syntax, the λ-calculus is as expressive as the Turing machines [Tur37, Bar92], and therefore it can be viewed as a paradigmatic programming language.

The set of λ-terms is built from the first-order terms using two operators: the binary abstraction operator $\lambda\_.\_$ and the binary application operator $(\_\ \_)$. For simplifying the readability of λ-terms, we assume that the application operator has greater priority than the abstraction operator.

Definition 32 (Syntax) *Given a set of variables $\mathcal{X}$ and a set of constants $\mathcal{K}$, the set of λ-terms, denoted $\Lambda^{\mathcal{K}}_{\mathcal{X}}$ or simply Λ, is defined as follows:*

$$\mathcal{T} \quad ::= \quad \mathcal{X} \mid \mathcal{K} \mid \lambda\mathcal{X}.\mathcal{T} \mid \mathcal{T}\ \mathcal{T}$$

The λ-*abstraction* is used for defining functions, in particular for abstracting the arguments of a function by variables. The *application*, together with the β-reduction defined later, is used for applying a function to its concrete arguments. The term t in $\lambda x.t$ is called the *body* of the λ-abstraction. We sometimes use the simplified notation $\lambda x_1 \dots x_n.t$ or $\lambda \overline{x}_n.t$ for the term $\lambda x_1 \dots \lambda x_n.t$.

If the set of constant $\mathcal{K}$ is empty, then the associated λ-calculus is called *pure*. Otherwise, it is called applied λ-calculus [HS86]. In an even more general syntax, function symbols are not simply constants but they are provided with a fixed arity, different from zero.

Example 11 (λ-terms)

- *The identity function* $id = \lambda x.x$.
- *The well known* λ*-term* $\Omega = (\lambda x.x\ x)\ (\lambda x.x\ x)$.
- *The fixed point combinator* $Y = \Big(\lambda xy.y\ (x\ x\ y)\Big)\ \Big(\lambda xy.y\ (x\ x\ y))\Big)$.

An occurrence of a variable x in a term t is *bound* if the variable appears in a sub-term of t of the form $\lambda x.u$. Otherwise the variable x is *free*.

Definition 33 (Free variables) *The set of* free variables *of a* λ*-term* t*, denoted* $\mathcal{FV}(t)$*, is recursively defined as follows:*

$$\begin{array}{lcllcl} \mathcal{FV}(x) & = & \{x\} & \mathcal{FV}(\lambda x.t) & = & \mathcal{FV}(t) \setminus \{x\} \\ \mathcal{FV}(f) & = & \emptyset & \mathcal{FV}(t_1\ t_2) & = & \mathcal{FV}(t_1) \cup \mathcal{FV}(t_2) \end{array}$$

First-order substitution, as defined in the first chapter, can yield to "dishonest" terms, like for example the term $(\lambda x.x\ y)\{y/x\} = \lambda x.x\ x$ where the free variable y becomes bound after its substitution by x. This can be avoided by using a suitable function, called α-conversion, which renames the bound variables of a term when needed.

Definition 34 (α-conversion) *Let* u *be a* λ*-term containing a sub-term* $\lambda x.t$ *and let* z *be a variable such that* $z \notin \mathcal{FV}(t)$*. The replacement of* $\lambda x.t$ *by* $\lambda z.t\{x/z\}$*, that is the* λ*-term* $\lambda z.t$ *where all the occurrences of the variable* x *in* t *are substituted by* z*, is called renaming of the bound variable* x *or* α-conversion*. Two terms* u, v *are* α*-equivalent, denoted* $u =_\alpha v$*, if* v *is obtained by applying to* u *a finite chain (possibly empty) of* α*-conversions.*

In the λ-calculus terms are always considered modulo α-conversion, that is to say that no distinction is made between two α-equivalent terms.

From the point of view of programming languages and their implementation, the bound variables represent the formal parameters in a procedure. An abstraction of the form $\lambda x.u$ represent a function whose value at a certain argument v is calculated by substituting v for x in u. This mechanism that instantiates the formal parameters with concrete arguments, *i.e.* λ-terms, is called β-reduction. We call *redex* a λ-term of the form $(\lambda x.t_1)\ t_2$ and we call $\omega \in \mathcal{P}os(t)$ a *redex position* in t if $t_{|\omega} = (\lambda x.t_1)\ t_2$.

In addition, since usually in mathematics we assume that two functions are equal if they give the same result for all arguments, the η-reduction identifies terms having the same applicative behaviour.

Definition 35 (β and η-reductions)

- *The relation between terms* $t_1 \mapsto_\beta t_2$ *(t_1 β-reduces in one step to t_2) is defined as the context closure of the relation generated by the β-rule:*

$$(\lambda x.u)\ v \to_\beta u\{x/v\}$$

- *The relation between terms $t_1 \mapsto_\eta t_2$ (t_1 η-reduces in one step to t_2) is defined as the context closure of the relation generated by the η-rule:*

$$(\lambda x.t\ x) \to_\eta t \quad \text{if } x \notin \mathcal{FV}(t)$$

A λ-term t is in β-normal form ($\beta\eta$-normal form respectively) if it contains no subterm $(\lambda x.u)\ v$ (and no subterm $\lambda x.t\ x$, with $x \notin \mathcal{FV}(t)$, respectively).

The union of the two relations is denoted by $\mapsto\!\!\!\!\!\rightarrow_{\beta\eta}$ and the generated relations are denoted $\mapsto\!\!\!\!\!\rightarrow_\beta, \mapsto\!\!\!\!\!\rightarrow_\eta, =_\beta, =_\eta, =_{\beta\eta}, \ldots$.

Example 12 *Consider the λ-term Y defined in Example 11. If we apply it to a term t we get*

$$\begin{array}{rcl} Y\ t & = & \Big(\lambda xy.y\ (x\ x\ y)\Big)\ \Big(\lambda xy.y\ (x\ x\ y))\Big)\ t \\ & \mapsto_\beta & \Big(\lambda y.y\ (\Big(\lambda xy.y\ (x\ x\ y))\Big)\ \Big(\lambda xy.y\ (x\ x\ y))\Big)\ y)\Big)\ t \\ & \mapsto_\beta & t\ (\Big(\Big(\lambda xy.y\ (x\ x\ y))\Big)\ \Big(\lambda xy.y\ (x\ x\ y))\Big)\ t) \quad = \quad t\ (Y\ t) \end{array}$$

which confirms the fact that Y behaves like a fixed point combinator, i.e. $Y\ t =_\beta t(Y\ t)$.

The λ-calculus provides a semantic framework for many programming languages like for example λProlog [MN86] or ML. In these cases, the notion of equality between λ-terms is defined modulo α-conversion and β-reduction. Unification, typically needed for theorem proving or for logic programming, is extended to higher-order unification [Hue75] which is undecidable from order two [Gol81].

Definition 36 (Higher-order unification and matching)

- *A substitution σ is solution of the* higher-order unification equation $t_1 =^?_{\beta\eta} t_2$ *if $\sigma(t_1) =_{\beta\eta} \sigma(t_2)$.*

- *A substitution σ is solution of the* higher-order match equation $t_1 \lessdot_{\beta\eta} t_2$ *if $\sigma(t_1) =_{\beta\eta} t_2$. We denote by Sol($t_1 \lessdot_{\beta\eta} t_2$) the set of all solutions of this higher-order match equation.*

A well-known decidable subclasses of this problem is obtained restricting the class of terms considered to λ-patterns.

Definition 37 (λ-pattern) *A λ-term t is called a* λ-pattern *if any of its free variables Z appears in a sub-term of t of the form $Z\ x_1 \ldots\ x_n$ where the variables $x_1, \ldots, x_n$, $n \geq 0$, are distinct and all bound in t.*

The first one to introduce the notion of λ-patterns and to define a unification algorithm for this class of terms was Miller in [Mil91]. This algorithm is defined in a simply typed setting (see Section 2.1.2), but it can be adapted also to a non-typed version of the λ-calculus. The algorithm determines whether or not solutions of a unification equation exist and characterises all of them, if they exist, by providing a unique, most general unifier (see Definition 8). Higher-order unification (and matching) problems on λ-patterns have thus been shown to be decidable, unitary [Mil91] and even linear [Qia93].

2.1.1 Properties

The λ-calculus is not strongly normalising. The classical example is the λ-term $\lambda x.(x\ x)$ applied to itself. Since a new redex is created at each step of reduction, this leads to an infinite derivation. On the other hand, the λ-calculus enjoys the confluence property.

Theorem 4 (Confluence [Hin78]) *The β-reduction, the η-reduction and the $\beta\eta$-reduction are confluent.*

There are two main methods for proving the confluence of the β-relation. The classical one uses the parallel version of the β-relation, which in one step reduces several redexes simultaneously. Another technique consists in using the notion of reduction called *complete development*. It turns out that the two methods are indeed equivalent, since the existence of a parallel reduction from t to t' implies the existence of a complete development from t to t' and vice versa.

We describe next in more detail the finite developments method since this proof technique will be used also in Chapter 8 for proving the confluence of the graph rewriting calculus.

The original proof using developments can be found in [Sch65]. Many other proofs have been proposed later, for example in [Hin78] and [Bar84]. An elegant version of the proof has been given in [Vri85] and an axiomatic proof that applies also to the λ-calculus can be found in [Mel96]. Some other proofs use an encoding of developments in a strongly normalising calculus, like in [Par90, Ghi96, vR96].

We follow here the proof lines of [Bar84]. The idea is to define a reduction relation $\mapsto_{Cpl}$ on λ-terms such that $\mapsto_\beta \subseteq \mapsto_{Cpl} \subseteq \mapsto\!\!\!\!\!\rightarrow_\beta$ and such that $\mapsto_{Cpl}$ satisfies the diamond property. One can define $\mapsto_{Cpl}$ as the relation that performs the complete reduction of a set of redexes chosen in the initial term (also called development).

Therefore, in order to define $\mapsto_{Cpl}$, we first define a version of the λ-calculus in which some redexes are underlined and then we make precise the notion of development.

Definition 38 *Given a set of variables $\mathcal{X}$ and a set of constants $\mathcal{K}$, let Λ' be the set of terms $\mathcal{T}$ defined as follows:*

$$\mathcal{T} \quad ::= \quad \mathcal{X} \mid \mathcal{K} \mid \lambda\mathcal{X}.\mathcal{T} \mid \mathcal{T}\ \mathcal{T} \mid (\underline{\lambda}\mathcal{X}.\mathcal{T})\ \mathcal{T}$$

Let $\underline{\beta}$ be the reduction rule defined as $(\underline{\lambda}x.u)v \rightarrow_{\underline{\beta}} u\{x/v\}$. We call $\underline{\lambda}$-calculus the calculus over Λ' associated to the reduction rule $\underline{\beta}$ and λ'-calculus the calculus over Λ' associated to the reduction rule $\beta' = \beta \cup \underline{\beta}$.

Let Δ be a set of redex positions in a λ-term t. If we are interested in following what happens to the redexes specified in Δ during the reduction, then we can lift t to $t' \in \Lambda'$ by underlying the redexes in Δ.

Lemma 5 (Lifting) *Let $I : \Lambda' \rightarrow \Lambda$ the function that forgets underlinings. Given a term $t' \in \Lambda'$ and a term $u \in \Lambda$, the β-reduction $I(t') \mapsto\!\!\!\!\!\rightarrow_\beta u$ can be lifted to the β'-reduction $t' \mapsto\!\!\!\!\!\rightarrow_{\beta'} u'$ with $I(u') = u$.*

We first will show that reductions in the λ'-calculus contracting only underlined redexes are always finite. For example the term $(\underline{\lambda}x.x\ x)\ (\lambda x.x\ x)$ in the λ'-calculus reduces in one step to $(\lambda x.x\ x)\ (\lambda x.x\ x)$ which is in normal form *w.r.t.* the rule $\underline{\beta}$. Note that the second $\lambda x.x\ x$ can not be underlined because it is not the first part of a redex, that is it is not applied to any term. We introduce next some notations and vocabulary.

Definition 39 *[Residual,Development]*

- *Given a term $t \in \Lambda$ and a set of redex positions Δ in t, we denote by $(t, \Delta) \in \Lambda'$ the term obtained from t by underlying the redex positions in Δ.*
- *Let $t, u \in \Lambda$, Δ be a set of redex positions in t and $t' = (t, \Delta) \in \Lambda'$. Let $\mathcal{R}$ be the β-reduction $t \mapsto\!\!\!\!\!\rightarrow_\beta u$ and $\mathcal{R}'$ be the β'-reduction $t' \mapsto\!\!\!\!\!\rightarrow_{\beta'} u'$ obtained by lifting the reduction $\mathcal{R}$. We call set of* residuals *of Δ relative to $\mathcal{R}$, denoted $\Delta/\mathcal{R}$, the (unique) set of redex positions Δ' in u such that $u' = (u, \Delta')$.*
- *Given a term $t_0 \in \Lambda$ and a set of redex positions Δ in t_0, a sequence of β-reductions $\mathcal{R}$: $t_0 \mapsto_\beta t_1 \mapsto_\beta \ldots$ is called a* development *of (t_0, Δ) if every redex reduced at the ith-step of $\mathcal{R}$ is a residual of a redex in Δ relative to the reduction from t_0 to t_{i-1}.*
- *Given a term $t \in \Lambda$ and a set of redex positions Δ in t, a sequence of β-reductions $\mathcal{R}$ is a* complete development *of (t, Δ) if $\mathcal{R}$ is a development such that $\Delta/\mathcal{R} = \emptyset$.*

Example 13 (Development) *Consider the term $t = id\ (\lambda y.y\ (id\ x))\ (xb) \in \Lambda$ where id denotes the identity $\lambda x.x$. We choose some redexes in t by underlying them. We obtain for example $t' = (t, \Delta) = id\ (\underline{\lambda} y.y(\underline{id}\ x))\ (xb) \in \Lambda'$ where $\underline{id}$ denotes the term $\underline{\lambda} x.x$. The set $\Delta = \{2, 2.1.2.2\}$ contains the occurrences of the two underlined redexes. We start reducing t'. For example we have*

$$t' = id\ (\underline{\lambda} y.y(\underline{id}\ x))\ (xb) \mapsto_{\underline{\beta}} id\ (\underline{\lambda} y.y\ x)\ (xb) = t'_1$$

After this first step, the set of residuals of Δ is $\Delta' = \{2\}$. Continuing the reduction $\mathcal{R}$ we obtain

$$t'_1 = id\ (\underline{\lambda} y.y\ x)\ (xb) \mapsto_{\underline{\beta}} id(xb\ x) = t'_2$$

Now $\Delta/\mathcal{R} = \emptyset$ and therefore the reduction is a complete development of the term t'. We could have chosen a different reduction

$$t' \mapsto_{\underline{\beta}} t'_1 \mapsto_\beta (\underline{\lambda} y.y\ x)\ (xb) \mapsto_{\underline{\beta}} (xb\ x) = t_2$$

In this case, the reduction is not a development of t' since the second redex reduced is not a residual of Δ'.

The following lemma specifies the correspondence between $\underline{\beta}$-reductions and developments.

Lemma 6 *Let $t \in \Lambda$ and Δ be a set of redex positions in t. A sequence of β-reductions $\mathcal{R}$ in the λ-calculus is a development of (t, Δ) if and only if its lifting $\mathcal{R}'$ in the λ'-calculus is a $\underline{\beta}$-reduction.*

It follows that proving that all developments are finite is equivalent to proving that the $\underline{\beta}$-relation is strongly normalising. In fact, the creation of "new" redexes makes β-reductions infinite, but if only (residuals of) old redexes are contracted, then a reduction always terminates. Moreover, we can prove that all complete developments terminate with the same term by proving that the $\underline{\beta}$-relation is confluent.

Lemma 7 *([Bar84]) $\underline{\beta}$ is strongly normalising and confluent.*

Proof : The proof of normalisation is done associating to any term a positive integer, called weight, that decreases during the reduction. The proof of the confluence property is done using the fact that strong normalisation and local confluence imply confluence. □

These properties of the $\underline{\beta}$-relation correspond in terms of developments to the following theorem.

Theorem 5 (Finite developments) *Let $t \in \Lambda$ and Δ be a set of redex positions in t.*

- *All developments of (t, Δ) are finite;*
- *All complete developments of (t, Δ) end with the same term.*

The previous theorem can be used for proving the confluence property for the β-relation. In order to do this, we define first a new reduction relation $\mapsto_{Cpl}$, that performs the complete development of a given set of redexes. We denote by $t' \downarrow_{\underline{\beta}}$ the (unique) $\underline{\beta}$-normal form of a term $t' \in \Lambda'$.

Definition 40 *Let t, u be two λ-terms. We define $t \mapsto_{Cpl} u$ if $u = t' \downarrow_{\underline{\beta}}$ with $t' = (t, \Delta)$ for some Δ.*

Theorem 6 *([Bar84]) The β-relation is confluent.*

Proof : By definition, the relation $\mapsto_{Cpl}$ has the same transitive closure than β and the diamond property for Cpl can be proved using Theorem 5. The diamond property of Cpl implies the diamond property of its transitive closure and thus the confluence of the β-relation.

□

2.1.2 Simply typed λ-calculus

The non termination of the λ-calculus corresponds, from a logical point of view, to the incoherence of the formal system. In fact, any term can be applied to any other without considering any notion of domain for functions.

The simply typed λ-calculus ($\lambda_{\rightarrow}$), is a more elaborated version of the λ-calculus that allows to avoid this kind of problems. The idea is to give to λ-terms representing functions some additional information, which is formalised in the notion of *type*. A function has type $\alpha_1 \rightarrow \alpha_2$ if it takes an argument in α_1 and returns an element of α_2. The typed λ-calculus has been proposed in two different ways: by Church, with types explicitly written for bound variables, and by Curry, with implicit types. In the following we will define the ($\lambda_{\rightarrow}$) *à la* Church.

Given a non empty set of basic types, also called atomic types, the type grammar is defined by

$$\alpha ::= \iota \mid \alpha_1 \rightarrow \alpha_2$$

where ι is an atomic type. Types of the form $\alpha_1 \rightarrow \alpha_2$ are called composed types and, as already mentioned, represent the set of functions from α_1 to α_2.

Not all terms defined by the λ-calculus syntax can be typed. We call them the *pre-terms*. Only well-typed (as defined by the typing rules below) pre-terms are called terms in the simply typed λ-calculus. The pre-terms grammar is defined as follows:

$$\mathcal{T} ::= \mathcal{X} \mid \lambda \mathcal{X} : \alpha . \mathcal{T} \mid \mathcal{T}\ \mathcal{T}$$

where α is a type.

A type *context* is a list of pairs (*variable* : *type of the variable*) such that any variable appears at most once in the list. A type *judgment* is a triple of the form $\Gamma \vdash t : \alpha$ meaning that t has type α under the typing hypothesis specified in Γ.

The *typing system* of the simply typed λ-calculus is given by the inference rules of Figure 2.1, that can be interpreted as follows: if the judgment(s) on the top line of a rule can be proved, then the judgment on the bottom line can also be proved.

$$(\text{Var})\frac{}{\Gamma, x : \alpha \vdash x : \alpha} \qquad (\text{Abs})\frac{\Gamma, x : \alpha_1 \vdash t : \alpha_2}{\Gamma \vdash \lambda x : \alpha_1 . t : \alpha_1 \to \alpha_2} \qquad (\text{App})\frac{\Gamma \vdash t : \alpha_1 \to \alpha_2 \quad \Gamma \vdash u : \alpha_1}{\Gamma \vdash t\ u : \alpha_2}$$

Figure 2.1: Typing rules of $\lambda_{\to}$

Given a term t in $\lambda_{\to}$, consider the term t' obtained from t erasing all the type symbols: we say that t' is a *typable* λ-term. Except for the type notation for bound variables, the definition of β-reduction and β-normal form does not change with respect to the λ-calculus. A useful notion on simply typed λ-term is the so-called long η-normal form. This special form has the effect that all subterms of a term are provided with the right number of arguments, according to their type.

Definition 41 (β-normal η-long form) *A simply typed λ-term t of type $\alpha_1 \to \ldots \to \alpha_n \to \iota$ is in β-normal η-long form (or long $\beta\eta$-normal form) if it is of the form $\lambda x_1 : \alpha_1 \ldots \lambda x_n : \alpha_n (a\ u_1 \ldots u_m)$ where the terms $u_1 \ldots u_m \in \lambda_{\to}$ are in β-normal η-long form.*

2.2 Higher-order rewrite systems

First-order term rewriting and λ-calculus present some interesting characteristics which are in some sense complementary. On the one hand, first-order term rewriting is not well-adapted to represent easily the composition of functions and functions as results of a procedure, and on the other hand the syntax of the λ-calculus often leads to non trivial encodings for algebraic data types. In order to enhance agility, several new rewrite formalisms combining the abstraction mechanism of the λ-calculus and the representation of abstract types with the syntax of first-order term rewriting, have been studied.

These systems can be grouped together under the name of *higher-order rewrite systems* and are essentially of two kinds: either they are defined as an extension of the λ-calculus with functional symbols, or they are defined adding to first-order rewrite systems an abstraction operator and a reduction relation similar to the β-reduction of the λ-calculus.

Algebraic extensions of the λ-calculus

The extension of λ-calculus with first order rewrite rules has started with the work of Breazu-Tannen [BT88] who have studied the confluence property for such a kind of combined systems. The normalisation property has been successively analysed by Breazu-Tannen and Gallier [GBT89] and Okada [Oka89].

In these extensions of λ-calculus, terms consist of classical λ-terms with the addition of first-order terms built from a given signature. The rewrite relation over terms is defined by the combination of the β-reduction and the reduction induced by the rewrite rules.

An example illustrating the advantages of this approach is the representation of natural numbers and the associated operation of addition. The direct encoding of integers in the λ-calculus is not intuitive at all: any natural number n is encoded in a λ-term $\lambda f x . f(f \ldots (fx) \ldots)$ with n occurrences of the symbol f and the symbol + encoded by the λ-term $\lambda np.(\lambda fx.mf(nfx))$. Using instead the algebraic representation of arithmetics given in Example 4, the integer n is represented by $s(\ldots(s(0))\ldots)$, with n occurrences of the symbol s, and the behavior of the symbol + is described by the rewrite rules of the first line below. The extension of the λ-calculus

with integer addition is thus defined by:

Terms	$t ::= x \mid \lambda x.t \mid t\ t \mid 0 \mid s(t) \mid t + t$
Reduction rules	$0 + u \to u \qquad s(t) + u \to s(t + u)$
	$(\lambda x.t)\ u \to_\beta t\{x/u\}$

It is clear that this approach is much more attractive than the direct encoding of integers in the λ-calculus. Data are much more easy to manipulate and, as a result, the new system is more efficient.

Term rewriting with abstraction

Starting from Klop's work on CRSs described in the next section, several other higher-order systems were proposed in the subsequent years. We mention next some of them.

The *Expression Reduction Systems* (ERSs) defined by Khasidashvili in [Kha90] are similar to CRSs. The differences *w.r.t.* the syntax of CRSs concern essentially the arity of metavariables, which are necessarily of arity zero in the ERSs, and an extra operator to deal explicitly with substitutions. More precisely, this operator defines an explicit environment where substitutions are represented, but they are not evaluated atomically, so ERSs is not a formalism with explicit substitutions.

This is the case instead for the *Explicit Reduction Systems* introduced by Pagano in [Pag97] that represent an algebraic extension of the λ-calculus with explicit substitutions. An extension of these systems with algebraic patterns and "choice" constructors, used to denote different possible structures allowed for an abstracted argument, is given by the *Explicit Reduction Systems with patterns* of J. Forest and D. Kesner [FK03].

A different approach to higher-order rewriting was taken by T. Nipkow in [Nip91] with the *Higher-order Rewrite Systems*, or HRSs, that use simply typed λ-calculus with β-reduction and η-expansion as a meta language. The HRSs are shown to have the same expressive power than CRSs in the work of Van Oostrom and Van Raamsdonk [VOVR93]. The main difference between the two systems is the meta language that is employed: untyped λ-calculus with developments for the CRSs, simply typed λ-calculus with η-expansion and reduction to normal form in the HRSs.

Wolfram defines in [Wol91] the *Higher-Order Term Rewriting Systems*. Like HRSs, they use simply typed λ-calculus as meta language but the rewrite rules are more general than those of the HRSs.

Asperti and Laneve define in [Lan93] and in [AL94] the *Interaction Systems* in order to extend the theory of optimal reductions of Levy [Lév78] in a more general setting than the λ-calculus. The Interaction Systems are in fact a subclass of CRSs and are quite similar to *ERS*.

A more general class of systems are the *Interaction Nets* introduced by Lafont in [Laf90], that are based on rewriting of networks rather than terms and have hence a graphical syntax. They provided Lamping [Lam90] the basis for a practical realisation of the partial sharing needed for Levy's optimal reductions in the λ-calculus.

For a uniform view and a comparison of some of these different higher-order formalisms, one can refer to the Ph.D. thesis of Van Raamsdonk [vR96].

2.3 Combinatory Reduction Systems

The *Combinatory Reduction Systems* (CRSs), introduced by J. W. Klop in 1980 [Klo80], were designed to combine first-order term rewriting with a mechanism of bound variables similar to the λ-abstraction. Our definitions of the main components of a CRS are based on the presentation of [KvOvR93].

2.3.1 Terms and metaterms

In the CRS syntax a distinction is made between the *metaterms*, built form metavariables of fixed arity and denoted with capital letters, and the classical *terms*.

Definition 42 (metaterms and terms) *Given a set of constants $\mathcal{F}$ and a set of metavariables $\mathcal{Z}$ of fixed arity, and a set of variables $\mathcal{X}$. The set of* CRS*-metaterms is defined as follows:*

$$\mathcal{MT} \quad ::= \quad \mathcal{X} \mid \mathcal{F}^n(\mathcal{MT}_1, \ldots, \mathcal{MT}_n) \mid \mathcal{Z}^n(\mathcal{MT}_1, \ldots, \mathcal{MT}_n) \mid [\mathcal{X}]\mathcal{MT}$$

where $\mathcal{F}^n \subset \mathcal{F}$ and $\mathcal{Z}^n \subset \mathcal{Z}$ are, respectively, the sets of functional symbols and metavariables of arity n.

The set $\mathcal{T}_{\mathcal{CRS}} \subset \mathcal{MT}$ of CRS*-terms is composed of all the metaterms without metavariables.*

We should point out that all metaterms are *well-formed*, *i.e.* the functional symbols and metavariables take exactly as many arguments as their arity. Comparing to first-order rewrite systems, in the syntax of a CRS we have two new concepts: the symbol $[\_]\_$ and the metavariables. The operator $[\_]\_$ denotes an abstraction similar to the λ-abstraction of the λ-calculus such that, in $[x]t$, the variable x is bound in t. In a metaterm of the form $[x]t$ we call t the scope of $[x]$. A variable x occurs *free* in a metaterm if it is not in the scope of an occurrence of $[x]$. A variable x occurs *bound* otherwise. The set of free variables of a metaterm A is written $\mathcal{FV}(A)$. As for the λ-calculus we work modulo α-conversion.

Metavariables (in the CRS rewrite rules defined below) behave as (free) variables of first-order rewrite systems. Metavariables cannot be bound by the abstraction operator but variables appearing as arguments of a metavariable can indeed be bound by this operator. For example, the only bound variable in $[x]Z(x)$ is x. The set of metavariables of a metaterm A is written $\mathcal{MV}(A)$.

A notion of context with one hole $\mathsf{Ctx}\{\Box\}$ similar to that defined in Section 1.1 is used for CRS-terms and metaterms.

Example 14 (Terms and Metaterms) *Some examples of terms and meta-terms:*

- $f([x]g(x,a)) \in \mathcal{T}_{\mathcal{CRS}}$ *with* $f \in \mathcal{F}^1$, $g \in \mathcal{F}^2$, $a \in \mathcal{F}^0$.
- $Z_1(Z_2) \in \mathcal{MT}$ *with* $Z_1 \in \mathcal{Z}^1$, $Z_2 \in \mathcal{Z}^0$.
- $f([x]Z(x,y)) \in \mathcal{MT}$ *with* $f \in \mathcal{F}^1$, $Z \in \mathcal{Z}^2$.

2.3.2 Substitution

The application of substitutions to metavariables is defined at the meta-level of the calculus and uses as meta-language the $\underline{\lambda}$-calculus, *i.e.* λ-calculus where the abstractions are underlined and reduction is performed by the $\underline{\beta}$-rule defined in Section 2.1.1. Unintended bindings of variables

by the $\underline{\lambda}$-abstractor operator are avoided using α-conversion. We should point out that a CRS-(meta)term is necessarily in $\underline{\beta}$-normal form by construction.

Performing a substitution in a CRS corresponds to applying an *assignment* (and consequently a set of substitutes) to a CRS-metaterm.

Definition 43 (Substitute) *An n-ary* substitute *is an expression of the form $\xi = \underline{\lambda}x_1 \dots x_n.u$ where $x_1, \dots, x_n$ are distinct variables and u is a* CRS*-term and its application to an n-tuple of* CRS*-terms $(t_1, \dots, t_n)$ yields the simultaneous substitution of $x_1, \dots, x_n$ by $t_1, \dots, t_n$ in u, denoted as follows: $(\underline{\lambda}x_1 \dots x_n.u)(t_1, \dots, t_n)\downarrow_{\underline{\beta}} = u\{x_1/t_1, \dots, x_n/t_n\}$.*

Definition 44 (Assignment) *An* assignment *$\sigma = \{(Z_1, \xi_1), \dots, (Z_n, \xi_n)\}$, is a finite set of pairs (metavariable, substitute) such that arity(Z_i) = arity(ξ_i) $\forall i \in \{1, \dots, n\}$. The domain of the assignment is $Dom(\sigma) = \{Z_1, \dots, Z_m\}$. The application of an assignment σ to a* CRS*-metaterm t, denoted $\sigma(t)$ or σt, is inductively defined by:*

$$\begin{array}{lcl@{\qquad}lcl} \sigma(x) & = & x & \sigma([x]t) & = & [x]\sigma(t) \\ \sigma(Z_i) & = & \xi_i \quad \text{if } (Z_i, \xi_i) \in \sigma & \sigma(f(t_1, \dots, t_n) & = & f(\sigma(t_1), \dots, \sigma(t_n)) \\ \sigma(Z_i) & = & Z_i \quad \text{if } Z_i \notin Dom(\sigma) & \sigma(Z_i(t_1, \dots, t_n)) & = & \sigma(Z_i)(\sigma(t_1), \dots, \sigma(t_n))\downarrow_{\underline{\beta}} \end{array}$$

Notice that since the assignments can instantiate only metavariables, they have no effect on variables. Since we work modulo the α-convention, unintended bindings of free variables are avoided by renaming bound variables; for example, if $\sigma(Z) = x$ then we can suppose that the variable y in $[y]Z$ is always different from x and thus $\sigma([y]Z) = [y]\sigma(Z)$.

The instantiation of a term is defined by replacing each metavariable by a substitute (*i.e.* a $\underline{\lambda}$-term) and by reducing all residuals of $\underline{\beta}$-redexes that are present in the initial term, *i.e.* performing a *complete development* (see Definition 39) on $\underline{\lambda}$-terms. Since in the λ-calculus all developments are finite (Theorem 5), the CRS substitution is well-defined. Note that the result of the application of an assignment to a metaterm is indeed a CRS-term.

2.3.3 Rules and rewritings

A CRS rewrite rule is a pair of metaterms. Their metavariables define the reduction schemes since they can be instantiated by any term. We consider as left-hand side of the rules only the CRS-metaterms satisfying the pattern definition:

Definition 45 (CRS-pattern) *A* CRS*-metaterm P is said to be a* CRS pattern *if any of its metavariables Z appears in a sub-metaterm of P of the form $Z(x_1, \dots, x_n)$ where the variables $x_1, \dots, x_n$, $n \geq 0$, are distinct and all bound in P.*

Moreover, the usual conditions used in first-order rewriting are imposed:

Definition 46 (Rewrite rules) *A set of* CRS *rewrite rules $\mathcal{R}$ consists of rules of the form $L \to R$ satisfying the following conditions:*

- *L and R are closed metaterms (they do not contain free variables);*
- *L has the form $f(A_1, \dots, A_n)$ with $A_1, \dots, A_n$ metaterms and $f \in \mathcal{F}_n$;*
- *$\mathcal{MV}(L) \supseteq \mathcal{MV}(R)$;*
- *L is a* CRS *pattern.*

The first three conditions are the ones already known from first-order rewriting: the first condition states that rules are built from metaterms; the second one specifies the structure of left-hand sides; the third one avoids the introduction of arbitrary terms. The last condition ensures the decidability and the uniqueness of the solution of the matching inherent to the application of the CRS rules. The same restrictions can be found in the Ph.D. thesis of Klop where he shows the confluence property for *regular* CRSs, *i.e.* CRSs were all rewrite rules are non-overlapping and whose left-hand sides are linear patterns.

Example 15 (β-rule in CRS) *The β-rule of λ-calculus $(\lambda x.t)u \to_\beta t\{x/u\}$ corresponds in CRS to the rewrite rule BetaCRS:*

$$App(Ab([x]Z(x)), Z_1) \to Z(Z_1)$$

where $App \in \mathcal{F}_2$ and $Ab \in \mathcal{F}_1$ are the encodings of the application operator and of the abstraction operator respectively.

The left-hand side and the right-hand side of a CRS rewrite rule are metaterms, but the rewrite relation induced by the rule is a relation on terms.

Definition 47 (Rewrite relation) *A CRS-term t rewrites to a CRS-term t' using the rewrite rule $L \to R$, denoted $t \to_{L\to R} t'$, if there exists an assignment σ and a context $\mathsf{Ctx}\{\Box\}$ such that $t = \mathsf{Ctx}\{\sigma(L)\}$ and $t' = \mathsf{Ctx}\{\sigma(R)\}$. The rewrite relation induced by a set of CRS rewrite rules $\mathcal{R}$ is the transitive closure of the union of the rewrite relations $\to_{L\to R}$ for all rewrite rules $L \to R \in \mathcal{R}$.*

Example 16 (Reduction) *Let us consider the CRS-term $f(t)$ with $t = App(Ab([x]f(x)), a)$. We apply to the sub-term t the BetaCRS rule (Example 15). A solution of the corresponding matching problem is the assignment $\sigma = \{(Z, \underline{\lambda}y.fy), (Z_1, a)\}$, since, when applying it to L, we obtain precisely t:*

$$\begin{aligned}\sigma(L) &= \sigma(App(Ab([x]Z(x)), Z_1))\\ &= App(Ab([x]\sigma(Z)(x), \sigma(Z_1))\\ &= App(Ab([x](\underline{\lambda}y.fy)(x), a))\downarrow_{\underline{\beta}}\\ &= App(Ab([x]f(x)), a) = t\end{aligned}$$

As result of the rule application, we obtain the instantiation by σ of the right-hand side R of the rule BetaCRS: $\sigma(R) = \sigma(Z(Z_1)) = (\sigma(Z))(\sigma(Z_1)) = (\underline{\lambda}y.fy)(a)\downarrow_{\underline{\beta}} = f(a)$.

Therefore we have $t \mapsto_{BetaCRS} f(a)$ and thus $f(t) \mapsto_{BetaCRS} f(f(a))$.

Indeed, one can see that the CRS formalism allow for two binding mechanisms. The first one is explicit in the syntax and denoted $[x]t$. The second is implicit and comes from the rewriting mechanism.

Conclusion

First-order term rewriting, described in the first chapter, and λ-calculus, briefly presented in this chapter, are two fundamental paradigms of computation and because of their complementarity, many frameworks have been designed with a view to integrate these two formalisms. Among them, we have presented the Combinatory Reduction Systems, a term rewrite system with an abstraction mechanism similar to the one used in the λ-calculus. This kind of systems will be considered also in Chapter 3, for a more detailed study on their matching, and in Chapter 5 for an encoding in the system called ρ-calculus, which is presented in Chapter 4.

Chapter 3

Matching in the CRS

In Section 2.3 we have defined CRS-terms and the rewrite relation induced on these terms by a set of CRS-rewrite rules. A rewrite step can be performed on a CRS-term when there exists a match , *i.e.* an assignment, between the left-hand side of a rewrite rule and the given term. From an operational point of view, we may be interested in finding a method to solve the matching problem and obtain this assignment. In other words, we would like to define an algorithm which, starting from a CRS-pattern and a CRS-term, is able to return a substitution of a certain form that applied to the pattern makes it equal to the input CRS-term. These kind of algorithms are usually called matching algorithms. Since to our knowledge a CRS matching algorithm has not been specified until now, the aim of this chapter is to clarify how we can produce a solution of a given CRS matching problem.

To this purpose, we first specify formally what a matching problem in CRSs consists in. Since the meta-language of CRS, *i.e.* the language in which the notions of substitution and rewrite step are expressed, is based on λ-calculus, we obtain a definition of CRS matching which naturally suggests the comparison with the higher-order matching in the λ-calculus. For this reason, we define transformations to and from the λ-calculus for the CRS components. We use then this encoding to correctly and completely define matching in CRSs exactly as the matching of the corresponding λ-terms.

The main result obtained with this approach is a proof of the uniqueness and decidability of the CRS matching. Furthermore, we think that this work can contribute to a better understanding of the behavior of CRSs, in particular the role of matching in the application of rewrite rules and the way CRS substitution works in the reduction of CRS-terms.

3.1 Higher-order and CRS matching

As we have seen in Section 2.3, a CRS-rule can be applied to a CRS-term t if there exists an assignment whose application to the left-hand side L of the rule leads to the term t. This assignment comes from the solution of the matching problem between the metaterms L and t. Thus, in order to provide a way to find the assignment, we need to define more precisely the notion of CRS matching.

Definition 48 (CRS Matching) *Let A_1 and A_2 be two* CRS *metaterms. An assignment σ is solution of the* CRS match equation $A_1 \prec\!\!<_{CRS} A_2$ *if $\sigma(A_1) \equiv A_2$. We denote by Sol($A_1 \prec\!\!<_{CRS} A_2$) the set of all solutions of $A_1 \prec\!\!<_{CRS} A_2$.*

Example 17 • *The matching problem $Z \prec\!\!<_{CRS} [x]f(x)$ has solution $\sigma = \{Z/[x]f(x)\}$.*

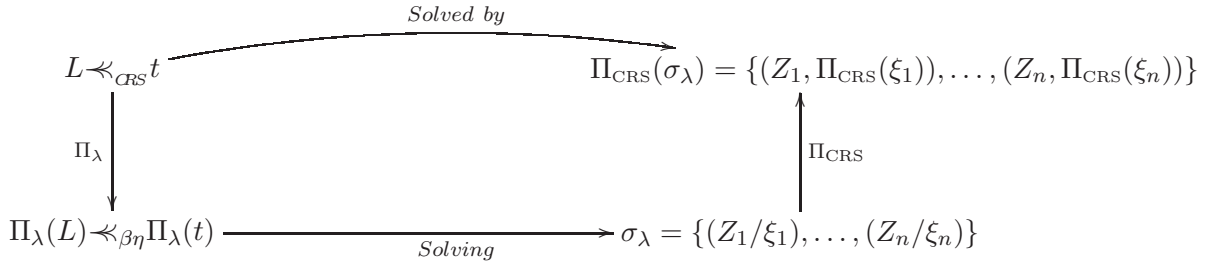


Figure 3.1: Translation of CRS matching problems

- *The matching problem* $[x]Z(x) \prec\!\!\prec_{CRS} [x]f(x)$ *has solution* $\sigma = \{Z/\underline{\lambda}y.f(y)\}$. *We can verify that this assignment is solution of the given match equation:*

$$\{Z/\underline{\lambda}y.f(y)\}([x]Z(x)) = [x]\{Z/\underline{\lambda}y.f(y)\}(Z)(x) = [x](\underline{\lambda}y.f(y))(x)\downarrow_{\underline{\beta}} = [x]f(x)$$

Since matching is mainly used to check if a CRS rewrite rule can be applied to a term, in the following we will restrict to CRS-patterns on left-hand sides of CRS match equations and to CRS-terms on right-hand sides.

We should point out that some "hidden" $\underline{\beta}$-steps are possibly included in the application of an assignment (see Example 16). Therefore, in the CRS the substitution is performed modulo β-reduction applied at the meta-level, and the equality is tested syntactically. Since CRS matching is based on (meta)λ-calculus, it is natural to compare it with λ-higher-order matching, *i.e.* matching modulo the β and η-rules on λ-terms (see Definition 36). In general higher-order matching is undecidable, but higher-order pattern matching (treated here) is decidable and unitary as a consequence of the decidability of pattern unification [Mil91]. We can notice that CRSs use a notion of pattern similar to that defined for the λ-calculus and thus matching in CRS can be naturally compared to pattern higher-order matching.

In the rest of the chapter we describe how a CRS assignment can be algorithmically obtained in CRSs. We show that a solution of a CRS matching problem can be exhibited by translating the problem into a higher-order matching problem on λ-terms and translating back the obtained substitution.

In the following, slightly deviating from the notations of Section 2.1, we will denote free variables in the λ-calculus by capital letters in order to distinguish them from bound variables and to have a notation similar to that of the CRS. The approach we propose here is summarized by the translation schema depicted in Figure 3.1. The projection function Π_λ translates CRS-metaterms into simply typed λ-terms and thus CRS match equations into higher-order match equations on λ-terms. We can then use one of the already known algorithms for higher-order pattern matching in order to find the substitution σ_λ which solves this latter match equation. Finally, the (back) projection function Π_{CRS} translates λ-terms into CRS-terms and thus the obtained substitution σ_λ into a well-defined CRS-assignment.

3.2 From CRS to λ-calculus

We define here a translation function of CRS-metaterms into simply typed λ-terms adapted from [VOVR93], as well as a function that computes the corresponding positions in the translated terms. The types of the λ-terms are built from only one base type that we denote ι. The translation function is chosen to be injective.

Definition 49 *The function* Π_λ *translates* CRS*-terms into simply typed* λ*-terms*

- $\Pi_\lambda(x) = x$ *with* x *of type* ι $(x : \iota)$,
- $\Pi_\lambda(f(t_1, \ldots, t_n)) = f\ \Pi_\lambda(t_1) \ldots \Pi_\lambda(t_n)$ *with* $f : \iota \twoheadrightarrow \ldots \twoheadrightarrow \iota$ *(*n *type arrows),*
- $\Pi_\lambda(Z(t_1, \ldots, t_n)) = Z\ \Pi_\lambda(t_1) \ldots \Pi_\lambda(t_n)$ *with* $Z : \iota \twoheadrightarrow \ldots \twoheadrightarrow \iota$ *(*n *type arrows),*
- $\Pi_\lambda([x]t) = \Lambda\ \lambda x.\Pi_\lambda(t)$ *with* $\Lambda : (\iota \twoheadrightarrow \iota) \twoheadrightarrow \iota$

and assignments into substitutions:

- $\Pi_\lambda(\{\ldots, (Z, t), \ldots\}) = \{\ldots, Z/\Pi_\lambda(t), \ldots\}$,

and substitutes into λ*-terms:*

- $\Pi_\lambda(\underline{\lambda}\overline{x}_n.t) = \lambda\overline{x}_n.\Pi_\lambda(t)$.

The symbol Λ is introduced to collapse a functional symbol in such a way that the resulting term obtained after the translation is of base type. The set of λ-terms of base type can be thought of as the set of "well-formed" CRS-terms. In CRSs function symbols and metavariables are equipped with an arity and metaterms are formed by supplying these symbols with the right number of arguments. In the simply typed λ-calculus, function symbols are constants equipped with a type and similarly metavariables are variables with a given type, specified according to their arity in the CRS. If we consider a λ-term in η-long normal form of base type, by definition of η-long normal form, every operator has exactly as many arguments as prescribed by its type and thus the term is a well-formed CRS-term.

As already mentioned, since we are mainly interested in the matching problems induced by the application of CRS rules we concentrate on match equations of the form $L \prec\!\!\!\prec_{CRS} t$ where $L \in \mathcal{MT}$ is a closed pattern and $t \in \mathcal{T}_{\mathcal{CRS}}$. We can immediately notice that the λ-terms obtained when translating this kind of CRS-terms have certain properties.

Proposition 5 *For any closed pattern* $L \in \mathcal{MT}$ *with* $\mathcal{MV}(L) = \{Z_1, \ldots, Z_n\}$*, and any term* $t \in \mathcal{T}_{\mathcal{CRS}}$ *we have:*

(type) $\Pi_\lambda(L) : \iota$ *and* $\Pi_\lambda(t) : \iota$*;*

(long $\beta\eta$*)* $\Pi_\lambda(L)$ *and* $\Pi_\lambda(t)$ *are in long* $\beta\eta$*-normal form;*

(pattern) $\Pi_\lambda(L)$ *is a* λ*-pattern;*

(free-var) $\mathcal{FV}(\Pi_\lambda(L)) = \{Z_1, \ldots, Z_n\}$*;*

Proof : We prove the properties simultaneously by structural induction on the CRS metaterms L and the term t. We should notice that the translation function Π_λ yields no β-redexes (no applications of λ-abstractions to some other terms can be generated) and thus all the terms in the image of Π_λ are in β-normal form. Furthermore, we should also notice that the only applications contained in the λ-term obtained as translation of a CRS-term are of the form $(((f\ u_1)\ u_2) \ldots u_n)$, where f is a constant and $u_1, \ldots, u_n$ are λ-terms. This holds since, by definition, a CRS-term contains no explicit application operator and no metavariables.

- L can not be a variable since, by hypothesis, it is closed. If t is a variable x, then $\Pi_\lambda(t) = x$.

(type) By definition $x : \iota$.

(long $\beta\eta$) Obvious since $x : \iota$.

- if $L = f(t_1, \ldots, t_n)$, then $\Pi_\lambda(L) = f\ \Pi_\lambda(t_1) \ldots \Pi_\lambda(t_n)$.

  (type) By definition and by induction hypothesis for $t_1, \ldots, t_n$ we have $f : \iota \rightarrowtriangle \ldots \rightarrowtriangle \iota$ (n type arrows) and $\Pi_\lambda(t_i) : \iota$ for all $i = 1, \ldots, n$. Thus $f\ \Pi_\lambda(t_1) \ldots \Pi_\lambda(t_n) : \iota$.

  (long $\beta\eta$) By induction hypothesis for $t_1, \ldots, t_n$, $f\ \Pi_\lambda(t_1) \ldots \Pi_\lambda(t_n) : \iota$ is η-expanded.

  (pattern) By induction hypothesis for $t_1, \ldots, t_n$.

  (free-var) We have $\mathcal{FV}(f(t_1, \ldots, t_n)) = \mathcal{FV}(t_1) \cup \ldots \cup \mathcal{FV}(t_n) \stackrel{ih}{=} \mathcal{FV}(\Pi_\lambda(t_1)) \cup \ldots \cup \mathcal{FV}(\Pi_\lambda(t_n)) = \mathcal{FV}(f\ \Pi_\lambda(t_1) \ldots \Pi_\lambda(t_n))$

  We have a similar proof for t.

- if $L = Z(t_1, \ldots, t_n)$, then $\Pi_\lambda(L) = Z\ \Pi_\lambda(t_1) \ldots \Pi_\lambda(t_n)$.

  (type) By definition and by induction hypothesis for $t_1, \ldots, t_n$ we have $Z : \iota \rightarrowtriangle \ldots \rightarrowtriangle \iota$ (n type arrows) and $\Pi_\lambda(t_i) : \iota$ for all $i = 1, \ldots, n$. Thus $Z\ \Pi_\lambda(t_1) \ldots \Pi_\lambda(t_n) : \iota$.

  (long $\beta\eta$) By induction hypothesis for $t_1, \ldots, t_n$, $Z\ \Pi_\lambda(t_1) \ldots \Pi_\lambda(t_n) : \iota$ is η-expanded.

  (pattern) By induction hypothesis for $t_1, \ldots, t_n$. If $Z(t_1, \ldots, t_n)$ satisfies the pattern condition, so does $Z\ \Pi_\lambda(t_1) \ldots \Pi_\lambda(t_n)$, since distinct variables in L are mapped into distinct variables in $\Pi_\lambda(L)$ thanks to the injectivity of the transformation function Π_λ.

  (free-var) We have $\mathcal{FV}(Z(t_1, \ldots, t_n)) = \{Z\} \cup \mathcal{FV}(t_1) \cup \ldots \cup \mathcal{FV}(t_n) \stackrel{ih}{=} \{Z\} \cup \mathcal{FV}(\Pi_\lambda(t_1)) \cup \ldots \cup \mathcal{FV}(\Pi_\lambda(t_n)) = \mathcal{FV}(Z\ \Pi_\lambda(t_1) \ldots \Pi_\lambda(t_n))$

  The term t can not contain metavariables.

- if $L = [x]u$, then $\Pi_\lambda(L) = \Lambda\ \lambda x.\Pi_\lambda(u)$.

  (type) By definition and by induction hypothesis for u we have $\Lambda : (\iota \rightarrowtriangle \iota) \rightarrowtriangle \iota$ and $\Pi_\lambda(u) : \iota$. Thus $\lambda x.\Pi_\lambda(u) : \iota \rightarrowtriangle \iota$ and therefore $\Lambda\ \lambda x.\Pi_\lambda(u) : \iota$.

  (long $\beta\eta$) By induction hypothesis, u is in long $\beta\eta$-normal form. If $\Pi_\lambda(u) : \iota$ then $\lambda x.\Pi_\lambda(u) : \iota \rightarrowtriangle \iota$ and $\Lambda\ \lambda x.\Pi_\lambda(u) : \iota$ are both η-expanded.

  (pattern) By induction hypothesis for u, since in $L = [x]u$ no new metavariables are introduced.

  (free-var) We have $\mathcal{FV}([x]u) = \mathcal{FV}(u) \setminus \{x\} \stackrel{ih}{=} \mathcal{FV}(\Pi_\lambda(u)) \setminus \{x\} = \mathcal{FV}(\Lambda\ \lambda x.\Pi_\lambda(u))$.

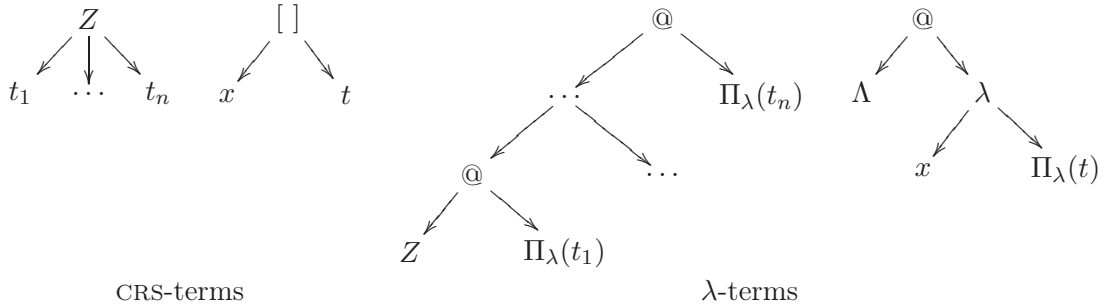
  We have a similar proof for t.

□

Once we have translated the CRS-metaterms appearing on the two sides of a CRS match equation, we can compute the solution of the obtained higher-order matching problem using, for example, the algorithm proposed in [Mil91]. Since the matching is performed against λ-patterns, the substitution that solves the match equation $\Pi_\lambda(L) \lessdot_{\beta\eta} \Pi_\lambda(t)$, if it exists, is unique and has the form $\sigma = \{Z_1/\xi_1, \ldots, Z_n/\xi_n\}$ where ξ_i, $i \in 1, \ldots, n$, are λ-terms in long $\beta\eta$-normal form.

The translation function Π_λ induces a correspondence between the sub-terms of a CRS-term and the sub-terms (of type ι) of its translation. The position ω of a sub-term B in a CRS-term

$A_{[B]_\omega}$ and the position of its translation $\Pi_\lambda(B)$ in the λ-term $\Pi_\lambda(A)$ are not the same. In fact, since n-ary metavariables and n-ary functions are translated into λ-calculus as applications of a variable or a function to n λ-terms, the n-ary tree of the CRS-term becomes a binary tree in the λ-calculus. Similarly, the introduction of the collapsing symbol Λ causes a difference in the positions of a CRS abstraction and in its translation.



CRS-terms λ-terms

We introduce therefore a function π_λ defining the sub-term positions after the translation into the λ-calculus.

Definition 50 *Let A be a* CRS*-metaterm and $\omega \in \mathcal{P}os(A)$ a position in the metaterm A . The function π_λ is inductively defined by:*

1. $\pi_\lambda(\epsilon, A) = \epsilon$,
2. $\pi_\lambda(n - i.\omega, Z(A_1, \ldots, A_n)) = \pi_\lambda(n - i.\omega, f(A_1, \ldots, A_n)) = 1^i.2.\pi_\lambda(\omega, A_{n-i})$, *where* $i \in \{0, \ldots, n-1\}$,
3. $\pi_\lambda(1.\epsilon, [x]A) = 2.1.\pi_\lambda(\epsilon, x)$,
4. $\pi_\lambda(2.\omega, [x]A) = 2.2.\pi_\lambda(\omega, A)$.

We prove next that this correspondence is bijective.

Lemma 8 *The function π_λ defines a bijective correspondence between subterms of a* CRS*-metaterm A and subterms of type ι of the corresponding λ-term $\Pi_\lambda(A)$.*

Proof : By structural induction on A.

- If A is a variable x, then $\Pi_\lambda(A) = x$ and the statement is trivially true.
- If $A = f(t_1, \ldots, t_n)$ (or $A = Z(t_1, \ldots, t_n)$), then $\Pi_\lambda(A) = f\ \Pi_\lambda(t_1) \ldots \Pi_\lambda(t_n)$, (respectively $\Pi_\lambda(A) = Z\ \Pi_\lambda(t_1) \ldots \Pi_\lambda(t_n)$). The correspondence at the head position is obvious. The positions in a subterm t_i of A are related to the corresponding positions in the term $\Pi_\lambda(A)$ by Definition 50 (2). We point out that terms of the form $f\ \Pi_\lambda(t_1) \ldots \Pi_\lambda(t_k)$ with $k \neq n$ are not of type ι.
- If $A = [x]u$, then $\Pi_\lambda(A) = \Lambda\ \lambda x.\Pi_\lambda(u)$. Again the correspondence at the head position is obvious. For a deeper position in the term, the correspondence is defined by Definition 50 (3) and (4). We stress the fact that both the subterms Λ and $\lambda x.\Pi_\lambda(u)$ are not of type ι, thus the correspondence is bijective.

□

3.3 Back from λ-calculus to CRS

The next step consists in defining a function Π_{CRS} which translates substitutions obtained as solution of a higher-order matching problem into CRS-assignments. Since we want to translate only substitutions obtained as solution of a higher-order matching problem corresponding to a CRS-matching problem, the translation function Π_{CRS} is defined only on simply typed λ-terms in long $\beta\eta$ normal form. We recall from the previous section that, since $\Lambda\ u$ has type ι and we consider terms in long $\beta\eta$-normal form, then u has the form $\lambda x.t$ with t of type ι.

Definition 51 *The function Π_{CRS} translates substitutions into assignments:*

- $\Pi_{CRS}(\{\ldots, Z/\xi, \ldots\}) = \{\ldots, (Z, \Pi_{CRS}(\xi)), \ldots\}$,
  with $arity(Z) = n$ if $Z : \iota \rightarrow \ldots \rightarrow \iota$ (n type arrows),

and λ-terms in long $\beta\eta$-normal form into CRS*-terms*

- $\Pi_{CRS}(x) = x$,
- $\Pi_{CRS}(Z\ t_1 \ldots t_n) = Z(\Pi_{CRS}(t_1), \ldots, \Pi_{CRS}(t_n))$ *if $Z : \iota \rightarrow \ldots \rightarrow \iota$ (n type arrows),*
- $\Pi_{CRS}(f\ t_1 \ldots t_n) = f(\Pi_{CRS}(t_1), \ldots, \Pi_{CRS}(t_n))$ *if $f : \iota \rightarrow \ldots \rightarrow \iota$ (n type arrows),*
- $\Pi_{CRS}(\Lambda\ \lambda x.t) = [x]\Pi_{CRS}(t)$,
- $\Pi_{CRS}(\lambda \overline{x}_n.t) = \underline{\lambda}\overline{x}_n.\Pi_{CRS}(t)$.

We show that the translation produces correct terms.

Proposition 6

1. *Given a λ-term $t : \iota$ in long $\beta\eta$-normal form, then $\Pi_{CRS}(t)$ is well-formed w.r.t. the* CRS *syntax.*
2. *Given a substitution in the λ-calculus $\{\ldots, Z/\xi, \ldots\}$ obtained as solution of the translation of a* CRS *matching problem, then $\{\ldots, (Z, \Pi_{CRS}(\xi)), \ldots\}$ is a well-formed* CRS *assignment.*

Proof :

1. By structural induction on the structure of a term in long $\beta\eta$-normal form.
   - If $t = x : \iota$ then $\Pi_{\text{CRS}}(x) = x$ and the statement is trivially verified.
   - If t has a constant as head symbol, say $f : \iota \rightarrow \ldots \rightarrow \iota$ (n type arrows), then $t = f\ t_1 \ldots t_n : \iota$ since t is in long $\beta\eta$-normal form. Consequently the symbol f has arity n in the CRS-syntax and the CRS-term we obtain is $\Pi_{\text{CRS}}(f\ t_1 \ldots t_n) = f(\Pi_{\text{CRS}}(t_1), \ldots, \Pi_{\text{CRS}}(t_n))$. Since $\Pi_{\text{CRS}}(t_1), \ldots, \Pi_{\text{CRS}}(t_n)$ are well-formed by induction hypothesis on $t_1, \ldots, t_n$ and f has arity n, $f(\Pi_{\text{CRS}}(t_1), \ldots, \Pi_{\text{CRS}}(t_n))$ is also well-formed.
   - If t has a variable Z as head symbol and $Z : \iota \rightarrow \ldots \rightarrow \iota$ (n type arrows), then the reasoning is similar as in the previous case.
   - If $t = \Lambda\ t' : \iota$ and $\Lambda : (\iota \rightarrow \iota) \rightarrow \iota$ then t' must be of type $\iota \rightarrow \iota$. Since t' is in long $\beta\eta$-normal form by hypothesis, the only possibility is $t' = \lambda x.u$. Thus we have $\Pi_{\text{CRS}}(\Lambda\ \lambda x.u) = [x]\Pi_{\text{CRS}}(u)$ and we conclude by using the induction hypothesis on u.

2. Since ξ is in long $\beta\eta$-normal form and has the same type as Z then ξ has the form $\lambda x_1 \dots x_n.u$ and thus $\Pi_{\text{CRS}}(\xi)$ has the form $\underline{\lambda} x_1 \dots x_n.\Pi_{\text{CRS}}(u)$ where n is the arity of the metavariable Z.

□

As for the previous translation function, we define a function π_{CRS} which computes the position of a given sub-term after translation.

Definition 52 *The function π_{CRS} taking as arguments a λ-term in long βη-normal form and a position (of a λ-term in long βη-normal form) in this term and returning a position in the Π_{CRS}-translation of the λ-term is inductively defined by:*

- $\pi_{CRS}(\epsilon, u) = \epsilon$,
- $\pi_{CRS}(1^i.2.\omega, Z\ u_1 \dots u_n) = n - i.\pi_{CRS}(\omega, u_{n-i})$ *where* $i \in \{0, \dots, n-1\}$,
- $\pi_{CRS}(1^i.2.\omega, f\ u_1 \dots u_n) = n - i.\pi_{CRS}(\omega, u_{n-i})$ *where* $i \in \{0, \dots, n-1\}$,
- $\pi_{CRS}(2.1.\epsilon, \Lambda\ \lambda x.t) = 1.\pi_{CRS}(\epsilon, x)$,
- $\pi_{CRS}(2.2.\omega, \Lambda\ \lambda x.t) = 2.\pi_{CRS}(\omega, t)$.

Notice that the positions of the Λ symbol in the λ-term are no longer valid in the corresponding CRS-term where they disappear.

We present next some simple examples of CRS matching problems and solutions with the corresponding translation in the λ-calculus. We will show in the next section that the approach we propose is correct and complete.

Example 18 *Given the CRS-metaterm $L = [x]Z(x)$ and the CRS-term $t = [x]f(x)$ we show how the proposed approach can be used in order to solve the CRS match equation $L \prec_{CRS} t$. By translating these two terms into λ-terms (the types are omitted) we obtain $\Pi_\lambda(L) = \Lambda\ \lambda x.Z\ x$ and $\Pi_\lambda(t) = \Lambda\ \lambda x.f\ x$. The solution of the match equation $\Pi_\lambda(L) \prec_{\beta\eta} \Pi_\lambda(t)$ is the substitution $\sigma = \{Z/\lambda y.f\ y\}$ and when translating it into a CRS assignment we obtain $\Pi_{CRS}(\sigma) = \{Z/\underline{\lambda} y.\Pi_{CRS}(f\ y)\} = \{Z/\underline{\lambda} y.f(y)\}$.*

It is easy to verify that this last assignment is solution of the CRS match equation $L \prec_{CRS} t$ (see Example 17).

Example 19 *We consider the CRS-metaterm $L = App([x]Z_1(x), Z_2)$ where $Z_1 \in \mathcal{Z}_1$ and $Z_2 \in \mathcal{Z}_0$ and the CRS-term $t = App([x][y]f(x,y), a)$ and we solve the CRS match equation $L \prec_{CRS} t$. The translation of these two terms into λ-calculus are $\Pi_\lambda(L) = App\ (\Lambda\ \lambda x.Z_1\ x)\ Z_2$ and $\Pi_\lambda(t) = App\ (\Lambda\ \lambda x.\Lambda\ \lambda y.f\ x\ y)\ a$ and the solution of the generated higher-order match equation $\Pi_\lambda(L) \prec_{\beta\eta} \Pi_\lambda(t)$ is the substitution $\sigma = \{Z_1/\lambda z.\Lambda\ \lambda y.f\ z\ y, Z_2/a\}$. When translating this substitution into an assignment we obtain*

$$\begin{aligned}\Pi_{CRS}(\sigma) &= \{(Z_1, \underline{\lambda} z.\Pi_{CRS}(\Lambda\ \lambda y.f\ z\ y)), (Z_2, a)\} \\ &= \{(Z_1, \underline{\lambda} z.[y]\Pi_{CRS}(f\ z\ y)), (Z_2, a)\} \\ &= \{(Z_1, \underline{\lambda} z.[y]f(z,y)), (Z_2, a)\}\end{aligned}$$

We can verify that this last assignment is solution of the CRS match equation $L \prec_{CRS} t$:

$$\begin{aligned}\Pi_{CRS}(\sigma)(L) &= \Pi_{CRS}(\sigma)(App([x]Z_1(x), Z_2)) \\ &= App([x]\Pi_{CRS}(\sigma)Z_1(x), \Pi_{CRS}(\sigma)(Z_2)) \\ &= App([x]\{(Z_1, \underline{\lambda} z.[y]f(z,y)), (Z_2, a)\}Z_1(x), \{(Z_1, \underline{\lambda} z.[y]f(z,y)), (Z_2, a)\}(Z_2)) \\ &= App([x](\underline{\lambda} z.[y]f(z,y))(x)\downarrow_{\underline{\beta}}, a\downarrow_{\underline{\beta}}) \\ &= App([x][y]f(x,y), a) = t\end{aligned}$$

3.4 Properties

We show in what follows the completeness and soundness of the translations, *i.e.* that a substitution is a solution of a CRS match equation if and only if its translation is a solution of the corresponding higher-order match equation. First of all, we state some auxiliary lemmas concerning essentially the context stability and the duality of the two translation functions Π_λ and Π_{CRS}. In the proofs, we will denote by "$\stackrel{ih}{=}$" the equalities obtained using the induction hypothesis.

Lemma 9 (Context stability) *Given a simply typed λ-term $u_{[v]_\omega} : \iota$ in long $\beta\eta$ normal form then for all ω such that $v : \iota$ we have*

$$u|_\omega = v \quad \Leftrightarrow \quad \Pi_{CRS}(u)|_{\pi_{CRS}(\omega,u)} = \Pi_{CRS}(v) \tag{1}$$

Given a CRS*-metaterm $A_{[B]_\omega}$, then*

$$A|_\omega = B \quad \Leftrightarrow \quad \Pi_\lambda(A)|_{\pi_\lambda(\omega,A)} = \Pi_\lambda(B) \tag{2}$$

Proof : For the first equality we proceed by structural induction on the λ-term u.
Base case: If $u = x$ or $u = f$ then $\omega = \epsilon$ and the result follows immediately.
Induction: u is an application. The cases where $\omega = \epsilon$ are obvious.

- If $u = (f\ u_1 \dots u_n)$, since ω is the position of a sub-term of type ι, then $\omega = 1^i.2.\omega'$ with ω' a position in u_{n-i} and we have:

$$\begin{aligned}
\Pi_{\text{CRS}}((f\ u_1 \dots u_n)_{[v]_{1^i.2.\omega'}}) &= \Pi_{\text{CRS}}(f\ u_1 \dots u_{n-i[v]_{\omega'}} \dots u_n) \\
&= f(\Pi_{\text{CRS}}(u_1), \dots, \Pi_{\text{CRS}}(u_{n-i[v]_{\omega'}}), \dots, \Pi_{\text{CRS}}(u_n)) \\
&\stackrel{ih}{=} f(\Pi_{\text{CRS}}(u_1), \dots, \Pi_{\text{CRS}}(u_{n-i})_{[\Pi_{\text{CRS}}(v)]_{\pi_{\text{CRS}}(\omega', u_{n-i})}}, \dots, \Pi_{\text{CRS}}(u_n)) \\
&= f(\Pi_{\text{CRS}}(u_1), \dots, \Pi_{\text{CRS}}(u_n))_{[\Pi_{\text{CRS}}(v)]_{n-i.\pi_{\text{CRS}}(\omega', u_{n-i})}} \\
&= \Pi_{\text{CRS}}(f\ u_1 \dots u_n)_{[\Pi_{\text{CRS}}(v)]_{\pi_{\text{CRS}}(1^i.2.\omega', f\ u_1 \dots u_n)}}
\end{aligned}$$

- If $u = (Z\ u_1 \dots u_n)$ then we proceed as in the previous case.

- If $u = \Lambda\ \lambda x.t$, since $\Lambda : (\iota \to \iota) \to \iota$, then $\omega = 2.\omega'$ with ω' a (non head) position in $\lambda x.t$ and thus we have $\Pi_{\text{CRS}}((\Lambda\ \lambda x.t)_{[v]_{2.\omega'}}) = \Pi_{\text{CRS}}(\Lambda\ (\lambda x.t)_{[v]_{\omega'}})$ with either $\omega' = 1.\epsilon$ and $v = x$ or $\omega' = 2.\omega''$ with ω'' a position in t:

  1. $\Pi_{\text{CRS}}((\Lambda\ \lambda x.t)_{[x]_{2.1.\epsilon}}) = ([x]\Pi_{\text{CRS}}(t))_{[x]_{1.\epsilon}} = \Pi_{\text{CRS}}(\Lambda\ \lambda x.t)_{[\Pi_{\text{CRS}}(x)]_{\pi_{\text{CRS}}(2.1.\epsilon, \Lambda\ \lambda x.t)}}$
  2. $$\begin{aligned}
     \Pi_{\text{CRS}}((\Lambda\ \lambda x.t)_{[v]_{2.2.\omega''}}) &= \Pi_{\text{CRS}}(\Lambda\ \lambda x.t_{[v]_{\omega''}}) = [x]\Pi_{\text{CRS}}(t_{[v]_{\omega''}}) \\
     &\stackrel{ih}{=} [x]\Pi_{\text{CRS}}(t)_{[\Pi_{\text{CRS}}(v)]_{\pi_{\text{CRS}}(\omega'', t)}} \\
     &= ([x]\Pi_{\text{CRS}}(t))_{[\Pi_{\text{CRS}}(v)]_{2.\pi_{\text{CRS}}(\omega'', t)}} \\
     &= \Pi_{\text{CRS}}(\Lambda\ \lambda x.t)_{[\Pi_{\text{CRS}}(v)]_{\pi_{\text{CRS}}(2.\omega'', \lambda x.t)}} \\
     &= \Pi_{\text{CRS}}(\Lambda\ \lambda x.t)_{[\Pi_{\text{CRS}}(v)]_{\pi_{\text{CRS}}(2.2.\omega'', \Lambda\ \lambda x.t)}}
     \end{aligned}$$

For the second equality we proceed by structural induction on the CRS-metaterm A.
Base case: If $A = x$ or $A = f$ then $\omega = \epsilon$ and the result follows immediately.
Induction: The cases where $\omega = \epsilon$ are obvious and for the others we have:

- If $A = f(A_1, \ldots, A_n)$ with $\omega = n - i.\omega'$ and $i \in \{0, \ldots, n-1\}$ then

$$\begin{aligned}\Pi_\lambda(f(A_1,\ldots,A_n)_{[B]_{n-i.\omega'}}) &= \Pi_\lambda(f(A_1,\ldots,A_{n-i[B]_{\omega'}},\ldots,A_n))\\ &= f\ \Pi_\lambda(A_1)\ldots\Pi_\lambda(A_{n-i[B]_{\omega'}})\ldots\Pi_\lambda(A_n)\\ &\stackrel{ih}{=} f\ \Pi_\lambda(A_1)\ldots\Pi_\lambda(A_{n-i})_{[\Pi_\lambda(B)]_{\pi_\lambda(\omega',A_{n-i})}}\ldots\Pi_\lambda(A_n)\\ &= (f\ \Pi_\lambda(A_1)\ldots\Pi_\lambda(A_n))_{[\Pi_\lambda(B)]_{1^i.2.\pi_\lambda(\omega',A_{n-i})}}\\ &= \Pi_\lambda(f(A_1,\ldots,A_n))_{[\Pi_\lambda(B)]_{\pi_\lambda(n-i.\omega',f(A_1,\ldots,A_n))}}\end{aligned}$$

- If $A = Z(A_1, \ldots, A_n)$ then we proceed as in the previous case.
- If $A = [x]A'$ then ω is either of the form $\omega = 1.\epsilon$ and $v = x$ or of the form $\omega = 2.\omega'$ with ω' a position in A':
  1. $\Pi_\lambda(([x]A')_{[x]_{1.\epsilon}}) = (\Lambda\ \lambda x.\Pi_\lambda(A'))_{[x]_{2.1.\epsilon}} = \Pi_\lambda([x]A')_{[\Pi_\lambda(x)]_{\pi_\lambda(1.\epsilon,[x]A')}}$
  2. $$\begin{aligned}\Pi_\lambda(([x]A')_{[B]_{2.\omega'}}) &= \Pi_\lambda([x]A'_{[B]_{\omega'}}) = \Lambda\ \lambda x.\Pi_\lambda(A'_{[B]_{\omega'}})\\ &\stackrel{ih}{=} \Lambda\ \lambda x.(\Pi_\lambda(A')_{[\Pi_\lambda(B)]_{\pi_\lambda(\omega',A')}})\\ &= (\Lambda\ \lambda x.\Pi_\lambda(A'))_{[\Pi_\lambda(B)]_{2.2.\pi_\lambda(\omega',A')}}\\ &= \Pi_\lambda([x]A')_{[\Pi_\lambda(B)]_{\pi_\lambda(2.\omega',[x]A')}}\end{aligned}$$

□

Lemma 10 (Inverse) *Given a simply typed λ-term $u : \iota$ in long $\beta\eta$ normal form then we have*

$$\Pi_\lambda(\Pi_{CRS}(u)) = u \tag{1}$$

Given a CRS*-metaterm $A \in \mathcal{MT}$, we have*

$$\Pi_{CRS}(\Pi_\lambda(A)) = A \tag{2}$$

Proof : For the first equality we proceed by structural induction on the λ-term u.
Base case: If $u = x$ then $\Pi_\lambda(\Pi_{\text{CRS}}(x)) = \Pi_\lambda(x) = x$. The case $u = f$ is similar.
Induction: The cases where $\omega = \epsilon$ are obvious. For the others we have:

- If $u = f\ t_1 \ldots t_n$ then, we obtain

$$\begin{aligned}\Pi_\lambda(\Pi_{\text{CRS}}(f\ t_1\ldots t_n)) &= \Pi_\lambda(f(\Pi_{\text{CRS}}(t_1),\ldots,\Pi_{\text{CRS}}(t_n)))\\ &= f\ \Pi_\lambda(\Pi_{\text{CRS}}(t_1))\ldots\Pi_\lambda(\Pi_{\text{CRS}}(t_n)) \stackrel{ih}{=} f\ t_1\ldots t_n\end{aligned}$$

- If $u = \Lambda\ \lambda x.t$ then $\Pi_\lambda(\Pi_{\text{CRS}}(\Lambda\ \lambda x.t)) = \Pi_\lambda([x]\Pi_{\text{CRS}}(t)) = \Lambda\ \lambda x.\Pi_\lambda(\Pi_{\text{CRS}}(t)) \stackrel{ih}{=} \Lambda\ \lambda x.t$

For the second equality we proceed by structural induction on the CRS-metaterm A.
Base case: If $A = x$ then $\Pi_{\text{CRS}}(\Pi_\lambda(x)) = \Pi_{\text{CRS}}(x) = x$. The case $A = f$ is similar.
Induction:

- If $A = f(A_1, \ldots, A_n)$ we obtain

$$\begin{aligned}\Pi_{\text{CRS}}(\Pi_\lambda(f(A_1,\ldots,A_n))) &= \Pi_{\text{CRS}}(f\ \Pi_\lambda(A_1)\ldots\Pi_\lambda(A_n))\\ &= f(\Pi_{\text{CRS}}(\Pi_\lambda(A_1)),\ldots,\Pi_{\text{CRS}}(\Pi_\lambda(A_n))) \stackrel{ih}{=} f(A_1,\ldots,A_n)\end{aligned}$$

- We proceed similarly if $A = Z(A_1, \ldots, A_n)$.
- If $A = [x]A'$ we obtain $\Pi_{\text{CRS}}(\Pi_\lambda([x]A')) = \Pi_{\text{CRS}}(\Lambda\ \lambda x.\Pi_\lambda(A')) = [x]\Pi_{\text{CRS}}(\Pi_\lambda(A')) \stackrel{ih}{=} [x]A'$

□

These properties of the projection functions are used to prove next that, given a CRS match equation, the solution of the translated match equation in the λ-calculus can be projected into a CRS assignment that is solution of the initial match equation in the CRS.

Theorem 7 (Soundness) *Let $L \in \mathcal{MT}$ be a closed* CRS*-pattern and $t \in \mathcal{T}_{\mathcal{CRS}}$ be a a* CRS*-term. Then*

$$\sigma \in \mathcal{S}ol(\Pi_\lambda(L) \lessdot_{\beta\eta} \Pi_\lambda(t)) \quad \Rightarrow \quad \Pi_{CRS}(\sigma) \in \mathcal{S}ol(L \lessdot_{\mathcal{CRS}} t)$$

Proof : According to Definitions 36 and 48 we have to prove that

$$\sigma(\Pi_\lambda(L)) =_{\beta\eta} \Pi_\lambda(t) \quad \Rightarrow \quad \Pi_{\text{CRS}}(\sigma)(L) \equiv t$$

and since $\Pi_\lambda(t)$ is in long $\beta\eta$-normal form (Proposition 5) then, we should prove that

$$\sigma(\Pi_\lambda(L)) \mapsto\!\!\!\!\!\rightarrow_\beta \Pi_\lambda(t) \quad \Rightarrow \quad \Pi_{\text{CRS}}(\sigma)(L) \equiv t$$

If L contains no CRS-metavariables then $\Pi_\lambda(L)$ contains no free variables. Consequently, σ acts as the identity and so, according to Definition 51 and since Π_{CRS} preserves the domain of the substitution, $\Pi_{\text{CRS}}(\sigma)$ acts as the identity as well. Since $\Pi_\lambda(L) \equiv \Pi_\lambda(t)$ and since Π_λ is injective then $L \equiv t$.

We should consider next the case where $Z_1, \ldots, Z_n$ are the n metavariables of the metaterm $L_{[Z_1(y_1,\ldots,y_{k_1})]_{\omega_1}\ldots[Z_n(y_1,\ldots,y_{k_n})]_{\omega_n}}$ with $\omega_1, \ldots, \omega_n$ sharing no prefix. Since we have $\mathcal{FV}(\Pi_\lambda(L)) = \{Z_1, \ldots, Z_n\}$ the substitution σ is of the form $\sigma = \{Z_1/\xi_1, \ldots, Z_n/\xi_n\}$ and since there is no interference between the n metavariable positions then, by Lemma 9(2),

$$\sigma(\Pi_\lambda(L_{[Z_1(y_1,\ldots,y_{k_1})]_{\omega_1}\ldots[Z_n(y_1,\ldots,y_{k_n})]_{\omega_n}})) = \sigma(\Pi_\lambda(L)_{[Z_1\ y_1\ \ldots\ y_{k_1}]_{\pi_\lambda(\omega_1,L)}\ldots[Z_n\ y_1\ \ldots\ y_{k_n}]_{\pi_\lambda(\omega_n,L)}})$$

and this latter term is equivalent to $\sigma(\Pi_\lambda(L))_{\lceil \xi_1\ y_1\ \ldots\ y_{k_1} \rceil_{\pi_\lambda(\omega_1,L)}\ldots\lceil \xi_n\ y_1\ \ldots\ y_{k_n} \rceil_{\pi_\lambda(\omega_n,L)}}$. Since the translation does not generate applications involving the sub-terms at the n positions, all we need to conclude the proof is the property for only one metavariable.

We have thus to prove that if $\sigma(\Pi_\lambda(L_{[Z(y_1,\ldots,y_k)]_\omega})) \mapsto\!\!\!\!\!\rightarrow_\beta \Pi_\lambda(t)$ then $\Pi_{\text{CRS}}(\sigma)(L_{[Z(y_1,\ldots,y_k)]_\omega}) \equiv t$. Since $\sigma = \{Z/\xi\}$ then, we can use Lemma 9(2) and obtain

$$\begin{aligned}
\sigma(\Pi_\lambda(L_{[Z(y_1,\ldots,y_k)]_\omega})) &= \sigma(\Pi_\lambda(L)_{[Z\ y_1\ \ldots\ y_k]_{\pi_\lambda(\omega,L)}}) \\
&= \Pi_\lambda(L)_{[\sigma(Z\ y_1\ \ldots\ y_k)]_{\pi_\lambda(\omega,L)}} \\
&\mapsto\!\!\!\!\!\rightarrow_\beta \Pi_\lambda(L)_{\lceil (\xi\ y_1\ \ldots\ y_k)\downarrow_\beta \rceil_{\pi_\lambda(\omega,L)}} \\
&= \Pi_\lambda(t).
\end{aligned}$$

Since ξ is in long $\beta\eta$-normal form and has the same type as $Z : \iota \twoheadrightarrow \ldots \twoheadrightarrow \iota$ (n type arrows), then ξ has the form $\lambda x_1 \ldots . x_k.u$ with u in normal form and thus

$$\Pi_\lambda(t) = \Pi_\lambda(L)_{[(u\{x_1/y_1,\ldots,x_k/y_k\})\downarrow_\beta]_{\pi_\lambda(\omega,L)}} = \Pi_\lambda(L)_{[u\{x_1/y_1,\ldots,x_k/y_k\}]_{\pi_\lambda(\omega,L)}}$$

and $\Pi_{\text{CRS}}(\sigma) = \{(Z, \Pi_{\text{CRS}}(\xi))\} = \{(Z, \underline{\lambda} x_1 \ldots . x_n.\Pi_{\text{CRS}}(u))\}$. Moreover, since u is in normal form it is easy to see that $\Pi_{\text{CRS}}(u)$ is in normal form as well and that $(\Pi_{\text{CRS}}(u)\{x_1/y_1, \ldots, x_k/y_k\})\downarrow_{\underline{\beta}}$ $= \Pi_{\text{CRS}}(u)\{x_1/y_1, \ldots, x_k/y_k\} = \Pi_{\text{CRS}}(u\{x_1/y_1, \ldots, x_k/y_k\})$. Then, we have

$$\begin{aligned}
\Pi_{\text{CRS}}(\sigma)(L_{[Z(y_1,\ldots,y_k)]_\omega}) &= L_{[(\Pi_{\text{CRS}}(\sigma)(Z))(y_1,\ldots,y_k)]_\omega} \\
&\mapsto\!\!\!\!\!\!\rightarrow_{\underline{\beta}} \; L_{[(\Pi_{\text{CRS}}(u)\{x_1/y_1,\ldots,x_k/y_k\})\downarrow_{\underline{\beta}}]_\omega} \\
&= L_{[\Pi_{\text{CRS}}(u\{x_1/y_1,\ldots,x_k/y_k\})]_\omega} \\
&= \Pi_{\text{CRS}}(\Pi_\lambda(L_{[\Pi_{\text{CRS}}(u\{x_1/y_1,\ldots,x_k/y_k\})]_\omega})) \text{ (by Lemma 10(2))} \\
&= \Pi_{\text{CRS}}(\Pi_\lambda(L)_{[\Pi_\lambda(\Pi_{\text{CRS}}(u\{x_1/y_1,\ldots,x_k/y_k\}))]_{\pi_\lambda(\omega,L)}}) \text{ (by Lemma 9(2))} \\
&= \Pi_{\text{CRS}}(\Pi_\lambda(L)_{[u\{x_1/y_1,\ldots,x_k/y_k\}]_{\pi_\lambda(\omega,L)}}) \text{ (by Lemma 10(1))} \\
&= \Pi_{\text{CRS}}(\Pi_\lambda(t)) \\
&= t \text{ (by Lemma 10(2)).}
\end{aligned}$$

□

The other direction of the implication also holds: a solution of a match equation in the CRS is also a solution of the match equation in the λ-calculus, after the respective translations.

Theorem 8 (Completeness) *Let $L \prec\!\!\!\prec_{CRS} t$ be a* CRS *match equation with $L \in \mathcal{MT}$ a closed pattern and $t \in \mathcal{T}_{\mathcal{CRS}}$, and σ be a* CRS *assignment.*

$$\sigma \in \mathcal{S}ol(L \prec\!\!\!\prec_{CRS} t) \quad \Rightarrow \quad \Pi_\lambda(\sigma) \in \mathcal{S}ol(\Pi_\lambda(L) \prec\!\!\!\prec_{\beta\eta} \Pi_\lambda(t))$$

Proof : The proof method is similar to the one used in Theorem 7. We need to prove that

$$\sigma L \equiv t \quad \Rightarrow \quad \Pi_\lambda(\sigma)(\Pi_\lambda(L)) \mapsto\!\!\!\!\!\!\rightarrow_\beta \Pi_\lambda(t)$$

If L contains no CRS-metavariables, then σ acts as the identity and so (by Definition 49 that preserves the domain of the substitution) $\Pi_\lambda(\sigma)$ acts as the identity as well. Since $L \equiv t$ then $\Pi_\lambda(L) = \Pi_\lambda(t)$.

We suppose next that $Z_1, \ldots, Z_n$ are the n metavariables of $L_{[Z_1(y_1,\ldots,y_{k_1})]_{\omega_1}\ldots[Z_n(y_1,\ldots,y_{k_n})]_{\omega_n}}$ with $\omega_1, \ldots, \omega_n$ sharing no prefix. Since there is no interference between the n metavariable positions, using a similar reasoning as in Theorem 7, we need to prove the property for only one metavariable Z of arity k at the position ω in L to conclude the proof.

By Definition 44, σ has the form $\{(Z, \underline{\lambda}x_1.\ldots.x_k.u)\}$, where u is a CRS-term, and thus $\Pi_\lambda(\sigma) = \{Z/\lambda x_1.\ldots.x_k.\Pi_\lambda(u)\}$ where the term $\Pi_\lambda(u) : \iota$ is in long $\beta\eta$-normal form. By hypothesis we know that $\sigma(L_{[Z(y_1,\ldots,y_k)]_\omega}) \mapsto\!\!\!\!\!\!\rightarrow_{\underline{\beta}} L_{[u\{x_1/y_1,\ldots,x_k/y_k\}]_\omega} = t$ and thus we have to prove that $\Pi_\lambda(\sigma)(\Pi_\lambda(L_{[Z(y_1,\ldots,y_k)]_\omega})) \mapsto\!\!\!\!\!\!\rightarrow_\beta \Pi_\lambda(t)$. We have

$$\begin{aligned}
\Pi_\lambda(\sigma)(\Pi_\lambda(L_{[Z(y_1,\ldots,y_k)]_\omega})) &= \Pi_\lambda(\sigma)(\Pi_\lambda(L)_{[Z\ y_1\ldots y_k]_{\pi_\lambda(\omega,L)}}) \quad \text{(by Lemma 9(2))} \\
&= \Pi_\lambda(L)_{[\Pi_\lambda(\sigma)(Z)\ y_1\ldots y_k]_{\pi_\lambda(\omega,L)}} \\
&\mapsto\!\!\!\!\!\!\rightarrow_\beta \Pi_\lambda(L)_{[\Pi_\lambda(u)\{x_1/y_1,\ldots,x_k/y_k\}\downarrow_\beta]_{\pi_\lambda(\omega,L)}} \\
&= \Pi_\lambda(L)_{[\Pi_\lambda(u\{x_1/y_1,\ldots,x_k/y_k\})]_{\pi_\lambda(\omega,L)}} \\
&= \Pi_\lambda(L_{[u\{x_1/y_1,\ldots,x_k/y_k\}]_\omega}) \text{ (by Lemma 10(2))} \\
&= \Pi_\lambda(t)
\end{aligned}$$

□

Corollary 1 CRS *match equations of the form $L \prec\!\!\!\prec_{CRS} t$ where L is a* CRS*-pattern and t a* CRS*-term are decidable and unitary.*

Conclusion

In this chapter we explained how a solution of a CRS matching problem can be provided. First we have given a precise definition of the notion of matching in CRSs which indeed is very similar to higher-order matching in the λ-calculus. Starting from this consideration, we defined the translation functions between the two systems and we have shown how a solution of a CRS matching problem can be exhibited by solving the corresponding higher-order matching problem on λ-terms. Finally, we have proved the soundness and correctness of our approach. The final result thus obtained is a proof of the decidability of the CRS matching and of the uniqueness of its solution.

Theorems 7 and 8, showing the soundness and completeness results, could be generalised to a matching problem of the form $\Pi_\lambda(A) \lessdot_{\beta\eta} \Pi_\lambda(t)$, respectively $A \lessdot_{CRS} t$, where the metaterm A is not necessarily a CRS pattern. This interesting question is currently under investigation.

Chapter 4

The rewriting calculus

We present here the central calculus of this thesis which is at the basis of the work done in the following chapters. This higher-order formalism, called rewriting calculus (or ρ-calculus) extends first-order term rewriting and λ-calculus.

From the λ-calculus, the ρ-calculus inherits its higher-order capabilities and the explicit treatment of functions and their application. The lambda abstraction is generalised to an abstraction on patterns, in such a way that term rewrite rules can be naturally represented. As a consequence of the definition of rewrite rules at the object level, the decision of redex reduction becomes explicit and the rewrite relation can be better controlled. The evaluation is based on matching which allow the calculus the possibility to express also non-reducibility. When matching is not performed syntactically, *e.g.* modulo an equational theory, the matching problem is in general non unitary. This consideration enlightens the non-deterministic nature of rewriting and the need to conveniently describe this aspect. In the ρ-calculus, results of matching are made explicit objects, allowing the calculus to handle non-determinism in a natural way.

We can thus summarise saying that the ρ-calculus main design concept is to make all the basic ingredients of rewriting explicit objects, in particular the notions of rule *formation*, *application* and *result*. The obtained calculus is quite expressive and offers a broad spectrum of applications. Its fundamental parameters, namely the subset of ρ-terms that can be used as patterns, the theory modulo which matching is performed and the structure under which the results of a rule application are returned, can be customised to describe in a uniform way different calculi. Typical examples are λ-calculus, term rewriting and object calculi.

The first version of the calculus, due to H. Cirstea and C. Kirchner [CK01], was introduced to give a semantics to the rewrite based language ELAN [CK98]. A simplified version has been proposed by the same authors in collaboration with L. Liquori in [CKL02]. More recently, several extensions have been proposed: a version handling explicitly the (application of) constraints [CFK04], briefly recalled in the last section of this chapter, and a (typed) imperative version of the calculus [LS04] and the corresponding certified interpreter. A polymorphic type system can be found in the previously mentioned paper [CKL02] and type checking and type inference issues have been studied in [LW04, Wac05].

4.1 Syntax

In the ρ-calculus, the usual λ-abstraction $\lambda x.t$ is generalised by a *rule abstraction* $p \twoheadrightarrow t$, also called a *ρ-rule* to support the relationship with term rewriting, where p can be a more general pattern than a single variable x. The application of a rule $p \twoheadrightarrow t$ to a term t' is denoted $(p \twoheadrightarrow t)\ t'$ and evaluates to a *delayed matching constraint* $t[p \ll t']$. This kind of constraint was introduced for the first time in [CKL02] and represents a constrained term t where the matching constraint $p \ll t'$ is waiting for evaluation. To handle non determinism, the operator "$\_\wr\_$" is introduced to group terms together into *structures*, usually used to represent sets of rewrite rules or results.

Given a set of variables $\mathcal{X}$, whose elements are denoted by $x, y, \ldots X, Y \ldots$, and a set of constants $\mathcal{K}$, whose elements are denoted by $a, b, \ldots f, g \ldots$, the set of ρ-terms is thus defined as follows:

$$\begin{array}{lcl} \mathcal{T} & ::= & \mathcal{X} \mid \mathcal{K} \mid \mathcal{T}\ \mathcal{T} \mid \mathcal{P} \twoheadrightarrow \mathcal{T} \mid \mathcal{T}[\mathcal{P} \ll \mathcal{T}] \mid \mathcal{T} \wr \mathcal{T} \mid \\ \mathcal{P} & \subseteq & \mathcal{T} \end{array}$$

To simplify the reading of ρ-terms, we will use some syntactical conventions. We assume that the application operator associates to the left, while all the other operators associate to the right. Algebraic terms, represented in a currified style using the application operator $(f\ t_1 \cdots t_n)$, with $f \in \mathcal{K}$, will often be written using the alias $f(t_1, \ldots, t_n)$, in order to recover the standard algebraic notation. To avoid the use of parenthesis, we define the priority of the application operator higher than the priority of "$\_[\_ \ll \_]$", which is higher than that of the abstraction "$\_ \twoheadrightarrow \_$", which is in turn higher than that of structure operator "$\_\wr\_$".

The general framework of the ρ-calculus can be made specific by instantiating the parameters of the calculus, *i.e.* by choosing

- an underlying theory for the "$\_\wr\_$" operator,
- the concrete subset of ρ-terms that can be used as patterns,
- the congruence on terms modulo which the matching is performed.

Different useful instances of the calculus will be exemplified in the following of this section and in the next section.

Example 20 (ρ-terms)

- *the ρ-term $x \twoheadrightarrow x$ represents the λ-term $\lambda x.x$ (the identity),*
- *the ρ-term $(x \twoheadrightarrow x\ x)\ (x \twoheadrightarrow x\ x)$ represents the λ-term $(\lambda x.(x\ x)\ \lambda x.(x\ x))$,*
- *the ρ-term $+(x, y) \twoheadrightarrow x$ has no corresponding easily writable term in the λ-calculus,*
- *the ρ-terms $+(0, y) \twoheadrightarrow y$ and $+(s(x), y) \twoheadrightarrow s(+(x, y))$ represent two rewrite rules defining addition on natural numbers.*

The ρ-calculus is an higher-order calculus that contains two kinds of binders: in $t_1 \twoheadrightarrow t_2$, the free variables of t_1 are bound in t_2 and in $t_3[t_1 \ll t_2]$, the free variables of t_1 are bound in t_3 but not in t_2. A formal definition of free and bound variables is given next.

Definition 53 (Free, bound and active variables) *The set of* free *variables of a term is defined as:*

$$\begin{array}{rcl} \mathcal{FV}(X) & = & \{X\} \\ \mathcal{FV}(a) & = & \emptyset \\ \mathcal{FV}(p \twoheadrightarrow t_1) & = & \mathcal{FV}(t_1) \setminus \mathcal{FV}(p) \\ \mathcal{FV}(t_3[p \ll t_2]) & = & (\mathcal{FV}(t_3) \setminus \mathcal{FV}(p)) \cup \mathcal{FV}(t_2) \\ \mathcal{FV}(t_1\, t_2) \;=\; \mathcal{FV}(t_1 \backslash t_2) & = & \mathcal{FV}(t_1) \cup \mathcal{FV}(t_2) \end{array}$$

The set of bound variables of a term is defined as follows:

$$\begin{array}{rcl} \mathcal{BV}(X) & = & \emptyset \\ \mathcal{BV}(a) & = & \emptyset \\ \mathcal{BV}(p \twoheadrightarrow t_1) & = & \mathcal{BV}(t_1) \cup \mathcal{BV}(p) \cup \mathcal{FV}(p) \\ \mathcal{BV}(t_3[p \ll t_2]) & = & \mathcal{BV}(t_3) \cup \mathcal{BV}(t_2) \cup \mathcal{BV}(p) \cup \mathcal{FV}(p) \\ \mathcal{BV}(t_1\, t_2) \;=\; \mathcal{BV}(t_1 \backslash t_2) & = & \mathcal{BV}(t_1) \cup \mathcal{BV}(t_2) \end{array}$$

A term is called closed *or* ground *if all its variables are bound. Given a ρ-term t, if at some occurrence of t the left-hand side of an application is a variable (like x in $t = x\ u$), we say that this variable is* active. *For example x is active in the ρ-term $t = x\ a\ b$, but not in $t = f\ x\ b$.*

As in any calculus involving binders, we work modulo the *α-convention* and modulo the *hygiene-convention* of Barendregt [Bar84], *i.e.* free and bound variables have different names.

The ρ-calculus is a flexible and expressive framework thanks to the possibility of customising its parameters according tho the use we want to make of the calculus. For example, different semantics can be obtained according to the theories associated to the structure operator. Typically, if an associative-commutative status is given to the this operator then we obtain results which can be interpreted as *multi-sets*. If also the idempotence axiom is added, then we recover the semantics of result *sets* [CK01].

Another parameter of the ρ-calculus is the set of patterns $\mathcal{P}$. In in full generality could be as large as the set of all terms $\mathcal{T}$. In many cases we restrict it to specific classes of terms, in order to ensure some properties of the calculus, like confluence, or some properties of the matching, like decidability. If $\mathcal{P}$ is the set of variables, then the ρ-abstraction is simply a λ-abstraction. A set of patterns commonly used are the so-called algebraic patterns which are terms of the form $f(t_1, \ldots, t_n)$ where $t_1, \ldots, t_n$ are algebraic. Another important class of terms consists of the ρ-patterns directly inspired from the classical λ-patterns that allow for unitary higher-order unification (see Definition 37).

Definition 54 (ρ-pattern) *A ρ-term p is called a* ρ-pattern *if it contains no structure operator and no matching constraints, if the left-hand sides of its abstractions are variables and if every of its free variable Z appears in a sub-term of p of the form $Z\ x_1 \ldots\ x_n$ where the variables $x_1, \ldots, x_n$, $n \geq 0$, are distinct and all bound in p,*

For example, $x \twoheadrightarrow f(Z\ x)$ is a ρ-pattern while $x \twoheadrightarrow (Z\ x\ x)$ and $g(Z\ x)$ are not since x appears twice on the left-hand side of the abstraction and x is not bound in the latter term. This kind of pattern will be used in Chapter 5 for the encoding of higher-order rewriting in the ρ-calculus.

Different instances of the calculus are obtained when instantiating the theory used at matching time by a congruence on terms. This congruence can be defined, for example, equationally (see Definition 26) when some operators of the signature are *e.g.* associative and commutative or by other means like, for example, considering the $\beta\eta$-equivalence. Nevertheless, one of the most usual and convenient approaches consists in introducing the congruence as a theory defined by a set of axioms. Therefore, we will sometimes call "matching theory" the congruence modulo which matching is performed.

We assume given a congruence $\mathbb{T}$ and we now recall the definition of matching problems in this general setting.

Definition 55 (Matching modulo $\mathbb{T}$) *Given a congruence $=_{\mathbb{T}}$ on the terms, a* $\mathbb{T}$-match equation *is a formula of the form $t_1 \prec\!\!<_{\mathbb{T}} t_2$, where t_1 and t_2 are two terms. A substitution σ is solution of the match equation $t_1 \prec\!\!<_{\mathbb{T}} t_2$ if $\sigma(t_1) =_{\mathbb{T}} t_2$. A* $\mathbb{T}$ matching-system *is a conjunction of* $\mathbb{T}$ *match equations. A substitution is solution of a* $\mathbb{T}$ *matching-system E if it is solution of all the* $\mathbb{T}$*-match equations in E. We denote by Sol(E) the set of all solutions of E.*

If the underlying congruence $\mathbb{T}$ is clear from the context (usually when matching is performed syntactically) the notation $t_1 \prec\!\!< t_2$ is used instead of $t_1 \prec\!\!<_{\mathbb{T}} t_2$.

Since in general we could consider arbitrary theories over ρ-terms, $\mathbb{T}$-matching is in general undecidable, even when restricted to first-order equational theories [JK91]. We are primarily interested here in the decidable cases. Among them we can mention higher-order-pattern matching that is decidable and unitary as a consequence of the decidability of pattern unification [Mil91, DHKP96].

General higher-order matching has been an open question for 30 years, since the first time the problem was explicitly stated and conjectured decidable in [Hue76], and nowadays is still open. Higher-order matching was first studied by Baxter and Huet [Hue76, Bax76] in the mid-seventies in their Ph.D. theses. In the same years, higher-order matching, under the name of "the range problem", was investigated also by Statman [Sta82], who reduced its decidability to decidability of λ-definability. Unfortunately, Loader [Loa] gave a negative answer to the latter question. Due to the difficulty of the general problem, several studies have been dedicated to the investigation of the problem with the order bound by a constant, where by order of a matching problem $t_1 \prec\!\!<_{\mathbb{T}} t_2$ we intend the highest functionality order of free variables occurring in t_1. At the current state of the art, higher-order matching is known to be decidable up to the fourth order [Dow92, Pad96].

Other interesting matching theories are those defined equationally, which are used in several areas of computer science, including theorem proving, logic programming and program verification. The first to observe that theorem provers should use matching in equational theories, like associative and commutative matching, was Plotkin in [Plo72]. Since then, the field of equational matching (and unification) has been extensively investigated. Several equational theories have been considered in the literature [Nip91, JK91, Rin96], like associativity, commutativity, idempotency, neutral element, Abelian group and boolean ring, and most of them yield to equational matching decision problems that are NP-complete [BKN87, KN92, GNW96]. A remarkable exception is the case of associative and commutative (AC) matching on linear terms, which is proved to be polynomial by means of graph matching techniques [BKN87].

Example 21 (Matching)

- *The matching-equation $a \prec\!\!<_{\emptyset} b$ has no solution.*
- *The matching-equation $head(x, y) \prec\!\!<_{\emptyset} head(a, b)$ has solution $\sigma = \{x/a, y/b\}$.*

$$
\begin{array}{lll}
(\rho) & (p \twoheadrightarrow t_2)t_3 & \rightarrow_{\rho} \quad t_2[p \ll t_3] \\
(\sigma) & t_2[p \ll t_3] & \rightarrow_{\sigma} \quad \sigma_1(t_2) \wr \ldots \wr \sigma_n(t_2) \wr \ldots \\
 & & \quad \textit{where } \sigma_i \in \mathit{Sol}(p \prec\!\!\prec_{\mathbb{T}} t_3) \\
(\delta) & (t_1 \wr t_2)\, t_3 & \rightarrow_{\delta} \quad t_1\, t_3 \wr t_2\, t_3
\end{array}
$$

Figure 4.1: Small-step reduction semantics of the ρ-calculus

- *The matching-equation* $+(x,y) \prec\!\!\prec_{\mathbb{T}_{\mathcal{AC}}} +(a,b)$*, where* $\mathbb{T}_{\mathcal{AC}}$ *is the associative and commutative theory, has the solutions* $\sigma_0 = \{x/a, y/b\}$ *and* $\sigma_1 = \{x/b, y/a\}$*.*

4.2 Semantics

The small-step reduction semantics of the ρ-calculus is defined by the reduction rules presented in Figure 4.1. The central idea of the (ρ)-rule of the calculus is that the application of a term $t_1 \twoheadrightarrow t_2$ to a term t_3, reduces to the delayed matching constraint $t_2[t_1 \ll t_3]$, while the application of the (σ)-rule consists in solving (modulo the congruence $\mathbb{T}$) the matching equation $t_1 \prec\!\!\prec_{\mathbb{T}} t_3$, and applying the obtained result to the term t_2. The rule (δ) deals with the distributivity of the application on the structures built with the "$\wr$" constructor.

It is worth noticing that according to the congruence modulo which matching is performed, the application of the (σ)-rule may produce a structure with a finite or infinite number of elements depending on the number of solutions for the corresponding matching problem. Usually we restrict to congruences leading to a unitary set of substitutions and thus, in this case, the result of reducing a delayed matching constraint is always at most a singleton.

In relation to the use we want to make of the calculus, matching failures can be treated in different ways. In several versions of the calculus [CKL01, CLW03] the application of an abstraction involving a matching failure reduces to a special term representing *the failure* term. In the version of the calculus presented here, the delayed matching constraints whose corresponding matching problem has no solution are in normal form. The introduction of this kind of constraints in the syntax of the ρ-calculus in [CKL02] was the first step toward an explicit handling of the matching related computations studied in [CFK04] and briefly recalled in the last section of this chapter.

We denote by $\rightarrow_{\rho\sigma\delta}$ the relation induced by the top-level rules of the ρ-calculus and by $\mapsto_{\rho\sigma\delta}$, $\mapsto\!\!\!\!\rightarrow_{\rho\sigma\delta}$ and $=_{\rho\sigma\delta}$ the derived relations.

Example 22 (Reductions)

- *In the* ρ*-calculus with syntactic matching, the* ρ*-term* $c[a \ll b]$ *is in normal form, since the matching-equation* $a \prec\!\!\prec_{\emptyset} b$ *has no solution.*

- *In the* ρ*-calculus with syntactic matching, the* ρ*-term* $(head(x,y) \twoheadrightarrow x)\ head(a,b)$ *reduces using the* (ρ) *rule to the term* $x[head(x,y) \ll head(a,b)]$ *which can be evaluated using the rule* (σ) *to the term* a*.*

- *In the* ρ*-calculus parametrised with the associative and commutative theory, the* ρ*-term* $(+(x,y) \twoheadrightarrow y)\ +(a,b)$ *reduces to* $y[+(x,y) \ll +(a,b)]$ *and using the* (σ) *rule reaches its normal form* $b \wr a$*.*

As mentioned in the previous section, specific instances of the general ρ-calculus can be obtained adjusting conveniently its parameters. Two instances of the ρ-calculus which will be useful in the following of this thesis are presented next.

ρ-calculus with algebraic patterns

- $\mathcal{P}$ is the set of algebraic terms,
- the congruence on terms modulo which the matching is performed is the syntactic equality and,
- the theory associated to the structure operator is empty.

We obtain an instance where the set of patterns coincide with the algebraic terms, *i.e.* all the ρ-rules are of the form $f(t_1, \dots t_n) \twoheadrightarrow t$ with $t_1, \dots t_n$ algebraic, and thus ρ-rules naturally correspond to classical term rewrite rules, when also the term t is algebraic.

Since syntactic matching is decidable and unitary, all structures obtained as result of the application of a ρ-rule contain at most only one element and thus the structure operator is not needed in the right-hand side of the (σ)-rule. The structure operator is useful to represent the application of several rewrite rules to a term. For example, a rewrite step at the head position of a term $f(a)$ with respect to a rewrite system containing the rules $f(a) \twoheadrightarrow a$ and $f(a) \twoheadrightarrow b$ is represented by the term $(f(a) \twoheadrightarrow a \wr f(a) \twoheadrightarrow b)\ f(a)$. The result of its evaluation is the term $a \wr b$ which is the representation of the non-deterministic behavior of first-order rewriting, where we can apply one rewrite rule or the other obtaining either a or b.

This instance of the ρ-calculus is thus suitable to encode first-order term rewriting derivations, as described in the next section.

$\rho Cal_{\P}$

A more elaborated instance of the ρ-calculus, called $\rho Cal_{\P}$, will be needed in Chapter 5 to encode higher-order rewriting. In order to define the $\rho Cal_{\P}$, we introduce the notations β_ρ and η_ρ to denote the two reduction rules, written in the ρ-calculus syntax, corresponding to the β- and η-rules of the λ-calculus:

$$\begin{array}{llll} (\beta_\rho) & (x \twoheadrightarrow t_2)\ t_3 & \rightarrow_{\rho\beta} & \sigma(t_2) \quad \textit{where } \sigma = \{x/t_3\} = \mathit{Sol}(x \lessdot t_3) \\ (\eta_\rho) & x \twoheadrightarrow (t\ x) & \rightarrow_{\rho\eta} & t \end{array}$$

We denote by $=_{\lambda_\rho}$ the generated congruence relation.

The $\rho Cal_{\P}$ is an instance of the ρ-calculus where

- $\mathcal{P}$ is the set of ρ-patterns in (β_ρ)-normal form and,
- the theory $\mathbb{T}$ modulo which the matching is performed is the congruence relation $=_{\lambda_\rho}$ and,
- the theory associated to the structure operator is empty.

When restricting the left-hand sides of matching-equations to ρ-patterns, matching modulo $=_{\lambda_\rho}$ is denoted $\lessdot_{\P}$.

The patterns of the $\rho Cal_{\P}$ are more elaborated than simple algebraic patterns, since they can contain abstractions. The calculus can express and compute matching problems on functions

which are treated in the same way as higher-order matching problems in the λ-calculus, *i.e.* modulo $\beta\eta$. The decidability of $\rho Cal_{\P}$ matching problems in their general form is an open problem, but the restriction to matching problems of the form $p \prec\!\!\prec_{\P} t$, where t is the ρ-term corresponding to a λ-term, can be shown decidable and unitary by comparison with the λ-calculus. Indeed, when translating the two terms p and t in the λ-calculus, we obtain exactly an higher-order matching problem on λ-patterns, which is known to be decidable and unitary and can be solved using one of the well-known algorithms for λ-calculus higher-order matching, *e.g.* [Mil91]. For the purposes of this thesis, this restricted class of matching problems will be sufficient, in particular in Chapter 5 where the relationship with $\rho Cal_{\P}$ and the Combinatory Reduction Systems is analysed.

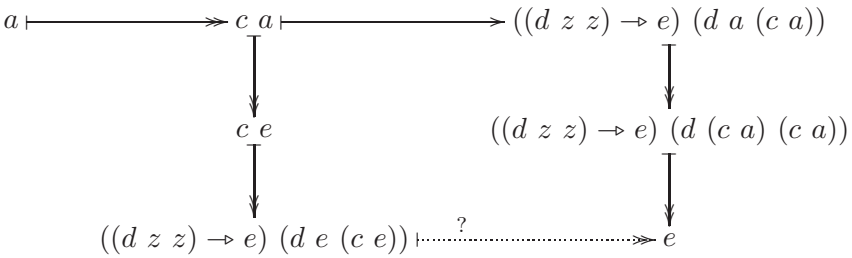
4.3 Properties and expressiveness

The ρ-calculus is not terminating, since it can express the λ-calculus and this one is not terminating. Confluence is not ensured either, in general. A number of examples can be found in Chapter 3 of [Cir00]. Nevertheless, some restrictions make the calculus confluent. For example, the set of possible reductions can be restricted to a subset of reductions, *i.e.* a strategy can be adopted to perform derivations [Cir00]. Another possibility in order to recover confluence is to restrict the set of patterns in the calculus to linear algebraic patterns [BCKL03]. The condition of linearity can not be dropped, as shown in next example [Wac03], originally given by J.W. Klop in the setting of the λ-calculus [Klo80].

Example 23 *We first define a ρ-term Y simulating a fixed point combinator exactly as in the λ-calculus (see Example 11) and therefore we have $Y\ t \mapsto\!\!\!\!\twoheadrightarrow_{\rho\sigma} t\ (Y\ t)$. Then we define the two terms*

$$\begin{aligned} c &\equiv Y\ \big(y \twoheadrightarrow x \twoheadrightarrow (d\ z\ z) \twoheadrightarrow e)\ (d\ x\ (y\ x))\big) \\ a &\equiv Y\ c \end{aligned}$$

and we have the following reductions:



The constant e is in normal form and the first abstraction in the term $((d\ z\ z) \twoheadrightarrow e)\ (d\ e\ (c\ e))$ can be eliminated only if $c\ e$ and e have a common reduct, due to the non linearity of the pattern $d\ z\ z$. We conclude by induction on the supposed length of a reduction from $c\ e$ to e that the two terms are not joinable.

As already mentioned, the ρ-calculus is expressive enough to generalise both λ-calculus and term rewriting. If the λ-calculus can be easily simulated by a particular instance of the ρ-calculus having the set of variables as set of patterns [Cir00], the simulation of term rewriting is not that direct. This is due to the fact that in the ρ-calculus rewrite rules are part of the term that gets rewritten and they are consumed during the derivation. Nevertheless, term rewriting derivations

can be encoded in the ρ-calculus choosing opportune ρ-terms that mimics the derivation, called derivation trace terms. We can define the general form of these terms using a notion of context for ρ-terms analogous to the one for first-order terms given in Definition 6.

Definition 56 (Derivation trace term) *Given a context* $\mathsf{Ctx}\{\Box\}$*, a derivation trace term is a* ρ*-term of the form* $\mathsf{Ctx}\{x\} \twoheadrightarrow \mathsf{Ctx}\{(l \twoheadrightarrow r)\ x\}$.

Given a term t and a rewrite rule $l \to r$ applied to t, in the ρ-calculus the derivation trace terms are used to make the rule application position explicit in t. Notice that the derivation trace term with an empty context corresponds to a rule application at the top position of t, since its application to t leads simply to $(l \twoheadrightarrow r)\ t$.

Example 24 (Addition) *In Example 4 we have shown a reduction in a term rewrite system R representing the addition on integers. The system can be represented in the ρ-calculus by the two rules $r_1 = 0 + x \twoheadrightarrow x$ and $r_2 = s(x) + y \twoheadrightarrow s(x + y)$, using an infix notation for the constant "+". The term rewrite reduction*

$$t_0 = s(0) + s(0) \ \mapsto_{r_2}\ s(0 + s(0)) = t_1 \ \mapsto_{r_1}\ s(s(0)) = t_2$$

can be obtained in the ρ-calculus as follows.

We first encode the step leading from t_0 to t_1. In order to do this, we apply the rule r_2 to the term $t_0 = s(0) + s(0)$ and we obtain:

$$\begin{array}{lcl} r_2\ t_0 & = & (s(x) + y \twoheadrightarrow s(x + y))\ (s(0) + s(0)) \\ & \mapsto_{\rho} & s(x + y)[s(x) + y \ll s(0) + s(0)] \\ & \mapsto_{\sigma} & s(x + y)\{x/0, y/s(0)\} = s(0 + s(0)) \\ & = & t_1 \end{array}$$

We proceed then by encoding the second step of the reduction leading from t_1 to t_2. To reduce t_1, we need to apply the rule r_1 to the subterm $0 + s(0)$. Differently from term rewriting, in the ρ-calculus the rule r_1 needs to be explicitly applied at the right position in the term, as a consequence of the explicit treatment of application in the calculus. We can do this by means of the derivation trace term $u = s(x) \twoheadrightarrow s(r_1\ x)$, that acts as a congruence rule and pushes the rule r_1 at the correct application position (under the head symbol s). We have

$$\begin{array}{lcl} u\ t_1 & = & (s(x) \twoheadrightarrow s(r_1\ x))\ (s(0 + s(0)) \\ & \mapsto^{*}_{\rho\sigma} & s(r_1\ (s(0 + s(0)))) \\ & \mapsto^{*}_{\rho\sigma} & s(s(0)) = t_2 \end{array}$$

The term r_2 can be seen as a particular case of derivation trace term in which the context is the empty context. The complete derivation can therefore be expressed starting from a unique ρ-term as follows $u\ (r_2\ t_0) \mapsto^{*}_{\rho\sigma} u\ (t_1) \mapsto^{*}_{\rho\sigma} t_2$.

The technique used in the previous example can be generalised to n rewrite rules subsequently applied at a given position in a term.

Theorem 9 ([CK01]) *Given a rewrite system R and two closed first order terms t_0 and t_n such that $t_0 \mapsto^{*}_{R} t_n$, then there exists some ρ-terms $u_0, \ldots, u_n$ such that $u_n\ (\ldots (u_0\ t_0) \ldots) \mapsto^{*}_{\rho\sigma} t_n$*

This method for simulating first-order term rewriting derivations will be used in the following chapter also for encoding higher-order derivations in the ρ-calculus.

Terms	A, B	::=	$\mathcal{X}$	(Variables)
		|	$\mathcal{K}$	(Constants)
		|	$A \twoheadrightarrow B$	(Abstraction)
		|	$A\ B$	(Functional application)
		|	$B\ \mathcal{C}$	(Constraint application)
		|	$A \wr B$	(Structure)
		|	$B\{X \ll A\}$	(Substitution application on terms)
Constraints	$\mathcal{C}, \mathcal{D}$	::=	$A \ll B$	(Match-equation)
		|	$\mathcal{C} \wedge \mathcal{D}$	(Conjunction of constraints)
		|	$\mathcal{C}\{X \ll A\}$	(Substitution application on constraints)

Figure 4.2: Syntax of $\rho_{\mathtt{x}}$

If we are interested in finding a possible derivation for a term t in the ρ-calculus without knowing *a priori* the reduction in the term rewrite system, we can use the approach based on fix points proposed in [CLW03] for the construction of an appropriate ρ-term only from a given set of rules.

Theorem 10 ([CLW03]) *Given a confluent and terminating rewrite system R, there exists a ρ-term u such that for all algebraic terms t, t' we have $u\ t \mapsto\!\!\!\!\twoheadrightarrow_{\rho\mathcal{S}} t'$ if and only if $t \mapsto\!\!\!\!\twoheadrightarrow_R t'$.*

4.4 The $\rho_{\mathtt{x}}$-calculus

A version of the ρ-calculus which is more suitable for being a theoretical back-end for implementation is the $\rho_{\mathtt{x}}$-calculus, introduced by H. Cirstea, G. Faure and C. Kirchner in [CFK04] with the aim of defining a extension of the ρ-calculus that handles explicitly matching and substitution application.

The syntax of the $\rho_{\mathtt{x}}$-calculus, presented in Figure 4.2, is an extension of that of the ρ-calculus in order to deal with explicit decomposition of matching equations and explicit treatment of substitutions. Constraints become first-class objects of the calculus and are conjunctions of matching problems, built using the operator "$\_ \wedge \_$" which is supposed to be associative, commutative, and idempotent. The usual application operator denoted by concatenation is extended to constraint application and an additional operator "$\_\{\_\}$" is introduced for denoting the application of substitutions to terms and constraints. The priorities for the operators are the same as in the ρ-calculus. As far as the new operator "$\_\{\_\}$" is concerned, it is assumed to be of greater priority than the constraint application operator.

We mention that two of the parameters of the ρ-calculus are still suitable for instantiation in the $\rho_{\mathtt{x}}$-calculus, namely the set of patterns and the theory underlying the "$\_\wr\_$" operator, but the treatment of matching at the object level imposes a choice *a priori* on the theory modulo which the matching is performed. At the current state, the set of evaluation rules of the $\rho_{\mathtt{x}}$-calculus is adapted to deal with syntactical matching. On the other hand, the theory associated to the conjunction operator for constraints can be seen as an additional parameter. However, to have an expressive and well-behaving calculus, the choices for this parameter are restricted. Usually, an associative-commutative and idempotent theory is used, in such a way that the order of the constraint in a term is not important and that the duplication of constraints is avoided.

In order to handle conveniently the new kind of constraints and the explicit substitution, several new rules need to be added with respect to the ρ-calculus semantics. The complete set

of $\rho_{\mathtt{x}}$-calculus evaluation rules is shown in Figure 4.3 and can be split into three categories: rules describing the application of structures and abstractions on ρ-terms (inherited from the ρ-calculus), rules that describe the solving of the matching problems in empty theory and rules defining the application of substitutions.

As for the version of the ρ-calculus presented in this chapter, in the $\rho_{\mathtt{x}}$-calculus the application of a rewrite rule (abstraction) to a term is always evaluated to a *delayed matching constraint* applied to the right-hand side of the rule. A constraint application is then transformed, if possible, into a collection of substitution applications using the *Constraint computation* set of rules. To go from constraints to substitutions, a constraint is simplified (using decomposition rules) until a sub part of the constraint is of the form $X \ll A$. If this non-decomposable constraint is independent of the remaining part of the constraint, we can "push it out" of the constraint and trigger its application as a substitution. The application of the substitution to a term is then treated explicitly using the *Substitution application* rules.

Example 25 (Application of a rewrite rule) *We describe the reduction in the $\rho_{\mathtt{x}}$-calculus of the application of the rewrite rule $r_2 = s(x) + y \twoheadrightarrow s(x + y)$ of Example 24 to the term $t_0 = s(0) + s(0)$.*

Term application

(ρ)	$(A \twoheadrightarrow B)\ C$	$\to$	$B\ (A \ll C)$
(δ)	$(A; B)\ C$	$\to$	$A\ C; B\ C$

Constraint computation

Decomposition

$(Decompose_{\wr})$	$A_1 \wr A_2 \ \ll\ B_1 \wr B_2$	$\to$	$A_1 \ll B_1 \wedge\ A_2 \ll B_2$
$(Decompose_{\mathcal{F}})$	$f(A_1, \ldots, A_n) \ll f(B_1, \ldots, B_n)$	$\to$	$A_1 \ll B_1 \wedge \ldots \wedge A_n \ll B_n$

Constraint application

$(ToSubst_{\wedge})$	$B(X \ll A \wedge \mathcal{C})$	$\to$	$B\{X \ll A\}\mathcal{C}$ if $X \notin \mathcal{D}om(\mathcal{C})$
$(ToSubst_{\ll})$	$B(X \ll A)$	$\to$	$B\{X \ll A\}$
$(ToSubst_{Id})$	$B(a \ll a)$	$\to$	B

Substitution application

$(Replace)$	$X\{X \ll A\}$	$\to$	A
$(Eliminate_{\mathcal{X}})$	$Y\{X \ll A\}$	$\to$	Y if $X \neq Y$
$(Eliminate_{\mathcal{F}})$	$f\{X \ll A\}$	$\to$	f
$(Share\ )$	$(B\ C)\{X \ll A\}$	$\to$	$B\{X \ll A\}\ C\{X \ll A\}$
$(Share_{\wr})$	$(B \wr C)\{X \ll A\}$	$\to$	$B\{X \ll A\} \wr C\{X \ll A\}$
$(Share_{\twoheadrightarrow})$	$(B \twoheadrightarrow C)\{X \ll A\}$	$\to$	$B \twoheadrightarrow (C\{X \ll A\})$
$(Share_{\ll})$	$(B \ll C)\{X \ll A\}$	$\to$	$B \ll C\{X \ll A\}$
$(Share_{\wedge})$	$(\mathcal{C} \wedge \mathcal{D})\{X \ll A\}$	$\to$	$\mathcal{C}\{X \ll A\} \wedge\ \mathcal{D}\{X \ll A\}$

Figure 4.3: Small-step semantics of the $\rho_{\mathtt{x}}$

$$
\begin{array}{lll}
 & (s(x)+y \twoheadrightarrow s(x+y))\ s(0)+s(0) & \\
\mapsto_{\rho} & s(x+y)\ (s(x)+y \ll s(0)+s(0)) & \\
\mapsto\!\!\!\!\mapsto_{Decompose_{\mathcal{F}}} & s(x+y)\ (x \ll 0 \wedge y \ll 0) & \\
\mapsto_{ToSubst_{\wedge}} & s(x+y)\{x \ll 0\}\ (y \ll 0) & \\
\mapsto_{ToSubst_{\ll}} & s(x+y)\{x \ll 0\}\{y \ll 0\} & \\
\mapsto\!\!\!\!\mapsto & s(0+y)\{y \ll 0\} & \textit{Substitution application} \\
\mapsto\!\!\!\!\mapsto & s(0+0) & \textit{Substitution application}
\end{array}
$$

We conclude by saying that the $\rho_{\mathtt{x}}$-calculus is an explicit calculus which enjoys the usual good properties of explicit substitutions (conservativity, termination). Moreover, when the set of patterns is restricted to algebraic linear patterns, the $\rho_{\mathtt{x}}$-calculus enjoy also the confluence property.

Conclusion

We presented in this chapter the ρ-calculus, a rewriting formalism combining higher-order capabilities with pattern matching features. The ρ-calculus has already been shown to be an expressive framework, able to simulate first-order rewriting, λ-calculus and object calculi. In the next chapter, we will prove that higher-order rewriting can be encoded in the ρ-calculus as well.

We briefly presented also a version of the ρ-calculus, called $\rho_{\mathtt{x}}$-calculus, dealing with explicit substitution and computation of matching constraints at the object level. Some ideas of this explicit calculus have been adapted for the definition in Chapter 7 of the graph rewriting calculus, a new version of the rewriting calculus enriched with terms having a graph-like structure. We can anticipate here that graph rewriting calculus terms are ρ-terms in which matching constraints are generalised to unification constraints handled using an appropriate set of evaluation rules. As far as the treatment of matching constraints is concerned, a set of rules inspired from the *Constraint computation* rules of the $\rho_{\mathtt{x}}$-calculus has been adopted in the graph rewriting calculus semantics.

Chapter 5

Expressing CRS in the ρ-calculus

Once a new calculus has been defined, it is useful to place it in the panorama of the already existing calculi with similar features and applications, in order to better understand its expressive power and efficiency with respect to other calculi. The rewriting calculus has already been shown to be a generalisation of λ-calculus and first-order term rewriting and, more generally, to have an interesting potential in expressing easily the usual computational formalisms. Its nature and its higher-order capabilities lead to a natural comparison with higher-order rewriting.

This chapter is thus concerned with the study of the relation between the rewriting calculus and higher-order rewrite systems, in particular with the *Combinatory Reduction Systems* defined in Section 2.3. CRSs, which are historically the first higher-order rewrite systems, have been source of inspiration for the numerous following works on higher-order rewriting. Our comparison of the ρ-calculus with CRSs is intended to analyse the behavior of these systems, with the aim of better understanding how the rewrite rules are applied to terms and how term reductions are performed in the CRSs, taking advantage also of the study on CRS matching developed in Chapter 3. To this purpose, we define a translation of the various CRS concepts, like terms, assignments, *etc.* to the corresponding notions in the ρ-calculus, and, using this translation, we propose a method that associates to every derivation of a CRS-term a term with a corresponding derivation in ρ-calculus.

5.1 The translation

We propose in this section a translation of CRS components into ρ-terms. We recall that, as we have seen in Section 2.3, the assignment application used for performing term reductions in CRSs (and thus the matching the CRS-reduction relies on) is based on λ-calculus. Consequently, to shallowly encode CRSs into the rewriting calculus the target ρ-calculus should use a matching congruence more complex than the syntactic one. For this reason, we choose the $\rho Cal_{\P}$ described in Section 4.2.0.0 as target calculus.

In the following we suppose that the set of constants of $\rho Cal_{\P}$ is the same as the set of functional symbols of CRSs and the set of variables of $\rho Cal_{\P}$ is the union of the sets of variables and metavariables of CRSs.

Definition 57 (Translation) *The translation $\mathcal{E}$ of a* CRS*-metaterm t into a term of $\rho Cal_{\P}$, denoted $\overline{t}$, is inductively defined as follows:*

- CRS*-metaterms into ρ-terms*

$$\begin{array}{lcl} \overline{x} & = & x \\ \overline{[x]t} & = & x \twoheadrightarrow \overline{t} \end{array} \qquad\qquad \begin{array}{lcl} \overline{f(t_1,\ldots,t_n)} & = & f(\overline{t_1},\ldots,\overline{t_n}) \\ \overline{Z(t_1,\ldots,t_n)} & = & Z\ \overline{t_1}\ \ldots\ \overline{t_n} \end{array}$$

- CRS *rewrite rules into ρ-terms:*

$$\overline{L \to R} \;=\; \overline{L} \twoheadrightarrow \overline{R}$$

- CRS *substitutes into ρ-terms:*

$$\overline{\underline{\lambda} x_1 \dots x_n.u} \;=\; x_1 \twoheadrightarrow (x_2 \twoheadrightarrow (\dots (x_n \twoheadrightarrow \overline{u}) \dots))$$

- CRS *assignments into ρ-substitutions:*

$$\overline{\{\dots, (Z, \xi), \dots\}} \;=\; \{\dots, Z/\overline{\xi}, \dots\}$$

One can notice that the CRS-abstraction operator "[_]_" is translated into a ρ-abstraction. Moreover, since in ρ-calculus rewrite rules are first class objects, a CRS rewrite rule is translated into a ρ-term and more precisely into a ρ-rule. The translation of the abstraction operator and of the rewrite rules of CRSs into the same abstraction operator of ρ-calculus corresponds to the uniform treatment of first and higher-order rewriting in the ρ-calculus. An n-ary CRS-metavariable (function) corresponds in $\rho Cal_{\P}$ to a variable (constant) applied to n ρ-terms.

The $\underline{\lambda}$-abstraction operator defined at the meta-level of CRS is also translated into the ρ-abstraction operator "_ ↠ _". This means that reductions performed in CRS at the meta-level (using $\underline{\lambda}$) correspond in $\rho Cal_{\P}$ to explicit reductions corresponding to the application of the abstraction operator "_ ↠ _".

Example 26 *We have already seen how the β-rule of λ-calculus can be translated into a* CRS*-rule (see Example 15). When translating this* BetaCRS *rule into the rewriting calculus the following term is obtained:*

$$\overline{BetaCRS} = \overline{App(Ab([x]Z(x)), Z_1) \to Z(Z_1)} = App(Ab(x \twoheadrightarrow (Z\ x)), Z_1) \twoheadrightarrow Z\ Z_1$$

where $App, Ab \in \mathcal{K}$ *and* $x, Z, Z_1 \in \mathcal{X}$.

Since there is no application symbol in the syntax of CRSs, the CRS-rules are never directly applied to a CRS-term. Nevertheless the CRS-rule is translated into a ρ-abstraction ensuring that the corresponding variables are bound in the translation and thus, that the pattern condition is preserved by the translation.

Proposition 7 *Given a* CRS*-metaterm* L*, if* L *is a* CRS*-pattern, then* $\overline{L}$ *is a ρ-pattern.*

Moreover, as a consequence of the definition of the translation, the ρ-terms obtained from CRS-metaterms contain no redexes and thus, are in normal form.

The position ω of a sub-metaterm B in a CRS-metaterm $A_{[B]_\omega}$ and the position of its translation $\overline{B}$ in the ρ-term $\overline{A}$ are not the same. This is due to the different use of (meta)variables and functional symbols in CRS-terms and ρ-terms. We use for the position transformations in a ρ-term a function π_ρ defined similarly to the function π_λ in Section 3.2 (Definition 50).

Definition 58 *Let* A *be a* CRS*-metaterm and* $\omega \in \mathcal{P}os(A)$ *a position in* A*. The function* π_ρ *is inductively by:*

- $\pi_\rho(\epsilon, A) = \epsilon$,
- $\pi_\rho((n-i).\omega, Z(A_1, \dots, A_n)) = 1^i.2.\pi_\rho(\omega, A_{n-i})$, *where* $i \in \{0, \dots, n-1\}$,
- $\pi_\rho((n-i).\omega, f(A_1, \dots, A_n)) = 1^i.2.\pi_\rho(\omega, A_{n-i})$, *where* $i \in \{0, \dots, n-1\}$,
- $\pi_\rho(1.\epsilon, [x]A) = 1.\pi_\rho(\epsilon, x)$,
- $\pi_\rho(2.\omega, [x]A) = 2.\pi_\rho(\omega, A)$.

5.2 Examples

In what follows we give some examples of CRS-reductions as well as the corresponding reductions in $\rho Cal_{\P}$. We show in the next section that all CRS-derivations can be expressed in the rewriting calculus using the translation we have proposed above.

Example 27 *We consider the* CRS*-reduction* $f(App(Ab([x]f(x)), a)) \mapsto\!\!\!\!\rightarrow_{BetaCRS} f(f(a))$ *shown in Example 16. We should point out that in this reduction the* CRS*-rule is not applied at the head position of the considered term but deeper in the term. In the rewriting calculus the application operator is explicit and thus, to obtain a similar reduction in* $\rho Cal_{\P}$*, we have to use a derivation trace term (see Definition 56) that pushes the application of the rule we want to apply down in the term. Therefore, we apply the* ρ*-term* $f(y) \twoheadrightarrow f(\overline{BetaCRS}\ y)$ *(with BetaCRS defined in Example 15) to the translation of the initial* CRS*-term and we obtain the following reduction:*

$$
\begin{array}{ll}
& (f(y) \twoheadrightarrow f(\overline{BetaCRS}\ y))\ (\overline{f(App(Ab([x]f(x)), a))}) \\
\mapsto_{\rho} & f(\overline{BetaCRS}\ y)[f(y) \ll f(App(Ab(x \twoheadrightarrow f(x)), a))] \\
\mapsto_{\sigma} & f(\overline{BetaCRS}\ App(Ab(x \twoheadrightarrow f(x)), a)) \\
= & f((App(Ab(x \twoheadrightarrow (Z\ x)), Z_1) \twoheadrightarrow Z\ Z_1)\ App(Ab(x \twoheadrightarrow f(x)), a)) \\
\mapsto_{\rho} & f(Z\ Z_1[App(Ab(x \twoheadrightarrow (Z\ x)), Z_1) \ll App(Ab(x \twoheadrightarrow f(x)), a)] \\
\mapsto_{\sigma} & f(Z\ Z_1\{Z/z \twoheadrightarrow f(z), Z_1/a\}) = f((z \twoheadrightarrow f(z))\ a) \\
\mapsto_{\rho} & f(f(z))[z \ll a] \\
\mapsto_{\sigma} & f(f(a))
\end{array}
$$

One can notice in Example 27 that the reductions in CRSs and in the rewriting calculus lead to the same final term, modulo the translation, but we do not have a one-to-one correspondence between the rewrite steps. In the previous example the steps from $f((z \twoheadrightarrow f(z))\ a)$ to $f(f(a))$ are explicit only in ρ-calculus. The same behavior is obtained in the following (more elaborated) example.

Example 28 (Summation) *We consider the finite sum* $\Sigma_{i=0}^{n} s(i)$ *where* n *is a natural number and* s *the successor function. It can be expressed as the following* CRS *rewrite rules set:*

$$
\begin{array}{rcll}
\mathcal{R} & = & \{ & S_0 : a(0, Y) \rightarrow Y\ ; \\
& & & S_1 : a(s(X), Y) \rightarrow s(a(X, Y))\ ; \\
& & & Rec_0 : \Sigma(0, [x]F(x)) \rightarrow F(0)\ ; \\
& & & Rec_1 : \Sigma(s(N), [x]F(x)) \rightarrow a(\Sigma(N, [x]F(x)), F(s(N)))\ \}
\end{array}
$$

where $0 \in \mathcal{F}_0$*,* $s \in \mathcal{F}_1$*,* $\Sigma, a \in \mathcal{F}_2$*,* $X, Y, N \in \mathcal{Z}_0$ *and* $F \in \mathcal{Z}_1$*. The* CRS*-rules* S_0 *and* S_1 *express the addition of natural numbers while the* CRS*-rules* Rec_0 *and* Rec_1 *express the summation in a recursive way.*

We consider the sum

$$\Sigma_{i=0}^{s(0)} s(i) = s(0) + s(s(0)) = s(s(s(0)))$$

The corresponding CRS*-term is* $t = \Sigma(s(0), [i]s(i))$*. We show next that reducing* t *we obtain the expected result* $s(s(s(0)))$*. We denote by* L_i *and* R_i *the left-hand side and the right-hand side of*

the rule Rec_i, for $i = 0, 1$. To perform the reduction of t we first apply the rule Rec_1 with the assignment $\sigma_1 = \{(N, 0), (F, \underline{\lambda}y.s(y))\}$. We have

$$\begin{array}{rcl} \sigma_1(L_1) & = & \sigma_1\ (\Sigma(s(N), [x]F(x))) \\ & = & \Sigma(s(\sigma_1(N)), [x]\sigma_1(F(x))) \\ & = & \Sigma(s(0), [x](\underline{\lambda}y.s(y))(x))\downarrow_{\underline{\beta}} \\ & = & \Sigma(s(0), [x]s(x)) \end{array}$$

and thus, modulo α-conversion, $\sigma_1(L_1) = t$. When instantiating the right-hand side we obtain

$$\begin{array}{rcl} \sigma_1(R_1) & = & \sigma_1\ (a(\Sigma(N, [x]F(x)), F(s(N)))) \\ & = & a(\Sigma(\sigma_1(N), [x](\sigma_1(F))(x)), (\sigma_1(F))(s(\sigma_1(N)))) \\ & = & a(\Sigma(0, [x](\underline{\lambda}y.s(y))(x)), (\underline{\lambda}y.s(y))(s(0)))\downarrow_{\underline{\beta}} \\ & = & a(\Sigma(0, [x]s(x)), s(s(0))) \end{array}$$

Next we apply the CRS*-rule Rec_0 to the sub-term $t_0 = \Sigma(0, [x]s(x))$ since the assignment $\sigma_0 = \{(F, \underline{\lambda}y.s(y))\}$ is solution of the matching between L_0 and t_0:*

$$\sigma_0(L_0) = \sigma_0\ (\Sigma(0, [x]F(x))) = \Sigma(0, [x]\sigma_0(F(x))) = \Sigma(0, [x](\underline{\lambda}y.s(y))(x))\downarrow_{\underline{\beta}} = \Sigma(0, [x]s(x)) = t_0$$

When applying σ_0 to the right-hand side of the rule we obtain

$$\sigma_0(R_0) = \sigma_0(F(0)) = (\sigma_0(F))(0) = (\underline{\lambda}y.s(y))(0)\ \rightarrow_{\underline{\beta}}\ s(0)$$

and therefore, the following reduction is obtained for the whole CRS*-term t:*

$$t \mapsto_{Rec_1} a(\Sigma(0, [x]s(x)), s(s(0))) \mapsto_{Rec_0} a(s(0), s(s(0)))$$

The last steps of the reduction follow easily using the rules S_1 and S_0:

$$a(s(0), s(s(0))) \mapsto_{S_1} s(a(0, s(s(0)))) \mapsto_{S_0} s(s(s(0)))$$

Now, we translate into $\rho Cal_{\P}$ the set $\mathcal{R}$ of CRS *rewrite rules:*

$$\begin{array}{rcl} \overline{S_0} & = & a(0, Y) \twoheadrightarrow Y \\ \overline{S_1} & = & a(s(X), Y) \twoheadrightarrow s(a(X, Y)) \\ \overline{Rec_0} & = & \Sigma(0, x \twoheadrightarrow (F\ x)) \twoheadrightarrow F\ 0 \\ \overline{Rec_1} & = & \Sigma(s(N), x \twoheadrightarrow (F\ x)) \twoheadrightarrow a(\Sigma(N, x \twoheadrightarrow (F\ x)), F\ s(N)) \end{array}$$

and the CRS*-term t:*

$$\overline{t} = \Sigma(s(0), i \twoheadrightarrow s(i))$$

We proceed as in Example 30 and starting from the reduction shown above we build the ρ-terms

$$u_1 = \overline{Rec_1} \text{ and } u_2 = (a(x, y) \twoheadrightarrow a(\overline{Rec_0}\ x,\ y))$$

corresponding to the application of the rules $\overline{Rec_1}$ and $\overline{Rec_0}$ respectively and we build the ρ-term

$$u_2\ (u_1\ \overline{t})$$

We first reduce the application of $\overline{Rec_1}$ to $\overline{t}$ using the substitution $\overline{\sigma_1} = \{N/0,\ F/y \twoheadrightarrow s(y)\}$:

$$\begin{array}{rcl} \overline{\sigma_1}\ (a(\Sigma(N, x \twoheadrightarrow (F\ x)), F\ s(N))) & = & a(\Sigma(0, x \twoheadrightarrow (y \twoheadrightarrow s(y))\ x), (y \twoheadrightarrow s(y))\ s(0)) \\ & \mapsto_{\rho\beta\delta} & a(\Sigma(0, x \twoheadrightarrow s(x)), s(s(0))) \end{array}$$

The term u_2 is then applied to this intermediate result:

$$\begin{array}{rcl} u_2\ a(\Sigma(0, x \twoheadrightarrow s(x)), s(s(0))) & = & (a(x,y) \twoheadrightarrow a(\overline{Rec_0}\ x, y))\ a(\Sigma(0, x \twoheadrightarrow s(x)), s(s(0))) \\ & \mapsto_{\rho\mathfrak{Cal}} & a(\overline{Rec_0}\ \Sigma(0, x \twoheadrightarrow s(x)), s(s(0))) \end{array}$$

The substitution $\overline{\sigma_2} = \{F/y \twoheadrightarrow s(y)\}$ is used for the application of the rule $\overline{Rec_0}$ and since

$$\overline{\sigma_2}(F\ 0) = (y \twoheadrightarrow s(y))\ 0\ \mapsto_{\rho\mathfrak{Cal}}\ s(0)$$

we obtain

$$u_2\ (u_1\ \overline{t}) \mapsto\!\!\!\!\!\rightarrow_{\rho\mathfrak{Cal}} a(\Sigma(0, x \twoheadrightarrow s(x)), s(s(0))) \mapsto\!\!\!\!\!\rightarrow_{\rho\mathfrak{Cal}} a(s(0), s(s(0)))$$

Finally, applying the (first-order) rewrite rules $\overline{S_0}$ and $\overline{S_1}$ yields the same final result as in CRS*:*

$$\overline{S_0}\ (\overline{S_1} a(s(0), s(s(0)))) \mapsto\!\!\!\!\!\rightarrow_{\rho\mathfrak{Cal}} \overline{S_0}\ s(a(0, s(s(0))) \mapsto\!\!\!\!\!\rightarrow_{\rho\mathfrak{Cal}} s(s(s(0)))$$

We consider now a CRS with a set of rewrite rules that express the computation of primitive recursive functions and we apply this set of rules to the function modelling the addition on natural numbers.

Example 29 (Recursion) *Given a* CRS *with the following set of rewrite rules $\mathcal{R}$:*

$$\begin{array}{rcll} \mathcal{R} & = & \{ & R_0 : rec(0, A, [x][y]F(x,y)) \to A\ ; \\ & & & R_1 : rec(s(N), A, [x][y]F(x,y)) \to F(N, rec(N, A, [x][y]F(x,y)))\ \} \end{array}$$

where $rec \in \mathcal{F}_3$, $s \in \mathcal{F}_1$, $A, N \in \mathcal{Z}_0$ and $F \in \mathcal{Z}_2$.

We consider the CRS*-term $t = rec(s(0), s(s(0)), [x][y]s(y))$ representing the addition of the two natural numbers $s(0)$ and $s(s(0))$ and we reduce the term t to the final result $s(s(s(0)))$. For this, we first apply to the* CRS*-term t the* CRS *rewrite rule R_1 using the assignment $\sigma_1 = \{(N,0), (A, s(s(0)), (F, \underline{\lambda} z_1 z_2.s(z_2))\}$ and we obtain*

$$\begin{array}{rcl} t & \mapsto_{R_1} & \sigma_1\ (F(N, rec(N, A, [x][y]F(x,y)))) \\ & = & (\sigma_1(F))(\sigma_1(N), rec(\sigma_1(N), \sigma_1(A), [x][y]\sigma_1(F(x,y))) \\ & = & (\underline{\lambda} z_1 z_2.s(z_2))(0, rec(0, s(s(0)), [x][y](\underline{\lambda} z_1 z_2.s(z_2))(x,y))\downarrow_{\underline{\beta}} \\ & = & (\underline{\lambda} z_1 z_2.s(z_2))(0, rec(0, s(s(0)), [x][y]s(y))\downarrow_{\underline{\beta}} \\ & = & s(rec(0, s(s(0), [x][y]s(y)))) \end{array}$$

Next we proceed by applying the rule R_0 to the sub-term $rec(0, s(s(0)), [x][y]s(y)))$ with the assignment $\sigma_0 = \{(A, s(s(0)), (F, \underline{\lambda} z_1 z_2.s(z_2))\}$ and we obtain

$$s(rec(0, s(s(0), [x][y]s(y)))) \mapsto_{R_0} s(s(s(0)))$$

We translate now the example in $\rho Cal_{\P}$ where the following two ρ-terms correspond to the two CRS *rewrite rules*

$$\begin{array}{rcl} \overline{R_0} & = & rec(0, A, x \twoheadrightarrow (y \twoheadrightarrow (F(x,y)))) \twoheadrightarrow A \\ \overline{R_1} & = & rec(s(N), A, x \twoheadrightarrow (y \twoheadrightarrow (F(x,y)))) \twoheadrightarrow F(N, rec(N, A, x \twoheadrightarrow (y \twoheadrightarrow (F(x,y))))) \end{array}$$

The CRS*-term t is translated into the following ρ-term:*

$$\overline{t} = rec(s(0), s(s(0), x \twoheadrightarrow (y \twoheadrightarrow s(y))))$$

As in the previous examples, starting from the CRS *reduction presented above, we build the following ρ-terms corresponding to the application of the rules* $\overline{R_1}$ *and* $\overline{R_0}$ *rules.*

$$u_1 = \overline{R_1} \text{ and } u_2 = (s(x) \twoheadrightarrow s(\overline{R_0}\ x))$$

and we build the ρ-term: $u_2\ (u_1\ \overline{t})$*. We start reducing this ρ-term by applying the ρ-rule* $\overline{R_1}$ *to* $\overline{t}$ *using the substitution* $\overline{\sigma_1} = \{F/z_1 \twoheadrightarrow z_2 \twoheadrightarrow s(z_2),\ N/0,\ A/s(s(0))\}$ *and we obtain:*

$$\begin{array}{rl} & \overline{\sigma_1}(F(N, rec(N, A, x \twoheadrightarrow (y \twoheadrightarrow (F(x,y)))))) \\ = & (\overline{\sigma_1}(F))\ (\overline{\sigma_1}(N), rec(\overline{\sigma_1}(N), \overline{\sigma_1}(A), x \twoheadrightarrow (y \twoheadrightarrow ((\overline{\sigma_1}(F))(x,y))))) \\ = & (z_1 \twoheadrightarrow z_2 \twoheadrightarrow s(z_2))\ (0, rec(0, s(s(0)), x \twoheadrightarrow (y \twoheadrightarrow ((z_1 \twoheadrightarrow z_2 \twoheadrightarrow s(z_2))\ x\ y))) \\ \mapsto_{\rho\beta\delta} & (z_1 \twoheadrightarrow z_2 \twoheadrightarrow s(z_2))\ (0, rec(0, s(s(0)), x \twoheadrightarrow (y \twoheadrightarrow s(y)))) \\ \mapsto_{\rho\beta\delta} & s(rec(0, s(s(0)), x \twoheadrightarrow (y \twoheadrightarrow s(y)))) \end{array}$$

When reducing the application of u_2 *to this intermediate result we obtain*

$$\begin{array}{rl} & (s(x) \twoheadrightarrow s(\overline{R_0}\ x))\ s(rec(0, s(s(0)), x \twoheadrightarrow (y \twoheadrightarrow s(y)))) \\ \mapsto_{\rho\beta\delta} & s(\overline{R_0}\ (rec(0, s(s(0)), x \twoheadrightarrow (y \twoheadrightarrow s(y))))) \end{array}$$

Finally we apply $\overline{R_0}$ *using the substitution* $\overline{\sigma_0} = \{F/z_1 \twoheadrightarrow z_2 \twoheadrightarrow s(z_2),\ A/s(s(0))\}$ *and we obtain the expected result.*

$$s(\overline{R_0}\ (rec(0, s(s(0)), x \twoheadrightarrow (y \twoheadrightarrow s(y))))) \mapsto\!\!\!\!\rightarrow_{\rho\beta\delta} s(s(s(0)))$$

In the next example we consider a CRS which extends the λ-calculus with the reduction rules for "Surjective Pairing". This yields to a CRS with four rewrite rules, one of which non-linear (the P rewrite rule where the metavariable X appears twice in the left-hand side).

Example 30 (λ-calculus with surjective pairing) *Given a* CRS *with the following set of rewrite rules* $\mathcal{R}$*:*

$$\begin{array}{rcll} \mathcal{R} & = & \{ & P_0 : \Pi_0(\Pi(X_1, X_2)) \to X_1, \\ & & & P_1 : \Pi_1(\Pi(X_1, X_2)) \to X_2, \\ & & & P : \Pi(\Pi_0 X, \Pi_1 X) \to X, \\ & & & \mathit{BetaCRS}\ \} \end{array}$$

where $X_1, X_2 \in \mathcal{Z}_0$*,* $\Pi \in \mathcal{F}_2$ *(pair function),* $\Pi_0, \Pi_1 \in \mathcal{F}_1$ *(projections) and BetaCRS as in Example 15.*

We consider the CRS*-term* $t = App(Ab([z]\Pi(\Pi_1 z, \Pi_0 z)), \Pi(x_1, x_2))$*. Intuitively, we can see the term* t *as the application of the function that swaps the elements of a pair* $Ab([z]\Pi(\Pi_1 z, \Pi_0 z))$ *to the pair* $\Pi(x_1, x_2)$*). One can check that the assignment* $\sigma = \{(Z, \underline{\lambda} z.\Pi(\Pi_1 z, \Pi_0 z), (Z_1, \Pi(x_1, x_2))\}$ *can be used to reduce the* CRS*-term* t *using the rule* BetaCRS*:*

$$\sigma(Z(Z_1)) = (\sigma(Z))(\sigma(Z_1)) = (\underline{\lambda} z.\Pi(\Pi_1 z, \Pi_0 z))(\Pi(x_1, x_2)) \downarrow_{\underline{\beta}} = \Pi(\Pi_1(\Pi(x_1, x_2)), \Pi_0(\Pi(x_1, x_2)))$$

Next we can apply the rules P_1 *and* P_0 *to the first and second arguments of* Π *respectively, using the assignment* $\sigma' = \{(X_1, x_1), (X_2, x_2)\}$ *and we obtain the final result:*

$$\Pi(\Pi_1(\Pi(x_1, x_2)), \Pi_0(\Pi(x_1, x_2))) \mapsto_{P_1} \Pi(x_2, \Pi_0(\Pi(x_1, x_2))) \mapsto_{P_0} \Pi(x_2, x_1)$$

When we translate the above set of CRS *rewrite rules into* $\rho Cal_{\P}$ *we obtain*

$$\begin{array}{rcl} \overline{P_0} & = & \Pi_0(\Pi(X_1, X_2)) \twoheadrightarrow X_1 \\ \overline{P_1} & = & \Pi_1(\Pi(X_1, X_2)) \twoheadrightarrow X_2 \\ \overline{P} & = & \Pi(\Pi_0 X_1, \Pi_1 X_1) \twoheadrightarrow X_1 \\ \overline{\mathit{BetaCRS}} & = & App(Ab(x \twoheadrightarrow (Z\ x)), Z_1) \twoheadrightarrow Z\ Z_1 \end{array}$$

and the translation of the initial CRS*-term* t *is:*

$$\overline{t} = App(Ab(z \twoheadrightarrow \Pi(\Pi_1 z, \Pi_0 z)), \Pi(x_1\ x_2))$$

Starting from the CRS *reduction above, we build the* ρ*-terms* u_1 *and* u_2 *corresponding to the applications of the rules* $\overline{BetaCRS}$ *and* $\overline{P_0}, \overline{P_1}$ *respectively. As in Example 27 we have to use two derivation trace terms to simulate the application of the last two rules and we obtain the* ρ*-terms*

$$u_1 = \overline{BetaCRS} \text{ and } u_2 = \Pi(x, y) \twoheadrightarrow \Pi(\overline{P_1}\ x, \overline{P_0}\ y)$$

The ρ*-terms* u_1, u_2 *are called* derivation trace terms *(see Theorem 11) and are used to apply the translation of the* CRS*-rules at the correct positions in the term. Starting from these terms we build the* ρ*-term* $u_2\ (u_1\ \overline{t})$*.*

When reducing the term $u_1\ \overline{t}$ *we use the substitution* $\overline{\sigma} = \{Z/z \twoheadrightarrow \Pi(\Pi_1 z, \Pi_0 z), Z_1/\Pi(x_1\ x_2)\}$ *and since*

$$\begin{array}{rcl} \overline{\sigma}(Z\ Z_1) & = & \overline{\sigma}(Z)\ \overline{\sigma}(Z_1)\ =\ (z \twoheadrightarrow \Pi(\Pi_1 z, \Pi_0 z))\ \Pi(x_1, x_2) \\ & \mapsto_{\rho\sigma} & \Pi(\Pi_1(\Pi(x_1, x_2)), \Pi_0(\Pi(x_1, x_2))) \end{array}$$

we obtain

$$\begin{array}{rcl} u_1\ \overline{t} & = & (App(Ab(x \twoheadrightarrow (Z\ x)), Z_1) \twoheadrightarrow Z\ Z_1)\ App(Ab(z \twoheadrightarrow \Pi(\Pi_1 z, \Pi_0 z)), \Pi(x_1, x_2)) \\ & \mapsto_{\rho} & Z\ Z_1[App(Ab(x \twoheadrightarrow (Z\ x)), Z_1)) \ll App(Ab(z \twoheadrightarrow \Pi(\Pi_1 z, \Pi_0 z)), \Pi(x_1, x_2))] \\ & \mapsto_{\sigma} & \overline{\sigma}(Z\ Z_1) \\ & \mapsto_{\rho\sigma\delta} & \Pi(\Pi_1(\Pi(x_1, x_2)), \Pi_0(\Pi(x_1, x_2))) \end{array}$$

The application of u_2 *to this intermediate result can be then reduced as follows:*

$$\begin{array}{rcl} u_2\ (u_1\ \overline{t}) & \mapsto_{\rho\sigma\delta} & (\Pi(x, y) \twoheadrightarrow \Pi(\overline{P_1}\ x, \overline{P_0}\ y))\ \Pi(\Pi_1(\Pi(x_1, x_2)), \Pi_0(\Pi(x_1, x_2))) \\ & \mapsto_{\rho\sigma} & \Pi(\overline{P_1}\ \Pi_1(\Pi(x_1, x_2)),\ \overline{P_0}\ \Pi_0(\Pi(x_1, x_2))) \end{array}$$

The last reduction consists in applying the ρ*-rules* $\overline{P_0}$ *and* $\overline{P_1}$*. For doing this we use the substitution* $\overline{\sigma'} = \{X_1/x_1, X_2/x_2\}$ *and we obtain*

$$u_2\ (u_1\ \overline{t}) \mapsto_{\rho\sigma\delta} \Pi(\overline{P_1}\ \Pi_1(\Pi(x_1, x_2)),\ \overline{P_0}\ \Pi_0(\Pi(x_1, x_2))) \mapsto\!\!\!\!\!\!\rightarrow_{\rho\sigma\delta} \Pi(x_2, x_1)$$

which is the translation of the result of the original CRS *reduction.*

5.3 Properties

We show in what follows that we can always build a ρ-term encoding a given CRS-derivation. Moreover, we will prove that all ρ-reductions of this ρ-term are correct, *i.e.* they lead to the same final term and this term is the translation of the term obtained when performing the CRS-derivation. We first state some auxiliary lemmas concerning the context stability of the translation function and the correspondence between the substitutions in CRS and ρ-calculus.

Lemma 11 (Context stability) *Let* $A_{[B]_\omega}$ *be a* CRS*-metaterm, then:*

$$A|_\omega = B \quad \Leftrightarrow \quad \overline{A}|_{\pi_\rho(\omega, A)} = \overline{B}$$

Proof : The case where $\omega = \epsilon$ is obvious. For the other cases we proceed by structural induction on the CRS-metaterm $A_{[B]_\omega}$.

- If $A = x$ or $A = f$ then $\omega = \epsilon$ and the lemma holds trivially.
- If $A = [x]s$ then ω is either of the form $\omega = 1.\epsilon$ or of the form $\omega = 2.\omega'$ with ω' a position in s.
  1. $\overline{([x]s)_{[x]_{1.\epsilon}}} = (x \to \overline{s})_{[x]_{1.\epsilon}} = \overline{[x]s}_{[\overline{x}]_{1.\epsilon}} = \overline{[x]s}_{[\overline{x}]_{\pi_\rho(1.\epsilon,[x]s)}}$
  2. $\overline{([x]s)_{[B]_{2.\omega'}}} = \overline{[x](s_{[B]_{\omega'}})} = x \twoheadrightarrow \overline{s_{[B]_{\omega'}}}$
     $\stackrel{ih}{=} x \twoheadrightarrow (\overline{s}_{[\overline{B}]_{\pi_\rho(\omega',s)}}) = (x \twoheadrightarrow \overline{s})_{[\overline{B}]_{2.\pi_\rho(\omega',[x]s)}} = \overline{[x]s}_{[\overline{B}]_{\pi_\rho(2.\omega',[x]s)}}$
- If $A = Z(A_1, \ldots, A_n)$ and $\omega = (n-i).\omega'$, where $i \in \{0, \ldots, n-1\}$, then

$$\begin{aligned}\overline{Z(A_1,\ldots,A_n)_{[B]_{n-i.\omega'}}} &= \overline{Z(A_1,\ldots,A_{n-i[B]_{\omega'}},\ldots,A_n)} = Z\ \overline{A_1}\ldots\overline{A_{n-i[B]_{\omega'}}}\ldots\overline{A_n}\\ &\stackrel{ih}{=} Z\ \overline{A_1}\ldots\overline{A_{n-i}}_{[\overline{B}]_{\pi_\rho(\omega',A_{n-i})}}\ldots\overline{A_n} = (Z\ \overline{A_1}\ldots\overline{A_n})_{[\overline{B}]_{1^i.2.\pi_\rho(\omega',A_{n-i})}}\\ &= \overline{Z(A_1,\ldots,A_n)}_{[\overline{B}]_{\pi_\rho(n-i.\omega',Z(A_1,\ldots,A_n))}}\end{aligned}$$

- If $A = f(A_1, \ldots, A_n)$ then we proceed as in the previous case.

□

The following lemma states the correctness of the translation with respect to the assignment and substitution applications in CRS and rewriting calculus respectively.

Lemma 12 *Let A be a* CRS*-metaterm and $t_1, \ldots, t_n$ be* CRS*-terms, let $x_1, \ldots, x_n$ be distinct variables and $\sigma = \{x_1/t_1, \ldots, x_n/t_n\}$ be a* CRS *assignment. Then*

$$\overline{\sigma(A)} \ = \ \overline{\sigma}(\overline{A})$$

Proof : We proceed by structural induction on the CRS-metaterm A.

- If $A = x$, with $x \notin Dom(\sigma)$ or $A = f$ then $\sigma(A) = A$ and the lemma holds trivially. If $A = x_i$ then $\overline{\sigma(x_i)} = \overline{t_i} = \overline{\sigma}(\overline{x_i})$.
- If $A = [x]s$ then we can suppose, by α-conversion, that $x \notin Dom(\sigma)$ and we have

$$\overline{\sigma([x]s)} = \overline{[x]\sigma(s)} = x \to \overline{\sigma(s)} \stackrel{ih}{=} x \to \overline{\sigma}(\overline{s}) = \overline{\sigma}(x \to \overline{s}) = \overline{\sigma}(\overline{[x]s})$$

- If $A = Z(A_1, \ldots, A_n)$, then
  $\overline{\sigma(Z(A_1,\ldots,A_n))} = \overline{\sigma(Z)(\sigma(A_1),\ldots,\sigma(A_n))} = \overline{Z(\sigma(A_1),\ldots,\sigma(A_n))} = \overline{Z}\ \overline{\sigma(A_1)}\ \ldots\ \overline{\sigma(A_n)}$
  $\stackrel{ih}{=} \overline{Z}\ \overline{\sigma}(\overline{A_1})\ \ldots\ \overline{\sigma}(\overline{A_n}) = \overline{\sigma}(\overline{Z}\ \overline{A_1}\ \ldots\overline{A_n}) = \overline{\sigma}(\overline{Z(A_1,\ldots,A_n)})$.
- If $A = f(A_1, \ldots, A_n)$ then we proceed as in the previous case.

□

We show now that the translation *preserves* the matching solution, *i.e.* for every assignment σ solution of a CRS matching problem, its translation into a ρ-substitution is a solution of the corresponding matching problem in ρ-calculus. As a consequence, given an assignment used for the application of a CRS rewrite rule to a CRS-term, the translation of the rule into $\rho Cal_{\P}$ can be applied to the translation of the corresponding term using as substitution the translation of the CRS assignment.

Lemma 13 *Let A be a* CRS*-metaterm and σ an assignment. Then*

$$\overline{\sigma}(\overline{A}) \ \mapsto\!\!\!\!\!\rightarrow_{\lambda_\rho} \ \overline{\sigma(A)}$$

Proof : By structural induction on the term A.

- If A is a variable x then, since $x \notin \mathcal{MV}$ and thus $x \notin Dom(\overline{\sigma})$ we have

$$\overline{\sigma}(\overline{x}) = \overline{\sigma}(x) = x = \overline{x} = \overline{\sigma(x)}$$

- If $A = [x]s$ then we can use the induction hypothesis on s and we obtain

$$\overline{\sigma}(\overline{[x]s}) = \overline{\sigma}(x \twoheadrightarrow \overline{s}) = x \twoheadrightarrow (\overline{\sigma}(\overline{s})) \mapsto\!\!\!\!\!\rightarrow_{\lambda_\rho} x \twoheadrightarrow \overline{\sigma(s)} = \overline{[x]\sigma(s)} = \overline{\sigma([x]s)}$$

- If $A = f(A_1, \ldots, A_n)$ then we can use the induction hypothesis on the subterms $A_1, \ldots, A_n$:

$$\begin{array}{rcl} \overline{\sigma}(\overline{f(A_1, \ldots, A_n)}) & = & \overline{\sigma}(f(\overline{A_1}, \ldots, \overline{A_n})) = f(\overline{\sigma}(\overline{A_1}), \ldots, \overline{\sigma}(\overline{A_n})) \\ & \mapsto\!\!\!\!\!\rightarrow_{\lambda_\rho} & f(\overline{\sigma(A_1)}, \ldots, \overline{\sigma(A_n)}) \\ & = & \overline{f(\sigma(A_1), \ldots, \sigma(A_n))} = \overline{\sigma(f(A_1, \ldots, A_n))} \end{array}$$

- If $A = Z(A_1, \ldots, A_n)$ then we have two cases:

  1. If Z is of arity zero, then $\sigma = \{\ldots, (Z, \xi), \ldots\}$ and ξ contains no $\underline{\beta}$-abstraction. Thus, we have $\overline{\sigma}(\overline{Z}) = \overline{\sigma}(Z) = Z\{Z/\overline{\xi}\} = \overline{\xi} = \overline{\sigma(Z)}$.
  2. If Z is of arity n, then $\sigma = \{\ldots, (Z, \underline{\lambda} x_1 \ldots x_n.u), \ldots\}$ and its translation $\overline{\sigma} = \{\ldots, (Z, x_1 \twoheadrightarrow (x_2 \twoheadrightarrow (\ldots (x_n \twoheadrightarrow \overline{u}) \ldots))), \ldots\}$. We suppose that $A_1, \ldots, A_n$ contain no occurrences of $x_1 \ldots x_n$ which is always possible by α-conversion. We have:

$$\begin{array}{rcl} \overline{\sigma}(\overline{Z(A_1, \ldots, A_n)}) & = & \overline{\sigma}(Z \ \overline{A_1} \ \ldots \ \overline{A_n}) = \overline{\sigma}(Z) \ \overline{\sigma}(\overline{A_1}) \ldots \overline{\sigma}(\overline{A_n}) \\ & = & (x_1 \twoheadrightarrow (x_2 \twoheadrightarrow (\ldots (x_n \twoheadrightarrow \overline{u}) \ldots))) \ \overline{\sigma}(\overline{A_1}) \ldots \overline{\sigma}(\overline{A_n}) \\ & \mapsto\!\!\!\!\!\rightarrow_{\lambda_\rho} & \overline{u}\{x_1/\overline{\sigma}(\overline{A_1})\} \ldots \{x_n/\overline{\sigma}(\overline{A_n})\} \\ & = & \overline{u}\{x_1/\overline{\sigma}(\overline{A_1}), \ldots, x_n/\overline{\sigma}(\overline{A_n})\} \quad (\textit{since } \forall i, x_i \notin \{A_1, \ldots, A_n\}) \\ & \mapsto\!\!\!\!\!\rightarrow_{\lambda_\rho} & \overline{u}\{x_1/\overline{\sigma(A_1)}, \ldots, x_n/\overline{\sigma(A_n)}\} \quad (\textit{by induction hypothesis}) \\ & = & \overline{u\{x_1/\sigma(A_1), \ldots, x_n/\sigma(A_n)\}} \quad (\textit{By Lemma } 12) \\ & = & \overline{(\underline{\lambda} x_1 \ldots x_n.u)(\sigma(A_1), \ldots, \sigma(A_n))\downarrow_{\underline{\beta}}} \\ & = & \overline{(\sigma(Z))(\sigma(A_1), \ldots, \sigma(A_n))} \\ & = & \overline{\sigma(Z(A_1, \ldots, A_n))} \end{array}$$

□

As an immediate consequence we obtain as well $\overline{\sigma}(\overline{A}) =_{\lambda_\rho} \overline{\sigma(A)}$. This result is important since in $\rho Cal_{\P}$ the matching is performed modulo $=_{\lambda_\rho}$ and thus, $\mapsto\!\!\!\!\!\rightarrow_{\lambda_\rho}$ reductions are performed when testing whether a substitution is the solution of a given matching problem.

The congruence $=_{\lambda_\rho}$ is used when the CRS-metaterm A contains metavariables of arity different from zero. In this case, the application of the assignment involves $\underline{\beta}$-reduction steps performed at the meta-level of the CRS-reduction and these steps correspond to explicit ρ-reductions. Therefore, we do not have a syntactical identity between the two terms in the statement of the above lemma, but a reduction.

Using the previous lemmas we can show that starting from a CRS-reduction a corresponding reduction in $\rho Cal_{\P}$ can be obtained and more precisely, a ρ-term encoding the CRS-derivation trace can be constructed. When a one-step CRS-reduction is considered, *i.e.* only one CRS-rule is applied, the corresponding ρ-term depends on the initial term to be reduced and on the applied rewrite rule. Furthermore, we show that every possible ρ-derivation resulting from the correct initial ρ-term terminates and converges to the correct result.

Theorem 11 *Given a* CRS*-rule* $L \to R$*, an assignment* σ*, the* CRS*-term* $t_{0_{[\sigma(L)]_\omega}}$ *and the* CRS*-term* $t_1 \equiv t_{0_{\lceil\sigma(R)\rceil_\omega}}$*, given the derivation trace term* $u_0 = \overline{t_0}_{[x]_{\pi_\rho(\omega,t_0)}} \twoheadrightarrow \overline{t_0}_{[\overline{L\to R}\ x]_{\pi_\rho(\omega,t_0)}}$ *then, every derivation resulting from* $u_0\ \overline{t_0}$ *terminates and converges to* $\overline{t_1}$*:*

$$(u_0\ \overline{t_0}) \mapsto\!\!\!\!\!\!\rightarrow_{\rho\sigma\beta}\ \overline{t_1}$$

Proof : Thanks to the form of the ρ-term u_0 that describes the application of the rewrite rule exactly at the needed position, the ρ-reduction follows relatively easily. For the sake of readability we only show the case $\omega = \epsilon$.

$$\begin{aligned}(u_0\ \overline{t_0}) &= (x \twoheadrightarrow \overline{L \to R}\ x)\ \overline{t_0} = (x \twoheadrightarrow \overline{L \to R}\ x)\ \overline{\sigma(L)} = (x \twoheadrightarrow (\overline{L} \twoheadrightarrow \overline{R})\ x)\ \overline{\sigma(L)}\\ &\mapsto_\rho ((\overline{L} \twoheadrightarrow \overline{R})\ x)\ [x \ll \overline{\sigma(L)}] \mapsto_\sigma (\overline{L} \twoheadrightarrow \overline{R})\ \overline{\sigma(L)} \mapsto_\rho \overline{R}\ [\overline{L} \ll \overline{\sigma(L)}] \mapsto_\sigma \overline{\sigma}(\overline{R}) \mapsto\!\!\!\!\!\!\rightarrow_{\rho\sigma\beta} \overline{\sigma(R)}\\ &= \overline{t_1}\end{aligned}$$

In the above reduction we have used the fact that, by Lemma 13, the two matching problems $\overline{L} \lessdot_{\P} \overline{\sigma(L)}$ and $\overline{L} \lessdot_{\P} \overline{\sigma}(\overline{L})$ have the same solution $\overline{\sigma}$. The Lemma 13 can be extended easily for $\rho Cal_{\P}$ reductions and this version is used for concluding the above reduction.

Another possible reduction is the following one:

$$\begin{aligned}(u_0\ \overline{t_0}) &= (x \twoheadrightarrow \overline{L \to R}\ x)\ \overline{t_0} = (x \twoheadrightarrow \overline{L \to R}\ x)\ \overline{\sigma(L)} = (x \twoheadrightarrow (\overline{L} \twoheadrightarrow \overline{R})\ x)\ \overline{\sigma(L)}\\ &\mapsto_\rho (x \twoheadrightarrow \overline{R}\ [\overline{L} \ll x])\ \overline{\sigma(L)} \mapsto_\rho \overline{R}\ [\overline{L} \ll x][x \ll \overline{\sigma(L)}] \mapsto_\sigma \overline{R}\ [\overline{L} \ll \overline{\sigma(L)}]\\ &\mapsto_\sigma \overline{\sigma}(\overline{R}) \mapsto\!\!\!\!\!\!\rightarrow_{\rho\sigma\beta} \overline{\sigma(R)} = \overline{t_1}\end{aligned}$$

The same arguments as in the previous case have been used in this reduction. These are the only possible derivations since the translations of CRS-terms contain no redexes and the matching problem $\overline{L} \ll_{\P} x$ in the latter derivation has no solution because L cannot be a variable.

The reduction for $\omega \neq \epsilon$ is similar, the only difference being at the matching level. In this case we have to use Lemma 11 to obtain the solution of the matching against the translation of the left-hand side of the rule. □

We can notice that we have a longer derivation scheme in $\rho Cal_{\P}$ than in CRSs. For every rewrite step in a CRS, the reduction of the corresponding $\rho Cal_{\P}$-term uses two (ρ)-rule steps plus two (σ)-rule steps for the application of the rewrite rule and some additional ($\rho\sigma\beta$) steps corresponding to the β-reduction steps performed at the meta-level of the CRS-reduction. We should point out that the matching in the $\rho Cal_{\P}$ may involve some derivations but these are performed at the meta-level.

We can generalise the previous theorem and build, using the derivations of a term t_0 *w.r.t.* a CRS, a ρ-term with a reduction similar to this CRS-derivation.

Proposition 8 *Let* t_0, t_n *be two* CRS*-terms such that* $t_0 \mapsto\!\!\!\!\!\!\rightarrow_{\mathcal{R}} t_n$ *and* $u_0, \ldots, u_n$ *the corresponding derivation trace terms in the* ρ*-calculus. Then every derivation resulting from* $(u_n\ \ldots\ (u_0\ \overline{t_0}))$ *terminates and converges to* $\overline{t_n}$*.*

Proof : Follows immediately by Theorem 11. □

The ρ-terms u_i of Proposition 8 can be built automatically starting from the CRS-reduction steps as stated in Theorem 11. It is obviously interesting to give a method for constructing this terms without knowing *a priori* the derivation steps from t_0 to t_n but only the set of CRS rewrite rules. A possible solution to this issue in the line of [CLW03] is discussed in the conclusions of this thesis.

5.4 Comparison of the two systems

The work done on the encoding of CRSs in the ρ-calculus enlightened some differences between the two systems. Rewriting is performed in a quite different way. In the CRSs we can distinguish two levels: the meta level, where notions like substitution, application and rewrite step are expressed, and the object level where terms and rewrite rules are defined. In the ρ-calculus, as explained next, most of these notions become explicit and are treated at the object level.

Let us first discuss the differences in the rewrite step generation. A rewrite step for a CRS-term is composed of two stages: first of all a match between a term t and the left-hand side L of a rewrite rule is performed, obtaining an assignment, then this assignment is applied to the right-hand side R of the rule. As a matter of fact, the application of the assignment consists in replacing the "holes" in the right-hand side of the rule, represented by metavariables, by a special kind of λ-terms. The computation of the result of this replacement is done at the meta level of the system by performing a development on $\underline{\lambda}$-terms.

On the other hand, in the $\rho Cal_{\P}$, once identified a redex $(L \twoheadrightarrow R)\ t$, this is first transformed into the corresponding constrained term $R[L \ll t]$ and then the process of rewriting is decomposed into two phases: the computation of the matching problem and the application of the obtained substitution to the term R. As far as the first phase is concerned, this is entirely performed at the meta level of the calculus. Matching in the $\rho Cal_{\P}$ is done modulo the congruence relation $=_{\lambda_\rho}$ and thus some reductions are possibly performed when checking whether a substitution is the solution of the matching problem, but these reductions are made at the meta level of the calculus. When the matching is successful, the result is a substitution that associates to each free variable of L a ρ-term. The application of the substitution is done at the object level and implies some $(\rho\varpi\delta)$ reduction steps whose number depends on the arity of the CRS-metavariables in the right-side R of the considered CRS rewrite rule.

In short, in the $\rho Cal_{\P}$ a rewrite step is composed first by an evaluation step, which is meant to make the matching problem explicit in the initial redex, and then by two phases conceptually similar to the ones of CRSs. However, differently from CRSs, only the first phase, *i.e.* the computation of the matching, is performed at the meta level of the calculus, while the application of the substitution is done at the object level. While for CRSs the application of the assignment includes implicit $\underline{\beta}$-reductions, in $\rho Cal_{\P}$ the corresponding $(\rho\varpi\delta)$ reduction steps are performed explicitly. It follows, as seen in Example 27 and in the proof of Theorem 11, that we generally have a longer reduction sequence in $\rho Cal_{\P}$ than in CRSs.

Considering now more generally the two rewrite relations, the explicit encoding we have presented indicates a certain gap between the two formalisms. "Walking through the context" is done implicitly in CRSs, while additional ρ-terms are needed to direct the reduction in ρ-calculus. This is a consequence of the fact that rewrite rules are defined at the object-level of ρ-calculus and they are applied explicitly. The reduction is then performed by the three evaluation rules of the calculus. On the contrary, in CRSs we have a set of rewrite rules separated from the set of terms. These rewrite rules are particular to the CRS considered and the application strategy

is left implicit. This is probably the main difference between the rewriting calculus and CRSs. Rewrite rules and consequently their control (application position) are defined at the object-level of ρ-calculus while for the CRSs the reduction strategy is not explicitly given.

5.5 Application to Higher-order Rewrite Systems

The encodings presented in this chapter for CRSs *w.r.t.* ρ-calculus can be combined with another work done by van Oostrom and van Raamsdonk in [VOVR93] about the comparison of CRSs and the systems called Higher-order Rewrite Systems (HRSs). We can obtain in this way an encoding of HRSs in the ρ-calculus.

As recalled in Section 2.2, HRSs, proposed by T. Nipkow in the early nineties [Nip91, Nip93], consist in an extension of the λ-calculus with algebraic functions and were introduced to give a logical framework to programming languages like λ-Prolog and theorem provers like Isabelle. A HRS $\mathcal{H} = (\mathcal{T}, \mathcal{R})$ is defined by a set of terms $\mathcal{T}$ and a set of rewrite rules $\mathcal{R}$. In [VOVR93] HRSs are shown to be as expressive as the CRSs, *i.e.* the rewrite relations induced by the two systems are equivalent and similar properties (*e.g.* the confluence) are obtained for the two systems.

Using the encoding proposed in [VOVR93] we can associate a CRS to any HRS and, using the translation defined in Section 5.1, we can encode the corresponding CRS into the ρ-calculus. We obtain thus a two-steps translation of the HRS into the ρ-calculus:

$$\text{HRS} \xrightarrow{\tau} \text{CRS} \xrightarrow{\mathcal{E}} \rho Cal_{\P}$$

where τ is the translation function from HRSs to CRSs and $\mathcal{E}$ is the function of Definition 57.

Concerning the preservation of the rewrite steps, we recall that HRSs have more "rewriting power" than CRSs, *i.e.* they can do more in one step. This is due to the different definition of the substitution mechanism in the two systems: for the CRSs, the application of an assignment to a term involves only a development on λ-terms, while for the HRSs a reduction to normal form in simply typed λ-calculus is performed. The solution adopted in [VOVR93] for preserving rewrite sequences is to add explicitly the BetaCRS rule to the set of CRS-rewrite rules obtained by translating the HRS-rules. This has no particular effect on the ρ-calculus side. The BetaCRS rule is treated as any other CRS-rule and it is translated by the function $\mathcal{E}$ in a term of $\rho Cal_{\P}$ as shown in Example 26.

Proposition 9 *Let t_0, t_n be two* HRS*-terms such that t_0 rewrites to t_n using the set $\mathcal{R}$ of* HRS*-rules. If $t'_0 = \mathcal{E}(\tau(t_0))$ and $t'_n = \mathcal{E}(\tau(t_n))$ are the translations of the two* HRS*-terms into ρ-terms then there exist m ρ-terms $u_0, \ldots, u_m$ such that every evaluation of $(u_m \ \ldots \ (u_0 \ t'_0))$ terminates into t'_n.*

Proof : By [VOVR93] we have that every HRS-rewrite step $t_0 \mapsto_R t_1$, with $R \in \mathcal{R}$, corresponds to a CRS-rewrite sequence $\tau(t_0) \mapsto_{\tau(R)} \mapsto_{\mathcal{B}}^! \tau(t_1)$, where $\mapsto_{\mathcal{B}}^!$ denotes a reduction to BetaCRS normal form. The ρ-terms $u_0, \ldots, u_m$ can be then built starting from this derivation using the approach described in Theorem 11. □

Conclusion

The rewriting calculus can be used to describe rewrite based languages and object oriented calculi in a natural and simple way. We have shown in this chapter that it also allows one to encode higher-order rewriting. In particular, we have concentrated our study on the analysis of the

relationship between ρ-calculus and CRSs. This has led to a simulation of CRSs reductions in the ρ-calculus. We have shown that any derivation of a term *w.r.t.* a given CRS can be represented by a corresponding ρ-term built automatically starting from the initial CRS-term and its reduction. Moreover we have proved the completeness of our approach, that is every derivation starting from such a kind of ρ-term converges to a term equivalent to the final CRS-term, modulo the translation between the two systems.

This work has made clearer some differences existing between the two formalisms. These differences concern notions like substitutions and their application, as well as the rewrite relation induced by the CRS rewrite rules and the ρ-calculus evaluation rules respectively.

The chapter terminates with an application of our result to another kind of higher-order rewrite systems than CRSs. In fact, we take advantage of another work of comparison between two higher-order rewrite systems, namely CRSs and the HRSs, to achieve an indirect representation of the HRSs in the rewriting calculus.

The results presented here have been published in [BCK03].

Chapter 6

Term graph rewriting

As we have seen in the first chapter of this thesis, term rewriting is a useful tool for the abstract modelling of functional languages. In the term rewriting setting, terms are often seen as trees, nevertheless in order to improve the efficiency of the implementation of functional languages, it is of fundamental interest to think and implement terms as graphs. In this case, the possibility of sharing subterms allows one to save space (by using multiple pointers to the same subterm instead of copying the subterm) and time during the computation (a redex appearing in a shared subterm will be reduced at most once). We can take as example a rewrite system $\mathcal{R} = \{r_1 = x * 0 \to 0, \quad r_2 = x * s(y) \to (x * y) + x\}$ defining the multiplication on naturals. If we represent it using term graphs, we will write the rule r_2 by duplicating the reference to x instead of duplicating x itself. In this way, when this rule is applied to the term $s(0) * s(0)$, the subterm $t = s(0)$ instantiating x will be allocated only once and not twice in the obtained result of the rewrite. Moreover, if t is not itself in normal form, *e.g.* $t = s(0) * 0$, as can be the case when the language has a call-by-need semantics, the value (normal form) of t is computed in one further r_1-step, while in standard term rewriting two rewrite steps would be necessary.

Graph rewriting is therefore a useful technique for the optimization of functional and declarative languages implementations [PJ87]. Moreover, the possibility to define cycles leads to an increased expressive power that allows one to represent naturally regular infinite data structures (*i.e.* structures with a finite number of different substructures). For example, a cyclic graph can be used to represent the circular list $ones = 1 : ones$, where "_:_" denotes the concatenation operator. We conclude that a "theory of cycles" is necessary if one wants to reason about compilation, optimisation and execution of programs.

Cyclic term-graph rewriting has been widely studied from different points of view which differ mainly for the mathematical framework in which the concepts of rule, match and rewriting step are formalised. A rigorous mathematical framework is given by the categorical approach studied in [Cor93, CG99b]. A point of view that reflects implementation of functional programming languages is described in [BvEG$^+$87]. An equational presentation is proposed in [AK96a]. For a survey on term graph rewriting one can refer to [SPvE93].

In this chapter we briefly describe these different approaches. We explain in more detail the equational framework that is used also for the definition of the cyclic λ-calculus [AK96b], an extension of the standard λ-calculus by adding a `letrec` like construct. This new feature allows the calculus to express λ-terms with sharing and cycles, consequently some appropriate transformation rules are added to the β-rule to explicitly manipulate this new kind of λ-terms.

6.1 Operational approach

Historically, the first presentation of term graphs is the one in [BvEG+87] where graphs are characterised by a set of nodes, a labelling function and a successor function.

Before giving the formal definition, we need to introduce some notation facilities. Let $\mathcal{F}$ be a set of symbols and let $\perp$ be a new symbol not appearing in $\mathcal{F}$. We denote by $\mathcal{F}_\perp$ the union $\mathcal{F} \cup \{\perp\}$ and we consider the partial order such that $\perp < f$ for any $f \in \mathcal{F}$. We define $arity(\perp) = 0$. We remind that, given a set S, we denote by S^* the free monoid over S with ϵ as neutral element and the operator of concatenation. For any element $e = e_1 \dots e_n \in S^*$, we write $|e|$ to indicate the *length* of e, namely $|e| = n$. Moreover the ith component of e is denoted by $[e]_i$. If $f : S_1 \to S_2$ is a function, we denote by $f^* : S_1^* \to S_2^*$ its monoidal extension.

Definition 59 (Term graph) *A* term graph *over $\mathcal{F}_\perp$ is a tuple $G = (N, s, l)$ where N is a set of* nodes*, $s : N \to N^*$ is the* successor function *and $l : N \to \mathcal{F}_\perp$ is the* labelling function *such that $|s(n)| = arity(l(n))$ for any $n \in N$.*

Intuitively, nodes labelled by $\perp$, called empty nodes, are intended to represent variables in the term graph. Different empty nodes represent different variables and multiple references to the same variable are represented in a term graph by multiple references to the same empty node. To identify empty nodes in textual representation, we may give them names as $x, y, z, \dots$. We denote by $\mathcal{V}ar(G) \subseteq N$ the set of all empty nodes (or variable nodes) in G, *i.e.* $n \in \mathcal{V}ar(G)$ if $l(n) = \perp$.

Definition 60 (Path) *In a term graph $G = (N, s, l)$ an* annotated path *from a node $n_1 \in N$ to a node $n_k \in N$, denoted $p : n_1 \to n_k$, is a sequence $n_1 i_1 n_2 i_2 \dots i_{k-1} n_k$ of nodes interleaved with integers, such that $[s]_{i_j}(n_j) = n_{j+1}$, for all $j \in \{1, \dots, k-1\}$. In this case the sequence of nodes $n_1 n_2 \dots n_k$ is called a* path *from n_1 to n_k and still denoted as $p : n_1 \to n_k$. We call $|p|$ the* length *of the path.*

The notion of path is useful for characterising term graphs with a special structure. For example a term graph G is called *acyclic* if there are no non-empty paths from one node to itself. It is called *cyclic* otherwise.

Definition 61 (Rooted term graph) *A* rooted term graph *over $\mathcal{F}$ is a tuple $G = (r, N, s, l)$ where (N, s, l) is a term graph and $r \in N$ is a node called the* root *of G.*

A node $n \in N$ is called a garbage node *if there is no path from the root r to n.*

A proper rooted term graph *is a rooted term graph where the set of garbage nodes is empty.*

Notice that, with respect to general graphs, the term graphs considered here are included into the sub-class of single rooted directed graphs. Given a (rooted) term graph G, we usually refer to its components as r_G, N_G, s_G and l_G respectively. In general we may have several paths from the root of a rooted term graph G to a node n in G. If there exists exactly one path from the root of G to any other node n of G, then G is a *tree*.

We will often depict graphs with the root as topmost node, displaying labels at their corresponding nodes and drawing successors as outgoing edges ordered from left to right. The label $\perp$ is denoted simply by "$\cdot$" in figures.

Example 31 *The term graphs defined in this example are depicted in Figure 6.1.*

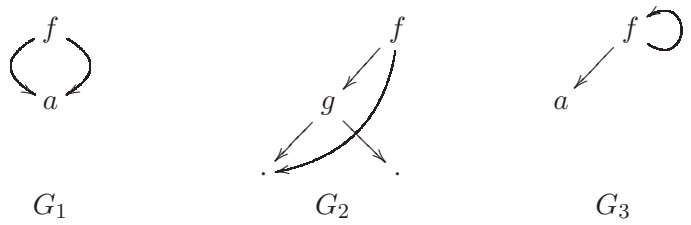


Figure 6.1: Some term graphs

- *Let $G_1 = (n_1, \{n_1, n_2\}, s, l)$ be a term graph where the functions s and l are defined on N as $s(n_1) = n_2 n_2$, $s(n_2) = \epsilon$ and $l(n_1) = f$, $l(n_2) = a$. The term graph G_1 correspond to the term $f(a, a)$ where the constant a is shared.*

- *Let $N = \{n_1, n_2, n_3, n_4\}$ and let the functions s and l be defined on N as follows:*

  $$s(n_1) = n_2 n_3,\ s(n_2) = n_3 n_4,\ s(n_3) = s(n_4) = \epsilon;$$
  $$l(n_1) = f,\ l(n_2) = g,\ l(n_3) = l(n_4) = \perp$$

  The term graph $G_2 = (n_1, N, s, l)$ correspond to the term $f(g(x, y), x)$.

- *Let $G_3 = (n_1, \{n_1, n_2\}, s, l)$ be a term graph where the functions s and l are defined on N as $s(n_1) = n_2 n_1$, $s(n_2) = \epsilon$ and $l(n_1) = f$, $l(n_2) = a$. The term graph G_3 has no term counterpart.*

Definition 62 (Subgraph) *A subgraph of a graph G rooted at $n \in N_G$ is the graph $G|n = (n, N, s, l)$ such that*

- *$N = \{m \in N_G$: there exists a path $p : n \to m\}$*
- *s and l are the restriction to N of the functions s_G and l_G, respectively.*

Usually we are interested in the subgraphs that are reachable from the root of the main graph, hence in the following we will consider only the connected parts of a graph. The isolated nodes can be eliminated using the operation of garbage collection, formally defined as follows.

Definition 63 (Garbage collection) *Let G be a rooted term graph. The proper rooted term graph of G obtained via garbage collection is defined as $gc(r, G) = (r, N, s, l)$ where:*

- *$r = r_G$*
- *$N = N_G - \{n : n$ is a garbage node in $G\}$*
- *s and l are the restrictions to N of the functions s_G and l_G, respectively.*

A match from a term graph G_1 to a term graph G_2 is a term graph homomorphism mapping nodes of G_1 to G_2 in such a way that the graphical structure and the labels are preserved.

Definition 64 (Term graph homomorphism) *Given the term graphs $G_1 = (N_1, s_1, l_1)$ and $G_2 = (N_2, s_2, l_2)$ over $\mathcal{F}_\perp$, a* term graph homomorphism *$f : G_1 \to G_2$ is a function $f : N_1 \to N_2$ such that for any $n \in N_1$*

1. $f^*(s_1(n)) = s_2(f(n))$ *for any non empty node* n*;*

2. $l_1(n) \leq l_2(f(n))$.

If G_1 *and* G_2 *are rooted term graphs, then a* rooted term graph homomorphism *maps the root of* G_1 *into the root of* G_2*, i.e., it must hold*

3. $f(r_1) = r_2$*;*

A strict homomorphism *is an homomorphism where the condition* (2.) *is replaced by the following:*

2. $l_1(n) = l_2(f(n))$*;*

A strict homomorphism between two term graphs is a particular case of a more general relation called *bisimilarity*.

Definition 65 (Term graph bisimulation) *Let* $G_1 = (N_1, s_1, l_1)$ *and* $G_2 = (N_2, s_2, l_2)$ *be two term graphs over* $\mathcal{F}_\perp$*. A* term graph bisimulation *is a relation* $\mathcal{B} \subseteq N_1 \times N_2$ *such that if two nodes belong to the relation, denoted* $n_1 \ \mathcal{B} \ n_2$*, then*

1. $l_1(n_1) = l_2(n_2)$*;*

2. $[s_1]_i(n_1) \ \mathcal{B} \ [s_2]_i(n_2)$ *for all* i *such that* $1 \leq i \leq arity(l_1(n_1))$

If such a relation exists, then the two graphs G_1 *and* G_2 *are called* bisimilar*. If* G_1 *and* G_2 *are rooted term graphs, then the relation holds if moreover the root of* G_1 *is bisimilar to the root of* G_2*, i.e.* $r_1 \ \mathcal{B} \ r_2$ *holds.*

Intuitively two term graphs are bisimilar if their unfolding represent the same term, possibly infinite if the term graphs are cyclic. Note that if the relation $\mathcal{B}$ is a function, then the bisimulation is a strict term graph homomorphism.

The notion of homomorphism between term graphs plays a basic role when defining graph rewriting. As already mentioned, a homomorphism must preserve the structure of a term graph, *i.e.* labels, successors and their order. A variable node can be mapped to any node, the intuition being that the variable matches the subgraph rooted at the image of the variable node. For example, the graphs G_1 and G_2 in Figure 6.2 are homomorphic using a function f that maps the root of G_1 to the root of G_2, the two nodes labeled s of G_1 to the same node with label s in G_2 and the node labeled 0 of G_1 to the node with the same label in G_2. Notice that the other way around the homomorphism does not exists, since the node s in G_2 cannot be mapped into two different nodes in G_1. In other words, a rooted homomorphism maps a term graph into another term graph having the same structure and possibly more sharing. In case of cycles, we can say that an homomorphism maps a cyclic term graph into a term graph that can be less "unravelled" than the first one.

The rewriting process over term graphs is more elaborated than over terms. This is due to the possibility of sharing that makes the notion of context not straightforward in term graphs. The matching part of the graph can indeed be connected to the rest of the graph by means of several references, other than its root, and all of them have to be considered during rewriting. Before describing a complete rewrite step, we define a term graph rewrite rule.

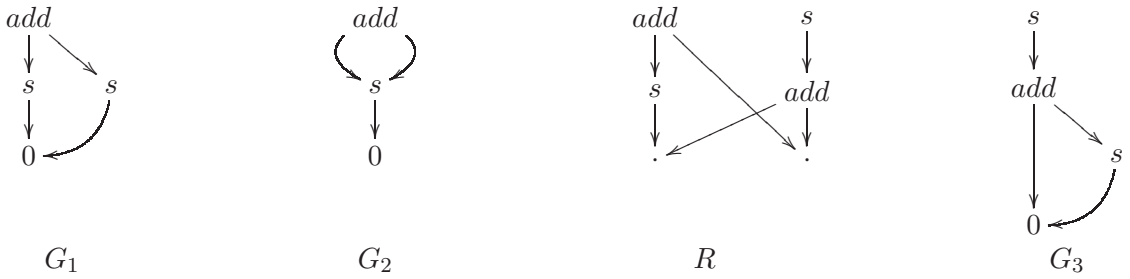


Figure 6.2: Examples of term graphs

Definition 66 (Term graph rewrite rule) *A* term graph rewrite rule *is a triple* (R, l, d)*, where* R *is a term graph and* $l, d \in N_R$ *are two fixed nodes. The term graph rooted at* l*, i.e.,* $gc(l, R)$ *and the term graph rooted at* d*, i.e.,* $gc(d, R)$ *are called the left- and right-hand side of* R*, respectively, and denoted by* R_l *and* R_r*.*

We assume on term graph rewrite rules some restrictions typical of term rewriting:

- R_l *is not a single variable node (empty node);*
- R_l *is a tree (which implies left-linearity);*
- *Every empty node is accessible from the root* l *of the left-hand side of* R*.*

In the following we will often denote (R, l, d) simply by R. An example of term graph rewrite rule representing the addition on natural numbers is depicted in Figure 6.2 (R).

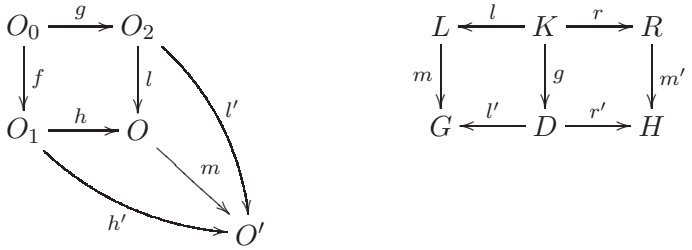
We can apply a rewrite rule R to a graph G if we find an occurrence of the left-hand side R_l of R in G. Formally this matching from R_l and (a subgraph of) G is given by a homomorphism f from R_l to G. We call the pair (R, f) a *redex*. When a redex has been determined, the rewriting is done by first copying the part of R_r not contained in R_l and linking it to the redex in G. Then, one performs the redirection of all the nodes pointing to the root of the occurrence of R_l to the root of the copy of the right-hand side R_r of the rule. Finally a garbage collection phase concludes the application. This process in three phases (built, redirection and garbage collection) can be formalised as in [BvEG$^+$87].

For example, the left-hand side R_l of the rule R in Figure 6.2 matches the graph G_1 and the rewriting leads to the graph G_3.

6.2 Categorical approach

We briefly describe in this section the algebraic approach, also called "double-pushout approach", which actually is a general (graph) rewriting formalism that can be applied to a particular category representing term graphs.

Definition 67 (Category [Fok92]) *A category* $\mathcal{C}$ *is a collection of* objects*, denoted* $O_1, O_2, \ldots$ *together with a collection of* arrows*, also called morphisms, denoted* $f, g, l \ldots$*. For any object* O*, a special arrow, called* identity *on* O *and denoted id, is distinguished. For any arrow* f*, a relation called* typing *is defined as* $f : O_1 \to O_2$ *where* $O_1 \to O_2$ *is the* type *of* f *and* O_1 *and* O_2 *are called the source and the target of* f*. Moreover, a partial operation called* composition *and denoted by "*$\_ ; \_$*" is defined over the set of arrows.*



Pushout Derivation

Figure 6.3: Rewriting in the algebraic approach

The above data are subject to the axioms listed in the following definition.

Definition 68 (Axioms) *Typing axioms:*

1. *(Uniqueness of type) If* $f : O_1 \to O_2$ *and* $f : O_1' \to O_2'$ *then* $O_1 = O_1'$ *and* $O_2 = O_2'$*;*
2. *(Composition of types) If* $f : O_1 \to O_2$ *and* $g : O_2 \to O_2'$ *then* $f \ ; \ g : O_1 \to O_2'$*;*
3. *(Identity type)* $id : O \to O$*.*

Axioms for arrow composition:

1. *(Associativity)* $(f \ ; \ g) \ ; \ l = f \ ; \ (g \ ; \ l)$
2. *(Identity)* $f \ ; \ id = f = id \ ; \ f$

Term graphs, as defined in the previous section, can be seen as a particular category $\mathcal{G}$, where the objects of the category are term graphs, denoted by $G, H, L, \ldots$ and the arrows of the category are term graph homomorphisms. The advantage of this approach is that techniques borrowed from category theory can be used to prove properties about graph transformations. The core of the algebraic approach is in fact the idea of expressing the *gluing of graphs* in categorical terms as a *pushout construction.*

Definition 69 (Pushout) *Given a category* $\mathcal{C}$ *and a pair of arrows* $f : O_0 \to O_1$ *and* $g : O_0 \to O_2$ *of* $\mathcal{C}$*, a triple* (O, h, l) *with* $h : O_1 \to O$ *and* $l : O_2 \to O$ *as in Figure 6.3 is called a pushout of* (f, g) *if the two following conditions are satisfied:*

1. *(Commutativity)* $f \ ; \ h = g \ ; \ l$
2. *(Universality) For any object* O' *and arrows* h' *and* l' *with* $h' : O_1 \to O'$ *and* $l' : O_2 \to O'$ *such that* $f \ ; \ h' = g \ ; \ l'$*, there exists a unique arrow* $m : O \to O'$ *such that* $h \ ; \ m = h'$ *and* $l \ ; \ m = l'$*.*

A graph direct derivation, or a *production*, as it is usually called, can be formalised in the categorical setting as a single pushout *SPO* [Löw93] or double pushout *DPO* [SEP73]. We present here the double pushout approach, which historically was the first one to be proposed. For a discussion on the differences between the two approaches the reader can refer to [EHK+97].

Definition 70 (Production, derivation) *In the DPO approach a* production $p : L \leftarrow K \rightarrow R$ *is a pair of graph homomorphisms* $l : K \rightarrow L$ *and* $r : K \rightarrow R$ *where* L, K, R *are finite graphs and are called the* left-hand side*, the* interface *and the* right-hand side *respectively.*

A direct derivation *from* G *to* H *exists if and only if the diagram in Figure 6.3 can be constructed, where both squares are required to be pushouts in the category* $\mathcal{G}$*. In this case* D *is called the* context graph*.*

Given a production, the graph K can be seen as an "interface", in the sense that it is not affected by the rewrite step itself, but it is necessary for specifying how the right-hand side R is glued with the graph D. Intuitively, the context graph D is obtained from the given graph G by deleting all elements of G which have a pre-image in L but not in K. The second pushout diagram models the insertion into H of all elements of R that do not have a pre-image in K. Consequently, thanks to the interface graph K, this construction allows one to handle in a formal and elegant way the "three phases" mechanism as described for the operational approach in the previous section.

A different categorical presentation of term graph rewriting, inspired from the work of B. Lawvere [Law63] has been given in the setting of 2-categories. A 2-category $\mathcal{C}$ is a category where the collection of arrows between any two objects O_1 and O_2 is itself a category, denoted $\mathcal{C}[O_1, O_2]$. The arrows of the category $\mathcal{C}[O_1, O_2]$ are called *cells*. It was shown by Corradini and Gadduccci in [CG99a, CG99b] that cyclic term graphs can be represented as arrows of a traced gs-monoidal 2-category, suitably generated from the graph signature, and term graph rewrite rules can be represented as 2-cells. To give an idea of the reason why cyclic term graphs are conveniently modelled in this specific category, we mention that the *trace structure* is the categorical counterpart of the operation called *feedback* on term graphs that is used to model cycles. The acronym *gs* stands for graph substitution since horizontal composition is induced by a suitable operation of substitution for graphs.

The free 2-category generated by such cells represents then term graph rewrite sequences, as defined by the *operational* approach, in a faithful way. The only exception concerns circular redexes: take for example the collapsing rewrite rule $f(x) \rightarrow x$ and apply it to the graph having one node labelled f and a self-looping edge. Following the definition of rewriting given in the previous section, such graph reduces to itself. Intuitively, this is motivated by the fact that this graph corresponds to the infinite term $f(f(\ldots))$ and the application of the given rewrite rule has as effect the erasing of the first symbol f. The result is a term that again has an infinite number of symbols f. Using the categorical approach, the graph above reduces using the given rule to a single node without label and without successor nodes, usually called *black hole* and denoted by "$\bullet$" (see Section 6.3). This empty graph is interpreted as a cycle of length zero or a completely undefined term.

The handling of circular redexes has been a matter of some discussion in the literature. Some authors agree with the first interpretation above [KKSd94, BvEG$^+$87], but some definitions of term graph rewriting differ in this point [CD97, Ken87]. Both interpretations discussed here are meaningful from the point of view of infinitary term rewriting. In the first case the looping graph corresponding to the term $f(f(\ldots))$ is the limit of an infinite sequence of application of the collapsing rule for f, *i.e.* $f(f(\ldots)) \rightarrow f(f(\ldots)) \rightarrow \ldots$. In the second case, the black hole is the result of the simultaneous application of the collapsing rule $f(x) \rightarrow x$ to an infinite number of symbols f. Some elements of comparison between the two approaches can be found in [CG99b, CD97].

6.3 Equational approach

We consider here an equational presentation in the style of [AK96a]. A similar representation was used by Courcelle for characterising the set of regular infinite trees, *i.e.* the trees with a finite number of subtrees [Cou83].

A natural way to represent a term graph is by associating unique names to its nodes and by specifying their interconnections through a set of recursion equations. Given a first order signature $\Sigma = (\mathcal{F}, \mathcal{X})$, a term graph over Σ is a system of equations of the form

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

with $x_i \in \mathcal{X}$ and $t_i \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, for $i = 1, \ldots, n$, and where the recursion variables x_i are supposed pairwise distinct. The variable x_1 on the left represents the root of the term graph. The variables $x_1, \ldots, x_n$ are bound in the term graph by the associated recursion equation. The other variables occurring in the term graph G are called free. A term graph without free variables is called closed and two graphs which differ only for the name of bound variables, are considered equal. We denote the set of free variables of G by $\mathcal{FV}(G)$ and the collection of all variables appearing in G by $\mathcal{V}ar(G)$. We call the list of equations the *body* of the term graph and we denote it by E_G, or simply E, when the graph G is clear from the context. The empty list is denoted by ϵ.

For example, the graphs depicted in Figure 6.1 can be represented by the following systems:

i) $\{x_1 \mid x_1 = f(x_2, x_2), x_2 = a\}$

ii) $\{x_1 \mid x_1 = f(g(x_2, x_3), x_2)\}$

iii) $\{x_1 \mid x_1 = f(a, x_1)\}$

Cycles may appear in the system, like in *iii*). Degenerated cycles, *i.e.* equations of the form $x = x$ that correspond to empty graphs, are replaced by $x = \bullet$ (black hole). The fresh constant "$\bullet$" that has been originally introduced in [AK96a] is a possible solution to the non confluence problem generated by the so called *collapsing rules*, *e.g.* the rewrite rules $f(x) \to x$ and $g(y) \to y$ which have a single variable on the right-hand side. Intuitively, it is not difficult to see that if we apply these rules to the term $\{x \mid x = f(g(x))\}$ we obtain either $\{x \mid x = f(x)\}$ or $\{x \mid x = g(x)\}$. A simple solution is to proceed in the reduction and rewrite both the terms to $\{x \mid x = x\}$, an undefined kind of expression for which the new object "$\bullet$" is introduced.

A term graph is said to be in *flat form* if all its recursion equations are of the form $x = f(x_1, \ldots, x_n)$, where the variables $x, x_1, \ldots, x_n$ are not necessarily distinct from each other. For example, the term graph *i*) above is in flat form while the term graphs *ii*) and *iii*) are not. Each term graph G can be transformed into a unique flattened system denoting the same graph by adding equations which give a name to "hidden" nodes and by removing trivial equations between variables of the kind $x = y$. Applying this transformation to the term graph *ii*) we obtain $\{x_1 \mid x_1 = f(y, x_2), y = g(x_2, x_3)\}$ and similarly the term graph *iii*) is transformed into $\{x_1 \mid x_1 = f(y, x_1), y = a\}$. In the work of Courcelle [Cou83], a system $\{x_1 \mid x_1 = t_1, \ldots, x_n = t_n\}$ in flat form is called *regular system* and it is shown to have a unique solution for the unknowns $x_1, \ldots, x_n$. Courcelle proves that all components of a solution of a regular systems are regular trees and every regular tree is a component of the unique solution of some regular system, obtaining in this way a complete and correct characterisation of regular trees.

Regular trees can also be seen as term graphs, as it is formalised in [SPvE93] where it is shown that there exists a one-to-one correspondence between a recursion system in flat form and

a term graph defined by its sets of nodes, a successor and a label function. To help the intuition, we will thus use the graphical representation of term graphs described in the previous section also for systems of equations in flat form.

We call *cyclic* a term graph in flat form whose body contains a sequence of the form

$$x_1 = f_1(y_1^1, \ldots, x_2, \ldots y_{k_1}^1), x_2 = f_2(y_1^2, \ldots, x_3, \ldots, y_{k_2}^2), \ldots, x_n = f_n(y_1^n, \ldots, x_1, \ldots, y_{k_n}^n)$$

where $n, k_i \in \mathbb{N}$, $i = 1, \ldots, n$. We call a term graph in flat form *acyclic* otherwise.

The rewrite mechanism is specified by means of term graph rewrite rules. The definitions of rewrite rule, match and rewrite step given in the first section for the operational approach are adapted to the equational setting. From now on we will consider only term graphs in flat form and without useless equations (garbage) that will be removed automatically during rewriting.

Definition 71 (Term graph rewrite rule) *A term graph rewrite rule is a pair of term graphs (L, R) such that L and R have the same root, L is not a single variable and $\mathcal{FV}(R) \subseteq \mathcal{FV}(L)$. We say that a rewrite rule is* left-linear *if L is acyclic and every variable appears at most once in the right-hand side of the recursion equations of L.*

In the following we will restrict to left-linear rewrite rules. This means that the term graph $\{x \mid x = g(x)\}$ is not allowed as left-hand side of a rewrite rule, since it is cyclic, and the term graph $\{x \mid x = g(y, y), y = a\}$ is not allowed since the variable y appears twice in the right-hand side of the recursion equations. Indeed, the left-hand sides of left-linear rewrite rules must be trees.

Definition 72 (TGR) *A term graph rewrite system $TGR = (\Sigma, \mathcal{R})$ consists of a signature Σ and a set of rewrite rules $\mathcal{R}$ over this signature.*

A rewrite rule can be applied to a term graph if there exists a match between its left-hand side and the graph. We point out that, since match is often a synonymous of homomorphism in graph rewriting, a rule matches also graphs containing more sharing than its left-hand side. For term graphs in flat form, considered here, the homomorphism σ is simply a variable substitution.

Definition 73 (Substitution, Matching and Redex)

- *A substitution $\sigma = \{y_1/x_1, \ldots, y_n/x_n\}$ is a map from variables to variables. Its application to a term graph G, denoted $\sigma(G)$, is inductively defined as follows:*

$$\sigma(\{z_1 \mid z_1 = t_1, \ldots, z_n = t_n\}) = \{\sigma(z_1) \mid \sigma(z_1) = \sigma(t_1), \ldots, \sigma(z_n) = \sigma(t_n)\}$$

$$\sigma(z) = \begin{cases} x_i & \text{if } z = y_i \in \{y_1, \ldots, y_n\} \\ z & \text{otherwise} \end{cases} \qquad \sigma(f(t_1, \ldots, t_n)) = f(\sigma(t_1), \ldots, \sigma(t_n))$$

- *A homomorphism (matching) from a term graph L to a term graph G is a substitution σ such that $\sigma(L) \subseteq G$, where the inclusion means that all recursion equations of $\sigma(L)$ are present in G, i.e. if $\sigma(L) = \{x_1 \mid E\}$ then $G = \{x_1' \mid E, E'\}$*

- *A redex in a term graph G is a pair $((L, R), \sigma)$ where (L, R) is a rule and σ is an homomorphism from the left-hand side L of the rule to G. If x is the root of L, we call $\sigma(x)$ the head of the redex.*

We introduce next the notions of path and position, similar to the ones used in the previous section, later used to define a rewrite step.

Definition 74 (Path, position) *A* path *in a graph G is a sequence of symbols interleaved by integers $p = f_1 i_1 f_2 \dots i_{n-1} f_n$ such that f_{j+1} is the i_j-th argument of f_j, for all $j = 0, \dots, n-1$. We call $|p| = n$ the* length *of the path p. The sequence of integers $i_1 \dots i_{n-1}$ is called the* position *of the node labeled f_n and still denoted with the letter p.*

One of the advantages of the equational representation is that the notion of context can be conveniently expressed syntactically. Similarly to what we have done for terms, we introduce the notations $G_{|p}$ for the subterm of G at the position p in G, while $G_{[G']_p}$ specifies that G contains a term graph G' at the position p. In the same situation, if z is the root of G' and $z = t$ is the corresponding equation we will also write $G_{[z=t]_p}$. By the notation $G_{\lceil G' \rceil_p}$ we denote the term graph G where the subgraph $G_{|p}$, or more precisely the equation defining the root of $G_{|p}$, has been replaced by G'. Given for instance the two term graphs $G_{[z=t]_p}$ and $G' = z[E_{G'}]$, the term graph $G_{\lceil G' \rceil_p}$ is obtained from G by replacing the equation $z = t$ by $E_{G'}$, and then possibly performing garbage collection. Intuitively, the term graph G' is attached to the node z in G. With respect to term rewriting, the main difference is that there may be more than one path in G to reach the position p and that the subgraph G' may have several links to G, other than its root.

The notions of path and position are used to define a rewrite step.

Definition 75 (Rewrite step) *Let $((L, R), \sigma)$ be a redex occurring in G. A rewrite step which reduces the redex above consists of removing the equation specified by the head of the redex and of replacing it by the body of $\sigma(R)$, with a fresh choice of bound variables. Using a context notation: $G_{[\sigma(x)=t]} \to G_{\lceil \sigma(R) \rceil}$*

We give next an example of rewriting. Note that only the root equation gets rewritten and it is replaced by several equations. Renaming is necessary to avoid collisions with other variables already present in the system and this is quite natural since variables in the recursion equations are implicitly existentially quantified.

Example 32 (Rewriting) *Let $G_1 = \{x_1 \mid x_1 = add(x_2, x_3), x_2 = s(x_4), x_3 = s(x_4), x_4 = 0\}$ be a closed term graph in flat form and let $(L, R) = (\{y_1 \mid y_1 = add(y_2, z_2), y_2 = s(z_1)\}, \{y_1 \mid y_1 = s(y_2), y_2 = add(z_1, z_2)\})$ be a rewrite rule (see Figure 6.2) where y_i denote the bound variables and z_i denote the free variables, $i = 1, 2$. A matching between L and G_1 is given by the substitution $\sigma = \{y_1/x_1, y_2/x_2, z_1/x_4, z_2/x_3\}$. The rewrite step is performed at the root of G_1. We have*

$$\begin{array}{rcl} G_1 & = & \{x_1 \mid \underline{x_1 = add(x_2, x_3)}, x_2 = s(x_4), x_3 = s(x_4), x_4 = 0\} \\ & \to & \{x_1 \mid \underline{x_1 = s(x_2'), x_2' = add(x_4, x_3)}, x_2 = s(x_4), x_3 = s(x_4), x_4 = 0\} = G_3' \end{array}$$

where the underlined equation in G_1 is rewritten into the underlined equations of G_3'. The resulting term graph G_3, obtained from G_3' after garbage collection, is also depicted in Figure 6.2.

6.4 The cyclic lambda calculus

The cyclic λ-calculus introduced by Ariola and Klop [AK96b] generalises the ordinary λ-calculus by allowing sharing and cycles, thus providing an equational framework for higher-order term graph rewriting.

Terms, called λ-graphs, are represented as systems of (possibly nested) recursion equations on standard λ-terms. If the system is used without restrictions on the rules, the confluence is lost. The authors restore it by controlling the operations on the recursion equations, as we will see later. The resulting calculus, called $\lambda\phi$, is powerful enough to express the classical λ-calculus [Bar84] and also the $\lambda\mu$-calculus [Par92] and the $\lambda\sigma$-calculus with names [ACCL91].

Given a signature $\Sigma = (\mathcal{F}, \mathcal{X})$, the syntax of $\lambda\phi$ over Σ is the following:

$$\mathcal{T} \quad ::= \quad \mathcal{X} \mid \mathcal{F}^n(\mathcal{T}_1, \ldots \mathcal{T}_n) \mid \mathcal{T}\ \mathcal{T} \mid \lambda\mathcal{X}.\mathcal{T} \mid \langle \mathcal{T}_0 \mid \mathcal{X}_1 = \mathcal{T}_1, \ldots, \mathcal{X}_n = \mathcal{T}_n \rangle$$

The set of $\lambda\phi$-terms is composed of the ordinary λ-terms (*i.e.* variables, functions of fixed arity, applications, abstractions) and of new terms built using the construct: $\langle t_0 \mid x_1 = t_1, \ldots, x_n = t_n \rangle$, where we suppose the recursion variables x_i, $i = 1, \ldots, n$ all distinct. Intuitively, this can be read as the construct `letrec` $x_1 = t_1, \ldots, x_n = t_n$ `in` t_0. Sometimes the recursion variables $x_1, \ldots, x_n$ can be referred to as the *names* of the terms $t_1, \ldots, t_n$, respectively. We denote by E an unordered sequence of equations $x_1 = t_1, \ldots, x_n = t_n$ and by ϵ the empty sequence. The notation $x =_\circ x$ is an abbreviation for the sequence of recursion equations $x = x_1, \ldots, x_n = x$.

Variables in λ-graphs are bound if they appear in the scope of a lambda abstraction or a recursion equation. The scope of a λ, as in the classical λ-calculus, is the body of the abstraction, while the scope of a recursion equation $x = t$ is the λ-graph $\langle t_0 \mid \ldots, x = t, \ldots \rangle$ where the equation is defined. As usual, variables that are not bound are called free.

Definition 76 (Free variables) *The set of* free variables *of a λ-graph t, denoted $\mathcal{FV}(t)$, is recursively defined as follows:*

$$\begin{array}{lcllcl}
\mathcal{FV}(x) & = & \{x\} & \mathcal{FV}(\lambda x.t) & = & \mathcal{FV}(t) \setminus \{x\} \\
\mathcal{FV}(f(t_1, \ldots, t_n)) & = & \bigcup_{i=1}^n \mathcal{FV}(t_i) & \mathcal{FV}(t_1\ t_2) & = & \mathcal{FV}(t_1) \cup \mathcal{FV}(t_2)
\end{array}$$

$$\mathcal{FV}(\langle t_0 \mid\ x_1 = t_1, \ldots, x_n = t_n \rangle) = \bigcup_{i=0}^n \mathcal{FV}(t_i) \setminus \{x_1, \ldots, x_n\}$$

The classical notion of α-conversion given for the λ-calculus must be adapted to take into account the two binders of the calculus, *e.g.* following the general ideas in [Blo01]. We first define first-order variable substitution, also called syntactic variable replacement, on λ-graphs.

Definition 77 (Variable substitution) *A variable substitution σ is a mapping from the set of variables to the set of variables. A finite substitution has the form $\sigma = \{y_1/z_1 \ldots y_m/z_m\}$ The application of a substitution σ to a λ-graph t, denoted by $\sigma(t)$ or $t\sigma$, is recursively defined as follows:*

$$\begin{array}{lcllcl}
\sigma(x_i) & = & \left\{ \begin{array}{ll} z_i & \text{if } x_i \in \{y_1, \ldots, y_m\} \\ x_i & \text{otherwise} \end{array} \right. & \sigma(\lambda x.t) & = & \lambda\sigma(x).\sigma(t) \\
\sigma(t_0\ t_1) & = & \sigma(t_0)\ \sigma(t_1) & \sigma(f(t_1, \ldots, t_m)) & = & f(\sigma(t_1), \ldots, \sigma(t_m))
\end{array}$$

$$\sigma(\langle t_0 \mid\ x_1 = t_1, \ldots, x_n = t_n \rangle) \quad = \quad \langle \sigma(t_0) \mid \sigma(x_1) = \sigma(t_1), \ldots, \sigma(x_n) = \sigma(t_n) \rangle$$

Definition 78 (α-conversion) *The α-conversion is defined on λ-graphs as follows.*

$$\begin{array}{lcl}
\lambda x_1.t & =_\alpha & \sigma(\lambda x_1.t) \\
\langle t_0 \mid\ x_1 = t_1, \ldots, x_n = t_n \rangle & =_\alpha & \sigma(\langle t_0 \mid\ x_1 = t_1, \ldots, x_n = t_n \rangle)
\end{array}$$

where $\sigma(y) = \left\{ \begin{array}{ll} z_i & \text{if } y = x_i \\ y & \text{otherwise} \end{array} \right.$ *and we suppose the variables z_i all distinct and fresh, i.e. different from $x_1, \ldots, x_n$ and not occurring in $t_0, \ldots, t_n$.*

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

Figure 6.4: Evaluation rules of the $\lambda\phi$-calculus

In the following we will always consider λ-graphs modulo α-conversion. This allow us to define higher-order substitution on λ-graphs.

Definition 79 (Higher-order substitution) *A variable substitution σ is a mapping from the set of variables to the set of variables. A finite substitution has the form $\sigma = \{y_1/t_1 \ldots y_m/t_m\}$ The application of a substitution σ to a λ-graph t, denoted by $\sigma(t)$ or $t\sigma$, is recursively defined as follows:*

$$\sigma(x_i) = \begin{cases} t_i & \text{if } x_i \in \{y_1, \ldots, y_m\} \\ x_i & \text{otherwise} \end{cases} \qquad \begin{array}{lcl} \sigma(\lambda x.t) & = & \lambda x.\sigma(t) \\ \sigma(f(t_1, \ldots, t_m)) & = & f(\sigma(t_1), \ldots, \sigma(t_m)) \end{array}$$

$$\sigma(t_0\ t_1) = \sigma(t_0)\ \sigma(t_1)$$

$$\sigma(\langle t_0 \mid\ x_1 = t_1, \ldots, x_n = t_n\rangle) = \langle \sigma(t_0) \mid x_1 = \sigma(t_1), \ldots, x_n = \sigma(t_n)\rangle$$

This defines higher-order substitution, as we work modulo α-conversion: when applying a substitution to an abstraction $\lambda x.t$ (or a recursion equation $x = t$), we are sure that the variable x does not belong to the domain of the substitution.

λ-graphs can be transformed using the reduction rules given in Figure 6.4. Some extensions of this basic set of rules can be considered [AK96b] by adding either distribution rules ($\lambda\phi_1$) or merging and elimination rules ($\lambda\phi_2$) for the box construct $\langle \_ \mid \_ \rangle$. In the following we will concentrate our attention on the basic system of Figure 6.4 and we will denote by $\mapsto_{\lambda\phi}$ and $\mapsto\!\!\!\!\to_{\lambda\phi}$ the rewrite relations it induces on λ-graphs.

The notion of β-redex does not change with respect to the one defined for the λ-calculus, therefore the (β) rule is similar to the classical β-rule, except that here the parameter x does not take immediately its actual value t_2. At first, the variable x bound by "λ" becomes bound by a recursion equation, subsequently the occurences of x in t_1 can be replaced by their value using the substitutions rules, namely the (*external sub*) and (*acyclic sub*) rules, which make a copy of a graph associated to a recursion variable. The (*black hole*) rules are a particular case of substitution rules in which undefined expression are replaced by the constant "$\bullet$". The (*garbage collect*) rules are used to eliminate useless recursion equations and empty lists. The condition $E' \neq \epsilon$ avoids trivial non-terminating reductions and the proviso $E' \perp (E, t)$, to be read as E' is orthogonal to the sequence of equations E and the term t, ensures that E' has no connections with the rest of the graph. More formally, a sequence of equations E' is defined as *orthogonal* to E and t if the recursion bound variables of E' do not intersect the set of free variables of E and t.

Some of the rules, namely the (*acyclic sub*) rule and the second (*black hole*) rule, are imposed a particular restriction on their application. We define next the pre-order $\leq$ on recursion variables

used for the definition of this restriction. We recall that a context $\mathsf{Ctx}\{\Box\}$ is a term with a single hole $\Box$ in the place of a subterm. Filling the context $\mathsf{Ctx}\{\Box\}$ with a term t yields the term $\mathsf{Ctx}\{t\}$.

Definition 80 *Let x and y be two recursion variables in a list of constraints E. We define the least pre-order $\geq$ on recursion variables as $x \geq y$ if $x = \mathsf{Ctx}\{y\}$, for some context $\mathsf{Ctx}\{\Box\}$. The equivalence induced by the pre-order is denoted $\equiv$ and we say that x and y are cyclically equivalent ($x \equiv y$) if $x \geq y \geq x$ (they lie on a common cycle). We write $x > y$ if $x \geq y$ and $x \not\equiv y$.*

The restriction on the order of recursion variables on the mentioned rules is introduced to ensure the confluence of the system. Without this condition, a counterexample to confluence, whose complete analysis can be found in [AK96b], is given by a term having cyclic configurations of (β) redexes.

Example 33 (Confluent reduction) *Consider the λ-graph $t = \langle x_1 \mid x_1 = \lambda y_1.x_2\ s(y_1), x_2 = \lambda y_2.x_1\ s(y_2)\rangle$ having two mutually nested cyclic redexes. We show first how this graph leads to two diverging reductions, if the proviso $y > x$ is not considered in the (acyclic sub) rule. We denote by $\mapsto_{\beta+}$ the reduction of a (β) redex using the sequence of steps $\mapsto_\beta \mapsto_{es} \mapsto_{gc}$. We have the reduction*

$$
\begin{aligned}
& \langle x_1 \mid x_1 = \lambda y_1.\underline{x_2}\ s(y_1), \underline{x_2 = \lambda y_2.x_1\ s(y_2)}\rangle \\
\mapsto_{ac}\ & \langle x_1 \mid x_1 = \lambda y_1.\underline{(\lambda y_2.x_1\ s(y_2))\ s(y_1)}, x_2 = \lambda y_2.x_1\ s(y_2)\rangle \\
\mapsto_{\beta+}\ & \langle x_1 \mid x_1 = \lambda y_1.x_1\ s(s(y_1)), \underline{x_2 = \lambda y_2.x_1\ s(y_2)}\rangle \\
\mapsto_{gc}\ & \langle x_1 \mid x_1 = \lambda y_1.x_1\ s(s(y_1))\rangle\ =\ t_1 \\
\mapsto_{ac}\ & \ldots
\end{aligned}
$$

and a different possible reduction

$$
\begin{aligned}
& \langle x_1 \mid \underline{x_1 = \lambda y_1.x_2\ s(y_1)}, x_2 = \lambda y_2.\underline{x_1}\ s(y_2)\rangle \\
\mapsto_{ac}\ & \langle x_1 \mid x_1 = \lambda y_1.x_2\ s(y_1), x_2 = \lambda y_2.\underline{(\lambda y_1.x_2\ s(y_1))\ s(y_2)}\rangle \\
\mapsto_{\beta+}\ & \langle x_1 \mid x_1 = \lambda y_1.x_2\ s(y_1), x_2 = \lambda y_2.x_2\ s(s(y_2))\rangle\ =\ t_2 \\
\mapsto_{ac}\ & \ldots
\end{aligned}
$$

It is clear that the two graphs t_1 and t_2 are irreversibly separated with respect to any possible sequence of rewritings, since the first one will always have a odd number and the second one an even number of symbols s.

We show now the reduction of t in the $\lambda\phi$-calculus. To ease the notation, we denote the two recursion equations $x_1 = \lambda y_1.x_2\ s(y_1)$ and $x_2 = \lambda y_2.x_1\ s(y_2)$ by E_1 and E_2 respectively.

$$
\begin{aligned}
t = \quad & \langle \underline{x_1} \mid \underline{E_1}, E_2\rangle \\
\mapsto_{es}\ & \langle \lambda y_1.\underline{x_2}\ s(y_1) \mid E_1, \underline{E_2}\rangle \\
\mapsto_{es}\ & \langle \lambda y_1.\underline{(\lambda y_2.x_1\ s(y_2))\ s(y_1)} \mid E_1, E_2\rangle \\
\mapsto_{\beta+}\ & \langle \lambda y_1.\underline{x_1}\ s(s(y_1)) \mid \underline{E_1}, E_2\rangle \\
\mapsto_{es}\ & \langle \lambda y_1.\underline{(\lambda y_1.x_2\ s(y_1))\ s(s(y_1))} \mid E_1, E_2\rangle \\
\mapsto_{\beta+}\ & \langle \lambda y_1.x_2\ s(s(s(y_2))) \mid E_1, E_2\rangle \\
\mapsto_{es}\ & \ldots
\end{aligned}
$$

Notice that in this case the only possible substitutions are the ones performed by the (external sub) rule. The (acyclic sub) rule can not be applied since the constraint on the order is not satisfied: we have $x_2 > x_1$ and $x_1 > x_2$, thus $x_1 \equiv x_2$, meaning that the two λ-terms with names x_1 and x_2 are on the same cycle.

Conclusion

In this chapter we presented different approaches to term graph rewriting, namely the operational approach, the categorical approach and the equational approach, giving the basic notions of term graph, rewrite rule, matching and rewrite step in the different frameworks.

The three approaches mainly differ for the formalism used for modeling sharing and cycles and for the adopted graph rewriting mechanism. The classical operational approach stresses the computational aspects of rewriting. Graphs are intended to provide an efficient representation of the "corresponding" (rational) terms and the rewriting process is split into a three-phase construction (matching, replacement using pointers manipulations and garbage collection). The approach reflects the actual implementation of functional programming languages and can be is useful in the analysis of time and space issues.

On the other hand, the algebraic approach is a general rewriting framework that applies, among others, to the category of term graphs. The categorical description enlights the algebraic structure of term graphs and of term graph rewriting, since it is largely independent from the " syntactical" representation. An advantage of this approach is the fact that numerous concepts and theorems based only on categorical properties and developed in other contexts are immediately available also for the category of term graphs.

The third approach provides an equational presentations of term graph rewriting, in which a term graph is represented as a system of recursion equations and rewriting is modeled by means of equational transformations on these systems. As in the first approach, the interest is for the terms corresponding to the manipulated term graphs, but some operational details of the rewriting mechanism are abstracted away, thus allowing for a more elegant formal treatment.

In the same spirit, another term graph rewriting formalism based on λ-calculus with cyclic λ-terms is the *Cyclic λ-calculus*, presented in the last section. The classical λ-calculus is generalised with a `letrec` like construct for modelling cyclic structures expressed in an equational style. The classical β-rule of the λ-calculus is adapted to the equational setting and some additional rules are introduced to deal with the new graph features of the terms. This calculus has been one of our main references in the development of the graph rewriting calculus introduced in the next chapter. We chose this calculus as starting point of our work for several reasons. On the one hand, like the ρ-calculus, it supports higher-order terms. On the other hand, recursion equations can be seen as a new form of constraint in a ρ-term and thus the generalisation of the ρ-calculus to include this new unification constraint can be done naturally.

Chapter 7

The graph rewriting calculus

In Chapter 4 and Chapter 6 we presented, respectively, the ρ-calculus, a generalisation of both term rewriting and λ-calculus, and term graph rewriting, insisting in particular on an extension of the λ-calculus with explicit recursion called cyclic λ-calculus.

The ρ-calculus and the cyclic λ-calculus, are both intended to provide an abstract model of computation for functional programming languages and therefore they share some common features, like the representation of functions by means of an abstraction operator and some degree of control on the rewrite relation thanks to the explicit application operator. They also have some complementary useful properties, like the possibility of discrimination on terms through the matching features in the ρ-calculus and the efficiency, in terms of space and speed of the reductions, in the cyclic λ-calculus, due to the use of graph-like structures.

Taking inspiration from these two calculi, we introduce the ρ_{g}-calculus, a common framework integrating in a natural way higher-order capabilities, graphical structures and matching features. We give in this chapter a general introduction to the ρ_{g}-calculus, enlightening on the one hand its similarities with the standard ρ-calculus and other systems, and on the other hand its new features. Chapters 8 and 9 are dedicated, respectively, to analyse the confluence property for this calculus and its expressive capabilities.

7.1 General overview

In term rewriting, terms are not described at the same level as rewrite rules and, in particular, the application of a rule to a term is defined at the meta level. Consequently, we have no control on the choice of the rewrite rule which is applied to a term, nor on the application position of this rule in the term. This lack of control can be overcome by dealing explicitly with the notions of rewrite rule, rewrite rule application and result of such application. The uniform treatment of these components at the object level is one of the main characteristics of the ρ-calculus, together with the notion of generalised abstraction that allows one to have in the left-hand side of the binder "$\_ \twoheadrightarrow \_$" more elaborated terms than simple variables, as in $f(x) \twoheadrightarrow x$ which abstracts on terms with shape $f(\ldots)$. All these features are provided also by the ρ_{g}-calculus. ρ_{g}-terms, also called ρ_{g}-graphs and denoted with capital letters $G, H, \ldots$, are built using a first-order signature and the same operators used for defining ρ-terms: an application operator denoted by concatenation, an abstraction operator "$\_ \twoheadrightarrow \_$", an operator for grouping terms together, denoted "$\_ \wr \_$" and a constraint application operator "$\_[\_]$". As explained for the ρ-calculus, this last operator is useful, for example, to keep track of the failures in rule applications, as it happens when the rule $f(x) \twoheadrightarrow x$ is applied to the term $g(a)$, denoted $(f(x) \twoheadrightarrow x)\ g(a)$. The

trace of the unsuccessful matching is kept in the reduction using a delayed matching constraint $x[f(x) \ll g(a)]$ which cannot be reduced furthermore.

A new aspect of the $\rho_{\mathtt{g}}$-calculus is the generalisation of this kind of constraints from singletons of the form $G_0 \ll G_1$ to lists of constraints written $G_0 \ll G_1, \ldots, G_{n-1} \ll G_n$, built using the "_, _" operator. This generalisation had already been considered for the $\rho_{\mathtt{x}}$-calculus (see Section 4.4) and it is useful for performing matching at the object level, in the sense that a matching problem, like $f(x, y) \ll f(a, b)$ is explicitly solved by subsequent decompositions into simpler constraints until only elementary problems $x \ll a, y \ll b$ remain in the same list of constraints.

Lists of constraints can be also used to model graph-like structures. In order to do this, besides the matching constraints, $\rho_{\mathtt{g}}$-terms are equipped with a new kind of constraint, *i.e.* recursion equations of the form $x = G$. These equations, which resemble those of the cyclic λ-calculus, are used to represent sharing and cyclic features in a $\rho_{\mathtt{g}}$-term. For example, in the graph $g(x, x)[x = a]$, we can think at the variable x on the left-hand side of the equation as the name given to the node labelled a in the graph. The two occurrences of x which appear as arguments of the function g can be thus seen as two pointers to the same node a. We can of course have more complex expressions than a simple constant on the right-hand side of an equation: we can share a whole term as in $g(x, x)[x = f(a, g(b))]$ or we can model cycles by defining looping pointers as in $x[x = f(x)]$. The fact that a constraint may be composed by several equations allows us to express also the so-called "twisted sharing", by means of two equations mutually pointing to each other like in $x[x = f(y), y = g(x)]$, and to express cycles of length greater than one, like in $x[x = f(y), y = g(z), z = f(y, x)]$. We mention that the order of constraints in the list is irrelevant, thus the previous graph has to be considered equivalent to $x[y = g(z), x = f(y), z = f(y, x)]$.

To summarise, $\rho_{\mathtt{g}}$-terms can be associated to lists of constraints, whose general form is $G_0 \ll G_1, x_1 = H_1, \ldots, G_{n-1} \ll G_n, x_m = H_m$. This increases the expressiveness of the $\rho_{\mathtt{g}}$-calculus with respect to the plain ρ-calculus, allowing to deal with explicit matching and to represent regular infinite entities as cyclic term graphs.

$\rho_{\mathtt{g}}$-terms can be manipulated using a set of appropriate transformation rules. Comparing with the evaluations rules of the ρ-calculus, those of the $\rho_{\mathtt{g}}$-calculus are enriched with a set of rules dealing with the term graph features of $\rho_{\mathtt{g}}$-terms, and a set of rules computing the solution of matching constraints. In the ρ-calculus, a successful reduction is obtained starting for instance from the term $(f(x) \twoheadrightarrow g(x))\ f(a)$. Reducing this term, we get $g(x)[f(x) \ll f(a)]$. The evaluation of the matching constraint leads to the substitution $\{x/a\}$ which applied to the left-hand side of the constraint, leads to the final result $g(a)$ of the rewriting. The steps needed for solving the matching and applying the substitution are done at the meta level using the (σ) rule. Instead in the $\rho_{\mathtt{g}}$-calculus the computation of matching is done explicitly by decomposing the original matching problem into simpler problems and eventually into elementary problems that can be immediately solved by turning them into equations, if the matching is successful. for instance, in the example above, the constraint $f(x) \ll f(a)$ is simplified into $x \ll a$ and finally solved in $x = a$. It is worth noticing that the solved form of a matching constraint is again a constraint, namely a recursion equation. As a matter of fact, a substitution is thus represented by a recursion equation added to the list of constraints of the term. Therefore, the $\rho_{\mathtt{g}}$-term $(f(x) \twoheadrightarrow g(x))\ f(a)$ evaluates, once the matching is solved, to $g(x)[x = a]$. Also the application of the substitution to the term is then performed explicitly by using a convenient context notation to specify the occurrence at which the substitution has to take place. In this example, substituting a for the occurrence of x under the symbol g we obtain $g(a)[x = a]$. Finally, garbage collection is used to eliminate the equation $x = a$ which is henceforth superfluous since x does not occur any more

Terms

$\mathcal{G}$	::=	$\mathcal{X}$	(Variables)
	|	$\mathcal{K}$	(Constants)
	|	$\mathcal{G} \twoheadrightarrow \mathcal{G}$	(Abstraction)
	|	$\mathcal{G}\ \mathcal{G}$	(Functional application)
	|	$\mathcal{G} \wr \mathcal{G}$	(Structure)
	|	$\mathcal{G}[\mathcal{C}]$	(Constraint application)

Constraints

$\mathcal{C}$	::=	ϵ	(Empty constraint)
	|	$\mathcal{X} = \mathcal{G}$	(Recursion equation)
	|	$\mathcal{G} \ll \mathcal{G}$	(Match equation)
	|	$\mathcal{C}, \mathcal{C}$	(Conjunction of constraints)

Figure 7.1: Syntax of the ρ_{g}-calculus

in the term. We obtain in this way the final term $g(a)$.

It is worth observing that, differently from what happens for usual term rewriting, here we do not necessarily want to apply immediately the substitution to the term $g(x)$. From a semantic point of view, the term $g(x)[x = a]$ is equivalent to the term $g(a)$, the only difference being that in the first case the value information for x is kept in the associated constraint environment. Delaying the application of a substitution is particularly interesting in case of sharing. For example in the ρ_{g}-graph $g(x, x)[x = a]$ the two instances of the constant a are intuitively shared, while after the application of the substitution we obtain $g(a, a)$ which can be seen as a tree where the two constants a are "distinct". This makes clear the interest of a strategy that can be used to maintain sharing in the reductions as long as possible, allowing the application of substitutions only when this is really needed, for example for unfreezing other redexes in the graph.

7.2 The syntax

The syntax of the ρ_{g}-calculus, describing formally how the objects handled in the calculus are built, is presented in Figure 7.1. The ρ_{g}-terms are built from a set of variables, a set of constants and a set of operators, most of which were already in the syntax of the ρ-calculus. As in the ρ-calculus, in the ρ_{g}-calculus terms of the form $f\ G_1 \dots G_n$ are usually denoted by $f(G_1, \dots, G_n)$ and operators have different priorities. The operators ordered from higher to lower priority are: application "_ _", '_[_]" "_ ↠ _", "_≀_", "_ ≪ _", "_ = _" and "_, _".

Rewrite rules can be expressed using the abstraction operator "_ ↠ _" which leads to terms of the form $G_1 \twoheadrightarrow G_2$, where G_1 is called the *pattern*. In the most general definition no restrictions are imposed on the rewrite rules, *i.e.* a pattern may be any ρ_{g}-term. We will see in the following that we will need to restrict to specific subsets of patterns in order to obtain some nice properties for the calculus. Terms can be grouped together into *structures* built using the operator "_≀_". As in the ρ-calculus, this operator is useful for representing the (non-deterministic) application of a set of rewrite rules and consequently, the set of possible results arising from the different choices. We can distinguish two different application operators: the functional application operator, denoted simply by concatenation (and by @ in graphical presentations), and the constraint application operator, denoted by "_[_]".

The ρ_{g}-calculus deals explicitly with matching constraints and introduces also a new kind of constraint, the recursion equations. A matching constraint $G_1 \ll G_2$ is generated when reducing the application of a rewrite rule to a term, like for example the application $(f(x) \twoheadrightarrow g(x))\ f(a)$ which reduces to $g(x)[f(x) \ll f(a)]$. A recursion equation is a constraint of the form $x = G$ which can be thought as an environment associated to a term. It is used to define terms with

sharing and cycles and can be seen also as a delayed substitution. For example in the term $G = +(x, x)[x = s(0)]$ the term $s(0)$ is considered shared and we can think of $s(0)$ as the value associated to the variable x in the term G.

In short, ρ_{g}-calculus constraints, denoted by $E, F, \ldots$, are conjunctions (built using the operator "$\_, \_$") of match equations and recursion equations. The empty constraint is denoted by ϵ. The operator "$\_, \_$" is considered associative, commutative and idempotent, with ϵ as neutral element. This means that we consider equal two ρ_{g}-terms that differs only for the order of the constraints in their lists, like $x[x = f(y), y = g(x)]$ and $x[y = g(x), x = f(y)]$ since, intuitively, they represents the same term graph. Similarly, we suppose the operator "$\_, \_$" to be associative, thus $x[\big(x = f(y), y = g(z)\big), z = a]$ is equivalent to $x[x = f(y), \big(y = g(z), z = a\big)]$ and thus usually brackets are omitted. The idempotency axioms avoids the duplication of identical constraints, thus $x[x \ll f(y), x \ll f(y)]$ is equivalent to $x[x \ll f(y)]$. Duplication can occur during the reduction of terms containing non-linear patterns, like for example the term $(f(x, x) \twoheadrightarrow x)\ f(a, a)$. In this example, the generated matching constraint $f(x, x) \ll f(a, a)$ can be decomposed into two identical constraints leading to the term $x[x \ll a, x \ll a]$ which is simplified by idempotency into $x[x \ll a]$.

A list of constraints where recursion equations have mutual references to each other, like in $x[x = f(y), y = g(x)]$, is used to represent cyclic terms. To define more precisely the notion of cycle, we first define a context $\mathsf{Ctx}\{\Box\}$, analogous to the notion of context already introduced in Section 1.1 for first-order terms.

Definition 81 (Context) *A context $\mathsf{Ctx}\{\Box\}$ is a ρ_{g}-term containing exactly one empty place, that is one occurrence of the constant $\Box$, called "hole". The result of replacing the hole $\Box$ in $\mathsf{Ctx}\{\Box\}$ by a ρ_{g}-term G is denoted by $\mathsf{Ctx}\{G\}$.*

Note that insertion into a context is not a capture-free substitution, *i.e.* variables occurring free in G may become bound in $\mathsf{Ctx}\{G\}$.

Definition 82 (Order, cycle) *We denote by $\leq$ the least pre-order on recursion variables such that $x \geq y$ if $\mathsf{Ctx}_1\{x\} \lll \mathsf{Ctx}_2\{y\}$ for some contexts $\mathsf{Ctx}_i\{\Box\}$, $i = 1, 2$, where the symbol $\lll$ can be the recursion operator $=$ or the match operator $\ll$. The equivalence induced by the pre-order is denoted $\equiv$ and we say that x and y are cyclically equivalent ($x \equiv y$) if $x \geq y \geq x$. We write $x > y$ if $x \geq y$ and $x \not\equiv y$.*

A cycle *is a sequences of constraints of the form $\mathsf{Ctx}_0\{x_0\} \lll \mathsf{Ctx}_1\{x_1\}, \mathsf{Ctx}_2\{x_1\} \lll \mathsf{Ctx}_3\{x_2\}, \ldots, \mathsf{Ctx}_m\{x_n\} \lll \mathsf{Ctx}_{m+1}\{x_0\}$, with $n, m \in \mathbb{N}$, where $x_0 \equiv x_1 \equiv \ldots \equiv x_n$. We say that a ρ_{g}-term is* acyclic *if it contains no cycle.*

Degenerated cycles, *i.e.* sequences of the form $x = x_1, \ldots, x_n = x$ are denoted using the abbreviation $x =_\circ x$. Undefined terms that correspond to the expression $x[x =_\circ x]$ (self-loop) are denoted by the constant $\bullet$ (black hole) already introduced by Ariola and Klop using the equational approach and corresponding to the $\bot$ used in the categorical approach by Corradini (see Chapter 6).

We call a ρ_{g}-graph *well-formed* if each variable occurs at most once as left-hand side of a recursion equation. All the ρ_{g}-graphs considered in the sequel will be implicitly well-formed.

We give next several examples of ρ_{g}-terms. Slightly anticipating the definition that will be given in Section 7.3, in order to support the intuition of the reader a graphical representation of these terms is given in Figure 7.2.

Example 34 (Some ρ_{g}-terms)

1. *In the rule* $2 * f(x) \twoheadrightarrow (y + y)[y = f(x)]$ *the sharing in the right-hand side avoids the copying of the object instantiating* $f(x)$*, when the rule is applied to a* $\rho_{\mathtt{g}}$*-term.*
2. *The* $\rho_{\mathtt{g}}$*-term* $cons(head(x), x)[x = cons(0, x)]$ *represents an infinite list of zeros.*
3. *The* $\rho_{\mathtt{g}}$*-term* $G = f(x, y)[x = g(y), y = g(x)]$ *is an example of twisted sharing that can be expressed using a* `letrec`*-like construct. Notice that* G *is a cyclic term.*

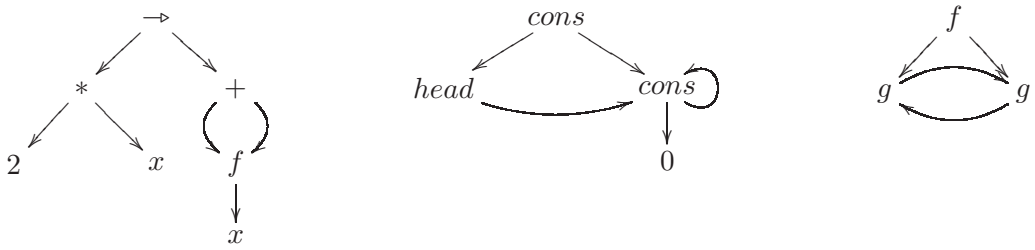


Figure 7.2: Some $\rho_{\mathtt{g}}$-terms

In order to model functionality and graph features conveniently, the $\rho_{\mathtt{g}}$-calculus is equipped with three different binders: the abstraction, the recursion and the match. Consequently, the notions of free and bound variables seen for the ρ-calculus must be adapted accordingly. To ease the definition, we introduce the domain of a constraint $\mathcal{C}$, denoted $\mathcal{DV}(\mathcal{C})$, as the set of variables (potentially) defined by the recursion and matching equations it contains. The set $\mathcal{DV}(\mathcal{C})$ includes, for any recursion equation $x = G$ in $\mathcal{C}$, the variable x and for any match $G_1 \ll G_2$ in $\mathcal{C}$, the set of free variables of G_1.

Definition 83 (Free, bound, and defined variables) *Given a* $\rho_{\mathtt{g}}$*-term* G*, its free variables, denoted* $\mathcal{FV}(G)$*, and its bound variables, denoted* $\mathcal{BV}(G)$*, are recursively defined below:*

G	$\mathcal{BV}(G)$	$\mathcal{FV}(G)$
x	$\emptyset$	$\{x\}$
k	$\emptyset$	$\emptyset$
$G_1\ G_2$	$\mathcal{BV}(G_1) \cup \mathcal{BV}(G_2)$	$\mathcal{FV}(G_1) \cup \mathcal{FV}(G_2)$
$G_1 \backslash G_2$	$\mathcal{BV}(G_1) \cup \mathcal{BV}(G_2)$	$\mathcal{FV}(G_1) \cup \mathcal{FV}(G_2)$
$G_1 \twoheadrightarrow G_2$	$\mathcal{FV}(G_1) \cup \mathcal{BV}(G_1) \cup \mathcal{BV}(G_2)$	$\mathcal{FV}(G_2) \setminus \mathcal{FV}(G_1)$
$G_0[E]$	$\mathcal{BV}(G_0) \cup \mathcal{BV}(E)$	$(\mathcal{FV}(G_0) \cup \mathcal{FV}(E)) \setminus \mathcal{DV}(E)$

For a given constraint E*, the free variables, denoted* $\mathcal{FV}(E)$*, the bound variables, denoted* $\mathcal{BV}(E)$*, and the defined variables, denoted* $\mathcal{DV}(E)$*, are defined as follows:*

E	$\mathcal{BV}(E)$	$\mathcal{FV}(E)$	$\mathcal{DV}(E)$
ϵ	$\emptyset$	$\emptyset$	$\emptyset$
$x = G_0$	$x \cup \mathcal{BV}(G_0)$	$\mathcal{FV}(G_0)$	$\{x\}$
$G_1 \ll G_2$	$\mathcal{FV}(G_1) \cup \mathcal{BV}(G_1) \cup \mathcal{BV}(G_2)$	$\mathcal{FV}(G_2)$	$\mathcal{FV}(G_1)$
E_1, E_2	$\mathcal{BV}(E_1) \cup \mathcal{BV}(E_2)$	$\mathcal{FV}(E_1) \cup \mathcal{FV}(E_2)$	$\mathcal{DV}(E_1) \cup \mathcal{DV}(E_2)$

It is worth remarking that the set of bound variables in the subterm G of a constraint application $G[E]$ is the domain of E plus the bound variables of G. For example, the bound variables of the term $(f(y) \twoheadrightarrow y)\ (g(x, z)[x \ll f(a)])$ are the variables x and y. Note also that

the visibility of a recursion variable is limited to the ρ_{g}-term appearing in the list of constraints where the recursion variable is defined and the ρ_{g}-term to which this list is applied. For example, in the term $f(x,y)[x = g(y)[y = a]]$ the variable y defined in the recursion equation binds its occurrence in $g(y)$ but not in $f(x,y)$. To avoid confusion and guarantee that free and bound variables have always different names in a ρ_{g}-term, an appropriate operation of renaming called α-conversion for its similarity with the α-conversion of the λ-calculus, is introduced. Using α-conversion, the previous term becomes $f(x,z)[x = g(y)[y = a]]$ where it is clear that variable z is free. The operation of α-conversion is useful also for avoiding the capture of free variables as a consequence of the application of a substitution to a term, like the free variable y in $y[x \ll a]$ which becomes bound in the term $\sigma(y[x \ll a]) = y[x \ll a]\{y/x\} = x[x \ll a]$.

Before defining α-conversion, we formalise in the natural way the notion of variable substitution in the setting of the ρ_{g}-calculus.

Definition 84 (Variable substitution) *A substitution σ is a mapping from the set of variables to the set of variables. A finite substitution has the form $\sigma = \{y_1/z_1, \ldots, y_m/z_m\}$. The application of a substitution σ to a ρ_{g}-term G, denoted by $\sigma(G)$ or $G\sigma$, is recursively defined as follows:*

$$\begin{array}{lcl} \sigma(x_i) & = & \left\{ \begin{array}{ll} z_i & \text{if } x_i \in \{y_1, \ldots, y_m\} \\ x_i & \text{otherwise} \end{array} \right. \\ \sigma(G_0\ G_1) & = & \sigma(G_0)\ \sigma(G_1) \end{array} \qquad \begin{array}{lcl} \sigma(G_0 \wr G_1) & = & \sigma(G_0) \wr \sigma(G_1) \\ \sigma(G_0 \twoheadrightarrow G_1) & = & \sigma(G_0) \twoheadrightarrow \sigma(G_1) \\ \sigma(G[E]) & = & \sigma(G)[\sigma(E)] \end{array}$$

The application of a substitution σ to a list of constraints is defined as follows:

$$\begin{array}{lcl} \sigma(\epsilon) & = & \epsilon \\ \sigma(E_1, E_2) & = & \sigma(E_1), \sigma(E_2) \\ \sigma(G_1 \ll G_2) & = & \sigma(G_1) \ll \sigma(G_2) \\ \sigma(x = G) & = & \sigma(x) = \sigma(G) \end{array}$$

Definition 85 (α-conversion) *The α-conversion is defined on ρ_{g}-terms as follows:*

- *α-conversion for an abstraction, where z is supposed to be fresh in G_0 and G_1:*

$$G_0 \twoheadrightarrow G_1 \ =_\alpha \ \sigma(G_0 \twoheadrightarrow G_1) \qquad \text{with } \sigma(y) = \left\{ \begin{array}{ll} z & \text{if } y \in \mathcal{FV}(G_0) \\ y & \text{otherwise} \end{array} \right.$$

- *α-conversion for a term with constraints:*

$$\begin{array}{rl} & G_0[x_1 = H_1, \ldots, x_n = H_n, G_1 \ll K_1, \ldots, G_n \ll K_n] \\ =_\alpha & \sigma(G_0[x_1 = H_1, \ldots, x_n = H_n, G_1 \ll K_1, \ldots, G_n \ll K_n]) \end{array}$$

$$\text{with } \sigma(y) = \left\{ \begin{array}{ll} z_i & \text{if } \ y = x_i \ \text{ or } \ y \in \mathcal{FV}(G_i),\ i = 1, \ldots, n \\ y & \text{otherwise} \end{array} \right.$$

where we suppose the variables z_i all distinct and fresh, i.e. different from $x_1, \ldots, x_n$ and not occurring in $H_1, \ldots, H_n, G_0, \ldots, G_n, K_1, \ldots, K_n$.

The notion of α-conversion allow us to define higher-order substitution on ρ_{g}-terms.

Definition 86 (Higher-order substitution) *A substitution σ is a mapping from the set of variables to the set of ρ_{g}-terms. A finite substitution has the form $\sigma = \{y_1/G_1, \ldots, y_m/G_m\}$. The application of a substitution σ to a ρ_{g}-term G, denoted by $\sigma(G)$ or $G\sigma$, is recursively defined as follows:*

$$\begin{array}{lcl} \sigma(x_i) & = & \begin{cases} G_i & \text{if } x_i \in \{y_1, \ldots, y_m\} \\ x_i & \text{otherwise} \end{cases} \\ \sigma(G_0\ G_1) & = & \sigma(G_0)\ \sigma(G_1) \end{array} \qquad \begin{array}{lcl} \sigma(G_0 \wr G_1) & = & \sigma(G_0) \wr \sigma(G_1) \\ \sigma(G_0 \twoheadrightarrow G_1) & = & G_0 \twoheadrightarrow \sigma(G_1) \\ \sigma(G[E]) & = & \sigma(G)[\sigma(E)] \end{array}$$

The application of a substitution σ to a list of constraints is defined as follows:

$$\begin{array}{lcl} \sigma(\epsilon) & = & \epsilon \\ \sigma(E_1, E_2) & = & \sigma(E_1), \sigma(E_2) \\ \sigma(G_1 \ll G_2) & = & G_1 \ll \sigma(G_2) \\ \sigma(x = G) & = & x = \sigma(G) \end{array}$$

It is worth noticing that the definition of α-conversion, as well as the definition of free and bound variables, generalise the respective definitions given for the cyclic λ-calculus, whose terms can be recovered from ρ_{g}-terms by restricting the left-hand sides of abstractions to single variables and eliminating the structure and the match equations from constraints.

Since sometimes the notion of free (and bound) variables in not very intuitive, due to the presence of different binders in the calculus and to the fact that the sets of variables of the different constraints in a list are not necessarily disjoint, we give some examples about the visibility of bound variables and the need of renaming variables.

Example 35 (Free and bound variables should not have the same name)
Given the ρ_{g}-term $z[z = x \twoheadrightarrow y, y = x{+}x]$, one may think to naively replace the variable y by $x{+}x$ in the right-hand side of the abstraction, leading to a variable capture. This could happen because the previous term does not respect our naming conventions: the variable capture is no longer possible if we consider the legal ρ_{g}-term $z[z = x_1 \twoheadrightarrow y, y = x + x]$ obtained after α-conversion. In order to have the occurrences of the variable x appearing in the second constraint bounded by the arrow, we should use a nested constraint as in the ρ_{g}-term $z[z = x \twoheadrightarrow (y[y = x + x])]$.

Example 36 (Different bound variables should have different names)
Intuitively, according to the notions of free and bound variable, in a term there cannot be any sharing between the left-hand side of rewrite rules and the rest of a ρ_{g}-term. In other words, the left-hand side of a rewrite rule is self-contained. Sharing inside the left-hand side is allowed and no restrictions are imposed on the right-hand side. For example, in $f(y, y \twoheadrightarrow g(y))[y = x]$ the first occurrence of y is bound by the recursion variable, while the scope of the y in the abstraction "$\_ \twoheadrightarrow \_$" is limited to the right-hand side of the abstraction itself. The ρ_{g}-term should be in fact written (by α-conversion) as $f(y, z \twoheadrightarrow g(z))[y = x]$.

Form now on, when needed we will freely α-convert bound variables in a ρ_{g}-term. This naming conventions allows us to disregard some terms and thus to apply replacements (like for the evaluation rules in Figure 7.6) quite straightforwardly, since no variable capture needs to be considered.

Besides the naming conventions, we require for our purposes some structural properties on ρ_{g}-terms.

Definition 87 (Algebraic terms) *We call* algebraic *the ρ_{g}-terms defined by the following grammar:*

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

We will consider in the following only ρ_{g}-terms having algebraic acyclic patterns as left-hand sides of abstractions and match equations. For instance, the ρ_{g}-term $f(y)[y = g(y)] \twoheadrightarrow a$ is not allowed since the abstraction has a cyclic left-hand side. These restrictions are common from term graph rewriting, where usually the left-hand sides of rewrite rules are restricted *e.g.* to trees. Moreover, these restrictions are important to make the calculus enjoying some nice properties, like for example confluence, as it is detailed in the next chapter.

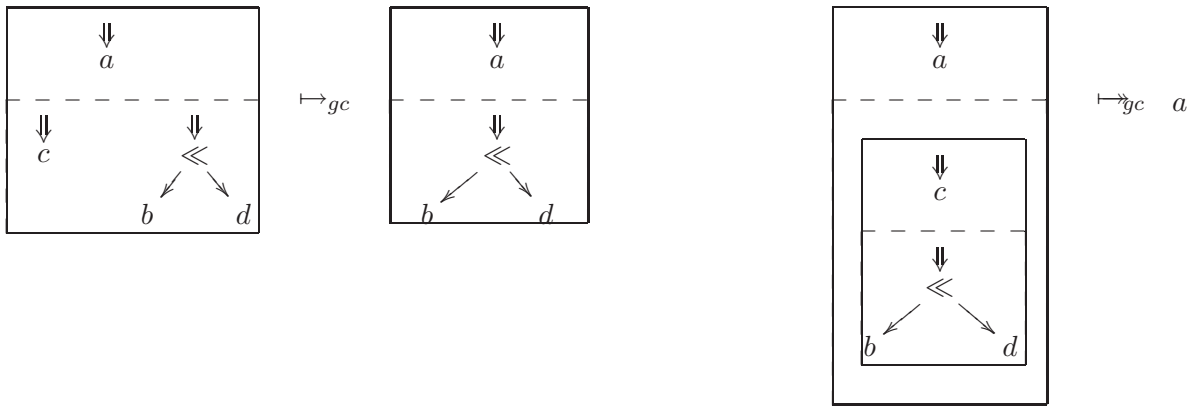
7.3 The graphical representation

Since ρ_{g}-terms describe objects with a graphical flavour, it can be convenient sometimes to illustrate them by a picture which is usually more intuitive than a long syntactical representation difficult to parse. We define in this section a graphical notation for ρ_{g}-terms whose aim is to help the intuition of the reader by associating pictures to ρ_{g}-terms.

Roughly, we can say that any term without constraints is represented as a directed acyclic graph without sharing in the obvious way, that is as a tree. A term with recursion equations $G[x_1 = G_1, \ldots, x_n = G_n]$ is read as a `letrec` construct $\texttt{letrec}\ x_1 = G_1, \ldots, x_n = G_n\ \texttt{in}\ G$ and represented as a term graph, that is a tree with possibly looping edges, as done in Section 6.1. The representation of the correspondence between a variable in the right-hand side of a rule and its binding occurrence in the pattern is done by keeping the variable names, instead of using back-pointers as it is done in other graphical interpretations of higher-order terms (see *e.g.* [Blo01]).

These two simple representations do not extend straightforwardly to general ρ_{g}-terms, possibly including matching constraints. Indeed, matching constraints can appear in a ρ_{g}-term, but they are not really part of its graphical structure, at least before having been solved. Recursion equations can be naturally interpreted as the addition of sharing and cycles to a standard tree. Instead, it is not completely clear, at first, what match equations should correspond to. We can think at them as graphs put in an environment (the list of constraints) and not directly rooted at the main graph. In particular, there may be some residual matchings even in a term in normal form for which the computation has not completely succeeded. From a graphical point of view, this term corresponds to a non-connected graph which has some isolated subgraphs, kept in its environment in order to express the matching failure. In the pictures, we chose to represent the environment, whose syntactical counterpart is the operator "_[_]", as a box. Given a ρ_{g}-term $G[E]$, in the upper part of the box we put the graph G, while in the lower part (under the dotted line) we put the representation of the constraints E. Since in a graph we can have nested lists of constraints, boxes can be nested in the pictures. Double vertical arrows denote the root of the main graph and the root of the subgraphs appearing in the environment.

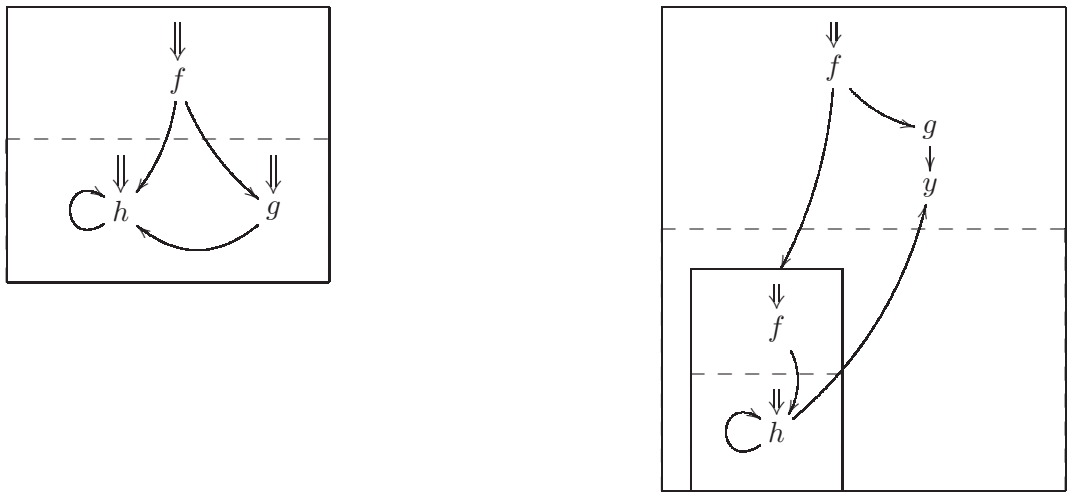
Some examples of this representation are shown in Figure 7.3 which depicts two ρ_{g}-terms and their transformation after a step of garbage collection. Note that in the first case the matching constraint is still present after the reduction, while in the second case all the environment is deleted. These reductions clarify hence why a hierarchical representation of environments is necessary in order to distinguish terms leading to successful computations from terms leading to unsuccessful ones.



1) $a[x = c, b \ll d]$ 2) $a[x = c[b \ll d]]$

Figure 7.3: ρ_{g}-terms with matching constraints

In Figure 7.4 we represent two ρ_{g}-terms with cycles. Notice here that the second graph contains an edge crossing a box to reach the free variable y. Edges can cross out a box to reach free variables or values defined in an upper list of constraints, like the edge from the constant h to the value a in the term $f(x, y)[x = g(z, z)[z = h(y)], y = a]$. On the other hand, no edge is allowed to enter a box for a graph representing a ρ_{g}-term. This restriction can be understood by noticing that a box in a ρ_{g}-term corresponds to the area of visibility of variables bound in the constraint represented by that box.



1) $f(x, y)[x = h(x), y = g(x)]$ 2) $f(x, g(y))[x = f(z)[z = h(z, y)]]$

Figure 7.4: Graphs with cycles

Finally in Figure 7.5 we represent two graphs with sharing. The first graph can be understood as the representation of the ρ_{g}-term $(2 * x) \twoheadrightarrow (y + y)[y = s(x)]$. Notice that in this term the list of constraints refers only to the right-hand side of the abstraction, and thus, in the corresponding graphical representation, the box frames only the right-hand subterm of the rule. On the other

hand, the second graph does not correspond to a well-formed $\rho_{\mathtt{g}}$-term, since no sharing is allowed between the left-hand side and the right-hand side of a rewrite rule, according to the well-formed conditions (see Example 36).

As already mentioned, $\rho_{\mathtt{g}}$-terms are grouped into equivalence classes. Intuitively, terms in the same class can be seen as terms that have different syntactical representations to describe the same graphical structure. Indeed, if we consider $\rho_{\mathtt{g}}$-terms whose lists of constraints are linear, *i.e.* a constraint appears at most once in the same list, as the terms of the linear $\rho_{\mathtt{g}}$-calculus considered in the next chapter, we can notice that two $\rho_{\mathtt{g}}$-terms belonging to the same equivalence class have the same graphical interpretation, modulo the permutation of the graphs appearing in the lower part of boxes.

As mentioned at the beginning, the graphical representation of $\rho_{\mathtt{g}}$-terms has been mainly defined to help the intuition of the reader. For this reason, in the following we try to keep images as simple as possible by omitting boxes for graphs that do not contain match equations or garbage.

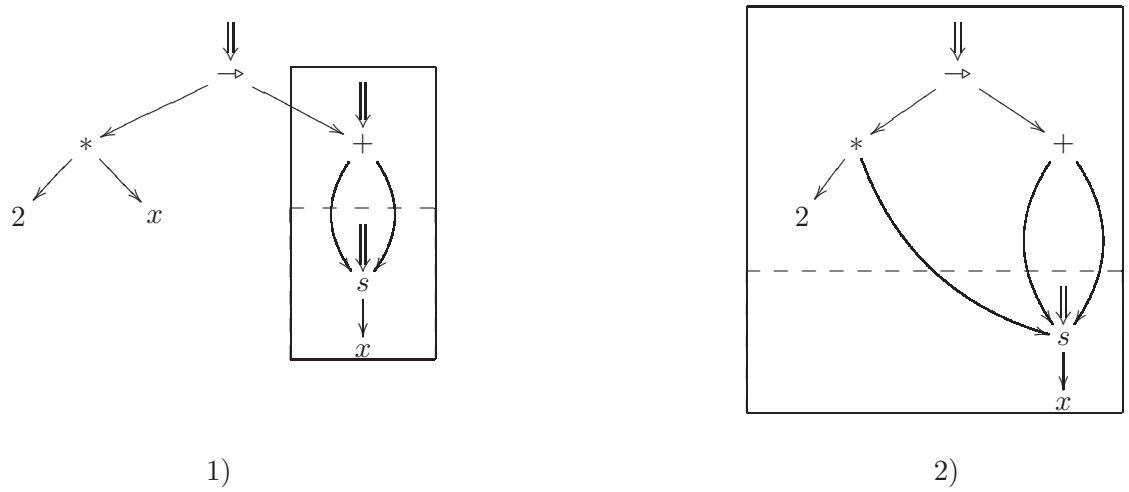


Figure 7.5: Rewrite rules with sharing

7.4 The semantics

The semantics of the $\rho_{\mathtt{g}}$-calculus can be seen, in a sense, a generalisation of those of the ρ-calculus and of the $\rho_{\mathtt{x}}$-calculus. In the classical ρ-calculus, when reducing the application of a delayed matching constraint to a term, the corresponding matching problem is solved and the resulting substitutions are applied at the meta-level of the calculus. In the $\rho_{\mathtt{x}}$-calculus, this reduction is decomposed into two phases, clearly separated and both treated explicitly, one computing substitutions and the other one describing the application of these substitutions. In the $\rho_{\mathtt{g}}$-calculus, the computation of substitutions solving a matching constraint is performed explicitly and, if the computation is successful, the result is a set of recursion equations added to the list of constraints of the term. The substitution can then be applied by copying the values associated to the recursion variables at the suitable positions in the term. This means that the substitution is not applied immediately to the term but kept in the environment for a possible delayed application. In this way, the two different kind of constraints of the calculus, match and recursion equations, are handled in a uniform way, since they are both part of the same environment. This is not

BASIC RULES:

$$
\begin{array}{lll}
(\rho) & (P \twoheadrightarrow G_2)\ G_3 & \to_\rho \ G_2[P \ll G_3] \\
 & (P \twoheadrightarrow G_2)[E]\ G_3 & \to_\rho \ G_2[P \ll G_3, E] \\
(\delta) & (G_1 \wr G_2)\ G_3 & \to_\delta \ G_1\ G_3 \wr G_2\ G_3 \\
 & (G_1 \wr G_2)[E]\ G_3 & \to_\delta \ (G_1\ G_3 \wr G_2\ G_3)[E]
\end{array}
$$

MATCHING RULES:

$$
\begin{array}{lll}
(\mathit{propagate}) & P \ll (G[E]) & \to_p \ P \ll G, E \quad \mathit{if}\ P \notin \mathcal{X} \\
(\mathit{decompose}) & f(G_1, \ldots, G_n) \ll f(G'_1, \ldots, G'_n) & \to_{dk} \ G_1 \ll G'_1, \ldots, G_n \ll G'_n \\
 & & \mathit{with}\ n \geq 0 \\
(\mathit{solved}) & x \ll G, E & \to_s \ x = G, E \quad \mathit{if}\ x \notin \mathcal{DV}(E)
\end{array}
$$

GRAPH RULES:

$$
\begin{array}{lll}
(\mathit{external\ sub}) & \mathsf{Ctx}\{y\}[y = G, E] & \to_{es} \ \mathsf{Ctx}\{G\}[y = G, E] \\
(\mathit{acyclic\ sub}) & G[P \lll \mathsf{Ctx}\{y\}, y = G_1, E] & \to_{ac} \ G[P \lll \mathsf{Ctx}\{G_1\}, y = G_1, E] \\
 & & \mathit{if}\ x > y,\ \forall x \in \mathcal{FV}(P) \\
 & & \mathit{where} \lll \in \{=, \ll\} \\
(\mathit{garbage}) & G[E, x = G'] & \to_{gc} \ G[E] \\
 & & \mathit{if}\ x \notin \mathcal{FV}(E) \cup \mathcal{FV}(G) \\
 & G[\epsilon] & \to_{gc} \ G \\
(\mathit{black\ hole}) & \mathsf{Ctx}\{x\}[x =_\circ x, E] & \to_{bh} \ \mathsf{Ctx}\{\bullet\}[x =_\circ x, E] \\
 & G[P \lll \mathsf{Ctx}\{y\}, y =_\circ y, E] & \to_{bh} \ G[P \lll \mathsf{Ctx}\{\bullet\}, y =_\circ y, E] \\
 & & \mathit{if}\ \ x > y,\ \forall x \in \mathcal{FV}(P)
\end{array}
$$

Figure 7.6: Small-step semantics of the ρ_{g}-calculus

the case in the ρ_{x}-calculus, where two different environments with the corresponding application operators are specified, one for matching constraints "_(_)" and the other for substitutions "_{_}". We also point out that in the ρ-calculus or in the ρ_{x}-calculus, a terminating reduction is considered successful if it ends with a term without constraints, while in the ρ_{g}-calculus the presence of constraints, namely recursion equations, is advisable for maintaining the sharing features in the term and does not mean that the reduction has failed.

At a glance, the evaluation rules of the ρ_{g}-calculus, presented in Figure 7.6, consist of three categories:

- BASIC RULES describing the application of abstractions and structures on ρ_{g}-terms.
- MATCHING RULES describing the solving of match equations.
- GRAPH RULES handling the replacements and the garbage collection.

Before detailing the behaviour of each rule, we remind the reader that reductions formally take place over equivalence classes of ρ_{g}-terms defined modulo the theory specified for the conjunction operator for constraints, that is associativity, commutativity, idempotence and neutral axiom for ϵ. The evaluation rules are thus applied to ρ_{g}-terms using syntactic matching modulo the theory of the list conjunction operator. This fact leads to some technical difficulties that will be considered more explicitly in the next chapter while proving the confluence of the calculus.

7.4.1 Basic rules

$$
\begin{array}{lll}
(\rho) & (P \twoheadrightarrow G_2)\ G_3 & \to_\rho\ G_2[P \ll G_3] \\
 & (P \twoheadrightarrow G_2)[E]\ G_3 & \to_\rho\ G_2[P \ll G_3, E] \\
(\delta) & (G_1 \wr G_2)\ G_3 & \to_\delta\ G_1\ G_3 \wr G_2\ G_3 \\
 & (G_1 \wr G_2)[E]\ G_3 & \to_\delta\ (G_1\ G_3 \wr G_2\ G_3)[E]
\end{array}
$$

The first two rules (ρ) and (δ) come from the ρ-calculus. Rule (δ) deals with the distributivity of the application on the structures built using the "$\_\wr\_$" operator while rule (ρ) triggers the application of a rewrite rule to a ρ_{g}-term by applying the appropriate constraint to the right-hand side of the rule. For each of these rules an additional one taking into account the possible presence of constraints is added. Without these additional rules, ρ_{g}-terms like $f(y) \twoheadrightarrow x\ f(y)[x = f(y) \twoheadrightarrow x\ f(y)]\ f(a)$ (that can encode a recursive application as in Example 40) could not be reduced. Alternatively, appropriate distributivity rules could be introduced but this approach is not considered in this thesis.

7.4.2 Matching rules

$$
\begin{array}{lll}
(\mathit{propagate}) & P \ll (G[E]) & \to_p\ P \ll G, E \quad \mathit{if}\ P \neq x \\
(\mathit{decompose}) & f(G_1, \ldots, G_n) \ll f(G'_1, \ldots, G'_n) & \to_{dk}\ G_1 \ll G'_1, \ldots, G_n \ll G'_n \\
 & & \mathit{with}\ n \geq 0 \\
(\mathit{solved}) & x \ll G, E & \to_s\ x = G, E \quad \mathit{if}\ x \notin \mathcal{DV}(E)
\end{array}
$$

The Matching rules and in particular the rule (*decompose*) are strongly related to the theory modulo which we want to compute the solutions of the matching. Here we have chosen to present the ρ_{g}-calculus with an empty theory which matching problem is known to be decidable and unitary.

Due to the restrictions imposed on the left-hand sides of rewrite rules, we only need to decompose algebraic terms. However, taking inspiration from the work on rational term unification by Courcelle [Cou83], an appropriate set of rules can be defined in order to deal with cyclic left-hand sides, as sketched in the conclusions of the thesis.

The goal of this set of rules is to produce a constraint of the form $x_1 = G_1, \ldots, x_n = G_n$ starting from a matching equation. This is possible when the left and right-hand sides of the matching equation are algebraic but some replacements might be needed (as defined by the Graph rules) as soon as the terms contain sharing or cycles.

A matching equation containing constraints is reduced (by the (*propagate*) rule) to the same matching equation without the constraints, which are propagated to a higher level. Since left-hand sides of matching equations are acyclic, there is no need for an evaluation rule propagating the constraints from the left-hand side of the matching equation; the possible constraints on this side of the matching can be "pushed" in the term using the substitution and garbage collection rules in the Graph rules. Algebraic terms are decomposed using the (*decompose*) rule until the two sides of the match begin with two different constant symbols or an elementary match equation $x \ll G$ is reached. The match constraint $x \ll G$ is then considered solved and transformed in a recursion equation $x = G$ by the (*solved*) rule if there is no other constraint of the form $x = G'$ or $\mathsf{Ctx}\{x\} \ll G'$, with $G \neq G'$, in the list of constraints. For example, the constraint $x \ll a, x \ll b$, which could be generated from $f(x, x) \ll f(a, b)$, cannot be reduced showing that the original (non-linear) matching problem has no solution.

7.4.3 Graph rules

(*external sub*)	$\mathsf{Ctx}\{y\}[y = G, E]$	$\to_{es}$	$\mathsf{Ctx}\{G\}[y = G, E]$
(*acyclic sub*)	$G[P \lll \mathsf{Ctx}\{y\}, y = G_1, E]$	$\to_{ac}$	$G[P \lll \mathsf{Ctx}\{G_1\}, y = G_1, E]$ *if* $x > y,\ \forall x \in \mathcal{FV}(P)$ *where* $\lll \in \{=, \ll\}$
(*garbage*)	$G[E, x = G']$	$\to_{gc}$	$G[E]$ *if* $x \notin \mathcal{FV}(E) \cup \mathcal{FV}(G)$
	$G[\epsilon]$	$\to_{gc}$	G
(*black hole*)	$\mathsf{Ctx}\{x\}[x =_\circ x, E]$	$\to_{bh}$	$\mathsf{Ctx}\{\bullet\}[x =_\circ x, E]$
	$G[P \lll \mathsf{Ctx}\{y\}, y =_\circ y, E]$	$\to_{bh}$	$G[P \lll \mathsf{Ctx}\{\bullet\}, y =_\circ y, E]$ *if* $x > y,\ \forall x \in \mathcal{FV}(P)$

The Graph rules are inspired to those of the cyclic λ-calculus of Ariola and Klop. The first two rules (*external sub*) and (*acyclic sub*) make a copy of a ρ_{g}-term associated to a recursion variable into a term that is inside the scope of the corresponding constraint. This is important when a redex should be made explicit (*e.g.* in $x\ a[x = a \twoheadrightarrow b]$) or when a matching equation should be solved (*e.g.* in $a[a \ll x, x = a]$).

Notice that we allow substitution only upwards *w.r.t.* the order on variables introduced in Definition 82. This restriction on the application of the (*acyclic sub*) rule is needed in order to ensure confluence. Without this condition this property is surely lost since the counterexample for the cyclic λ-calculus in Example 33 can be expressed also in the ρ_{g}-calculus: it is sufficient to translate the initial λ-graph in the corresponding ρ_{g}-term $z_1[z_1 = x \twoheadrightarrow z_2\ s(x), z_2 = y \twoheadrightarrow z_1\ s(y)]$ to get exactly the same reductions leading to two different normal forms $z_1[z_1 = x \twoheadrightarrow z_1\ s(s(x))]$ and $z_1[z_1 = x \twoheadrightarrow z_2\ s(x), z_2 = y \twoheadrightarrow z_2\ s(s(y))]$. We will see in Chapter 8 that this restriction together with some restrictions on the form of the left-hand side of the rules will be the key ingredients to ensure the confluence of the ρ_{g}-calculus.

The (*garbage*) rules gets rid of recursion equations that represent non connected parts of the ρ_{g}-term. Matching constraints are not eliminated, thus keeping the trace of possible matching failures during an unsuccessful reduction.

Finally, the (*black hole*) rules are a special case of the substitution rules and replace undefined ρ_{g}-terms by the constant $\bullet$.

We will use the notation $\mapsto_{\rho g}$ ($\mapsto_{\mathcal{M}}$) and $\mapsto\!\!\!\!\twoheadrightarrow_{\rho g}$ ($\mapsto\!\!\!\!\twoheadrightarrow_{\mathcal{M}}$) for the relations induced by the set of rules of Figure 7.6 and by the subset of Matching rules, respectively. For any two rules r and s belonging to this set, we will write $\mapsto\!\!\!\!\twoheadrightarrow_{r,s}$ to express the two steps $\mapsto_r \mapsto_s$.

7.5 Examples of reductions

To better understand the behavior of the different operators with respect to rewriting, we will present and comment in this section some examples of reductions in the ρ_{g}-calculus.

We start with an example about the computation of two non-linear matching problems. The reader can notice how in the case of unsuccessful matchings the (*solved*) rule cannot be applied since its proviso is not satisfied. As a result, the match equations are not transformed into recursion equations and the computation is stopped.

Example 37 (Non-linearity) *The matching involving non-linear patterns can lead to a normal form that is either a constraint consisting only of recursion equations (representing a successful*

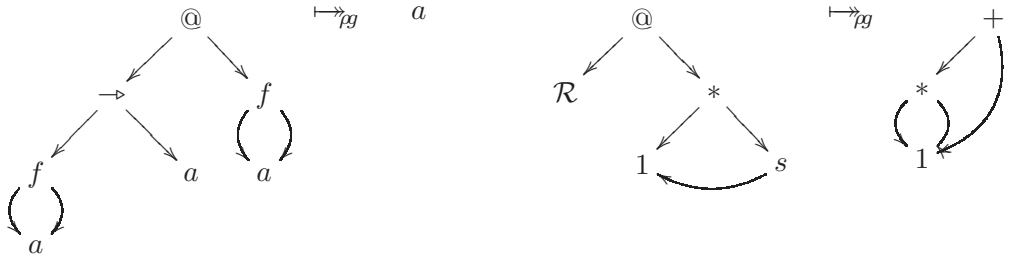


Figure 7.7: Examples of reductions

matching) or a constraint that does contain some matching equations (representing a matching failure).

$$\begin{array}{ll} & f(y,y) \ll f(a,a) \\ \mapsto_{dk} & y \ll a \quad (\textit{by idempotency}) \\ \mapsto_{s} & y = a \end{array} \qquad \begin{array}{ll} & f(y,y) \ll f(a,b) \\ \mapsto_{dk} & y \ll a, y \ll b \end{array}$$

The next example describes the reduction of a term which includes the application of a rewrite rule having some sharing in its left-hand side. As in the previous example, the match equation have a non-linear left-hand side and therefore the idempotency property of the "_, _" operator is needed during the computation to eliminate identical constraints appearing twice in the list. Moreover, the neutral element axiom (NE) is used to make the application of the (*garbage*) rule possible.

Example 38 (A simple reduction with sharing) *A graphical representation is given in Figure 7.7(a).*

$$\begin{array}{lll} & (f(x,x)[x=a] \twoheadrightarrow a)\ (f(y,y)[y=a]) & \\ \mapsto_{p} & a[f(x,x)[x=a] \ll f(y,y)[y=a]] & \\ \mapsto_{es} & a[f(a,a)[x=a] \ll f(y,y)[y=a]] & \\ = & a[f(a,a)[x=a,\epsilon] \ll f(y,y)[y=a]] & (\textit{by NE}) \\ \mapsto_{gc} & a[f(a,a)[\epsilon] \ll f(y,y)[y=a]] & \\ \mapsto_{gc} & a[f(a,a) \ll f(y,y)[y=a]] & \\ \mapsto_{p} & a[f(a,a) \ll f(y,y), y=a] & \\ \mapsto_{dk} & a[a \ll y, a \ll y, y=a] & \\ = & a[a \ll y, y=a] & (\textit{by idempotency}) \\ \mapsto_{ac} & a[a \ll a, y=a] & \\ \mapsto_{dk} & a[y=a] & \\ = & a[y=a,\epsilon] & (\textit{by NE}) \\ \mapsto_{gc} & a[\epsilon] & \\ \mapsto_{gc} & a & \end{array}$$

Note that, as a consequence of the definition of the Matching rules, the left-hand side of the match equation $f(x,x)[x=a]$ needs to be rewritten in the more compact form $f(a,a)$ (without lists of constraints) before starting the decomposition of the term. From the point of view of the structure of the graph, this means that the sharing of the left-hand side is lost during the reduction. This could be avoided by modifying the definition of the Matching rules, for example along the lines of the ideas described in Section 7.7. Anyway, the lost of sharing in the

right-hand side of a match equation does not affect the sharing of the right-hand side, which can be maintained in the result of the rewriting, as shown in the next example.

Example 39 (Multiplication) *If we use an infix notation for the symbol "$\_ * \_$" the following ρ_g-term corresponds to the application of the rewrite rule $\mathcal{R} = x * s(y) \twoheadrightarrow (x * y + x)$, modelling multiplication, to the term $1 * s(1)$ where the constant 1 is shared. The result is shown graphically in Figure 7.7(b).*

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

The reduction could continue by applying the (*acyclic sub*) rule followed by the (*garbage*) rule until the normal form $1*1+1$ of the term is reached. However, in this case the computation is not very efficient since the sharing in the final result is completely lost. We will see in Section 7.6 how we can define a reduction strategy which consider the term $(z * z + z)[z = 1]$ as completely reduced.

In the next example, we consider two definitions of the fixed point combinator Y of the λ-calculus (see Examples 11 and 12), the first one based on a term rewrite rule and the second, more efficient, one based on a term graph rule. We show how both these definitions can be expressed in the ρ_{g}-calculus and we compare the reductions they generate.

Example 40 (Fixed point combinator) *Consider the term rewrite rule $R_Y = Y\ x \to x\ (Y\ x)$ which expresses the behaviour of the fixed point combinator Y of the λ-calculus. Given the a term t, we have the infinite rewrite sequence*

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

which, in a sense which can be formalized as in [KKSdV91, Cor93], converges to the infinite term $t\ (t\ (t\ (\ldots)))$.

We can represent the Y-combinator in the ρ_{g}-calculus as the following term:

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

If we denote $R = z \twoheadrightarrow z\ (x\ z)$, for any ρ_{g}term G we have the following reduction:

$$
\begin{array}{ll}
 & Y\ G \\
\mapsto_{es} & (z \twoheadrightarrow z\ (x\ z))[x = R]\ G \\
\mapsto_{p} & z\ (x\ z)[z \ll G, x = R] \\
\mapsto_{s} & z\ (x\ z)[z = G, x = R] \\
\mapsto\!\!\!\!\rightarrow_{es} & G\ (x\ G)[z = G, x = R] \\
\mapsto_{gc} & G\ (x\ G)[x = R] \\
\mapsto\!\!\!\!\rightarrow_{pg} & G(G \ldots (x\ G))[x = R] \\
\mapsto\!\!\!\!\rightarrow_{pg} & \ldots
\end{array}
$$

Continuing the reduction, this will "converge" to the term of Figure 7.8(a).

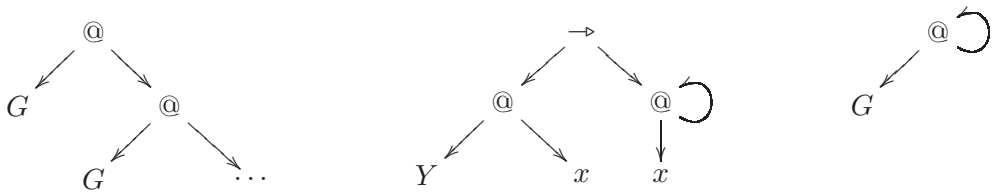


Figure 7.8: Example of reductions

We can have a more efficient implementation of the same term reduction using a method introduced by Turner [Tur79] that models the rule R_Y by means of the cyclic term depicted in Figure 7.8(b). This gives in the ρ_{g}-calculus the ρ_{g}-term

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

The reduction in this case is the following:

$$\begin{array}{ll} & Y_T\ G \\ \mapsto_{p} & z[z = x\ z][x \ll G] \\ \mapsto_{s} & z[z = x\ z][x = G] \\ \mapsto_{es} & z[z = G\ z][x = G] \\ \mapsto_{gc} & z[z = G\ z] \end{array}$$

The resulting ρ_{g}-term is depicted in Figure 7.8(c). If we "unravel", in the intuitive sense, this cyclic ρ_{g}-term we obtain the infinite term shown in Figure 7.8(a).

This second reduction captures the fact that a finite sequence of rewritings on cyclic ρ_{g}-terms can correspond to an infinite term reduction sequence. Talking again about cyclic term reductions, we will analyse next, in the setting of the ρ_{g}-calculus, the collapsing problem already addressed in Section 6.2 for other term graph rewrite systems.

Example 41 (Cyclic redexes) *As discussed in Section 6.2, the application of collapsing rules of the kind $f(y) \twoheadrightarrow y$ to a cyclic term $x[x = f(x)]$ can lead to two different results according to the interpretation that is given to rewriting. Either the term $x[x = f(x)]$ rewrites to itself, or it rewrites to the undefined term $x[x = \bullet]$.*

In the ρ_{g}-calculus we can actually model both behaviors, thanks to the definition of the rewrite rules at the object level and to the explicit treatment of their application.

1. *To obtain the first result, we apply the rule $f(y) \twoheadrightarrow y$ at the top level of the term. This corresponds intuitively to the application of the rule at the head position of the infinite term $f(f(\ldots))$. Since the rewriting erases the top symbol f, the result is still a term with an infinite number of symbols f.*

$$\begin{array}{ll} & (f(y) \twoheadrightarrow y)\ x[x = f(x)] \\ \mapsto_{\rho} & y[f(y) \ll x[x = f(x)]] \\ \mapsto_{p} & y[f(y) \ll x, x = f(x)] \\ \mapsto_{ac} & y[f(y) \ll f(x), x = f(x)] \\ \mapsto_{dk} & y[y \ll x, x = f(x)] \\ \mapsto_{s} & y[y = x, x = f(x)] \\ \mapsto_{es,gc} & x[x = f(x)] \end{array}$$

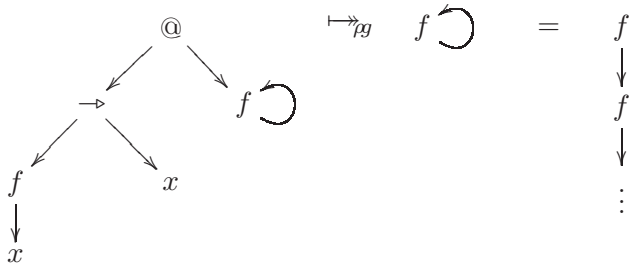


Figure 7.9: Reduction to a cyclic term

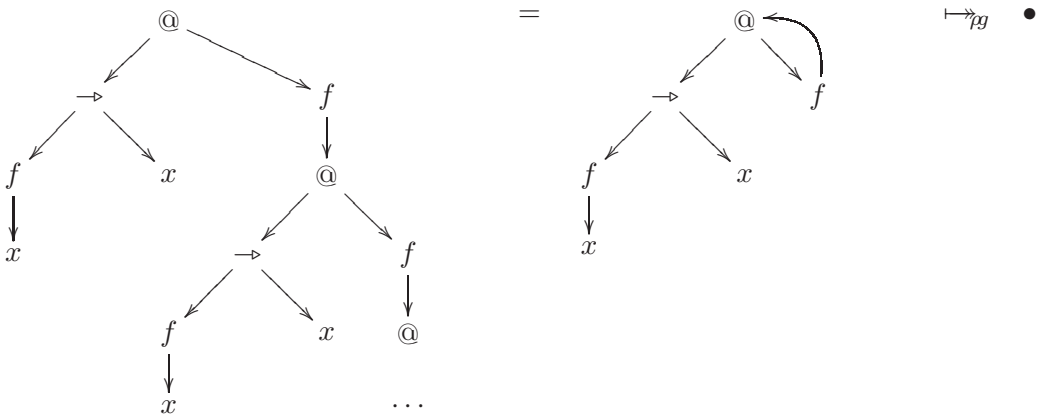


Figure 7.10: Reduction to black hole

2. *To obtain the second result, we apply the rule inside the cycle of the term* $x[x = f(x)]$. *This means that we apply the rule* at each position *of the symbol* f, *simultaneously. The rewriting eliminates thus in one step all the symbols* f *and leads to the undefined term "black hole".*

$$
\begin{array}{ll}
 & x[x = (f(y) \twoheadrightarrow y)\ f(x)] \\
\mapsto_{\rho} & x[x = y[f(y) \ll f(x)]] \\
\mapsto_{dk} & x[x = y[y \ll x]] \\
\mapsto_{s} & x[x = y[y = x]] \\
\mapsto_{es,gc} & x[x = x] \\
\mapsto_{bh} & x[x = \bullet]
\end{array}
$$

To give a graphical intuition we depicted in Figure 7.9 and Figure 7.10 respectively, the two ρ_{g}*-terms, their corresponding infinite* ρ*-terms as well as the result of the two reductions.*

7.6 A strategy to maintain sharing

With a view to a future efficient implementation of the calculus, it would be interesting to study suitable strategies that aim at keeping the sharing information as long as possible in ρ_{g}-terms. Considering the evaluation rules as defined in Section 7.4, we propose the definition of a sharing strategy at the meta level of the calculus.

Such strategy can be obtained, for example, by delaying the reductions the applications of the substitution rules, (*external sub*) and (*acyclic sub*), which can break the sharing by duplicating

terms. Basically, the idea consists of applying the substitution rules only if needed for generating new redexes for basic or matching rules. In addition, substitutions rules can be used to "remove" trivial recursion equations of the kind $x = y$. We allow for example the application of the (*external sub*) rule to the terms $x\ a[x = f(x) \twoheadrightarrow x]$ or $x\ a[x = a \wr (a \twoheadrightarrow b)]$, since this is useful for creating, respectively, a new (ρ) redex and a new (δ) redex. We do not allow its application to the terms $f(x,x)[x = g(x)]$ or $x[x = f(x)]$ that are considered in normal form. Similarly, we do not want to apply the (*acyclic sub*) rule to terms like $y[y = f(x,x), x = g(x)]$, but we need it to continue the reduction in terms like $y[y = x\ a, x = f(x) \twoheadrightarrow x]$ and $a[a \ll y, y = a]$.

We can say thus that we allow the application of the (*acyclic sub*) and (*external sub*) rules only to create new redexes for the Basic rules and to possibly unfreeze match equations where otherwise the computation of the matching is stuck.

There is one more situation in which we want to apply the substitution rules, that is when we have trivial recursion equations where both sides are single variables, like in $x * y + x[x = z, y = z, z = 1]$. In this case, we may want to simplify the term to $(z * z + z)[z = 1]$ in which useless names have been eliminated by garbage collection.

To summarise, we formally define next the reduction strategy we can adopt in the ρ_{g}-calculus to maintain the sharing information during the reduction as long as possible.

Definition 88 *The strategy* SharingStrat *consists of performing a step of reduction using the evaluation rules (external sub) or (acyclic sub) in a ρ_{g}-term G only if:*

- *it instantiates a variable in active position (see Definition 53) by an abstraction or a structure, or*
- *it instantiates a variable in a stuck match equation,*
- *it instantiates a variable by a variable.*

A further question that naturally arises is whether the properties of the calculus, as for example the confluence shown in the next chapter for the general calculus, hold under this evaluation strategy. On the other hand, we are interested in studying strategies that ensure the validity of properties, *e.g.* termination, that the calculus, considered with no restrictions on the application of the rules, does not enjoy. An extended analysis of different reduction strategies and their properties will be the object of future research.

7.7 Potential alternatives for the Matching rules

We can think to modify the set of evaluation rules in order to make the order of rule applications more flexible, in particular for what concerns the substitution rules during the computation of matching problems. Indeed, according to the current rules defined in Figure 7.6, for solving a matching $f(x,x)[x = a] \ll f(y,y)$ we need first to transform the left-hand side into the tree $f(a,a)$ by applying twice the (*external sub*) rule (followed by garbage collection) and only afterwards we can proceed in the computation by applying the (*decompose*) rule. It is worth exploring some ways to make the computation of a matching problem more efficient, for example, by delaying in the reductions the applications of the substitution rules, (*external sub*) and (*acyclic sub*), which can break the sharing by duplicating terms.

We have analysed two possibilities: the first one consists in adding to the set of Matching rules a new (*propagate*) rule, and the second possibility consists in modifying the definition of the (*decompose*) rule.

We can notice that the flattening of the constraints in the left-hand side of a match equation would allow one the decomposition of the equation without the need of completely unsharing the left-hand side. This observation led us to propose an operational semantics with an additional (*propagate*) rule operating on the left-hand side of a match.

$$G[E] \ll G' \ \rightarrow_{p'} \ G \ll G', E$$

With this new rule, given the match equation $f(x,x)[x = a] \ll f(y,y)$ we would obtain the reduction

$$f(x,x)[x = a] \ll f(y,y) \ \mapsto_{p'} \ f(x,x) \ll f(y,y), x = a \ \mapsto_{dk} \ x \ll y, x = a$$

We immediatly notice here that the variable x in the last list of constraints becomes bound by both the match operator and the recursion equation. To solve this problem, a solution would be to associate explicitly to any abstraction and match operator a list of variables containing the variables bound by the operator, following an approach similar to that used for the Pure Pattern Typed Systems in [BCKL03]. At the light of this consideration, it is clear that the choice of adding this rule to the ρ_{g}-calculus semantics needs further investigations, since it would require a considerable change in the definition of the calculus.

The second possibility we have explored consists in changing slightly the definition of the (*decompose*) rule in order to allow its application even when the term in the left-hand side of a matching constraint contains constraints. This approach is closer to the $\lambda\phi_1$ calculus in [AK96b] where some distribution rules are added to the set of evaluation rules of the basic $\lambda\phi$ (defined in Section 6.4). In this version of the ρ_{g}-calculus, the (*decompose*) rule would be replaced by the two following rules:

$$\begin{array}{lll} f(G_1, \ldots, G_n)[E] \ll f(G'_1, \ldots, G'_n), E' & \rightarrow_{dk} & G_1[E] \ll G'_1, \ldots, G_n[E] \ll G'_n, E' \\ & & \textit{with } n \geq 0 \\ f[E] \ll f & \rightarrow_{e} & E \end{array}$$

This answer is a good solution in the aim of having derivations where the application of the substitution rules is not required as soon as a matching problem appears in the reduced term. Unfortunately, using this semantics, evaluation become more space consuming than in the original calculus, since each step of decomposition implies the duplication of the entire list of constraints appearing in the left-hand side of the match.

Other solutions for making the computation of the matching more efficient, as well as a version of the MATCHING rules able to deal with cyclic left-hand sides in match equations, are currently under study.

Conclusion

We have introduced the ρ_{g}-calculus, a higher-order calculus that naturally generalises the ρ-calculus by using unification constraints in addition to the standard ρ-calculus matching constraints. This leads to an expressive higher-order calculus which can naturally handle and compute over graph-like structures.

We have defined the main components of the ρ_{g}-calculus, namely the syntax and the evaluation rules, and we have presented several examples of terms and reductions.

In the ρ_{g}-calculus, terms are associated to a list of constraints which is composed of recursion equations, used to express sharing and cycles, and matching constraints, arising from the fact that

matching is made explicit and performed at the object-level. This defines a higher-order calculus where terms can contain variables bound by a recursion equation, a matching constraint, or an abstraction (similar to the one of the ρ-calculus). Some work had been dedicated to formalise the notions of free and bound variable in a ρ_{g}-term, adapting their definition according to the three binders of the calculus.

The evaluation rules describe the application of ρ_{g}-terms, the behaviour of matching constraints as well as the application of substitutions and garbage collection. Evaluation strategies can guide the application of these rules and we have proposed in particular a strategy that aims at maintaining the sharing information as long as possible during the reduction.

Finally, the graph-like structure of ρ_{g}-terms naturally led to the definition of a graphical representation which takes into consideration the lists of constraints, possibly including matching constraints, associated to a term. These results have been partially presented at Termgraph'04 [BBCK05].

Chapter 8

Confluence of the graph rewriting calculus

Since the ρ_{g}-calculus generalises the ρ-calculus and the λ-calculus, it is not surprising that it does not enjoy termination. In addition to the infinite reductions generated from ρ_{g}-terms corresponding to λ-terms like Ω, in the ρ_{g}-calculus we can have non-terminating reductions induced by the presence of cycles in the terms, like in

$$x[x = f(x)] \mapsto_{es} f(x)[x = f(x)] \mapsto_{es} \dots$$

We will not prove specific results for non-terminating reductions, but we will prove an important property for terminating reductions, namely confluence over equivalence classes of ρ_{g}-terms in a linear ρ_{g}-calculus. Any two terminating reductions starting from the same ρ_{g}-term reduce to two terms that, even if not syntactically equal, are very similar, in the sense that they belong to the same equivalence class. In other words, a term has not a unique normal form, but all its normal forms are equivalent modulo the theory specified for the constraint conjunction operator (associativity, commutativity, idempotency and neutral element).

We prove in this chapter that the ρ_{g}-calculus, or more precisely the rewrite relation defined over equivalence classes of ρ_{g}-terms, enjoys the confluence property, under some linearity restrictions on patterns. These restrictions will allow us to avoid the classical counter-example to confluence [Klo80] and to drop the idempotency axiom for the constraint conjunction operator, a fact which will be a key technical ingredient in the proof.

We develop an original proof method that generalises the proof of confluence of the cyclic λ-calculus [AK96b] to the setting of rewriting modulo an equational theory [Ohl98] and moreover it adapts the proof to deal with terms containing patterns and matching equations. The proof uses the concept of "developments" and the property of "finiteness of developments" as defined in the theory of classical λ-calculus (see Section 2.1.1).

8.1 General presentation

The confluence for higher-order systems dealing with non-linear matching is a difficult issue since we usually obtain non-joinable critical pairs as shown by Klop in the setting of the λ-calculus [Klo80]. Klop's counterexample can be encoded in the ρ-calculus (see Example 23) showing that the non-linear ρ-calculus is not confluent, if no evaluation strategy is imposed on the reductions. The counterexample is still valid when generalising the ρ-calculus to the ρ_{g}-calculus, therefore in the following we restrict to a linear ρ_{g}-calculus.

Definition 89 (Linear ρ_{g}-calculus) *The class of (algebraic)* linear patterns *is defined as follows:*

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

where we assume that $FV(\mathcal{L}_i) \cap FV(\mathcal{L}_j) = \emptyset$ for $i \neq j$. A constraint $[L_1 \lll G_1, \ldots, L_n \lll G_n]$, where $\lll \in \{=, \ll\}$, is linear *if all patterns $L_1, \ldots, L_n$ are linear and $FV(L_i) \cap FV(L_j) = \emptyset$ for $i \neq j$. The* linear ρ_{g}-calculus *is the ρ_{g}-calculus where all the patterns in the left-hand side of abstractions and all constraints are linear.*

ρ_{g}-terms are grouped into equivalence classes defined according to the theory specified for the constraint conjunction operator and rewriting is performed over these classes. In the general ρ_{g}-calculus, the operator "_, _" is supposed to be associative, commutative and idempotent, with ϵ as neutral element. However, in the linear ρ_{g}-calculus, idempotency is not needed since constraints of the form $x \ll G, x \ll G$ are not allowed (and cannot arise from reductions) and constraints of the form $f \ll f, f \ll f$ can be solved by applying twice the *decompose* rule. Therefore, in the linear ρ_{g}-calculus, rewriting can be thought of as acting over equivalence classes of ρ_{g}-terms with respect to the congruence relation, denoted by $\sim_{AC_\epsilon}$ or simply AC_ϵ, generated by the associativity, commutativity and neutral element axioms for the "_, _" operator.

Following the notation introduced in Section 1.4, the relation induced over AC_ϵ-equivalence classes is written $\mapsto_{\rho_g/AC_\epsilon}$. Concretely, in most of the proofs we will use the notion of rewriting modulo AC_ϵ *à la* Peterson and Stickel [PS81], denoted $\mapsto_{\rho_g, AC_\epsilon}$ (see Definition 29). On the one hand, this notion of rewriting is more convenient, from a computational point of view, than AC_ϵ-class rewriting. On the other hand, under suitable assumptions satisfied by our calculus, *i.e.* linearity and compatibility as defined in Lemma 14, the confluence of the (ρ_g, AC_ϵ) relation implies the confluence of the ρ_g/AC_ϵ relation, as detailed in Section 8.4.

According to the definition of $\mapsto_{\rho_g, AC_\epsilon}$, matching modulo AC_ϵ is performed at each step of rewriting. We mention that matching modulo AC_ϵ may lead to infinitely many solutions, but the complete set of solution is finitary and has as canonical representative the solution in which terms are normalised *w.r.t.* the neutral element [Kir90].

The confluence proof is quite elaborated and we decompose it in a number of lemmata to achieve the final result. Its complexity is mainly due to the non-termination of the system and to the fact that equivalence modulo AC_ϵ on terms has to be considered throughout the proof.

We start by proving a fundamental compatibility lemma showing that the ρ_{g}-calculus rewrite relation is particularly well-behaved *w.r.t.* the congruence relation AC_ϵ. Then, we proceed by proving several lemmata that lead to the confluence of the relation $(\rho_{\mathsf{g}}, AC_\epsilon)$ and finally we conclude on the confluence of $\rho_{\mathsf{g}}/AC_\epsilon$.

The compatibility lemma ensures that the ρ_{g}-calculus evaluation rules interact nicely with the congruence relation AC_ϵ: if there exists a rewrite step from a ρ_{g}-term G, then the "same" step can be performed starting from any term AC_ϵ-equivalent to G.

Lemma 14 (Compatibility of ρ_g) *Compatibility with AC_ϵ holds for any rule r of the ρ_{g}-calculus.*

$$\leftarrow\!\!\!\!\!\!\rightarrow_{r, AC_\epsilon} \cdot \sim_{AC_\epsilon} \subseteq \sim_{AC_\epsilon} \cdot \leftarrow\!\!\!\!\!\!\rightarrow_{r, AC_\epsilon}$$

Proof : By case analysis on the rules of the ρ_{g}-calculus. Consider, for instance, the diagram for the (*acyclic sub*) rule with a commutation step.

$$\begin{array}{ccc} G[G_0 \lll \mathsf{Ctx}\{y\}, y = G_1, F] & \sim^1_{AC_\epsilon} & G[y = G_1, G_0 \lll \mathsf{Ctx}\{y\}, F] \\ \downarrow_{ac, AC_\epsilon} & & \vdots_{ac, AC_\epsilon} \\ G[G_0 \lll \mathsf{Ctx}\{G_1\}, y = G_1, F] & \sim_{AC_\epsilon} & G[y = G_1, G_0 \lll \mathsf{Ctx}\{G_1\}, F] \end{array}$$

A different order of the constraints in the list does not prevent the application of the (*acyclic sub*) rule, thanks to the fact that matching is performed modulo AC_ϵ. Moreover, the extension variable E in the definition of the (*acyclic sub*) rule ensures the applicability of the rule to terms having an arbitrary number of constraints in the list. In particular, the extension variable E can be instantiated by ϵ if the term to reduce is simply $G[G_0 \lll \mathsf{Ctx}\{y\}, y = G_1]$. In this case the application of the rule is possible since there exists a match between the term $[G_0 \lll \mathsf{Ctx}\{y\}, y = G_1, \epsilon]$, equivalent to the given term using the neutral element axiom, and the left-hand side of the (*acyclic sub*) rule.

The diagram can thus be easily closed. The same reasoning can be applied for the other evaluation rules. The extension to several steps of $\sim_{AC_\epsilon}$ trivially holds. □

We point out that since compatibility holds for any evaluation rule of the ρ_{g}-calculus, then it also holds for any subset of rules and also for the entire ρ_{g}-calculus semantics. This property will be one of the key hypothesis in many lemmata of the proof and in particular it will play a basic role when proving that the confluence of the ρ_g, AC_ϵ relation can be generalized to the confluence of the ρ_g/AC_ϵ relation.

We recall that the compatibility property for a rewrite relation modulo a set of axioms is stronger than the coherence or local coherence properties defined in Section 1.4.3.

Remark 1 (Compatibility vs local coherence) *As mentioned at the beginning of the chapter, the linearity assumption allows us to avoid the use of idempotency for the conjunction operator for constraints. It is worth noticing that the compatibility property for the full ρ_{g}-calculus does not hold if we consider the idempotency in the congruence relation. We provide next a counterexample, where G and H are two ρ_{g}-terms and H' is obtained from H after one step of reduction.*

$$\begin{array}{ccc} G[y \ll H, y \ll H] & \sim_{ACI_\epsilon} & G[y \ll H] \\ \downarrow_{\rho_g, ACI_\epsilon} & & \downarrow_{\rho_g, ACI_\epsilon} \\ G[y \ll H', y \ll H] & \not\sim_{ACI_\epsilon} & G[y \ll H'] \end{array}$$

To recover the equivalence between the two rewritten terms we need a further reduction step for the term $G[y \ll H', y \ll H]$, as shown in the next diagram. This corresponds to the property called local coherence (see Definition 31).

$$\begin{array}{ccc} G[y \ll H, y \ll H] & \sim_{ACI_\epsilon} & G[y \ll H] \\ \downarrow_{\rho_g, ACI_\epsilon} & & \\ G[y \ll H', y \ll H] & & \vdots_{\rho_g, ACI_\epsilon} \\ \vdots_{\rho_g, ACI_\epsilon} & & \\ G[y \ll H', y \ll H'] & \sim_{ACI_\epsilon} & G[y \ll H'] \end{array}$$

However, this weaker property would not be sufficient to prove the confluence of the ρ_g-relation on equivalence classes of ρ_{g}-terms, since the termination hypothesis for the relation would be necessary, as proved in [JK84] for a general rewrite system.

For the proof of confluence of the ρ_g, AC_ϵ relation we use a technique inspired from the one adopted in [AK96b] for the confluence of the cyclic λ-calculus. The larger number of evaluation rules of the $\rho_{\mathtt{g}}$-calculus and the explicit treatment of the congruence relation on terms make the proof for the $\rho_{\mathtt{g}}$-calculus much more elaborated. The main idea is to split the rules into two subsets, to show separately their confluence and then to prove the confluence of the union using a commutation lemma for the two sets of rules. In the $\rho_{\mathtt{g}}$-calculus rules are split into the following two subsets:

- the Σ-*rules*, including the two substitution rules (*external sub*) and (*acyclic sub*), plus the (δ) rule;
- the τ-*rules*, including all the remaining rules of the $\rho_{\mathtt{g}}$-calculus.

The Σ-rules include the substitution rules which represent the non-terminating rules of the $\rho_{\mathtt{g}}$-calculus. Also the (δ) rule is included in the Σ-rules, although it could be added to the τ-rules keeping this set of rules terminating. This choice motivated by the fact that, because of its non-linearity, adding the (δ) rule to the τ-rules would have caused relevant problems in the proof of the final commutation lemma.

The remaining part of the proof is structured in three parts that are detailed in the next sections.

1) Section 8.2: *confluence modulo AC_ϵ of the relation induced by the τ-rules.*

   This is done by using the fact that a relation strongly normalizing and locally confluent modulo AC_ϵ is confluent modulo AC_ϵ, if the compatibility property holds (see Proposition 1). To prove the strong normalization for the τ-rules classical rewriting techniques, described in Section 1.3.2, are applied. The local confluence modulo AC_ϵ is rather easy to prove by analysis of the critical pairs.

2) Section 8.3: *confluence modulo AC_ϵ of the relation induced by the Σ-rules.*

   This is the more complex part of the proof. The idea is to follow the complete development method of the λ-calculus, as defined in Section 2.1.1, *i.e.* to define a terminating version of the Σ relation (the development) and to use its properties for deducing the confluence of the original rewrite relation.

3) Section 8.4: *confluence modulo AC_ϵ of the relation induced by the union of the two sets.*

   General confluence holds since we can prove the commutation modulo AC_ϵ of the τ-rules with the Σ-rules (see Proposition 2).

   From the confluence of the relation ρ_g, AC_ϵ we can deduce the confluence of the relation ρ_g/AC_ϵ on AC_ϵ-equivalent classes of $\rho_{\mathtt{g}}$-terms. This is a consequence of the fact that the compatibility with AC_ϵ property holds for the $\rho_{\mathtt{g}}$-calculus.

In the rest of the chapter, we adopt the convention to simply write AC_ϵ or $\sim$ for $\sim_{AC_\epsilon}$ and $\mapsto_R$ for $\mapsto_{R,AC_\epsilon}$, where R may be any subset of rules of the $\rho_{\mathtt{g}}$-calculus.

The outline of the proof is depicted in Figure 8.1, where all the lemmata are mentioned, except the compatibility lemma which is left implicit, since it is used for almost all the intermediary results. The property appearing in the picture are formally defined for a general rewriting system in Section 1.4.3.

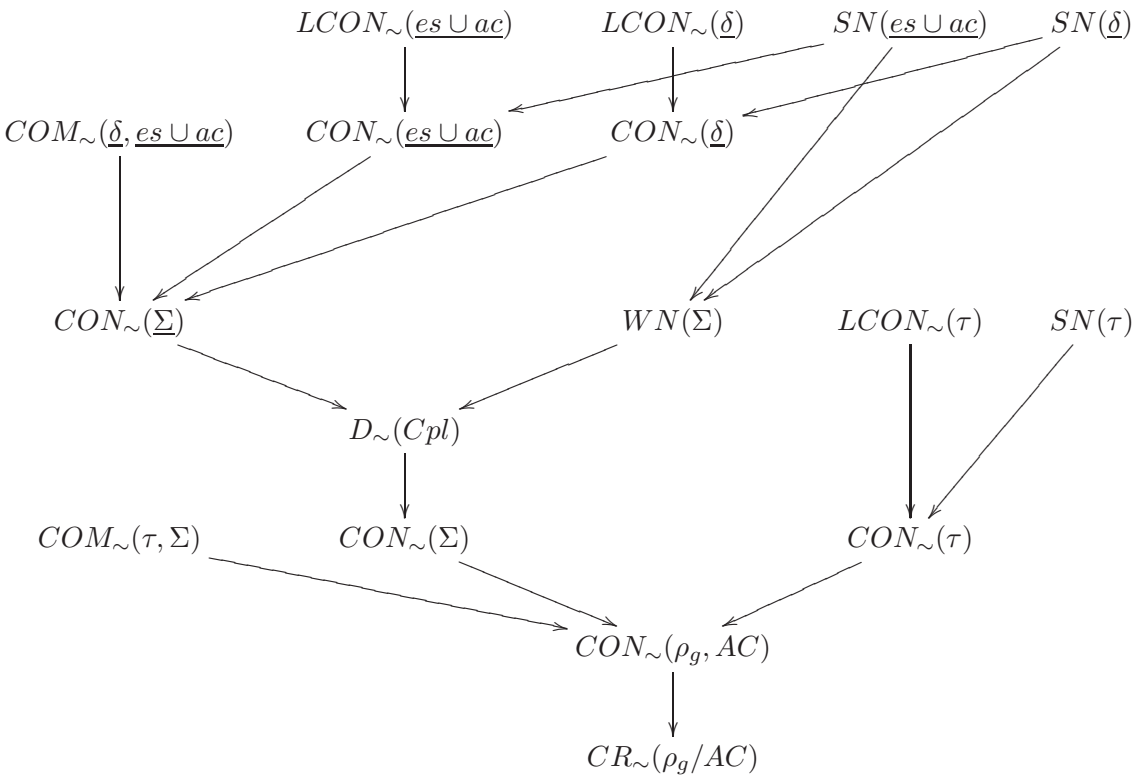


Figure 8.1: Confluence proof scheme

8.2 Confluence modulo AC_ϵ for the τ-rules

The confluence modulo AC_ϵ for the relation τ induced by the τ-rules can be proved using the properties of strong normalization and local confluence modulo AC_ϵ of this relation.

In the first part of the section we show the strong normalization of the relation τ by using a reduction order based on a polynomial interpretation on the ρ_{g}-graphs.

In the second part of the section, the local confluence modulo AC_ϵ is proved for the relation τ by case analysis of the critical pairs. On the basis of these results, we can then conclude the confluence of the relation τ.

We start by showing that τ is strongly normalising. In order to do that, a polynomial interpretation on the components of the ρ_{g}-calculus syntax is defined.

Definition 90 (Polynomial interpretation) *We consider the following polynomial interpretation of ρ_{g}-graphs (assuming the standard order on natural numbers):*

$$
\begin{array}{lcl}
Size(\epsilon) & = & 0 \\
Size(\bullet) & = & 1 \\
Size(x) & = & Size(f) \;=\; 2 \textit{ for all } x \in \mathcal{X} \textit{ and } f \in \mathcal{K} \setminus \{\bullet\} \\
Size(G_1 * G_2) & = & Size(G_1) + Size(G_2) + 1 \textit{ where } * \in \{\ \_\wr\_,\ \_\ \_,\ \_ \ll \_\ \}
\end{array}
$$

$$\begin{aligned}
Size(G_1 \twoheadrightarrow G_2) &= Size(G_1) + Size(G_2) + 2\\
Size(G[E]) &= Size(G) + Size(E) + 1\\
Size(E, E') &= Size(E) + Size(E')\\
Size(x = G) &= Size(x) + Size(G)
\end{aligned}$$

We point out that the polynomial interpretation is compatible *w.r.t.* to neutrality of ϵ for the constraint conjunction operator. In fact $E, \epsilon = E$ and $Size(E, \epsilon) = Size(E) + Size(\epsilon) = Size(E) + 0 = Size(E)$. Similarly, it is compatible *w.r.t.* the associativity and commutativity of the conjunction and *w.r.t.* α-conversion. Moreover, function $Size(\cdot)$ can be easily seen to be monotonic and closed under contexts.

Lemma 15 (Context stability) *Let G_1 and G_2 be two ρ_g-terms. If $Size(G_1) > Size(G_2)$ then $Size(\mathsf{Ctx}\{G_1\}) > Size(\mathsf{Ctx}\{G_2\})$, for all contexts $\mathsf{Ctx}\{\Box\}$.*

Proof : Since the addition is increasing on naturals, the lemma is clearly satisfied. □

We next show that for all the rules in τ, the polynomial interpretation of the left-hand side is bigger than that of the right-hand side for any substitution of the rule (meta-)variables by positive naturals. As a consequence, we get the termination of the τ relation.

Lemma 16 *For all terms G_1 and G_2 such that $G_1 \mapsto_\tau G_2$ using any of the τ-rules, we have $Size(G_1) > Size(G_2)$.*

Proof : We proceed by analysing separately all the rules.

(ρ) $Size((P \twoheadrightarrow G_2)\ G_3) = Size(P) + Size(G_2) + Size(G_3) + 3$
$> Size(P) + Size(G_2) + Size(G_3) + 2 = Size(G_2[P \ll G_3])$

(propagate) $Size(P \ll (G[E])) = Size(P) + Size(G) + Size(E) + 2$
$> Size(P) + Size(G) + Size(E) + 1 = Size(P \ll G, E)$

(decompose) $Size(f(G_1, \ldots, G_n) \ll f(G'_1, \ldots, G'_n), E)$
$= 2 * Size(f) + \Sigma_{i=1}^n Size(G_i) + \Sigma_{i=1}^n Size(G'_i) + Size(E) + 1 + 2 * (n-1)$
$= 3 + \Sigma_{i=1}^n Size(G_i) + \Sigma_{i=1}^n Size(G'_i) + Size(E) + 2 * n$
$> \Sigma_{i=1}^n Size(G_i) + \Sigma_{i=1}^n Size(G'_i) + Size(E) + n$
$= Size(G_1 \ll G'_1, \ldots, G_n \ll G'_n, E)$

(solved) $Size(x \ll G, E) = Size(x) + Size(G) + Size(E) + 1$
$> Size(x) + Size(G) + Size(E) = Size(x = G, E)$

(garbage) $Size(G[E, x = G']) = Size(G) + Size(E) + Size(x) + Size(G') + 1$
$> Size(G) + Size(E) + 1 = Size(G[E])$

(garbage) $Size(G[\epsilon]) = Size(G) + 1 > Size(G)$

(black hole) $Size(\mathsf{Ctx}\{x\}[x =_\circ x, E]) = Size(\mathsf{Ctx}\{x\}) + Size(x =_\circ x) + Size(E) + 1$
$> Size(\mathsf{Ctx}\{\bullet\}) + Size(x =_\circ x) + Size(E) + 1$ *by Lemma 15*
$= Size(\mathsf{Ctx}\{\bullet\}[x =_\circ x, E])$

(black hole) $Size(G[P \lll \mathsf{Ctx}\{y\}, y =_\circ y, E])$
$= Size(G) + Size(P) + Size(\mathsf{Ctx}\{y\}) + Size(y =_\circ y) + Size(E) + 2$
$> Size(G) + Size(P) + Size(\mathsf{Ctx}\{\bullet\}) + Size(y =_\circ y) + Size(E) + 2$
by Lemma 15
$= Size(G[P \lll \mathsf{Ctx}\{\bullet\}, y =_\circ y, E])$

□

Lemma 17 (SN(τ)) *The relation τ is strongly normalizing.*

Proof : Clearly $Size(\cdot)$ associates a natural number to any constraint and ρ_{g}-graph (precisely $Size(E) \geq 0$ for any constraint E and $Size(G) \geq 1$ for any ρ_{g}-graph G). Now, by Lemma 16, for any rule $L \to R$ in τ we have $Size(L) > Size(R)$ for all interpretations of the meta-variables of L and R over natural numbers. Hence, by Lemma 15, for all ρ_{g}-graphs G_1 and G_2 such that $G_1 \mapsto_\tau G_2$ we have $Size(G_1) > Size(G_2)$. Therefore the relation τ is strongly normalising. □

The result on the termination of the relation achieved, we can analyse its local confluence modulo AC_ϵ. In order to prove this property, we proceed with the analysis of the critical pairs generated by the τ-rules.

Lemma 18 (LCON$_\sim$(τ)) *The relation τ is locally confluent modulo AC_ϵ.*

Proof : When the two steps from G to G_1 and from G to G_2 do not overlap, the lemma is easy. We inspect next the possible critical pairs. The (*decompose*) rule and the (*garbage*) rule generate only trivial critical pairs with the other τ-rules. The (ρ) rule generates a joinable critical pair with the (*black hole*) rule as shown in the next diagram:

$$\begin{array}{ccc} (P \twoheadrightarrow \mathsf{Ctx}\{x\})[x =_\circ x, E]\ G_3 & \xrightarrow[bh]{} & (P \twoheadrightarrow \mathsf{Ctx}\{\bullet\})[x =_\circ x, E]\ G_3 \\ \Big\downarrow{\scriptstyle \rho} & & \Big\downarrow{\scriptstyle \rho} \\ \mathsf{Ctx}\{x\}[P \ll G_3, x =_\circ x, E] & \xrightarrow[bh]{} & \mathsf{Ctx}\{\bullet\}[P \ll G_3, x =_\circ x, E] \end{array}$$

The (*propagate*) rule generates a joinable critical pair with the (*black hole*) rule:

$$\begin{array}{ccc} P \ll \mathsf{Ctx}\{x\}[x =_\circ x, E] & \xrightarrow[bh]{} & P \ll \mathsf{Ctx}\{\bullet\}[x =_\circ x, E] \\ \Big\downarrow{\scriptstyle p} & & \Big\downarrow{\scriptstyle p} \\ P \ll \mathsf{Ctx}\{x\}, x =_\circ x, E & \xrightarrow[bh]{} & P \ll \mathsf{Ctx}\{\bullet\}, x =_\circ x, E \end{array}$$

Another possible critical pair between the (*propagate*) and the (*solved*) rule is fixed adding the proviso $P \neq x$ in the (*propagate*) rule.

Finally, the (*solved*) rule generates a joinable critical pair with the (*black hole*) rule:

$$\begin{array}{ccc} y \ll \mathsf{Ctx}\{x\}, x =_\circ x, E & \xrightarrow[bh]{} & y \ll \mathsf{Ctx}\{\bullet\}, x =_\circ x, E \\ \Big\downarrow{\scriptstyle s} & & \Big\downarrow{\scriptstyle s} \\ y = \mathsf{Ctx}\{x\}, x =_\circ x, E & \xrightarrow[bh]{} & y = \mathsf{Ctx}\{\bullet\}, x =_\circ x, E \end{array}$$

It is clear from the previous cases the importance of having defined the (*black hole*) rule in such a way that it can replace variables not only in recursion equations, but also in match equations.

□

Similarly to standard term rewriting, we can use local confluence and strong normalisation to conclude about the confluence of a relation. Since here we are in the context of rewriting modulo a set of equations, beside the previous two properties, the compatibility property is also needed to ensure the confluence modulo AC_ϵ of the τ relation.

Proposition 10 (CON$_\sim(\tau)$) *The relation τ is confluent modulo AC_ϵ.*

Proof : Since the compatibility property is stronger than the local coherence property, we can conclude by Proposition 1 using Lemma 14, Lemma 17 and Lemma 18. □

8.3 Confluence modulo AC_ϵ for the Σ-rules

We present in this section the more elaborated part of the proof, namely the proof of confluence modulo AC_ϵ for the relation Σ induced by the Σ-rules. The difficulties in this issue arise from the fact that the rewrite relation Σ is not strongly normalizing. In particular, we notice that the rewrite relations induced by the substitution rules are both not terminating in the presence of cycles:

$$x[x = f(y), y = g(y)] \mapsto_{ac} x[x = f(g(y)), y = g(y)] \mapsto_{ac} \dots$$
$$y[y = g(y)] \mapsto_{es} g(y)[y = g(y)] \mapsto_{es} \dots$$

Consequently, in this case proving the local confluence is not sufficient, since the techniques used in the previous section for the τ relation cannot be exploited.

Taking inspiration from the confluence proof of the cyclic λ-calculus in [AK96b], we chose to use the so-called *complete development method* of the λ-calculus described in Section 2.1.1, adapting it to the relation Σ. The idea of this proof technique consists, first, in defining a new rewrite relation Cpl with the same transitive closure as the Σ relation and, secondly, in proving some properties of this relation, namely the diamond property modulo AC_ϵ, that will allow us to conclude on the confluence of the original Σ relation. Intuitively, a step of Cpl rewriting on a term G consists of the complete reduction of a set of redexes initially fixed in G. In other words, some redexes are marked in G and a complete development of these redexes is performed by the Cpl relation. Concretely, an underlining function is used to mark the redexes and the reductions on underlined redexes are then performed using the following underlined version of the Σ-rules:

$$\begin{array}{lll}
(\underline{\textit{external sub}}) & \mathsf{Ctx}\{\underline{y}\}[y = G, E] & \to_{\underline{es}} \ \mathsf{Ctx}\{G\}[y = G, E] \\
(\underline{\textit{acyclic sub}}) & G[G_0 \lll \mathsf{Ctx}\{\underline{y}\}, y = G_1, E] & \to_{\underline{ac}} \ G[G_0 \lll \mathsf{Ctx}\{G_1\}, y = G_1, E] \\
 & & \quad \textit{if } x > \underline{y},\ \forall x \in \mathcal{FV}(G_0) \\
(\underline{\delta}) & (G_1 \underline{\wr} G_2)\ G_3 & \to_{\underline{\delta}} \ G_1\ G_3 \underline{\wr} G_2\ G_3 \\
 & (G_1 \underline{\wr} G_2)[E]\ G_3 & \to_{\underline{\delta}} \ (G_1\ G_3 \underline{\wr} G_2\ G_3)[E]
\end{array}$$

We call the new rewrite relation $\underline{\Sigma}$ and the associated calculus $\underline{\Sigma}$-calculus. Terms belonging to the $\underline{\Sigma}$-calculus are ρ_{g}-terms in which some recursion variables belonging to a $\underline{\Sigma}$-redex are underlined.

Example 42 (Terms of the $\underline{\Sigma}$-calculus)

- $\underline{x}[x = f(x)]$ *is a legal term.*
- $x[x = f(\underline{x})]$ *is not a legal term, since* $x \equiv \underline{x}$ *and thus the proviso for the application of the* ($\underline{acyclic\ sub}$) *rule is not verified.*
- *Similarly,* $f(x)[x \ll f(\underline{y}), y = g(\underline{z}, \underline{z}), z = \underline{y}]$ *is not a legal term, since* $x > \underline{y}$ *but* $\underline{z} \equiv \underline{y}$.
- $f(x)[x \ll f(\underline{y}), y = g(\underline{z}, \underline{z}), z = y]$ *is a legal term, since here* $x > \underline{y}$ *and also* $y > \underline{z}$.

The *Cpl* rewrite relation is then defined as follows.

Definition 91 (*Cpl* relation) *Given the terms* G_1 *and* G_2 *in the* Σ*-calculus, we have that* $G_1 \mapsto\!\!\!\!\rightarrow_{Cpl} G_2$ *if there exists an underlining* G_1' *of* G_1 *such that* $G_1' \mapsto\!\!\!\!\rightarrow_{\underline{\Sigma}} G_2$ *and* G_2 *is in normal form w.r.t. the relation* $\underline{\Sigma}$.

Example 43 (Reductions in the $\underline{\Sigma}$-calculus)

- *The term* $x[x = f(\underline{y}), y = g(y)]$ *reaches the* $\underline{\Sigma}$ *normal form* $x[x = f(g(y)), y = g(y)]$ *in one* ($\underline{ac}$)*-step.*
- *We have the reduction* $G_1 = x[f(x, y) \ll f(\underline{z}, \underline{z}), z = g(\underline{w}), w = a] \mapsto_{\underline{\Sigma}} x[f(x, y) \ll f(\underline{z}, \underline{z}), z = g(a), w = a] \mapsto_{\underline{\Sigma}} x[f(x, y) \ll f(g(a), g(a)), z = g(a), w = a] = G_2$ *and thus* $G_1 \mapsto_{Cpl} G_2$.

To ensure the fact that for every possible choice of redexes in G_1 we have a corresponding *Cpl* reduction, we need to prove that for every term there exists a normal form *w.r.t.* the $\underline{\Sigma}$ reduction, *i.e.* we must prove that $\underline{\Sigma}$ is weakly normalizing. To this aim, we prove first that the (δ) rule and the underlined substitution rules separately are strongly normalizing.

Lemma 19 $SN(\underline{\delta})$ *and* $SN(\underline{es \cup ac})$ *hold.*

Proof : The strong normalisation of the relation induced by the ($\underline{\delta}$) rule can be proved by using the multiset path ordering induced by the following precedence on the operators of the ρ_{g}-calculus:

$$\_\ \_ \succ \_\wr\_ \succ \_[\_] \succ \_ \ll \_ \succ \_,\_ \succ \_ = \_$$

For the first of the two ($\underline{\delta}$) rules we have to verify that $(G_1 \underline{\wr} G_2)\ G_3 >_{mpo} G_1\ G_3 \wr G_2\ G_3$. According to the given precedence, the the top application operator of the left-hand side is greater than the top operator of the right-hand side, *i.e.* $(\_\ \_) \succ (\_\wr\_)$, and thus we split the proof into two cases:

- $(G_1 \underline{\wr} G_2)\ G_3 >_{mpo} G_1\ G_3$.
  The two sides of the inequality have the same top operator and therefore we consider the two multisets $\{G_1 \underline{\wr} G_2, G_3\}$ and $\{G_1, G_3\}$ respectively. It is easy to see that the first one is greater then the second.
- $(G_1 \underline{\wr} G_2)\ G_3 >_{mpo} G_2\ G_3$.
  The same reasoning as in the previous case applies.

For the second one of the ($\underline{\delta}$) rules, we have to show that $(G_1 \underline{\wr} G_2)[E]\ G_3 >_{mpo} (G_1\ G_3 \wr G_2\ G_3)[E]$. We have $(\_\ \_) \succ (\_[\_])$ and so we show

- $(G_1 \underline{\wr} G_2)[E]\ G_3 >_{mpo} E$.
  This is true since E is contained in the left-hand side.

- $(G_1 \underline{\wr} G_2)[E]\ G_3 >_{mpo} (G_1\ G_3 \wr G_2\ G_3)$.
  Here we have $(\_\ \_) \succ (\_ \wr \_)$ therefore we decompose the right-hand side and we get two cases $(G_1 \underline{\wr} G_2)[E]\ G_3 >_{mpo} G_1\ G_3$ and $(G_1 \underline{\wr} G_2)[E]\ G_3 >_{mpo} G_2\ G_3$ that are similar to the ones already analysed for the first rule.

For proving the termination of the $(\underline{es \cup ac})$ relation we exploit a technique similar to that in [AK96b]. We define the weight associated to a term of the $\underline{\Sigma}$-calculus as the multiset of all its underlined recursion variables and we show that the weight decreases during the reduction using the multiset ordering and the fact that the ordering $>$ among recursion variables (see Definition 82) is well defined. We analyse the two different cases:

- If we substitute an underlined recursion variable by a term containing no underlined variables, then the weight trivially decreases. For example $\underline{x}[x = f(x)]$ has weight $\{\underline{x}\}$ while its reduct $f(x)[x = f(x)]$ has weight $\emptyset$.

- If we substitute an underlined recursion variable $\underline{x}$ by a term containing one or more recursion variables $\underline{y_1}, \ldots, \underline{y_n}$, then we have $\underline{x} > \underline{y_i}$ for all $i = 1, \ldots, n$ otherwise the term would not be a legal $\underline{\Sigma}$-calculus term. It follows that the multiset of the reduced term is smaller than the one associated to the initial term. Consider for example the reduction

  $$G = x[x = \mathsf{C}_0\{\underline{y_1}\}, y_1 = \mathsf{C}_1\{\underline{y_2}\}, y_2 = G'] \mapsto_{\underline{ac}} x[x = \mathsf{Ctx}_0\{\mathsf{C}_1\{\underline{y_2}\}\}, y_1 = \mathsf{C}_1\{\underline{y_2}\}, y_2 = G']$$

  The multiset associated to G is $\{\underline{y_1}, \underline{y_2}\}$. By the definition of the order on recursion variables, we have $x > \underline{y_1}$ and $y_1 > \underline{y_2}$, so the multiset $\{\underline{y_2}, \underline{y_2}\}$ associated to G after reduction is smaller. Notice that $y_1 \neq y_2$ otherwise the proviso of the (*acyclic sub*) rule would not be satisfied and G would not be a legal term. For the same reason, no $\underline{y_1}$ is allowed on the right-hand side of the recursion equation for y_2.

□

Proposition 11 (WN($\underline{\Sigma}$)) *The relation $\underline{\Sigma}$ is weakly normalizing.*

Proof : By using the rewriting strategy where $(\underline{\delta})$ has greater priority than $(\underline{es \cup ac})$. By Lemma 19 we know that $(\underline{\delta})$ and $(\underline{es \cup ac})$ are strongly normalising. Observe that the $(\underline{es \cup ac})$ relation does not generate $(\underline{\delta})$ redexes. Hence we can normalise a term G first *w.r.t.* the $(\underline{\delta})$ relation and then *w.r.t.* the $(\underline{es \cup ac})$ relation obtaining thus a finite reduction of G. □

The next goal is to prove the diamond property of the *Cpl* relation. In order to do this, the confluence modulo AC_ϵ of the $\underline{\Sigma}$ relation is needed. Since we know that both the relations $(\underline{\delta})$ and $(\underline{es \cup ac})$ are strongly normalizing, we prove their local confluence modulo AC_ϵ by analysis of the critical pairs and then we conclude on their confluence modulo AC_ϵ. The confluence modulo AC_ϵ of the $\underline{\Sigma}$ relation will then follow using a commutation lemma.

Lemma 20 *$LCON_\sim(\underline{\delta})$ and $LCON_\sim(\underline{es \cup ac})$ hold.*

Proof : If the redexes are disjoint the result is easy to prove. We proceed by analysis of the critical pairs. The critical pairs of the $(\underline{\delta})$ rule with itself are trivial. Among the critical pairs of the $(\underline{external\ sub})$ and $(\underline{acyclic\ sub})$ rules, we show next the diagrams for two interesting cases. We consider the case where $\lll$ is equal to $=$. The case with $\lll$ equal to $\ll$ can be treated in the same way. To ease the notation, from now on we will write just $\mathsf{C}_i\{G\}$ for a context $\mathsf{Ctx}_i\{G\}$ in the critical pairs diagrams.

- We consider the critical pair generated from a term having a list of constraints containing two ($\underline{ac}$) redexes with a cyclic plane in common. Notice that the recursion variable $\underline{z}$ can be duplicated by the first ($\underline{ac}$)-step.

$$\begin{array}{ccc} G_0[y = \mathsf{C}_1\{\underline{x}\}, x = \mathsf{C}_2\{\underline{z}\}, z = G_1] & \xrightarrow[\underline{ac}]{} & G_0[y = \mathsf{C}_1\{\underline{x}\}, x = \mathsf{C}_2\{G_1\}, z = G_1] \\ \downarrow \underline{ac} & & \vdots\, \underline{ac} \\ G_0[y = \mathsf{C}_1\{\mathsf{C}_2\{\underline{z}\}\}, x = \mathsf{C}_2\{\underline{z}\}, z = G_1] & \xrightarrow[\underline{ac}]{} & G_0[y = \mathsf{C}_1\{\mathsf{C}_2\{G_1\}\}, x = \mathsf{C}_2\{G_1\}, z = G_1] \end{array}$$

- We consider the critical pair in which the term duplicated by an ($\underline{es}$) step contains an ($\underline{ac}$) redex. Notice that we need both the substitution rules, *i.e.* $\underline{ac \cup es}$, to close the diagram.

$$\begin{array}{ccc} \mathsf{C}_0\{\underline{y}\}[y = \mathsf{C}_1\{\underline{x}\}, x = \mathsf{C}_2\{x\}] & \xrightarrow[\underline{es}]{} & \mathsf{C}_0\{\mathsf{C}_1\{\underline{x}\}\}[y = \mathsf{C}_1\{\underline{x}\}, x = \mathsf{C}_2\{x\}] \\ \downarrow \underline{ac} & & \vdots\, \underline{ac \cup es} \\ \mathsf{C}_0\{\underline{y}\}[y = \mathsf{C}_1\{\mathsf{C}_2\{x\}\}, x = \mathsf{C}_2\{x\}] & \xrightarrow[\underline{ac}]{} & \mathsf{C}_0\{\mathsf{C}_1\{x\}\}[y = \mathsf{C}_1\{\mathsf{C}_2\{x\}\}, x = \mathsf{C}_2\{x\}] \end{array}$$

□

At this point, it is worth noticing that the local compatibility with AC_ϵ holds for the underlined version of the rules. This property, together with the local confluence modulo AC_ϵ and the strong normalisation for the ($\underline{\delta}$) and the ($\underline{es \cup ac}$) relations is sufficient to prove their confluence.

Lemma 21 *$CPB_\sim(\underline{\delta})$ and $CPB_\sim(\underline{es \cup ac})$ hold.*

Proof : By Lemma 14 we know that the property holds for the original version of the rules, without underlining. Since equivalence steps in the AC_ϵ theory have no effect with respect to the underlining, which means that an underlined redex remains an underlined redex after one or more steps of AC_ϵ equality, we can conclude that the lemma is true also for the underlined rules. □

Lemma 22 *$CON_\sim(\underline{\delta})$ and $CON_\sim(\underline{es \cup ac})$ hold.*

Proof : By Proposition 1 using Lemma 19, Lemma 20, Lemma 21 and the fact that the compatibility property implies the local coherence property. □

After having proved the confluence modulo AC_ϵ of the two subsets of rules independently, following Proposition 2, we need a commutation lemma to be able to conclude about the confluence of the union of the two subsets.

Lemma 23 *$COM_\sim(\underline{\delta}, \underline{es \cup ac})$ hold.*

Proof : General commutation is not easy to prove, thus we prove a simpler property which implies commutation. By Lemma 21, we know that the compatibility property holds for our relations. Unfortunately, the two relations are not strongly commuting, since both of them can duplicate redexes of the other one. Nevertheless, the relations do not interfere with each other, in the sense that a redex for *e.g.* the ($\underline{\delta}$) rule will still be present (possibly duplicated) after one or several steps of ($\underline{es \cup ac}$). Therefore, we will use Proposition 3 and we need simply to verify the property

$$\leftarrow_{\underline{es \cup ac}} \cdot \mapsto\!\!\!\!\rightarrow_{\underline{\delta}} \;\subseteq\; \mapsto\!\!\!\!\rightarrow_{\underline{\delta}} \cdot \sim_E \cdot \twoheadleftarrow_{\underline{es \cup ac}}$$

We do this by analysis of the critical pairs. We show the critical pairs between the ($\underline{\delta}$) rule and the ($\underline{es}$) rule, the critical pairs between the ($\underline{\delta}$) rule and the ($\underline{ac}$) rule being similar. In particular, we present in the next diagram a critical pair in which the ($\underline{\delta}$) and the ($\underline{es}$) redexes are not disjoint. We recall that there exists no infinite ($\underline{\delta}$) or ($\underline{es \cup ac}$) reduction by Lemma 19. For the sake of simplicity, in the diagram we show a single ($\underline{\delta}$) step. A longer derivation would bring to a further duplication of the ($\underline{es \cup ac}$) redex but the diagram could be closed in a similar way.

$$\begin{array}{ccc}
\mathsf{C}_0\{(G_1 \underline{\wr} G_2)\ \mathsf{C}_1\{\underline{x}\}\}[x = G, E] & \xrightarrow[\underline{\delta}]{} & \mathsf{C}_0\{(G_1\ \mathsf{C}_1\{\underline{x}\}) \underline{\wr} (G_2\ \mathsf{C}_1\{\underline{x}\})[x = G, E] \\
\Big\downarrow{\scriptstyle \underline{es \cup ac}} & & \Big\downarrow{\scriptstyle \underline{es \cup ac}} \\
\mathsf{C}_0\{(G_1 \underline{\wr} G_2)\ \mathsf{C}_1\{G\}\}[x = G, E] & \xrightarrow[\underline{\delta}]{} & \mathsf{C}_0\{(G_1\ \mathsf{C}_1\{G\}) \underline{\wr} (G_2\ \mathsf{C}_1\{G\})[x = G, E]
\end{array}$$

□

Taking advantage of the previous three lemmata, it is now possible to show the confluence modulo AC_ϵ of the $\underline{\Sigma}$ relation.

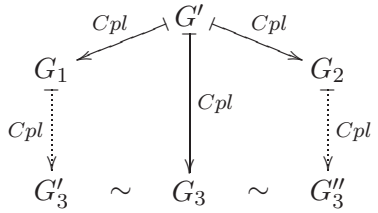
Proposition 12 ($\mathbf{CON}_\sim(\underline{\Sigma})$) *The reduction relation $\underline{\Sigma}$ is confluent modulo AC_ϵ.*

Proof : By Proposition 2, using Lemma 21, Lemma 22 and Lemma 23. □

Using the weak termination of the relation $\underline{\Sigma}$ and its confluence modulo AC_ϵ, we can finally prove that the diamond property modulo AC_ϵ holds for the Cpl relation.

Lemma 24 ($\mathbf{D}_\sim(Cpl)$) *The rewrite relation Cpl enjoys the diamond property modulo AC_ϵ.*

Proof : Given a term G', let $S = S_1 \cup S_2$ be a set of underlined redexes in G' such that we have $G' \mapsto\!\!\!\!\!\rightarrow_{Cpl} G_3$ reducing all the underlined redexes in S. Let G_1 and G_2 be the two partial developments relative to S_1 and S_2 respectively, *i.e.* $G' \mapsto\!\!\!\!\!\rightarrow_{Cpl} G_1$ reducing only the redexes in S_1 and $G' \mapsto\!\!\!\!\!\rightarrow_{Cpl} G_2$ reducing only the redexes in S_2. In both cases, since $WN(\underline{\Sigma})$ holds by Proposition 11, we can continue reducing the remaining underlined redexes and obtaining $G_1 \mapsto\!\!\!\!\!\rightarrow_{Cpl} G'_3$ and $G_2 \mapsto\!\!\!\!\!\rightarrow_{Cpl} G''_3$. Since all the steps in the Cpl reduction are $\underline{\Sigma}$ steps, using the fact that G_3, G'_3 and G''_3 are completely reduced *w.r.t.* $\underline{\Sigma}$ and that $CON_\sim(\underline{\Sigma})$ holds by Proposition 12, we can conclude on the equivalence of G_3, G'_3 and G''_3.



□

The confluence of the Σ relation follows easily by noticing that the Σ relation and the Cpl relation have the same transitive closure.

Proposition 13 ($\mathbf{CON}_\sim(\Sigma)$) *The reduction relation Σ is confluent modulo AC_ϵ.*

Proof : The result follows by Lemma 24, since if Cpl satisfies the diamond property modulo AC_ϵ, so does its transitive closure and it is not difficult to show that the transitive closure of the relation Cpl is the same as that of the relation Σ. For doing this we use the inclusions $\mapsto_\Sigma \subseteq \mapsto\!\!\!\!\!\rightarrow_{Cpl} \subseteq \mapsto\!\!\!\!\!\rightarrow_\Sigma$. The first inclusion can be proved by underlining the redex reduced by the Σ step. The second inclusion follows trivially from the definition of the Cpl relation. □

8.4 General confluence

So far we have shown the confluence of the two subsets of rules τ and Σ separately. In the last part of the proof we consider the union of the these two subsets. General confluence holds since we can prove the commutation modulo AC_ϵ between the τ-rules and the Σ-rules.

Lemma 25 ($\mathbf{COM}_\sim(\tau, \Sigma)$) *The relations τ and Σ commute modulo AC_ϵ.*

Proof : Since the Σ-rules do not terminate, it is easier to show strong commutation between the two sets of rules instead of general commutation.

$$\begin{array}{ccc} G & \overset{\tau}{\longmapsto} & G_1 \\ {\scriptstyle\Sigma}\downarrow & & {\scriptstyle\Sigma}\,{\scriptstyle 0/1}\,\vdots \\ G_2 & \overset{\tau}{\longmapsto\!\!\!\!\!\rightarrow} & G'_1 \;\sim\; G'_2 \end{array}$$

We can then conclude by Proposition 4, using the compatibility with AC_ϵ for the two relations τ and Σ of Lemma 14. The possibility of closing the diagram using at most one step for the Σ-rules is ensured by the fact that none of the τ-rules is duplicating.

If the applied Σ-rule is the (δ) rule, the diagram can be easily closed, since the τ-rules do not interfere with (δ) redexes (the generated critical pairs are trivial). Only the (*garbage*) rule can interact with the (δ) redex by eliminating it and in this case the diagram is closed with zero (δ) steps.

If the applied Σ-rule is a substitution rule, we analyse next the interesting critical pairs.

- The τ-rule applied to G is the (*propagate*) rule. The only interesting case is the following where the two Σ-rules applied are different.

$$\begin{array}{ccc} P \ll (\mathsf{Ctx}\{y\}[y = H, E]) & \overset{p}{\longmapsto} & P \ll \mathsf{Ctx}\{y\}, y = H, E \\ {\scriptstyle es}\downarrow & & {\scriptstyle ac}\,\vdots \\ P \ll (\mathsf{Ctx}\{H\}[y = H, E]) & \overset{p}{\longmapsto\!\!\!\!\!\rightarrow} & P \ll \mathsf{Ctx}\{H\}, y = H, E \end{array}$$

- The τ-rule applied to G is the (*decompose*) rule. In this case the term G is of the form $H[f(H_1, \ldots, H_n) \ll f(\mathsf{Ctx}\{y\}, \ldots, H'_n), y = H', E]$. The (*decompose*) rule transforms the match equation in a set of simpler constraints $H_1 \ll \mathsf{Ctx}\{y\}$,...,$H_n \ll H'_n$ in the same list. Since the (*acyclic sub*) rule is applied using matching modulo AC_ϵ, the substitution generated from $y = H'$ can be equivalently performed either before or after the decomposition.

- The τ-rule applied to G is the (*solved*) rule. In this case, there are no differences between G_1 and G from the point of view of the application of a substitution rule.

- The τ-rule applied to G is the (*garbage*) rule. The particularity here is that we can have zero steps of the Σ rules for closing the diagram when the substitution redex is part of the subgraph which is eliminated by garbage collection.

- The τ-rule applied to G is the (*black hole*) rule.

We may have an overlap of the (*external sub*) rule and the (*black hole*) rule if the term duplicated by the substitution is a variable.

$$\begin{array}{ccc} \mathsf{Ctx}\{y\}[y=y,E] & \xrightarrow[bh]{} & \mathsf{Ctx}\{\bullet\}[y=y,E] \\ {\scriptstyle es}\downarrow & & {\scriptstyle es}\,\vdots\,0 \\ \mathsf{Ctx}\{y\}[y=y,E] & \overset{bh}{\twoheadrightarrow} & \mathsf{Ctx}\{\bullet\}[y=y,E] \end{array}$$

If the cycle has length greater than one, *i.e.* it is expressed by more than one recursion equation, the matching modulo AC_ϵ allows to apply the (*black hole*) rule even when the recursion equations are not in the right order in the list and this can happen as a consequence of the application of the (*external sub*) rule.

$$\begin{array}{ccc} \mathsf{Ctx}\{y\}[y=x,x=y,E] & \xrightarrow[bh]{} & \mathsf{Ctx}\{\bullet\}[y=x,x=y,E] \\ {\scriptstyle es}\downarrow & & {\scriptstyle es}\,\vdots\,0 \\ \mathsf{Ctx}\{x\}[y=x,x=y,E] & \overset{bh}{\twoheadrightarrow} & \mathsf{Ctx}\{\bullet\}[y=x,x=y,E] \end{array}$$

We have similar cases for the (*acyclic sub*) rule.

□

The confluence modulo AC_ϵ of the the sets of rules τ and Σ, the commutation modulo AC_ϵ of the two sets, together with their compatibility property with AC_ϵ ensure the confluence of their union.

Theorem 12 (CON$_\sim(\rho_g, AC_\epsilon)$) *In the linear ρ_{g}-calculus, the rewrite relation ρ_g modulo AC_ϵ is confluent modulo AC_ϵ.*

Proof : By Proposition 2 using Proposition 10, Proposition 13, Lemma 14 and Lemma 25. □

As mentioned in the first section, what we aim at is a more general result about rewriting on AC_ϵ-equivalence classes of ρ_g-terms. In [JK84] it is shown how from the confluence of a rewrite relation modulo a set of axioms E we can deduce the Church-Rosser property of the relation on E-equivalence classes. This result does not directly apply to the ρ_{g}-calculus because the system is not strongly normalizing. Anyway, thanks to the strong property of compatibility of ρ_{g} with AC_ϵ, the Church-Rosser property on AC_ϵ-equivalence classes for the ρ_{g}-calculus rewrite relation can be achieved from the latter theorem without the need of the strong normalization hypothesis.

Theorem 13 ($CR_\sim(\rho_g/AC_\epsilon)$) *The linear ρ_{g}-calculus is Church-Rosser modulo AC_ϵ.*

Proof : By induction on the length of the reduction.
To emphasise the first step, we decompose the reduction $\twoheadleftarrow\!\!\twoheadrightarrow^n_{\rho_g/AC_\epsilon}$ into $\twoheadleftarrow\!\!\twoheadrightarrow^1_{\rho_g/AC_\epsilon}\ \twoheadleftarrow\!\!\twoheadrightarrow^{n-1}_{\rho_g/AC_\epsilon}$. We have three possibilities for the first step. For each case, we show the Church-Rosser diagram in Figure 8.2. The dotted arrows in the diagram correspond to derivations of the relation $\rho_g, AC_\epsilon \subseteq \rho_g/AC_\epsilon$, following Definition 30. To close the diagram, we thus use the local confluence modulo AC_ϵ of the relation ρ_g, AC_ϵ showed in Theorem 12 (denoted LC), the compatibility property (CP) showed in Lemma 14 and the induction hypothesis (IH).

□

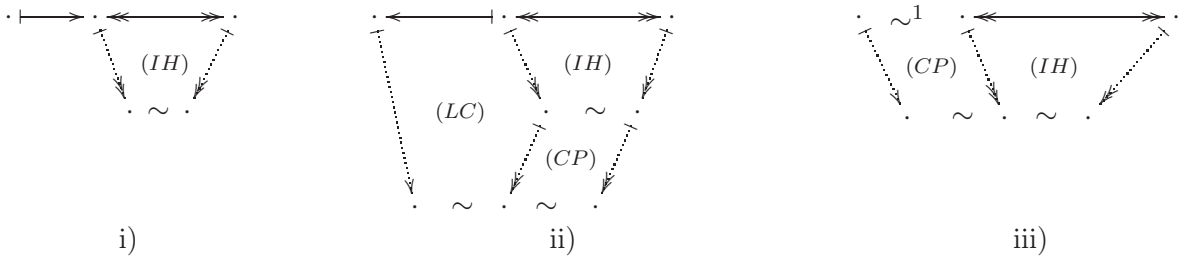


Figure 8.2: Church-Rosser property for ρ_g/AC_ϵ

8.5 Summary: outline of the proof

Confluence is first shown for the relation (ρ_g, AC_ϵ). In order to do this, the ρ_{g}-calculus rules are split into two subsets, τ and Σ, and confluence is proved independently for each set:

- The τ-rules
  1. $CPB_\sim(\tau)$ Lemma 14
  2. $LCON_\sim(\tau)$ Lemma 18
  3. $SN(\tau)$ Lemma 17

  $\Longrightarrow$ $CON_\sim(\tau)$ Lemma 10

- The Σ-rules

  We define a reduction $\underline{\Sigma} = \underline{\delta} \cup \underline{(es \cup ac)}$ and we show its termination:

  – $WN(\underline{\Sigma})$ Lemma 11.

  We show confluence for the underlined delta and substitution rules separately:

  – $CON_\sim(\underline{\delta})$ and $CON_\sim(\underline{es \cup ac})$ Lemma 22

  1. $SN(\underline{\delta})$ and $SN(\underline{es \cup ac})$ Lemma 19
  2. $LCON_\sim(\underline{\delta})$ and $LCON_\sim(\underline{es \cup ac})$ Lemma 20
  3. $CPB_\sim(\underline{\delta})$ and $CPB_\sim(\underline{es \cup ac})$ Lemma 21

  $\Longrightarrow$ $CON_\sim(\underline{\delta})$ and $CON_\sim(\underline{es \cup ac})$ Lemma 22

  and we conclude on the confluence of the union:

  1. $CON_\sim(\underline{\delta})$ and $CON_\sim(\underline{es \cup ac})$
  2. $COM_\sim(\underline{\delta}, (\underline{es \cup ac}))$ Lemma 23
  3. $CPB_{AC_\epsilon}(\underline{\delta})$ and $CPB_{AC_\epsilon}(\underline{es \cup ac})$ Lemma 21

  $\Longrightarrow$ $CON_\sim(\underline{\Sigma})$ Lemma 12

  Weak termination and confluence of the $\underline{\Sigma}$ relation enable us to define a new rewrite relation, called Cpl, which has the the same transitive closure as $\underline{\Sigma}$. The Cpl relation is shown to enjoy the diamond property $D_\sim(Cpl)$, Lemma 24. Using this result, we obtain the confluence of the Σ-rules: $CON_\sim(\Sigma)$ Lemma 13

We use then the confluence of the two sets of rules to have general confluence for the (ρ_g, AC_ϵ) relation :

1. $CON_\sim(\Sigma)$ and $CON_\sim(\tau)$ Lemma 10 and Lemma 12
2. $COM_\sim(\tau, \Sigma)$ Lemma 25
3. $CPB_\sim(\tau)$ and $CPB_\sim(\Sigma)$ Lemma 14

$\Longrightarrow$ $CON_\sim(\rho_g, AC_\epsilon)$ Theorem 12

We conclude by showing the Church-Rosser property modulo AC_ϵ for the ρ_g-calculus rewrite relation on equivalence classes of ρ_g-terms, denoted $CR_\sim(\rho_g/AC_\epsilon)$.

1. $CPB_\sim(\rho_\mathsf{g})$ Lemma 14
2. $CON_\sim(\rho_g, AC_\epsilon)$ Theorem 12

$\Longrightarrow$ $CR_\sim(\rho_g/AC_\epsilon)$ Theorem 13.

Conclusion

In this chapter we presented the proof of the confluence for the ρ_g-calculus restricted to linear algebraic patterns. Since two linear ρ_g-terms are considered equivalent modulo the order of the constraints in their list and the presence of empty constraints, they are grouped into equivalence classes defined modulo associativity (A), commutativity (C) and neutral element (ϵ) of the conjunction operator for lists. Thus, we have proved, more precisely, the Church-Rosser property modulo AC_ϵ for the ρ_g rewrite relation acting over AC_ϵ-equivalence classes, denoted ρ_g/AC_ϵ.

To achieve this result, the more operational relation (ρ_g, AC_ϵ), which performs matching modulo AC_ϵ on single ρ_g-terms, is used throughout the proof. The evaluation rules of the ρ_g-calculus are split into two subsets of rules for which confluence is first proved independently. Then, this intermediary result, together with a commutation lemma, is used for proving the confluence of the union of the two subsets. The first subset, including non-terminating rules, is shown to be confluent adapting the finite development method defined for the λ-calculus to terms containing constraints, while the second subset is shown to be confluent using classical propositions known from term rewriting adapted to the setting of rewriting modulo a set of equations. For example, the confluence of the union of two confluent and commuting relations or the confluence of a terminating and locally confluent relation hold if moreover a further property, ensuring the well-behaviour of the rewrite relation with the congruence relation modulo which rewriting is performed, is satisfied. This property, called compatibility, holds for any subset of rules of the ρ_g-calculus and is used in many lemmata of the proof. It is a key ingredient also of the most important theorem of the chapter, Theorem 13, where the confluence of the ρ_g/AC_ϵ relation is proved, by deducing it from the confluence of the (ρ_g, AC_ϵ) relation.

Partial results of the work presented here have been published in [Ber05].

Chapter 9

Expressivity of the ρ_{g}-calculus

The ρ_{g}-calculus is a higher-order framework integrating the pattern matching capabilities of the plain ρ-calculus with the term graph features of the cyclic λ-calculus. We develop in this chapter a formal comparison between the ρ_{g}-calculus and these two other formalisms.

Reductions performed in the ρ-calculus and reductions starting from the same term in the ρ_{g}-calculus are compared discussing their similarities and explaining their differences.

The representation of λ-graphs and their reductions in the ρ_{g}-calculus is realised defining a translation from λ-graphs into ρ_{g}-terms and showing that conservativity holds *w.r.t.* the two calculi.

Moreover, since the ρ_{g}-calculus is able to manipulate terms with sharing and cycles, the question about the relation with term graph rewriting [BvEG$^+$87, AK96a] naturally arises. We analyse the relationship between the reduction of a first-order term graph with respect to a set of term graph rewrite rules and the reduction of the corresponding ρ_{g}-term.

9.1 ρ_{g}-calculus versus ρ-calculus

We describe in this section how standard ρ-reductions, *i.e.* reductions performed on terms seen as trees, can be equivalently performed in the ρ_{g}-calculus.

We start by noticing that the set of terms of the ρ-calculus is a strict subset of the set of terms of the ρ_{g}-calculus. The main difference for ρ-terms is the restriction of the list of constraints to a single constraint necessarily of the form $\_ \ll \_$ (delayed matching constraint).

A first correspondence between the two calculi can be established at the level of the matching algorithm. The MATCHING RULES of the ρ_{g}-calculus are well-behaved with respect to the ρ-calculus matching algorithm restricted to algebraic patterns (see [CKL02]), as shown in the next lemma. This ensures that a successful matching in the ρ-calculus will certainly have a successful counterpart in the ρ_{g}-calculus.

Lemma 26 *Let T be an algebraic ρ-term with $\mathcal{FV}(T) = \{x_1, \ldots, x_n\}$ and let $T \ll U$ be a matching problem with solution $\sigma = \{x_1/U_1, \ldots, x_n/U_n\}$, i.e. $\sigma(T) = U$. Then we have the reduction $T \ll U \mapsto\!\!\!\!\rightarrow_{\mathcal{M}} x_1 = U_1, \ldots, x_n = U_n$.*

Proof : We show by structural induction on the term T that there exists a reduction

$$T \ll U \;\mapsto\!\!\!\!\rightarrow_{\mathcal{M}}\; x_1 \ll U_1, \ldots, x_n \ll U_n$$

where the x_i's are all distinct. The thesis follows, since we can apply successively n steps of the *solved* rule to transform the n match equations into n recursive equations.

Basic case: The term T is a variable or a constant. The case where $T = x$ is trivial.
If $T = a$ then $\sigma = \{\}$ and $U = a$. In the ρ_{g}-calculus we have $a \ll a \mapsto_e \epsilon$ and the property obviously holds.

Induction case: $T = f(T_1, \ldots, T_m)$ with $m > 0$.
Since a substitution σ exists and the matching is syntactic, we have $U = f(V_1, \ldots, V_m)$ and $\sigma(f(T_1, \ldots, T_m)) = f(\sigma(T_1), \ldots, \sigma(T_m))$ with $\sigma(T_i) = V_i$, for $i = 1 \ldots m$. By induction hypothesis, for any i, if $\mathcal{FV}(T_i) = \{x_1^i, \ldots, x_{k_i}^i\} \subseteq \mathcal{FV}(T)$, then $T_i \ll V_i \mapsto\!\!\!\!\!\rightarrow_{\mathcal{M}} x_1^i \ll \sigma(x_1^i), \ldots, x_{k_i}^i \ll \sigma(x_{k_i}^i)$. Joining the various reductions we have

$$f(T_1, \ldots, T_m) \ll f(V_1, \ldots, V_m) \;\mapsto_{dk}\; T_1 \ll V_1, \ldots, T_m \ll V_m \;\mapsto\!\!\!\!\!\rightarrow_{\mathcal{M}}\; x_1 \ll \sigma(x_1), \ldots, x_n \ll \sigma(x_n)$$

To understand the last step note that in the list

$$x_1^1 \ll \sigma(x_1^1), \ldots, x_{k_1}^1 \ll \sigma(x_{k_1}^1), \ldots, x_1^m \ll \sigma(x_m^1), \ldots, x_{k_m}^m \ll \sigma(x_{k_m}^m)$$

constraints with the same left-hand side variable have identical right-hand sides. Hence, by idempotency, this list coincides with $x_1 \ll \sigma(x_1), \ldots, x_n \ll \sigma(x_n)$.

□

Taking advantage of the previous lemma, we can show that every rewrite step in the ρ-calculus can be simulated in the ρ_{g}-calculus and consequently that for every ρ-reduction there exists a corresponding reduction in the ρ_{g}-calculus.

Lemma 27 *Let T and T' be ρ-terms. If there exists a one step reduction $T \mapsto\!\!\!\!\!\rightarrow_{\rho\sigma\delta} T'$ in the ρ-calculus then there exists a corresponding reduction $T \mapsto\!\!\!\!\!\rightarrow_{\rho g} T'$ in the ρ_{g}-calculus.*

Proof : We show that for each reduction step in the ρ-calculus we have a corresponding sequence of reduction steps in the ρ_{g}-calculus.

- If $T \mapsto_{\rho} T'$ or $T \mapsto_{\delta} T'$ in the ρ-calculus, then we trivially have the same reduction in the ρ_{g}-calculus using the corresponding rules.
- If $T = T_2[T_1 \ll T_3] \mapsto_{\sigma} \sigma(T_2) = T'$ where T_1 is a ρ-calculus pattern and the substitution $\sigma = \{U_1/x_1, \ldots, U_m/x_m\}$ is solution of the matching then, in the ρ_{g}-calculus the corresponding reduction is the following:

$$\begin{array}{ll} T = & T_2[T_1 \ll T_3] \\ \mapsto\!\!\!\!\!\rightarrow_{\mathcal{M}} & T_2[x_1 = U_1, \ldots, x_m = U_m] \quad \textit{(by Lemma 26)} \\ \mapsto\!\!\!\!\!\rightarrow_{es} & \sigma(T_2)[x_1 = U_1, \ldots, x_m = U_m] \\ \mapsto\!\!\!\!\!\rightarrow_{gc} & \sigma(T_2)[\epsilon] \\ \mapsto_{gc} & \sigma(T_2) = T' \end{array}$$

□

Theorem 14 (Completeness) *Let T and T' be ρ-terms. If there exists a reduction $T \mapsto\!\!\!\!\!\rightarrow_{\rho\sigma\delta} T'$ in the ρ-calculus then $T \mapsto\!\!\!\!\!\rightarrow_{\rho g} T'$ in the ρ_{g}-calculus.*

Proof : Follows from Lemma 27. □

In the case of matching failures, the two calculi handle errors in a slightly different way, even if, in both cases, matching clashes are not reduced and kept as constraint application failures. In particular we can have a deeper decomposition of a matching problem in the ρ_{g}-calculus than in the ρ-calculus and thus it can happen that a ρ-term in normal form can be further reduced in the ρ_{g}-calculus.

Example 44 (Matching failures in ρ-calculus and ρ_g-calculus) *In both calculi, non successful reductions lead to a non solvable match equation in the list of constraints of the term.*

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

Notice that in the ρ-calculus, since the matching algorithm cannot compute a substitution solving the match equation $f(a) \ll f(c)$, the (σ) rule cannot be applied and thus the reduction is stuck. On the other hand, in the ρ_g-calculus the Matching rules *can partially decompose the match equation until the clash $a \ll c$ is reached.*

About conservativity

One can ask whether the other way of the implication in Theorem 14 also holds, that is if a reduction starting from a term without sharing and cycles in the ρ_g-calculus can be always simulated in the ρ-calculus.

One can observe that when the ρ-calculus is expressed in the ρ_g-calculus, a step in the ρ-calculus may be simulated by a sequence of "small" steps in the ρ_g-calculus, in an analogous way in which the λ-calculus is simulated by the λ-calculus with explicit substitutions [ACCL91]. This is due to the nature of the ρ_g-calculus, which by definition is a more concrete calculus than the ρ-calculus. For this reason, we cannot expect every step of the refined system to be reflected in a single step of the ρ-calculus. Concerning ρ_g-rewrite sequences in their whole, it is not difficult to realise that a general conservative result does not hold in this case either. This is not very surprising, since that matching is treated at the object level in the ρ_g-calculus, while this is left at the meta level in the ρ-calculus. This implies that the evaluation of match equations is more fine-grained in the ρ_g-calculus than in the ρ-calculus. As a matter of fact, this leads in the ρ_g-calculus to rewrite sequences in which information obtained from partially solved match equations is possibly used to continue the reduction, as shown in the next example.

Example 45 *The following reduction is stuck in the ρ-calculus after the application of the first ρ-rule. Instead, in the ρ_g-calculus, values obtained from a partially solved match equation can be used to make explicit the second underlined redex and thus continue the reduction.*

$$\begin{array}{ll} & \underline{(f(x,a) \twoheadrightarrow x\ a)\ f(a \twoheadrightarrow c, b)} \\ \mapsto_{\rho} & x\ a[f(x,a) \ll f(a \twoheadrightarrow c, b)] \\ \mapsto_{dk} & x\ a[x \ll a \twoheadrightarrow c, a \ll b] \\ \mapsto_{s} & x\ a[x = a \twoheadrightarrow c, a \ll b] \\ \mapsto_{es,gc} & \underline{(a \twoheadrightarrow c)\ a}[a \ll b] \\ \mapsto\!\!\!\!\!\rightarrow_{fg} & c[a \ll b] \end{array}$$

Looking at the previous example, it is worth noticing that, even if the final ρ_g-term $c[a \ll b]$ and the final ρ-term $x\ a[f(x,a) \ll f(a \twoheadrightarrow c, b)]$ are different, the information about a failure in the computation is present in both of them.

Concerning the behavior of the two calculi in presence of successful computations, we think that a correspondence in the reductions is to be expected.

A first step in this direction shows that the Matching rules faithfully simulate the matching algorithm of the ρ-calculus.

Lemma 28 *Given a matching problem $T \ll U$ where T is an algebraic pattern and U a ρ-term with $\mathcal{V}ar(T) \cap \mathcal{V}ar(U) = \emptyset$, if we have the reduction $T \ll U \mapsto\!\!\!\!\!\rightarrow_{\mathcal{M}} x_1 = U_1, \ldots, x_n = U_n$ in the ρ_g-calculus, then there exists a substitution $\sigma = \{x_1/U_1, \ldots, x_n/U_n\}$ such that $\sigma(T) = U$.*

Proof : By structural induction on the term T.

Basic case: The term T is a variable or a constant. If $T = x$ then $x \ll U \mapsto_s x = U$ and thus trivially $\sigma = \{x/U\}$.
If $T = a$ then $U = a$ and $a \ll a \mapsto_e \epsilon$. Then the solution σ is the empty substitution.

Induction case: $T = f(T_1, \ldots, T_m)$ with $m > 0$.
Since by hypothesis we have a reduction in the ρ_{g}-calculus that leads to a term without matching constraints (successful matching), then necessarily $U = f(V_1, \ldots, V_m)$ and we have the reduction $f(T_1, \ldots, T_m) \ll f(V_1, \ldots, V_m) \ \mapsto_{dk} \ T_1 \ll V_1, \ldots, T_m \ll V_m \ \mapsto\!\!\!\!\!\rightarrow_{\mathcal{M}} \ x_1 \ll U_1, \ldots, x_n \ll U_n$. By induction hypothesis we know that $\sigma(T_i) = V_i$, for all $i = 1, \ldots, n$ and therefore $f(V_1, \ldots, V_m) = f(\sigma(T_1), \ldots, \sigma(T_m)) = \sigma(f(T_1, \ldots, T_m))$.

□

As we have already mentioned, the treatment of constraints in the two calculi is different. In the classical ρ-calculus, the application of a constraint reduces to a term in one step using the (σ)-rule. During this step, some work is hidden at the meta-level of the calculus, namely the solution of the matching and the application of the resulting substitution to a term. Intuitively, in the ρ_{g}-calculus a (σ)-reduction corresponds to a sequence of MATCHING RULES steps, which completely decomposes a matching constraint into recursion equations, and a sequence of GRAPH RULES steps that allow one to apply the substitution obtained from the matching (in the form of a list of recursion equations) to the term. Therefore, in short, we can say that a sequence of steps $\mapsto\!\!\!\!\!\rightarrow_{\mathcal{M}} \mapsto_{\mathcal{G}}$ in the ρ_{g}-calculus, where $\mathcal{G}$ denote the relation induced by the set of the GRAPH RULES, corresponds to an evaluation step $\mapsto_{\sigma}$ of the ρ-calculus. Unfortunately, in a ρ_{g}-reduction the ρ_{g}-rules concerning the same match equation are not necessarily applied one after the other, as exemplified next.

$$
\begin{array}{ll}
 & (y \twoheadrightarrow \ \underline{(f(a) \twoheadrightarrow b) \ f(y)}) \ a \\
\mapsto_{\rho} & (y \twoheadrightarrow \ b[f(a) \ll f(y)]) \ a \\
\mapsto\!\!\!\!\!\rightarrow_{dk} & \underline{(y \twoheadrightarrow \ b[a \ll y]) \ a} \\
\mapsto_{\rho} & b[a \ll y][y \ll a] \\
\mapsto_{s} & b[a \ll y][y = a] \\
\mapsto_{ac,gc} & b[a \ll a] \\
\mapsto\!\!\!\!\!\rightarrow_{dk} & b[\epsilon] \\
\mapsto gc & b
\end{array}
$$

However, this kind of reductions is not really problematic if we restrict to linear patterns, since we can exploit the confluence result of the previous chapter to reorder the ρ_{g}-reduction in such a way that each of the two redexes is evaluated separately from the other, thus obtaining two corresponding $\mapsto_{\rho}\mapsto_{\sigma}$ reductions in the ρ-calculus. In general, to obtain a correspondence with a reduction in the ρ-calculus we need to order the ρ_{g}-reductions using an outer-most strategy. In fact, in the reduction above, one can observe that the inner-most redex needs to wait for a value for the variable y coming from the solution of the outer-most redex.

We do not present here the proof of this conservativity result for successful reductions that will be object of future investigation.

9.2 ρ_{g}-calculus versus cyclic λ-calculus

The ρ_{g}-calculus can be seen as a generalisation of the cyclic λ-calculus in the same way as the ρ-calculus represents an extension of the classical λ-calculus. We present in this section a

translation from the cyclic λ-calculus, also called $\lambda\phi$-calculus (see Section 6.4), into ρ_{g}-terms. Using this translation, we prove a conservativity result with respect to the reductions in the two systems.

The terms of $\lambda\phi$ can be easily translated into terms of the ρ_{g}-calculus having as lists of constraints simply lists of recursion equations.

Definition 92 (Translation) *The translation of a $\lambda\phi$-term t into a ρ_{g}-term, denoted $\overline{t}$, is inductively defined as follows:*

$$\begin{array}{lcl@{\qquad}lcl} \overline{x} & = & x & \overline{f(t_1,\ldots,t_n)} & = & f(\overline{t_1},\ldots,\overline{t_n}) \\ \overline{\lambda x.t} & = & x \twoheadrightarrow \overline{t} & \overline{\langle t_0|\ x_1 = t_1,\ldots,x_n = t_n\rangle} & = & \overline{t_0}[x_1 = \overline{t_1},\ldots,x_n = \overline{t_n}] \\ \overline{t_0\ t_1} & = & \overline{t_0}\ \overline{t_1} & & & \end{array}$$

It is worth noticing that matching constraints are not needed in the $\lambda\phi$-calculus, since the left-hand sides of abstractions are variables and thus the matching is always trivially satisfied.

The translation is used to show how $\lambda\phi$-reductions can be simulated in the ρ_{g}-calculus. One can observe that the evaluation rules of the ρ_{g}-calculus are the generalisation of those of the $\lambda\phi$-calculus. The (β)-rule can be actually simulated using the (ρ)-rule plus the MATCHING RULES of the ρ_{g}-calculus, while the rest of the rules can be simulated using the corresponding rule in the GRAPH RULES of the ρ_{g}-calculus.

Lemma 29 *Let t_1 and t_2 be two $\lambda\phi$-terms. If $t_1 \mapsto_{\lambda\phi} t_2$ in the cyclic λ-calculus, then there exists a reduction $\overline{t_1} \mapsto\!\!\!\!\twoheadrightarrow_{\rho g} \overline{t_2}$ in the ρ_{g}-calculus.*

Proof : We proceed by analysing each reduction rule of $\lambda\phi$.

- (β)-rule. The $\lambda\phi$-reduction is of the form $t_1 = (\lambda x.s_1)\ s_2 \to_\beta \langle s_1|\ x = s_2\rangle = t_2$

  The corresponding reduction in the ρ_{g}-calculus is given by:

  $$\overline{t_1}\ =\ (x \twoheadrightarrow \overline{s_1})\ \overline{s_2}\ \mapsto_\rho\ \overline{s_1}[x \lll \overline{s_2}]\ \mapsto_s\ \overline{s_1}[x = \overline{s_2}]\ =\ \overline{t_2}$$

- (*external sub*) rule: trivial.
- (*acyclic sub*) rule: trivial ($\lll$ stands always for $=$ in this case).
- (*black hole*) rule: trivial.
- (*garbage collect*) rule: The proviso $E \perp (E', t)$ is equivalent to the one expressed using the definition of free variables in the ρ_{g}-calculus. The condition $E' \neq \epsilon$ is implicit in the ρ_{g}-calculus since we eliminate one recursion equation at time. For this reason, a single step of the *garbage collect* rule in $\lambda\phi$ can correspond to several steps of the corresponding *garbage* rule in the ρ_{g}-calculus: if $\langle t|E, E'\rangle \to_{gc} \langle t|E\rangle$ then $\overline{t}[E, E'] \mapsto\!\!\!\!\twoheadrightarrow_{gc} \overline{t}[E]$.

□

Theorem 15 (Completeness) *Let t_1 and t_2 be two $\lambda\phi$-terms. Given a reduction $t_1 \mapsto\!\!\!\!\twoheadrightarrow_{\lambda\phi} t_2$ in the cyclic λ-calculus, then there exists a corresponding reduction $\overline{t_1} \mapsto\!\!\!\!\twoheadrightarrow_{\rho g} \overline{t_2}$ in the ρ_{g}-calculus.*

Proof : Follows from Lemma 29. □

For the cyclic λ-calculus, the converse simulation is easier than in the case of the ρ-calculus, due to the fact that matching is always successful and can be trivially solved in one step using the (*solved*) rule.

It is worth remarking that ρ_{g}-reductions, starting with a ρ_{g}-term coming from the translation of a $\lambda\phi$-term, have a special form. In fact, from the considerations above, we can observe that the MATCHING RULES other than the (*solved*) rule are superfluous. Moreover, since terms do not contain structures, the (δ) rule is not needed either.

We first assume that rules in the ρ_{g}-calculus are applied according to a strategy in which the (*solved*) rule has greater priority than the other rules, meaning that a match equation is solved as soon as possible. We call this strategy *BetaStrat*. Form this result, general soundness will easily follow.

Theorem 16 (Soundness) *Let t_1 and t_2 be two $\lambda\phi_0$-terms. Given a reduction $\overline{t_1} \mapsto\!\!\!\!\!\rightarrow_{\rho g} t'_2$ in the ρ_{g}-calculus using the strategy* BetaStrat*, where t'_2 contains no match equations, then there exists a corresponding reduction $t_1 \mapsto\!\!\!\!\!\rightarrow_{\lambda\phi} t_2$ in the cyclic λ-calculus with $\overline{t_2} = t'_2$.*

Proof : By induction on the length of the reduction. We first notice that, since the initial term $\overline{t_1}$ is the encoding in the ρ_{g}-calculus of a term of the cyclic λ-calculus, it does not contain match equations.

We consider first a reduction of length one. The (ρ)-rule is not considered since the resulting term t'_2 would not respect the hypothesis.

- $\overline{t_1} \mapsto_{es} t'_2$. We have trivially the same reduction in the cyclic λ-calculus.
- $\overline{t_1} \mapsto_{ac} t'_2$. This step is performed using the (*acyclic sub*) rule in which $\lll$ is equal to $=$. Thus, we have trivially the same reduction in the cyclic λ-calculus.
- $\overline{t_1} \mapsto_{bh} t'_2$. The *black hole* rules can be seen as a particular case of the (*external sub*) rule and the (*acyclic sub*) rule.
- $\overline{t_1} \mapsto_{gc} t'_2$. In this case we have a similar reduction in the the $\lambda\phi$ using *garbage collect* rule where the variable E' is instantiated by a single recursion equation.

Let us consider now a reduction of length two.

- If the first step is done using a rule among the ones analysed before, than we can conclude immediately by induction hypothesis.
- If the first rule applied is the (ρ)-rule, then we have the reduction $\overline{t_1} \mapsto_{\rho} t'_1 \mapsto_s t'_2$. The (*solved*) step is the only possibility for the second and last step, since t'_1 contains a match equation and t'_2 does not. Without loss of generality, we can take as initial term the ρ_{g}-term $\overline{t_1} = ((x \twoheadrightarrow \overline{u}_1)[\overline{E}_1])\ \overline{u}_2[\overline{E}_2]$, where $\overline{E}_i = x_1 = \overline{v}^i_1, \ldots, x_n = \overline{v}^i_n$ with v^i_j $\lambda\phi$-terms, for $j = 1, \ldots, n$ and $i = 1, 2$.

  In this case the corresponding reduction in $\lambda\phi$ consists of a step of (β)-rule

$$t_1 = (x \twoheadrightarrow u_1)[E_1]\ u_2[E_2] \mapsto_{\beta} u_1[x = u_2[E_2], E_1] = t_2$$

Let us finally consider a reduction of length greater than two.

- If the first two step are performed using a rule other than the (ρ)-rule, than we can conclude immediately by induction hypothesis.

- If the first step is a (ρ)-step, then, since by hypothesis the reduction satisfies the strategy *BetaStrat*, the second step is necessarily a (*solved*) step. Thus we have

$$((x \twoheadrightarrow \overline{u}_1)[\overline{E}_1])\ \overline{u}_2[\overline{E}_2] \mapsto_\rho \overline{u}_1[x \ll u_2[\overline{E}_2], \overline{E}_1] \mapsto_s u_1[x = \overline{u}_2[\overline{E}_2], \overline{E}_1] \mapsto\!\!\!\!\!\rightarrow_{\not p g} t_2'$$

  The corresponding reduction in the $\lambda\phi$ is given by

$$t_1 = (x \twoheadrightarrow u_1)[E_1]\ u_2[E_2] \mapsto_\beta u_1[x = u_2[E_2], E_1] \mapsto\!\!\!\!\!\rightarrow_{\lambda\phi} t_2$$

  where the last steps $\mapsto\!\!\!\!\!\rightarrow_{\lambda\phi}$ hold by induction hypothesis.

- If the second step is a (ρ)-step, then we conclude similarly, using the hypothesis on the strategy and induction.

□

To prove the general soundness result, we show that for any ρ_{g}-reduction starting and ending with terms free of matching constraints, we can find a reduction using the *BetaStrat* strategy producing the same term.

Lemma 30 *Let t_1 be a $\lambda\phi$-term. Given a reduction $\overline{t_1} \mapsto\!\!\!\!\!\rightarrow_{\not p g} t_n$ in the ρ_{g}-calculus where t_n contains no match equations, then there exists a reduction from $\overline{t_1}$ to t_n using the strategy* BetaStrat.

Proof : If the given reduction does not contain a (ρ)-step, then the thesis follows trivially.

Suppose there exists a (ρ)-step in the reduction. To simplify the notation, we assume that this is the first step of the reduction and that the (ρ)-rule is applied at the top position in the term. The other cases can be treated in a similar way. Hence we have

$$\overline{t_1} = ((x \twoheadrightarrow u_1)[E_1])\ u_2[E_2] \mapsto_\rho u_1[x \ll u_2[E_2], E_1]\ = t_2 \mapsto\!\!\!\!\!\rightarrow_{\not p g} t_n$$

We proceed as follows. We underline the (ρ)-redex and we consider a ρ_{g}-semantics in which we replace the (ρ)-rule and the (*solved*) rule with their underlined version defined as follows

$$\begin{array}{lll}
(\underline{\rho}) & \underline{(G_1 \twoheadrightarrow G_2)\ G_3} & \rightarrow_{\underline{\rho}}\ G_2[G_1 \mathrel{\underline{\ll}} G_3] \\
 & \underline{(G_1 \twoheadrightarrow G_2)[E]\ G_3} & \rightarrow_{\underline{\rho}}\ G_2[G_1 \mathrel{\underline{\ll}} G_3, E] \\
(\underline{solved}) & x \mathrel{\underline{\ll}} G, E & \rightarrow_{\underline{s}}\ x = G, E \quad \textit{if } x \notin \mathcal{DV}(E)
\end{array}$$

Moreover, we add to the semantics an underlined version of the (*acyclic sub*) rule and of the (*black hole*) rule defined in the natural way, in such a way that they can act on underlined matchings only. We consider then the underlined version of the given reduction. We obtain in this way a reduction in which all the steps concerning the match equation generated from the (ρ)-step have been marked. From this one, we built a new reduction obtained by

- inserting a $\underline{s}$-step immediately after the ($\underline{\rho}$)-step,

- transforming the $\mathrel{\underline{\ll}}$ into $=$ in all the terms that follow of the $\underline{s}$-step and eliminating all the $\underline{s}$-steps already present in the reduction.

It is easy to see that the non-underlined version of this new reduction follows the *BetaStrat* strategy and is equivalent to the original reduction, in the sense that it leads to the same final term t_n.

$$((x \twoheadrightarrow u_1)[E_1])\ u_2[E_2] \mapsto_\rho u_1[x \ll u_2[E_2], E_1] \mapsto_s u_1[x = u_2[E_2], E_1] \mapsto\!\!\!\!\twoheadrightarrow_{\not\rho g} t_n$$

Observe that the steps from t_2 to t_n that do not act on the match equation $x \ll u_2[E_2]$ do not change *w.r.t.* the original reduction and that the steps which involve the match equation duplicating it, eliminating it or reducing in its right-hand side can be performed also on the recursion equation $x = u_2[E_2]$.

If the given reduction $\overline{t_1} \mapsto\!\!\!\!\twoheadrightarrow_{\not\rho g} t_n$ contains more than one (ρ)-step, a reduction satisfying the *BetaStrat* strategy can be obtained repeating the explained procedure for each (ρ)-step. □

9.3 Simulation of term graph rewriting into the ρ_g-calculus

The possibility of representing structures with cycles and sharing naturally leads to the question asking whether first-order term graph rewriting can be simulated in this context. In this section we provide a positive answer. For our purposes, we choose the equational description of term graph rewriting defined in Section 6.3, since it is closer to the approach used in the ρ_g-calculus. We recall that a term graph rewrite system $TGR = (\Sigma, \mathcal{R})$ is composed by a signature Σ over which the considered term graphs are built and a set of term graph rewrite rules $\mathcal{R}$. Both the term graphs over Σ and the set of rules are translated at the object level of the ρ_g-calculus, *i.e.* into ρ_g-terms.

Definition 93 *We define for the various components of a TGR their correspondent in the ρ_g-calculus.*

- *(Terms) Using the equational framework, the set of term graphs of a TGR is a strict subset of the set of terms of the ρ_g-calculus, modulo some obvious syntactic conventions. In particular, by abuse of notation, in the following we will sometimes confuse the two notations $\{x \mid E\}$ and $x[E]$.*
- *(Rewrite rules) A rewrite rule $(L, R) \in \mathcal{R}$ is translated into the corresponding ρ_g-term $L \twoheadrightarrow R$. Recall that we consider only left-linear term graph rewrite rules.*
- *(Substitution) A substitution $\sigma = \{x_1/G_1, \ldots x_n/G_n\}$ corresponds in the ρ_g-calculus to a list of constraints $E = (x_1 = G_1, \ldots, x_n = G_n)$ and its application to a term graph L corresponds to the addition of the list of constraints to the ρ_g-term L, i.e. to the ρ_g-term $L[E]$.*

As seen in Section 6.3, it is convenient to work with the restricted class of term graphs in flat form and without useless equations.

The structure of a ρ_g-term can be, in general, more complicated than the one of a flat term graph, since it can have nested lists of constraints and garbage. To recover the similarity in the form of terms, we define next the canonical form of a ρ_g-term G containing no abstractions and no match equations. To reach the canonical form, we perform the flattening and merging of the lists of equations of G and we introduce new recursion equations with fresh variables for every subterm of G. We obtain in this way a ρ_g-term in flat form, where the notion of flat form is defined analogously to the one of term graphs.

Definition 94 (Flattening) *Let G be a ρ_g-term containing no abstractions and no match equations. G can be reduced in flat form using the following set of rules:*

$$\begin{array}{lcll}
\mathsf{Ctx}\{x[E]\}[E'] & \to & \mathsf{Ctx}\{x\}[E', E] & \\
x[y = y'[E], E'] & \to & x[y = y', E, E'] & \\
f(G_1, \ldots, G_n)[E] & \to & x[x = f(G_1, \ldots, G_n), E] & \text{with } x \text{ fresh} \\
x[y = f(G_1, \ldots, G_i, \ldots, G_n), E] & \to & x[y = f(G_1, \ldots, y_i, \ldots, G_n), y_i = G_i, E] & \text{for } i = 1, \ldots n \\
 & & \text{with } G_i \notin \mathcal{X} \text{ and } y_i \text{ fresh} &
\end{array}$$

The canonical form of a ρ_{g}-term G, that will be denoted by $\overline{G}$, can then be obtained from its flat form by removing the useless equations. This can be done using the two substitution rules and the garbage collection rule of the ρ_{g}-calculus.

Definition 95 (Canonical form) *Let G be a ρ_g-term containing no abstractions and no match equations. We say that G is in canonical form if it is in flat form and it contains neither garbage equations nor trivial equations of the form $x = y$.*

It is easy to see that the canonical form of a ρ_{g}-term is unique, up to α-conversion and the AC_ϵ axioms for the constraint conjunction operator, and that a ρ_{g}-term with no abstractions and no match equations in canonical form corresponds to a term graph in flat form.

Before proving the correspondence of rewritings, we need a lemma showing that matching in the ρ_{g}-calculus is well-behaved *w.r.t.* the notion of term graph homomorphism.

Lemma 31 (Matching) *Let G be a closed term graph and let (L, R) be a left-linear rewrite rule, with $\mathcal{V}ar(L) = \{x_1, \ldots, x_m\}$. Assume that there is a rooted homomorphism from L to G, given by the variable renaming $\sigma = \{x_1/x'_1, \ldots, x_m/x'_m\}$.*
Let $E = (x_n = x'_n, \ldots, x_m = x'_m, E_G)$ with $\{x_n, \ldots, x_m\} = \mathcal{FV}(L)$. Then in the ρ_{g}-calculus we have the reduction $L \ll G \mapsto\!\!\!\!\!\rightarrow_{\rho g} E$ with $\tau(\overline{L[E]}) = G$, where τ is a variable renaming.

Proof : We consider functions of arity less or equal two. Note that this is not really a restriction since n-ary functions are encoded in the ρ_{g}-calculus as a sequence of nested binary applications.

Given the matching problem $L \ll G$, where $L = x_1[E_L]$ and $G = x'_1[E_G]$, in the ρ_{g}-calculus we have the reduction

$$L \ll G \;=\; x_1[E_L] \ll x'_1[E_G] \;\mapsto_p\; x_1[E_L] \ll x'_1, E_G \;\mapsto\!\!\!\!\!\rightarrow_{es,gc}\; T_L \ll x'_1, E_G$$

where T_L is a term without constraints, *i.e.*a tree, which can be reached since L is linear and acyclic by hypothesis.

We proceed by induction on the length of the list of recursion equations E_L of the term graph L, or, equivalently, on the height of T_L, seen as tree.

Base case. T_L is a variable x_1. We obtain the reduction $x_1 \ll x'_1, E_G \;\mapsto\!\!\!\!\!\rightarrow_{\mathsf{s}}\; x_1 = x'_1, E_G$.

Then it is immediate to verify that $\overline{L[E]} = \overline{x_1[x_1 = x'_1, E_G]}$ is equal to G using the variable renaming $\tau = \{x_1/x'_1\}$.

Induction. Let G be of the form $G = x'_1[x'_1 = f(x'_2, x'_3), x'_2 = T_2, x'_3 = T_3, E']$. Continuing the reduction of the match equation $L \ll G$ we obtain

$$\begin{array}{rcl}
T_L \ll x'_1, E_G & = & T_L \ll x'_1, x'_1 = f(x'_2, x'_3), \ldots \\
 & \mapsto_{ac} & T_L \ll f(x'_2, x'_3), E_G
\end{array}$$

(a) rewrite rule: (L, R) (b) rewrite step: $G \to G'$

Figure 9.1: Example of term graph rewriting

Since by hypothesis an homomorphism σ between L and G exists, we have $T_L = f(T'_2, T'_3)$ and thus

$$T_L \ll f(x'_2, x'_3), E_G \mapsto_{dk} T'_2 \ll x'_2, T'_3 \ll x'_3, E_G$$

Using the induction hypothesis and the fact that L is acyclic, we obtain the reductions $T'_2 \ll x'_2, E_G \mapsto\!\!\!\!\to_{\not g} E_2$ and $T'_3 \ll x'_3, E_G \mapsto\!\!\!\!\to_{\not g} E_3$ and the variable renamings τ_2 and τ_3 such that $\tau_2(\overline{T'_2[E_2]}) = T_2$ and $\tau_3(\overline{T'_3[E_3]}) = T_3$. Therefore, since $\mathcal{FV}(L) = \mathcal{FV}(T'_2) \cup \mathcal{FV}(T'_3)$, it is easy to verify that $L \ll G \mapsto\!\!\!\!\to_{\not g} E$, with $E = E_2, E_3, E_G$ and that the variable renaming $\tau = \tau_2\tau_3\{x_1/x'_1\}$ is such that $\tau(\overline{L[E]}) = G$. □

The previous lemma guarantees the fact that if there exists an homomorphism relation (represented as a variable renaming) between two term graphs, in the ρ_{g}-calculus we obtain the variable renaming (in the form of a list of recursion equations) as result of the evaluation of the matching problem generated from the two graphs. In other words, this means that if a rewrite rule can be applied to a term graph, the application is still possible after the translation of the rule and the term graph in the ρ_{g}-calculus.

Example 46 (Matching) *Consider the term graphs* $L = \{x_1 \mid x_1 = add(x_2, y_1), x_2 = s(y_2)\}$ *and* $G = \{z_0 \mid z_0 = add(z_1, z_2),\ z_1 = s(z_2), z_2 = 0\}$ *and the homomorphism from* L *to* G $\sigma = \{x_1/z_0,\ x_2/z_1,\ y_1/z_2,\ y_2/z_2\}$. *We show how the substitution* σ *can be obtained in the* ρ_{g}*-calculus starting from the matching problem* $L \ll G$.

$$\begin{array}{lll}
L \ll G & \mapsto_{p} & L \ll z_0, E_G \\
& \mapsto\!\!\!\!\to_{s,gc} & add(s(y_2), y_1) \ll z_0, E_G \\
& = & add(s(y_2), y_1) \ll z_0, z_0 = add(z_1, z_2), z_1 = s(z_2), z_2 = 0 \\
& \mapsto_{ac} & add(s(y_2), y_1) \ll add(z_1, z_2), z_0 = add(z_1, z_2), z_1 = s(z_2), z_2 = 0 \\
& \mapsto_{dk} & s(y_2) \ll z_1, y_1 \ll z_2, z_0 = add(z_1, z_2), z_1 = s(z_2), z_2 = 0 \\
& \mapsto\!\!\!\!\to_{ac,dk} & y_2 \ll z_2, y_1 \ll z_2, z_0 = add(z_1, z_2), z_1 = s(z_2), z_2 = 0 \\
& \mapsto\!\!\!\!\to_{s} & y_2 = z_2, y_1 = z_2, E_G
\end{array}$$

We can verify then that $\overline{L[y_2 = z_2, y_1 = z_2, E_G]}$ *is equal to* G *up to variable renaming. In fact, the transformation into the canonical form leads to the graph* $x_1[x_1 = add(x_2, z_2), x_2 = s(z_2), z_2 = 0]$ *and it is easy to see that the variable renaming* $\tau = \{x_1/z_0, x_2/z_1\}$ *makes this graph equal to* G.

We analyse next the relationship between the derivations of a term graph rewrite system $TGR = (\Sigma, \mathcal{R})$ and the reductions in the ρ_{g}-calculus. Given a term graph G in the TGR and its derivation *w.r.t.* the set of rules $\mathcal{R}$, we build from this derivation a ρ_{g}-term having a reduction in the ρ_{g}-calculus ending with the same term as the original reduction in the TGR.

Notice that, since in the ρ_{g}-calculus the rule application is at the object level, we need to define a *position trace* ρ_{g}*-term* H encoding the position of the redex in the given term graph G.

An similar notion of derivation trace term was introduced in Section 4.3 for terms of the standard ρ-calculus.

Lemma 32 (Simulation) *Let G be a ground term graph, let (L, R) be a left-linear rewrite rule rooted at z and let σ be an homomorphism from L to G such that $G_{\lceil\sigma(z)=t\rceil_p} \to G_{\lceil\sigma(R)\rceil_p}$ with $\sigma(z)$ not in a cycle, i.e. $\nexists y \in G$ such that $\sigma(z) \equiv y$.*

Define the ρ_{g}-term $H = G_{\lceil\sigma(z)=x\rceil_p} \twoheadrightarrow G_{\lceil\sigma(z)=(L\twoheadrightarrow R)\ x\rceil_p}$. Then in the ρ_{g}-calculus there exists a reduction $(H\ G) \mapsto\!\!\!\!\!\!\twoheadrightarrow_{\rho g} G'$ and a variable renaming τ such that $\tau(\overline{G'})$ is equal to $G_{\lceil\sigma(R)\rceil_p}$.

Proof : First of all, observe that by definition $G_{\lceil\sigma(z)=x\rceil_p}$ matches G. If the corresponding homomorphism is $\sigma' = \{x/x'\}$, by Lemma 31 after the resolution of the matching we obtain in the ρ_{g}-calculus a list of recursion equations $x = x', E_G$. We will denote such list simply by E'_G.

In the ρ_{g}-calculus we obtain the following reduction:

$$
\begin{array}{ll}
 & G_{\lceil\sigma(z)=x\rceil_p} \twoheadrightarrow G_{\lceil\sigma(z)=(L\twoheadrightarrow R)\ x\rceil_p}\ G \\
\mapsto_{\rho} & G_{\lceil\sigma(z)=(L\twoheadrightarrow R)\ x\rceil_p}[G_{\lceil\sigma(z)=x\rceil_p} \ll G] \\
\mapsto\!\!\!\!\!\!\twoheadrightarrow_{\rho g} & G_{\lceil\sigma(z)=(L\twoheadrightarrow R)\ x\rceil_p}[E'_G] \quad \textit{by Lemma 31} \\
\mapsto_{es} & G_{\lceil\sigma(z)=(L\twoheadrightarrow R)\ x'\rceil_p}[E'_G] \\
\mapsto_{\rho} & G_{\lceil\sigma(z)=R[L\ll x']\rceil_p}[E'_G] \\
\mapsto\!\!\!\!\!\!\twoheadrightarrow_{\rho g} & G_{\lceil\sigma(z)=R[y_1=y'_1,\ldots,y_n=y'_n]\rceil_p}[E'_G] \quad \textit{by Lemma 31} \\
\mapsto\!\!\!\!\!\!\twoheadrightarrow_{es,gc} & G_{\lceil R'\rceil_p}[E'_G] \ = \ G'_1
\end{array}
$$

where $\{y_1, \ldots, y_n\} = \mathcal{FV}(R)$ and R' is the term obtained from R by renaming y_i with y'_i, for $i = 1 \ldots n$. By using Lemma 31, it is not difficult to deduce that R' is equal to $\sigma(R)$ modulo α-conversion. We conclude by noticing that (the flat form of) $G_{\lceil\sigma(R)\rceil_p}[E'_G]$ equal up to variable renaming to $G_{\lceil\sigma(R)\rceil_p}$. □

If the head of the redex in G is inside a cycle, in Figure 9.3 we have illustrated a counter example for this result.

Notice that we could have separated the rule from its application position information in the ρ_{g}-term H by choosing $H = y \twoheadrightarrow (G_{\lceil\sigma(z)=x\rceil_p} \twoheadrightarrow G_{\lceil\sigma(z)=y\ x\rceil_p})$ and then by considering $H\ (L \twoheadrightarrow R)\ G$ as initial term of the reduction.

We point out that, in the proof of the previous lemma, starting with a ρ_{g}-term equal up to variable renaming to G, say G', we could have constructed a reduction in the ρ_{g}-calculus. Indeed in this case using the same reasoning above, we obtain as final term $G'_1 = G'_{\lceil R'\rceil_p}[E'_G]$ whose flat form is equal up to variable renaming to $G'_{\lceil\sigma(R)\rceil_p}$ which is in turn equal up to variable renaming to $G_{\lceil\sigma(R)\rceil_p}$ and thus the lemma still holds.

Corollary 2 *Given a term graph G and a left-linear rewrite rule (L, R) such that $G_{[\sigma(z)=t]_p} \to G_{\lceil\sigma(R)\rceil_p} = G_1$, with $\sigma(z)$ not in a cycle, then there exists a ρ_{g}-term H and a ρ_{g}-reduction $(H\ G') \mapsto\!\!\!\!\!\!\twoheadrightarrow_{\rho g} G'_1$ such that any term G' (whose flat form is) equal up to variable renaming to G reduces to a term G'_1 (whose flat form is) equal up to variable renaming to G_1.*

The final ρ_{g}-term we obtain is not exactly the same as the term graph resulting from the ρ_{g}-reduction in the TGR, and this is due to some unsharing steps that may occur in the reduction. In general, we have an homomorphism relation between the two graphs in presence of cycles, the ρ_{g}-term is possibly more "unravelled" than the term graph G'.

Example 47 (Addition) *Let (L, R), where $L = x_1\{x_1 = add(x_2, y_1),\ x_2 = s(y_2)\}$ and $R = x_1\{x_1 = s(x_2),\ x_2 = add(y_1, y_2)\}$, be a term graph rewrite rule describing the addition of natural numbers. We apply this rule to the term graph $G = z\{z = s(z_0), z_0 = add(z_1, z_2), z_1 = s(z_2), z_2 = 0\}$ using the variable renaming $\sigma = \{x_1/z_0, x_2/z_1, y_1/z_2, y_2/z_2\}$. We obtain thus the term graph $G' = z\{z = s(z_0), z_0 = s(z'_1), z'_1 = add(z_2, z_2), z_2 = 0\}$. For a graphical representation see Figure 9.1.*

The corresponding reduction in the ρ_{g}-calculus is as follows. First of all, since the rule is not applied at the head position of G, we need to define the ρ_{g}-term $H = s(x) \twoheadrightarrow s((L \twoheadrightarrow R)\ x)$ that pushes down the rewrite rule to the right application position, i.e. under the symbol s. Then applying the ρ_{g}-term H to G we obtain the following reduction:

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

The canonical form of G'' is then obtained removing the useless recursion equations in E_G and merging the lists of constraints. We get the graph $\overline{G''} = x[x = s(x_1), x_1 = s(x_2), x_2 = add(z_1, z_2), z_0 = 0]$ which is equal up to variable renaming to the term graph G'.

Theorem 17 (Completeness) *Given a n step reduction $G \mapsto^n G_n$ in a TGR, such that the heads of the n redexes are not in a cycle, then there exist n ρ_{g}-terms $H_1, \ldots, H_n$ such that $(H_n\ \ldots\ (H_1\ G)) \mapsto_{\not p g} G'_n$ and there exists a variable renaming τ such that $\tau(\overline{G'_n}) = G_n$.*

Proof : By induction on the length of the reduction, using Lemma 32 and Corollary 2.

About conservativity

We are interested in determining whether a reduction in a TGR can be found, given a reduction in the ρ_{g}-calculus defined on suitable terms. As already done for the ρ-calculus, a first important lemma to prove is the one concerning the relationship between the set of the MATCHING RULES of the ρ_{g}-calculus and the notion of homomorphism in TGRs. In the ρ_{g}-calculus, in order to solve a matching problem, before proceeding to its decomposition in simpler problems, a complete unsharing of the left-hand side of the rewrite rule is performed. This has some consequences on the meaning of the matching algorithm defined by the MATCHING RULES. We showed in Lemma 31 that for any term graph G homomorphic to G' the matching $G \ll G'$ is successful in the ρ_{g}-calculus, but the converse does not hold in general. In fact, it can be shown that if $G \ll G'$ is successful in the ρ_{g}-calculus, then there exists a substitution σ such that $\sigma(G)$ and G' are bisimilar although there could be no homomorphism from G to G'. This is understood by observing that in the ρ_{g}-calculus we actually always match the *tree* corresponding to the term graph G, to the term G'. As an example, take the term graph $f(a, a)$ which flat form is $G = \{x \mid x = f(y, z), y = a, z = a\}$ and the same term where the constant a is shared $G' = \{x' \mid x' = f(y', y'), y' = a\}$. There exists no variable renaming that can make G' homomorphic to G. However we obtain in the ρ_{g}-calculus the following successful reduction:

$$
\begin{array}{ll}
 & G' \ll G \\
= & x'[x' = f(y', y'), y' = a] \ll x[x = f(y, z), y = a, z = a] \\
\mapsto\!\!\!\!\rightarrow_{es,gc} & f(a, a) \ll x[x = f(y, z), y = a, z = a] \\
\mapsto\!\!\!\!\rightarrow_{\not pg} & a \ll a, x = f(y, z), y = a, z = a \quad (\textit{by idempotency}) \\
\mapsto_{dk} & x = f(y, z), y = a, z = a
\end{array}
$$

To capture homomorphism rather than bisimilarity, we would need to change the MATCHING RULES in order to define a more fine-grained matching algorithm. However, since we work under the (common) hypothesis of left-linearity for the rewrite rules, implying that the left-hand sides are trees, the two notions collapse. Consequently, the difference we stressed here is not relevant any more, since the left-hand sides of the rules are already trees by definition.

Lemma 33 *Let L be a linear term graph and G be a closed term graph. If in the ρ_{g}-calculus we have the reduction $L \ll G \mapsto\!\!\!\!\rightarrow_{\not pg} E$ with E a list of recursion equations, then the term graph L is homomorphic to the term graph G.*

Proof : By induction on the number of recursion equation of L. Using a proof scheme similar to that of Lemma 31, it is not difficult to show that if $L \ll G \mapsto\!\!\!\!\rightarrow_{\not pg} x_n = x'_n, \ldots, x_m = x'_m, E_G$ and $\tau(\overline{L[E]}) = G$ for a certain variable renaming τ, then the variable renaming $\sigma = \{x_n/x'_n, \ldots, x_m/x'_m\}\tau$ is an homomorphism between L and G. □

Using this lemma we can prove the soundness of our calculus *w.r.t.* linear TGRs.

Lemma 34 *Let $L \twoheadrightarrow R$ be the ρ_{g}-term corresponding to the left-linear term graph rewrite rule (L, R). If $(L \twoheadrightarrow R)\ G \mapsto\!\!\!\!\rightarrow_{\not pg} G'$, with G an acyclic ρ_{g}-term and G' a ρ_{g}-term corresponding to term graphs, then there exists a reduction $G \rightarrow G_1$ in the corresponding TGR, obtained by applying the rewrite rule (L, R) at the top position in G, with the term graph G_1 equal to $\tau(\overline{G'})$, where τ is a variable renaming.*

Proof : By hypothesis we know that the redex $(L \twoheadrightarrow R)\ G$ generated a matching problem $L \ll G$ which is successful. By Lemma 33, this implies that the two term graphs L and G are homomorphic and thus in the TGR there exists a reduction of the term graph G using the rewrite rule (L, R) at the top position. Let G_1 be the result of this reduction. By Lemma 32, there exists a corresponding reduction in the ρ_{g}-calculus such that $(L \twoheadrightarrow R)\ G \mapsto\!\!\!\!\rightarrow_{\not pg} G_2$ where $\overline{G_2}$ is equal up to variable renaming to G_1. Thus we have in the ρ_{g}-calculus two reductions $(L \twoheadrightarrow R)\ G \mapsto\!\!\!\!\rightarrow_{\not pg} G'$ and $(L \twoheadrightarrow R)\ G \mapsto\!\!\!\!\rightarrow_{\not pg} G_2$. By the confluence property of the linear ρ_{g}-calculus we can conclude that the two terms G_2 and G' reduce to two terms G_3 and G'_3 equivalent modulo AC_ϵ. Since the terms G_2 and G' are algebraic terms with associated a list of constraint containing only recursion equations, the only rules that can be applied in the reduction from G_2 to G_3, G' to G'_3 respectively, are substitutions that perform some unsharing in the term or garbage collection rules. For this reason, we can conclude that the flat form of G_2 is equal to the flat form of G_3, up to variable renaming, and similarly for G' and G'_3. Moreover, since G_3 and G'_3 are equivalent modulo AC_ϵ, their graphical structure is the same and thus any homomorphism (variable renaming) is preserved from G_3 to G'_3. From these observations, together with the fact that $\overline{G_2}$ is equal up to variable renaming to G_1, we conclude that $\overline{G'}$ is equal up to variable renaming to G_1. □

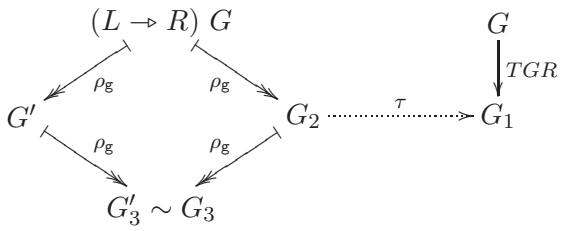


Figure 9.2: Soundness

The previous lemma generalises easily to ρ_{g}-calculus reductions where the rule $L \twoheadrightarrow R$ is applied at a deeper position in G.

Theorem 18 (Soundness) *Let $L \twoheadrightarrow R$ be the ρ_{g}-term corresponding to the left-linear term graph rewrite rule (L, R). Let G_1 be an acyclic ρ_{g}-term, G_0 and G'_1 be ρ_{g}-terms corresponding to term graphs. Assume that $G_{1\lceil L \twoheadrightarrow R\ G_0 \rceil_p} \mapsto\!\!\!\!\!\rightarrow_{\rho g} G'_1$. Then, if we define the term graph $G_2 = G_{1\lceil G_0 \rceil_p}$, there exists a reduction $G_2 \to G'_2$ obtained by applying the rewrite rule (L, R) at the position p in G_2, such that $\tau(\overline{G'_1})$ is equal to G'_2 for a suitable variable renaming τ.*

Proof : (**Sketch**) We first show the thesis for a particular ρ_{g}-calculus reduction and then we conclude for all the existing reductions $G_{1\lceil L \twoheadrightarrow R\ G_0 \rceil_p} \mapsto\!\!\!\!\!\rightarrow_{\rho g} G'_1$.

Since G'_1 corresponds to a term graph by hypothesis, G'_1 cannot contain matching constraints and thus the matching problem $L \ll G_0$ is successful. Consider the ρ_{g}-calculus reduction $G_{1\lceil L \twoheadrightarrow R\ G_0 \rceil_p} \mapsto_{\rho} G_{1\lceil R[L \ll G_0]\rceil_p} \mapsto\!\!\!\!\!\rightarrow_{\mathcal{M}} G_{1\lceil R[E]\rceil_p}$ where E is a list of recursion equations. If we call the term $R[E]$ simply G'_0, we have for this reduction $G'_1 = G_{1\lceil G'_0 \rceil_p}$.

By Lemma 34 we known that in the TGR there exists a reduction $G_0 \to G''_0$ and a variable renaming τ_1 such that $\tau_1(\overline{G'_0}) = G''_0$. By context closure, we obtain the reduction $G_{1\lceil G_0 \rceil_p} \to G_{1\lceil G''_0 \rceil_p}$ with $\tau_1(\overline{G_{1\lceil G'_0 \rceil_p}}) = G_{1\lceil G''_0 \rceil_p}$. Hence, we can conclude that the thesis holds for this particular ρ_{g}-calculus reduction.

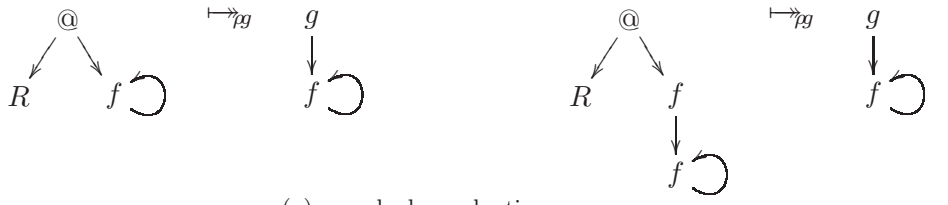
To prove the thesis in its generality, we use the confluence property of the linear ρ_{g}-calculus. Let G'_1 be the final term of the ρ_{g}-calculus reduction. By the confluence property, G'_1 and $G_{1\lceil G'_0 \rceil_p}$ reduce to two terms equivalent modulo AC_ϵ. Moreover, since G'_1 and $G_{1\lceil G'_0 \rceil_p}$ correspond to term graphs, the only rules that can be applied to continue the reduction are substitutions rules or garbage collection rules. Therefore, we conclude that the flat form of G'_1 and $G_{1\lceil G'_0 \rceil_p}$ are equal modulo a variable renaming τ_2. Hence we have $\tau_2(\overline{G'_1}) = \overline{G_{1\lceil G'_0 \rceil_p}}$ and consequently the variable renaming $\tau = \tau_1\tau_2$ is such that $\tau(\overline{G'_1}) = G'_2$.

□

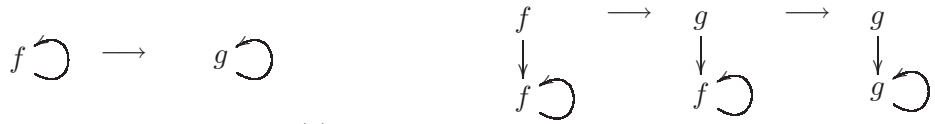
The soundness result holds for reductions in the ρ_{g}-calculus of a single redex of the form $(L \twoheadrightarrow R)\ G$, where L, R and G are the corresponding of term graphs, G is acyclic and L is acyclic and linear. A counter example of this result in the case of a cyclic ρ_{g}-term G is given next.

Example 48 *Consider the ρ_{g}-term $R = f(x) \twoheadrightarrow g(x)$ corresponding to the term graph rewrite rule $(\{x_1 \mid x_1 = f(x)\}, \{x_1 \mid x_1 = g(x)\})$.*

Given the cyclic ρ_{g}-term $G = y[y = f(y)]$, we have the following reduction:



(a) $\rho_{\mathtt{g}}$-calculus reductions



(b) TGR reductions using R

Figure 9.3: Counterexample for cyclic terms

$$
\begin{array}{ll}
 & f(x) \twoheadrightarrow g(x)\ y[y = f(y)] \\
\mapsto_{\rho g} & g(x)[f(x) \ll y, y = f(y)] \\
\mapsto_{\rho g} & g(x)[x = y, y = f(y)] \\
\mapsto_{\rho g} & g(y)[y = f(y)] = G_1
\end{array}
$$

Notice that only the top "f" symbol is rewritten to "g", while the rest of the term remains unchanged.

In a TGR, applying once the given rule to the term graph G we have instead the reduction $G = \{y \mid y = f(y)\} \rightarrow \{y \mid g(y)\} = G_2$, where all the "f" symbols are reduced simultaneously.

This different behaviour in the two formalisms may seem strange at the first look. However, the behaviour of the $\rho_{\mathtt{g}}$-calculus is coherent with the notion of term graphs considered modulo bisimulation and it is necessary to maintain the confluence of the calculus. If we take the $\rho_{\mathtt{g}}$-term $G' = f(y)[y = f(y)]$ which is bisimilar to G and can be obtained from G with a step of external substitution, then the $\rho_{\mathtt{g}}$-term $f(x) \twoheadrightarrow g(x)\ G'$ reduces again to G_1 (see Figure 9.3 (a)).

In the TGR, we would need to apply twice to G' the given rewrite rule in order to obtain a term graph bisimilar to G_2 (see Figure 9.3 (b)). This would not be possible in the $\rho_{\mathtt{g}}$-calculus where the application of a rule consumes the rule itself.

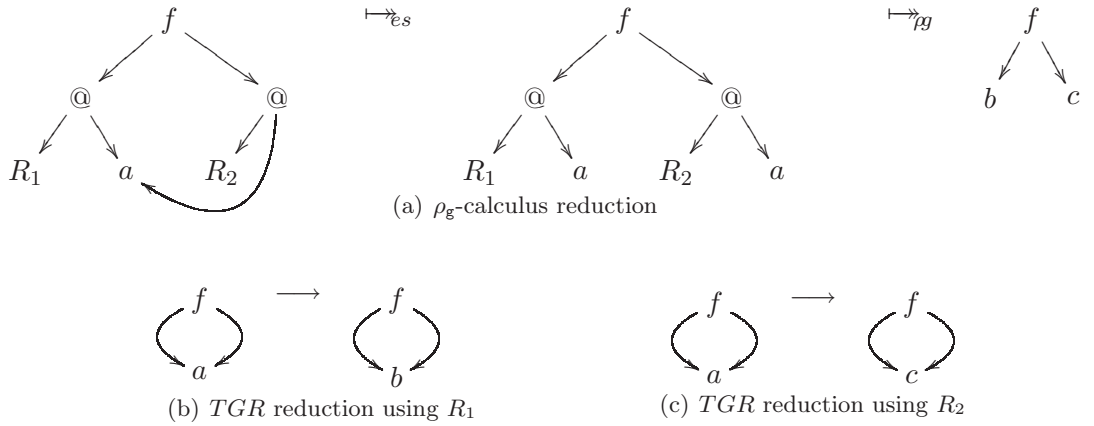
Unlike one could expect, general soundness does not follow from the previous theorem. Reductions of an arbitrary length in the $\rho_{\mathtt{g}}$-calculus do not always have a counterpart in a TGR. This is due to the unfolding steps that may occur in the reduction of a $\rho_{\mathtt{g}}$-term, as a result of the application of the substitution rules to the term. Analogously to what happens in the relation between term rewriting and term graph rewriting, not every result obtainable in the $\rho_{\mathtt{g}}$-calculus is also obtainable by term graph rewriting. We show next an example.

Example 49 *Consider the $\rho_{\mathtt{g}}$-terms $R_1 = a \twoheadrightarrow b$ and $R_2 = a \twoheadrightarrow c$ corresponding to the term graph rewrite rules $(\{x_1 \mid x_1 = a\}, \{x_1 \mid x_1 = b\})$ and $(\{x_2 \mid x_2 = a\}, \{x_2 \mid x_2 = c\})$.*

Given the $\rho_{\mathtt{g}}$-term $f((a \twoheadrightarrow b)x, (a \twoheadrightarrow c)x)[x = a]$, we have the reduction (see Figure 9.4):

$$
\begin{array}{ll}
 & f((a \twoheadrightarrow b)x, (a \twoheadrightarrow c)x)[x = a] \\
\mapsto_{es} & f((a \twoheadrightarrow b)a, (a \twoheadrightarrow c)a)[x = a] \\
\mapsto_{\rho g} & f(b, c)
\end{array}
$$

If we consider the term graph $\{y \mid y = f(x, x), x = a\}$ and we apply to it the term graph rewrite rules R_1 and R_2, we obtain either the reduction $\{y \mid y = f(x, x), x = a\} \rightarrow \{y \mid y = f(x, x), x =$



(a) ρ_g-calculus reduction

(b) TGR reduction using R_1

(c) TGR reduction using R_2

Figure 9.4: Counterexample to general soundness

$b\}$, *using the rule* R_1, *or the reduction* $\{y \mid y = f(x,x), x = a\} \to \{y \mid y = f(x,x), x = c\}$, *using the rule* R_2. *There exists no rewriting sequence leading to a final term in which the two argument of the symbol "*f*" are different.*

This can be explained by the fact that in the ρ_g*-calculus the element* a *is shared in the initial* ρ_g*-term, but the* (es)*-step produces two copies of* a *and each of them can be rewritten separately. This is not possible in the* TGR*, where the sharing of the subgraph* a *forces the synchronised rewriting of the node.*

A positive answer to general soundness can most likely be provided comparing ρ_g-calculus and left-linear TGR with *copying* [Plu99].

Conclusion

The ρ_g-calculus is an expressive formalism that has been shown to be a generalisation of both the plain ρ-calculus and the λ-calculus extended with explicit recursion, providing an homogeneous framework for pattern matching and higher-order graphical structures.

We have analysed in particular the relationship between ρ-calculus reductions and ρ_g-calculus reductions starting from ρ_g-terms that are simply trees. The explicit treatment of matching constraints in the ρ_g-calculus makes this correspondence valid in both directions only for successful computations.

We have defined then a translation from λ-graphs into ρ_g-terms and we have shown that reductions of corresponding terms are equivalent in the two calculi.

The ρ_g-calculus can be naturally seen also as an extension of term-graph rewriting. We have presented a method to build a ρ_g-term with a reduction similar to the one of a term graph, with respect to a given set of term graph rules.

These results have been published in [BBCK05, Ber05].

Conclusions et perspectives

The work developed in this thesis is dedicated, on the one hand, to a better understanding of the relationship between the rewriting calculus and other higher-order rewrite formalisms and, on the other hand, to an extension of the calculus with sharing and cyclic features aiming to an increased expressivity and more efficient implementations.

ρ-calculus and higher-order rewriting

The rewriting calculus is known to be an expressive higher-order framework, able to simulate the λ-calculus, term rewriting and object calculi; nevertheless so far no results existed about its relationship with higher-order term rewriting. This thesis provides a formal answer to this subject by comparing the rewriting calculus with the higher-order rewrite systems called *Combinatory Reduction Systems* (CRSs).

This work of comparison between the two formalisms allowed us to better understand the behaviour of both of them. In particular, it led us to consider an aspect of CRSs which had not been previously investigated, that is matching in CRSs. We have proposed in this thesis a formal definition of the notion of CRS matching problem and characterised all its solutions. We have done this by showing that a CRS matching problem is equivalent to a higher-order matching problem on λ-terms, in the sense that the two problems have the same solution, modulo the translation. General higher-order matching is a difficult issue, but CRS patterns are translated into λ-patterns *à la* Miller, and thus we obtain the decidability and the uniqueness of the solution of CRS pattern matching as a consequence of the well-known results on λ-calculus [Mil91].

The comparison between the ρ-calculus and CRSs has been formalised by defining a translation function between the two formalisms and by establishing a precise relationship between CRS and ρ-calculus rewrite reductions. We have shown that any CRS derivation can be conveniently encoded in the ρ-calculus. More precisely, we have proved that, given a CRS-term that reduces to another CRS-term using a set of CRS-rules, we can automatically build an appropriate rewriting calculus term from the CRS derivation. Any evaluation of this ρ-calculus term terminates and the results is the translation of the final CRS-term.

The rewriting calculus term used for encoding CRS derivations exploits information provided by the initial CRS-term, the set of CRS-rules and the reduction of this term *w.r.t.* this set of rules. It would be certainly interesting to build a similar ρ-term using just the initial CRS-term and the CRS-rules. The development of such a method needs the definition of iteration strategies and of strategies for the generic traversal of terms. It has been shown [CLW03] that one can build ρ-terms describing the application of first-order term rewriting systems *w.r.t.* a given reduction strategy like, for example, innermost or outermost. Intuitively, the ρ-term encoding a first-order rewrite system is a ρ-structure consisting of the corresponding rewrite rules wrapped in an iterator that allows for the repetitive application of the rules. We conjecture that this approach

based on (typed object oriented flavored) fix-points can be applied here for the construction of an appropriate ρ-term only from the set of CRS-rules. Then, the application of the obtained ρ-term to a given term encodes the reduction of this term *w.r.t.* the set of CRS-rules guided by a given reduction strategy. Nevertheless the application of this technique for the encoding of a first-order rewrite system is not directly applicable in the case of a higher-order rewrite system, since during the iteration rewrite rules are applied to a term from the top level down to its leafs, passing through the binders possibly present in the term. Therefore, the method of destructuring terms requires manipulations on the bound variables in order to make the process correct.

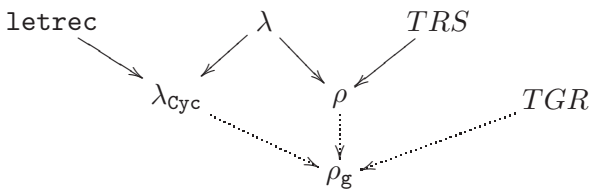
Since we are mainly interested in the expressive power of the ρ-calculus we have proposed a translation from CRS to ρ-calculus but the translation the other way around has not been explicitly defined here. We believe such a translation is possible but rather complex since the explicit control of rewrite rules in ρ-calculus should be somehow simulated in the corresponding CRS rewrite system.

We have considered in this thesis CRSs satisfying the pattern condition. It would be interesting to understand if the presented results can be extended to more general CRSs, namely if the correspondence between CRS-reductions and reductions in an appropriate version of ρ-calculus can be defined similarly. This is not a trivial generalisation of our results, since higher-order matching, used in the CRSs, is undecidable when no restrictions are imposed on patterns.

ρ-calculus with sharing and cyclic features

Besides these results concerning the ρ-calculus and higher-order rewriting, the main achievement of the second part of the thesis is the extension of the ρ-calculus with graph-like structures. This new calculus, called $\rho_{\mathtt{g}}$-calculus, handles terms with sharing and cyclic features that are transformed using a convenient set of evaluation rules.

The ρ-calculus was designed to fully integrate term rewriting (TRS) and classical λ-calculus. Similarly, we have shown in this thesis that the $\rho_{\mathtt{g}}$-calculus is expressive enough to simulate term graph rewriting (TGR) and λ-calculus with explicit recursion (modelled using a `letrec` like construct). Informally, we can organise the considered formalisms in a diagram as showed below:



The $\rho_{\mathtt{g}}$-calculus is thus a formalism which can take advantage of multiple contributions from different calculi and make these different features cohabit together. This approach led to a simple calculus that provides a uniform framework for graph-like structures, explicit matching and higher-order features.

An important achievement of the thesis is the proof of confluence of the calculus, which can be obtained under some linearity restrictions on patterns. Since different $\rho_{\mathtt{g}}$-calculus terms representing the same graphical structure are grouped into equivalence classes, the proof of confluence is done for the rewrite relation acting on these classes. The confluence result is obtained adapting the usual techniques for confluence of term rewrite systems to terms with constraints. Moreover, since the $\rho_{\mathtt{g}}$-calculus rewrite relation is not terminating, the "finite development method" of the

classical λ-calculus together with several properties of rewriting modulo a set of equations are needed to obtain the final result, making thus the complete proof quite elaborated.

The ρ_g-calculus, is closer to concrete implementations of rewrite-based languages, due to its capability to model sharing and cycles. An implementation of the ρ_g-calculus has not been realised yet and represents certainly an interesting issue for future work. Starting from the ongoing studies on the implementation of the explicit version of the ρ-calculus, for which an experimental interpreter in the support language TOM [MRV03] is already available [Fau05], we can naturally extend this implementation to include the new features of the the ρ_g-calculus. TOM is a pattern matching compiler particularly well-suited for describing algebraic transformations on terms and XML based documents. Following the experience of ELAN, a rewrite-based language whose semantics is provided by the ρ-calculus, TOM provides a framework integrating rewriting into existing programming environments like C, Java, and Caml and is thus well-suited for programming by pattern matching and implementing rule based systems.

An interesting topic of future research is a version of the ρ_g-calculus where the current restrictions on patterns are weakened. For example, in the thesis we require the left-hand sides of match-equations to be acyclic, *i.e.*, intuitively, to represent only finite terms. This is a common assumption, when term graph rewriting is thought of as an implementation of term rewriting. At the same time, an appealing problem is the generalization of ρ_g-calculus to deal with different, non syntactic, matching theories. General cyclic matching, that is matching involving cyclic left-hand sides, could be useful, for example, for the modeling of reactions on cyclic molecules or transformations on distribution nets. Cyclic matching can be properly formalised as a relation between graphs called *bisimilarity*, which intuitively can be understood as the equivalence of the infinite terms obtained by unravelling the two graphs. A ρ_g-calculus semantics adapted to conveniently handle this kind of matching computations is currently under study.

Furthermore, since a term of the ρ_g-calculus can be seen as a "compact" representation of a possibly infinite ρ-calculus term, it would be interesting to define an infinitary version of the ρ-calculus, taking inspiration, *e.g.* from the work on the infinitary λ-calculus [KKSd97] and on infinitary rewriting [KKSdV91, Cor93]. Intuitively, a single reduction in a cyclic ρ_g-calculus graph can correspond to the reduction of infinitely many redexes in the corresponding infinite term. Therefore, a precise relationship between the derivations in the ρ_g-calculus and the infinitary version of the ρ-calculus, respectively, should be investigated in order to have an adequacy result in the style of [KKSd94, CD97] and thus enforce the view of the ρ_g-calculus as efficient implementation of infinite terms and their rewriting.

The expressivity of the ρ_g-calculus offers a broad spectrum of applications for the calculus, as communication network applications and semantic web applications.

For example, the telecom network can be modelled by a graph whose configuration changes locally, according to the signals emitted and received by its nodes. The notion of local transformations is a characteristic typical of the ρ_g-calculus, where the explicit treatment of application and matching allow for the description of local rewriting.

On the other hand, the ρ_g-calculus may be used for describing the semantics and interoperability of web services. The list of constraints associated to terms and the hierarchical organisation of these lists can be used for representing information about resources in the World Wide Web, and thus the ρ_g-calculus can be seen as a starting point for a language in the style of RDF [MSB, HBEV04].

Another interesting domain can be the bio informatics and particularly the chemical-bio informatics. Reaction mechanisms between molecules and their results can be naturally modeled

using a term graph framework as the $\rho_{\mathtt{g}}$-calculus, where molecules can be represented by cyclic graphs and chemical reaction can be modeled as rules. Chains of reactions on a molecule become then rewrite reductions on the term representing the initial molecule. A first-step in this direction was made in [Iba04]. Cyclic molecules are encoded as special terms equipped with labels on their edges in order to keep trace of cycles, and chemical reactions correspond to rewriting sequences on terms. It would be interesting to understand if, along these lines, a simpler approach can be provided by the $\rho_{\mathtt{g}}$-calculus, which support cyclic terms and their evaluation.

Bibliography

[ACCL91] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Levy. Explicit substitutions. *Journal of Functional Programming*, 4(1):375–416, 1991.

[Acz78] P. Aczel. A general Church-Rosser theorem. Technical report, University of Manchester, 1978.

[AK96a] Zena M. Ariola and Jan W. Klop. Equational term graph rewriting. *Fundam. Inf.*, 26(3-4):207–240, 1996.

[AK96b] Zena M. Ariola and Jan W. Klop. Lambda calculus with explicit recursion. *Journal of Information and Computation* 139(2):154–233, 1997.

[AL94] A. Asperti and C. Laneve. Interaction systems i: The theory of optimal reductions. *Mathematical structures in Computer Science*, 4:457–504, 1994.

[Bar84] H. Barendregt. *The Lambda-Calculus, its syntax and semantics*. Studies in Logic and the Foundation of Mathematics. Elsevier Science Publishers B. V. (North-Holland), Amsterdam, 1984. Second edition.

[Bar92] H. Barendregt. *Handbook of Logic in Computer Science*, chapter Lambda Calculi with Types, pages 117–309. Clarendon Press, 1992.

[Bax76] L. D. Baxter. *The Complexity of Unification*. PhD thesis, University of Waterloo, 1976.

[BBCK05] C. Bertolissi, P. Baldan, H. Cirstea, and C. Kirchner. A rewriting calculus for cyclic higher-order term graphs. In *Proceedings of TERMGRAPH'04, Roma, Italy*, volume 127 of *Electronic Notes in Theoretical Computer Science*, pages 21–41. Elsevier Science, 2005.

[BCK03] C. Bertolissi, H. Cirstea, and C. Kirchner. Translating combinatory reduction systems into the rewriting calculus. *Electronic Notes in Theoretical Computer Science*, 86(2), 2003.

[BCKL03] G. Barthe, H. Cirstea, C. Kirchner, and L. Liquori. Pure Patterns Type Systems. In *Principles of Programming Languages - POPL2003, New Orleans, USA*. ACM, January 2003.

[BCL87] A. Ben Cherifa and P. Lescanne. Termination of rewriting systems by polynomial interpretations and its implementation. *Science of Computer Programming*, 9(2):137–160, October 1987.

[Ber05] C. Bertolissi. The graph rewriting calculus: confluence and expressiveness. In *Proceedings of ICTCS'05, Ninth Italian Conference on Theoretical Computer Science, Siena, Italy*, Lecture Notes in Computer Science. Springer-Verlag, 2005.

[BFG97] F. Barbanera, M. Fernández, and H. Geuvers. Modularity of Strong Normalisation and Confluence in the Algebraic λ-Cube. *Journal of Functional Programming*, 7(6):613–660, November 1997.

[Bir35] G. Birkhoff. On the structure of abstract algebras. *Proceedings Cambridge Phil. Soc.*, 31:433–454, 1935.

[BKN87] D. Benanav, D. Kapur, and P. Narendran. Complexity of matching problems. *J. Symb. Comput.*, 3(1/2):203–216, 1987.

[Bla01] F. Blanqui. *Type Theory and Rewriting*. PhD thesis, University Paris-Sud, 2001.

[Blo01] S. Blom. *Term Graph Rewriting - syntax and semantics*. PhD thesis, Vrije Universiteit, Amsterdam, 2001.

[BN98] F. Baader and T. Nipkow. *Term Rewriting and all That*. Cambridge University Press, 1998.

[BS96] E. Barendsen and S. Smetsers. Uniqueness typing for functional languages with graph rewriting semantics. *Mathematical Structures in Computer Science*, 6(6):579–612, 1996.

[BS01] F. Baader and W. Snyder. Unification theory. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 8, pages 445–532. Elsevier Science, 2001.

[BT88] V. Breazu-Tannen. Combining algebra and higher-order types. In *Proceedings 3rd IEEE Symposium on Logic in Computer Science, Edinburgh (UK)*, pages 82–90, 1988.

[BvEG+87] H. P. Barendregt, M.C.J.D. van Eekelen, J.R.W. Glauert, J.R. Kennaway, M.J. Plasmeijer, and M.R. Sleep. Term graph rewriting. In *PARLE Parallel Architectures and Languages Europe*, volume 259 of *Lecture Notes in Computer Science*, pages 141–158, Eindhoven, 1987. Springer-Verlag.

[CD97] A. Corradini and F. Drewes. (Cyclic) term graph rewriting is adequate for rational parallel term rewriting. Technical Report TR-97-14, Dipartimento di Informatica, Pisa, 1997.

[CFK04] H. Cirstea, G. Faure, and C. Kirchner. A rho-calculus of explicit constraint application. In *Proceedings of the 5th workshop on rewriting logic and applications*, volume 117. Electronic Notes in Theoretical Computer Science, 2004.

[CG99a] A. Corradini and F. Gadducci. An algebraic presentation of term graphs via gsmonoidal categories, 1999.

[CG99b] A. Corradini and F. Gadducci. Rewriting on cyclic structures: Equivalence of operational and categorical descriptions. *Theoretical Informatics and Applications*, 33:467–493, 1999.

[Chu41] A. Church. A Formulation of the Simple Theory of Types. *Journal of Symbolic Logic*, 5:56–68, 1941.

[Cir00] H. Cirstea. *Calcul de réécriture : fondements et applications.* Thèse de Doctorat d'Université, Université Henri Poincaré - Nancy I, 2000.

[CK98] H. Cirstea and C. Kirchner. The rewriting calculus as a semantics of ELAN. In J. Hsiang and A. Ohori, editors, *4th Asian Computing Science Conference*, volume 1538 of *Lecture Notes in Computer Science*, pages 8–10. Springer-Verlag, 1998.

[CK01] H. Cirstea and C. Kirchner. The rewriting calculus — Part I *and* II. *Logic Journal of the Interest Group in Pure and Applied Logics*, 9(3):427–498, May 2001.

[CKL01] H. Cirstea, C. Kirchner, and L. Liquori. Matching Power. In *Proc. of RTA*, volume 2051 of *LNCS*, pages 77–92. Springer-Verlag, 2001.

[CKL02] H. Cirstea, C. Kirchner, and L. Liquori. Rewriting calculus with(out) types. In Fabio Gadducci and Ugo Montanari, editors, *Proceedings of WRLA'02*, Pisa (Italy), September 2002. Volume 71 of Electronic Notes in Theoretical Computer Science.

[CLW03] H. Cirstea, L. Liquori, and B. Wack. Rewriting Calculus with Fixpoints: Untyped and First-Order Systems. In *Proc. of Types, International Workshop on Types for Proof and Programs*, volume 3085 of *Lecture Notes in Computer Sciences*, pages 147–161. Springer Verlag, 2003.

[Cor93] A. Corradini. Term rewriting in CT_Σ. In M. C. Gaudel and J. P. Jouannaud, editors, *Proceedings of TAPSOFT'93, Theory and Practice of Software Developm ent / 4th International Joint Conference CAAP/FASE*, pages 468–484. Springer, Berlin, Heidelberg, 1993.

[Cou83] B. Courcelle. Fundamental properties of infinite trees. *Theoretical Computer Science*, 25(2):95–169, 1983.

[DDHY92] D. Dill, A. Drexler, A. Hu, and C. Yang. Protocol verification as a hardware design aid. In *IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 522–525. IEEE computer society, 1992.

[Der82] N. Dershowitz. Orderings for term-rewriting systems. *Theoretical Computer Science*, 17:279–301, 1982.

[Deu96] A. Deursen. An Overview of ASF+SDF. In *Language Prototyping*, pages 1–31. World Scientific, 1996. ISBN 981-02-2732-9.

[DHKP96] G. Dowek, T. Hardin, C. Kirchner, and F. Pfenning. Unification via explicit substitutions: The case of higher-order patterns. In Michael Maher, editor, *Proceedings of JICSLP'96*, Bonn (Germany), September 1996. The MIT press.

[DJ90] N. Dershowitz and J.-P. Jouannaud. Rewrite Systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, chapter 6, pages 244–320. Elsevier Science Publishers B. V. (North-Holland), 1990.

[Dow92] G. Dowek. Third order matching is decidable. In *Proceedings of LICS'92*, Santa-Cruz (California, USA), June 1992.

[EHK+97] H. Ehrig, R. Heckel, M. Korff, M. Löwe, L. Ribeiro, A. Wagner, and A. Corradini. *Handbook of Graph Grammars and Computing by Graph Transformation*, chapter Algebraic Approach to Graph Transformation Part II: Single Pushout Approach and Comparison with Double Pushout Approach, pages 247–312. World Scientific Publishing, 1997.

[Fau05] G. Faure. RhomCal: an implementation of the explicit rewriting calculus. Available on: `http://rho.loria.fr/implementations.html`, 2005.

[FG86] F. Fages and G. Huet. Complete sets of unifiers and matchers in equational theories. *Theoretical Computer Science*, 43(2-3):189–200, 1986.

[FK03] J. Forest and D. Kesner. Expression reduction systems with patterns. In *Proceedings of Rewriting Techniques and Applications, 14th International Conference, RTA 2003*, volume 2706 of *Lecture Notes in Computer Sciences*, pages 107–122. Springer Verlag, 2003.

[Fok92] M.M. Fokkinga. A gentle introduction to category theory — the calculational approach. In *Lecture Notes of the STOP 1992 Summerschool on Constructive Algorithmics*, pages 1–72 of Part 1. University of Utrecht, September 1992.

[GBT89] J. Gallier and V. Breazu-Tannen. Polymorphic rewriting conserves algebraic strong normalization and confluence. In *16th Colloquium Automata, Languages and Programming*, volume 372 of *Lecture Notes in Computer Science*, pages 137–150. Springer-Verlag, 1989.

[Ghi96] S. Ghilezan. Generalized finitness of developments in typed lambda calculi. *Journal of Automata, Languages and Combinatorics*, 1(4):247–258, 1996.

[GNW96] Q. Guo, P. Narendran, and D. A. Wolfram. Unification and matching modulo nilpotence. In *Conference on Automated Deduction*, pages 261–274, 1996.

[Gol81] D. Goldfarb. The undecidability of the second order unification problem. *Theoretical Computer Science*, 13:225–230, 1981.

[HBEV04] P. Haase, J. Broekstra, A. Eberhart, and R. Volz. A comparison of RDF query languages. In *Proceedings of the Third International Semantic Web Conference, Hiroshima, Japan, 2004.*, NOV 2004.

[Hin64] J. R. Hindley. *The Church-Rosser property and a result in combinatory logic.* PhD thesis, University of Newcastle-upon-Tyne, 1964.

[Hin78] J.R. Hindley. Reductions of residuals are finite. *Transactions of the American Mathematical Society*, 240:345–361, 1978.

[HP91] B. Hoffmann and D. Plump. Implementing term rewriting by jungle evaluation. *RAIRO: R. A. I. R. O. Informatique Theorique et Applications/Theoretical Informatics and Applications*, 25, 1991.

[HS86] J. Roger Hindley and Johnathan P. Seldin. *Introduction to Combinators and Lambda-calculus.* Cambridge University, 1986.

[Hue75] G. Huet. A unification algorithm for typed lambda calculus. *Theoretical Computer Science*, 1(1):27–57, 1975.

[Hue76] G. Huet. *Résolution d'equations dans les langages d'ordre 1,2, ...,ω*. Thèse de Doctorat d'Etat, Université de Paris 7 (France), 1976.

[Hue80] G. Huet. Confluent reductions: Abstract properties and applications to term rewriting systems. *Journal of the ACM*, 27(4):797–821, October 1980. Preliminary version in 18th Symposium on Foundations of Computer Science, IEEE, 1977.

[Iba04] L. Ibanescu. *Programmation par règles et stratégies pour la génération automatique de mécanismes de combustion d'hydrocarbures polycycliques*. Thèse de Doctorat d'Université, Institut National Polytechnique de Lorraine, Nancy, France, June 2004.

[JK84] J.P. Jouannaud and H. Kirchner. Completion of a set of rules modulo a set of equations. *SIAM Journal of Computing*, 15(4):1155–1194, 1986.

[JK91] J.-P. Jouannaud and C. Kirchner. Solving equations in abstract algebras: A rule-based survey of unification. In J.-L. Lassez and G. Plotkin, editors, *Computational Logic: Essays in Honor of Alan Robinson*, pages 257–321. MIT Press, Cambridge, MA, 1991.

[JO97] J.P. Jouannaud and M. Okada. Abstract Data Type Systems. *Theoretical Computer Science*, 173(2):349–391, 1997.

[Kah87] G. Kahn. Natural semantics. Technical Report 601, INRIA Sophia-Antipolis, February 1987.

[Kah97] W. Kahl. Algebraic graph derivations for graphical calculi. In *Proceedings of Graph-Theoretic Concepts in Computer Science, Cadenabbia (Como), Italy, 1996*, volume 1197 of *Lecture Notes in Computer Science*, pages 224–238. Springer, 1997.

[Kah98] W. Kahl. Relational treatment of term graphs with bound variables. *Journal of the IGPL*, 6(2):259–303, 1998.

[KB70] D. E. Knuth and P. B. Bendix. Simple word problems in universal algebras. In J. Leech, editor, *Computational Problems in Abstract Algebra*, pages 263–297. Pergamon Press, Oxford, 1970.

[Ken87] R. Kennaway. On 'on graph rewritings'. *Theoretical Computer Science*, 52(1-2):37–58, 1987.

[Kha90] Z.O. Khasidashvili. Expression reduction systems. In *Proceedings of I. Vekua Institute of Applied Mathematics*, volume 36, pages 200–220, 1990.

[Kir90] Claude Kirchner, editor. *Unification*. Academic Press inc., 1990.

[KK99] C. Kirchner and H. Kirchner. Rewriting, solving, proving. A preliminary version of a book available at `www.loria.fr/~ckirchne/rsp.ps.gz`, 1999.

[KKSd94] J. R. Kennaway, J. W. Klop, M. R. Sleep, and F. J. de Vries. On the adequacy of graph rewriting for simulating term rewriting. *ACM Transactions on Programming Languages and Systems*, 16(3):493–523, May 1994.

[KKSd97] J. R. Kennaway, J. W. Klop, M. R. Sleep, and F. J. de Vries. Infinitary lambda calculus. *Theoretical Computer Science*, 175(1):93–125, 1997.

[KKSdV91] J. R. Kennaway, J. W. Klop, M. R. Sleep, and F. J. de Vries. Transfinite reductions in orthogonal term rewriting systems. In *Proceedings 4th Conference on Rewriting Techniques and Applications, Como (Italy)*, 1991. also Report CS-R9041, CWI, 1990.

[KL82] S. Kamin and J.-J. Lévy. Attempts for generalizing the recursive path ordering. *Inria, Rocquencourt*, 1982.

[Klo80] J. W. Klop. *Combinatory Reduction Systems*. PhD thesis, CWI, 1980.

[Klo90] J. W. Klop. Term Rewriting Systems. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 1, chapter 6. Oxford University Press, 1990.

[KN92] D. Kapur and P. Narendran. Complexity of unification problems with associative-commutative operators. *J. Autom. Reason.*, 9(2):261–288, 1992.

[KOR93] J. W. Klop, V. van Oostrom, and F. van Raamsdonk. Combinatory Reduction Systems: Introduction and Survey. *Theoretical Computer Science*, 121:279–308, 1993. Special issue in honour of Corrado Böhm.

[KvOvR93] J.W. Klop, V. van Oostrom, and F. van Raamsdonk. Combinatory reduction systems: introduction and survey. *Theoretical Computer Science*, 121:279–308, 1993.

[Laf90] Y. Lafont. Interaction nets. In *POPL '90: Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 95–108, 1990.

[Lam90] J. Lamping. An algorithm for optimal lambda calculus reduction. In *POPL '90: Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 16–30. ACM Press, 1990.

[Lan75] D. S. Lankford. Canonical algebraic simplifications. Technical report, Louisiana Tech. University, 1975.

[Lan79] D. S. Lankford. A unification algorithm for abelian group theory. Technical report, Louisiana Tech. University, 1979.

[Lan93] C. Laneve. *Optimality and Concurrency in Interaction Systems*. PhD thesis, Università di Pisa, Pisa, Italy, 1993.

[Law63] B. Lawvere. Functorial semantics of algebraic theories. *Proceedings of the National Academy of Sciences of the United States of America*, 50(1):869–872, 1963.

[Lév78] J.-J. Lévy. *Réductions correctes et optimales dans le lambda-calcul*. PhD thesis, Université de Paris VII, 1978.

[Loa] R. Loader. *The undecidability of λ-definability*. Kluwer Academic Press, to appear.

[Löw93] M. Löwe. Algebraic approach to single-pushout graph transformation. *Theoretical Computer Science*, 109(1–2):181–224, 1993.

[LS04] L. Liquori and B.P. Serpette. iRho: an Imperative Rewriting Calculus. In *Proc. of PPDP, ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, pages 167–178. The ACM Press, 2004.

[LW04] L. Liquori and B. Wack. The Polymorphic Rewriting-calculus [Type Checking *v.s.* Type Inference]. In *Proc. of WRLA*, volume 117 of Electronic Notes in Theoretical Computer Science. Elsevier Science, 2004.

[Mel96] P.-A. Melliès. *Description Abstraite des Systèmes de Réécriture*. PhD thesis, Université Paris 7, 1996.

[Mil91] D. Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *Journal of Logic and Computation*, 1(4):497–536, 1991.

[MN86] D. A. Miller and G. Nadathur. Higher-order logic programming. In E. Shapiro, editor, *Proceedings of the Third International Logic Programming Conference*, volume 225 of *Lecture Notes in Computer Science*, pages 448–462. Springer-Verlag, 1986.

[MRV03] P.-E. Moreau, D. Ringeissen, and M. Vittek. A pattern matching compiler for multiple target languages. In *Compiler Construction, 12th International Conference, CC 2003, Warsaw, Poland*, pages 61–76, 2003.

[MSB] E. Miller, R. Swick, and D. Brickley. Resource description framework (RDF). `http://www.w3.org/RDF/`.

[New42] M. H. A. Newman. On theories with a combinatorial definition of equivalence. In *Annals of Math*, volume 43, pages 223–243, 1942.

[Nip91] T. Nipkow. Combining matching algorithms: The regular case. *Journal of Symbolic Computation*, 12:633–653, 1991.

[Nip93] T. Nipkow. Orthogonal higher-order rewrite systems are confluent. In *Proceedings of the International Conference on Typed La mbda Calculi and Applications*, pages 306–317, 1993.

[NP98] T. Nipkow and C. Prehofer. Higher-order rewriting and equational reasoning. In W. Bibel and P. Schmitt, editors, *Automated Deduction — A Basis for Applications. Volume I: Foundations*. Kluwer, 1998.

[Ohl98] E. Ohlebusch. Church-Rosser theorems for abstract reduction modulo an equivalence relation. In Tobias Nipkow, editor, *Proceedings of RTA'98)*, volume 1379 of *Lecture Notes in Computer Science*, pages 17–31. Springer, 1998.

[Oka89] M. Okada. Strong normalizability for the combined system of the typed Lambda-calculus and an arbitrary convergent term rewrite system. In *Proceedings of the ACM-SIGSAM 1989, Portland (Oregon)*, pages 357–363. ACM Press, July 1989.

[Pad96] V. Padovani. *Filtrage d'ordre supérieur*. Thèse de Doctorat d'Université, Université Paris VII, 1996.

[Pag97] B. Pagano. *Des calculs de substitution explicites et de leur application a la compilation des langages fonctionnels*. Thèse de Doctorat d'Université, U. Paris VI, 1997.

[Par90] M. Parigot. Internal labellings in lambda-calculus. In *Proceedings of Mathematical Foundations of Computer Science 1990, MFCS'90, Banská Bystrica, Czechoslovakia*, pages 439–445, 1990.

[Par92] M. Parigot. $\lambda\mu$-calculus: An algorithmic interpretation of classical natural deduction. In A. Voronkov, editor, *Proceedings of LPAR'92, St Petersburg, Russia, July 1992*, volume 624 of *Lecture Notes in Artificial Intelligence*, pages 190–201. Springer-Verlag, Berlin, 1992.

[PJ87] S. Peyton-Jones. *The implementation of functional programming languages*. Prentice Hall, Inc., 1987.

[Plo72] G. Plotkin. Building in equational theories. In B. Meltzer and D. Michie, editors, *Machine Intelligence*, volume 7, pages 73–90, Edinburgh, Scotland, 1972. Edinburgh University Press.

[Plu99] D. Plump. *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 2, chapter Term graph rewriting, pages 3–61. World Scientific Publishing, 1999.

[PS81] G. Peterson and M. E. Stickel. Complete sets of reductions for some equational theories. *Journal of the ACM*, 28:233–264, 1981.

[Qia93] Z. Qian. Linear unification of higher-order patterns. In M.-C. Gaudel and J.-P. Jouannaud, editors, *Proceedings of TAPSOFT'93*, volume 668 of *Lecture Notes in Computer Science*, pages 391–405. Springer-Verlag, 1993.

[Rin96] C. Ringeissen. Combining Decision Algorithms for Matching in the Union of Disjoint Equational Theories. *Information and Computation*, 126(2):144–160, May 1996.

[Ros73] B. K. Rosen. Tree-manipulating systems and Church-Rosser theorems. *Journal of the ACM*, 20(1):160–187, 1973.

[Sch65] D. Schroer. *The Church-Rosser theorem*. PhD thesis, Cornell University, 1965.

[SEP73] H. J. Schneider, H. Ehrig, and M. Pfender. Graph-grammars - an algebraic approach. In IEEE, editor, *Proceedings of the Fourteenth Annual Symposium on Switching and Automata Theory*, pages 167–180. Iowa, 1973.

[SPvE93] M. R. Sleep, M. J. Plasmeijer, and M. C. J. D. van Eekelen, editors. *Term graph rewriting: theory and practice*. Wiley, London, 1993.

[Sta75] J. Staples. Church-Rosser theorems for replacement systems. *Algebra and Logic, Lectures Notes in Mathematics*, 450:291–307, 1975.

[Sta82] R. Statman. Completeness, invariance and lambda-definability. *Journal Symbolic Logic*, 47(1):17–26, 1982.

[Tur37] A. M. Turing. Computable numbers with an application to the entscheidungsproblem. *Proc. London Math. Soc.*, 42:230–65, 1937.

[Tur79] D. A. Turner. A new implementation technique for applicative languages. *Software and Practice and Experience*, 9:31–49, 1979.

[VOVR93] V. van Oostrom and F. van Raamsdonk. Comparing combinatory reduction systems and higher-order rewrite systems. In *Higher-Order Algebra, Logic and Term Rewriting (HOA)*, pages 276–304, 1993.

[vR96] F. van Raamsdonk. *Confluence and Normalisation of Higher-Order Rewriting*. PhD thesis, University of Amsterdam, 1996.

[Vri85] Roel C. de Vrijer. A direct proof of the finite developments theorem. *Journal Symbolic Logic*, 50(2):339–343, 1985.

[Wac03] B. Wack. Klop counter example in the rho-calculus. Draft notes, LORIA, Nancy, 2003.

[Wac05] B. Wack. *Calcul de réécriture : du typage à la déduction*. Thèse de Doctorat d'Université, Université Henri Poincaré - Nancy I, 2005.

[Wad71] C. P. Wadsworth. *Semantics and Pragmatics of the Lambda Calculus*. D.Phil. thesis, Oxford University, 1971.

[Wol91] D. A. Wolfram. Rewriting, and equational unification: The higher-order cases. In R. V. Book, editor, *Rewriting Techniques and Applications: Proc. of the 4th International Conference RTA-91*, pages 25–36. Springer, Berlin, Heidelberg, 1991.

[Wol93] D. A. Wolfram. *The Clausal Theory of Types*, volume 21 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1993.

Résumé

Ces dernières années, on a assisté au développement du *calcul de réécriture*, encore appelé ρ-calcul, qui intègre de façon uniforme la réécriture de premier ordre et le λ-calcul. Cette thèse est dédiée à l'étude des capacités d'expression du calcul de réécriture, avec un intérêt particulier pour la réécriture d'ordre supérieur et la possibilité de manipuler des graphes.

Dans la première partie de cette thèse, la relation entre le calcul de réécriture et la réécriture d'ordre supérieur, en particulier les *Combinatory Reduction Systems* (CRSs), est étudiée. Nous présentons d'abord un algorithme de filtrage original pour les CRSs qui utilise une traduction des termes CRS en λ-termes et le filtrage d'ordre supérieur classique du λ-calcul. Nous proposons ensuite un encodage des CRSs dans le ρ-calcul basé sur la traduction de chaque réduction CRS en une réduction correspondante dans le ρ-calcul.

Dans la deuxième partie, nous présentons une extension du ρ-calcul, appelé calcul de réécriture de graphes (ou ρ_{g}-calcul), qui gère des termes avec partage et cycles. Le calcul sur les termes est généralisé de manière naturelle en utilisant des contraintes d'unification en plus des contraintes de filtrage standard du ρ-calcul. Le ρ_{g}-calcul est alors montré confluent sur des classes d'équivalence de termes, sous certaines restrictions de linéarité sur les motifs, et assez expressif pour simuler la réécriture de termes graphes et le λ-calcul cyclique.

Mots-clés: calcul de réécriture de graphes, lambda calcul, réécriture de terme graphes, filtrage

Abstract

The last few years have seen the development of the *rewriting calculus* (also called ρ-calculus) that uniformly integrates first-order term rewriting and λ-calculus. This thesis is devoted to the study of the expressiveness of the rewriting calculus, with special interest for higher-order rewriting and the possibility of dealing with graph-like structures.

The first part of the thesis is dedicated to the relationship between the rewriting calculus and higher-order term rewriting, namely the *Combinatory Reduction Systems* (CRSs). First, an original matching algorithm for CRSs terms that uses a simple term translation and the classical higher-order pattern matching of lambda terms is proposed and then an encoding of CRSs in the ρ-calculus based on a translation of each possible CRS-reduction into a corresponding ρ-reduction is presented.

The second part of the thesis is devoted to an extension of the ρ-calculus, called *graph rewriting calculus* (or ρ_{g}-calculus), handling terms with sharing and cycles. The calculus over terms is naturally generalised by using unification constraints in addition to standard ρ-calculus matching constraints. The ρ_{g}-calculus is shown to be confluent over equivalence classes of terms, under some linearity restrictions on patterns, and expressive enough to simulate first-order term graph rewriting and cyclic λ-calculus

Keywords: graph rewriting calculus, lambda calculus, Combinatory Reduction Systems, term graph rewriting, matching