Calcul de réécriture explicite

Rapport de stage
DEA Sémantique, Preuves et Langages
Université Paris 7 Denis Diderot

Germain Faure (ENS Cachan antenne de Bretagne)
Encadrants : Thomas Jensen, Claude Kirchner, Horatiu Cirstea

2 juin - 12 septembre 2003

Remerciements

Thomas Jensen a accepté de m'encadrer sur un sujet a priori un peu éloigné de ses recherches. Je l'en remercie. Son regard extérieur m'a amené a prendre du recul et à me poser des questions que l'on ne se pose pas toujours.

Merci bien sûr à Claude Kirchner. J'aimerais qu'il trouve ici mes sincères remerciements pour la qualité scientifique de son encadrement mais aussi pour la qualité humaine de celui-ci. Lui et toutes les personnes avec qui j'ai eu la chance de travailler à Nancy (Horatiu Cirstea, Benjamin Wack, Luigi Liquori, Pierre-Etienne Moreau, Christophe Ringensein) n'ont jamais compté leur temps et ont toujours eu leur porte grande ouverte. Un merci tout spécial à Horatiu qui a passé, en plein milieu du mois d'août, de longues heures au téléphone pour m'aider à finaliser mon travail.

Merci à Daniel pour sa maîtrise de la langue anglaise.

Merci à tous mes amis pour leur joie et leur amitié.

"Last but not least", un grand merci à ma famille qui, discrètement mais efficacement, m'a soutenu dans ce travail.

Introduction

I would be embarrassed to define what "programming means". Early computers were literally "wired" together with little wires on a patch board to make connections for each "1", and the absence of a wire represented a "0". It is difficult to say whether such computers were programmed since what we call nowadays "net programming" is not so different.

In the late forties, Turing's analysis of computability in terms of machines provided the conceptual basis for the construction of physical computers. Different approaches to recursiveness generate different types of programming languages. For example, what kind of programming languages the quantum calculus generates is still an open question. So, what a programming language is, or more generally, what " to program" means is not a fully answered question. Nevertheless, the question is better understood if one restricts to classical recursion theory. This understanding was obtained thanks to the abstraction of programming languages. It can be quite surprising to see how a language can develop thanks to its mathematization. In [OT97], we can see the evolution of Algol due to the categorical theory approach, the study of the λ-calculus [Bar84] is the most popular example.

One straightforward problem when one studies the core of a programming language is that we stay far from implementations. Many works have been trying to narrow the gap between the λ-calculus and implementations. Actually, one should say: "these gaps" since we can distinguish two different kinds of works: First, those who try to make explicit all ingredients of the λ-calculus. This was, in the 90's, the beginning of calculus with explicit substitutions [ACCL91]. Secondly, there were new calculus which enrich the λ-calculus: we can mention for example the π-calculus [SW01], a calculus for mobile processes, or calculus with explicit operators for pattern matching and substitutions [For02]. One drawback of all these calculus is that they do not deal explicitly with errors whereas error management takes a huge place in programming, compiling...

Many Ph.D theses begin with the leitmotiv "the lambda-calculus, the functional programming paradigm..." and the search on the web: "Lambda Calculus" and "Functional Programming" leads to 12 200 web pages dealing with these topics. Nevertheless, the typed lambda-calculus is *strongly normalizing*[1] whereas most programming languages use *fix points* to encode recursion. CAML users often think that without the `let rec` construction, recursion and fix points cannot be expressed. We will see in the Example 2.1 a counter example to that prejudice.

Pattern matching occurs implicitly in many languages through the simple parameter passing mechanism, and explicitly in languages like Prolog and ML, where it can be quite sophisticated [Mil91]. It is somewhat astonishing that one of the most commonly used models of computation, the lambda calculus, uses only trivial pattern matching. This has been extended, initially for programming concerns, either by the introduction of patterns in lambda calculi [Pey87, vO90], or by the introduction of matching and rewrite rules in functional programming languages. And indeed, many works address the integration of term rewriting with lambda calculus, either by enriching first-order rewriting with higher-order capabilities [KOR93, Wol93, NP98], or by adding to lambda calculus algebraic features allowing one, in particular, to deal with equality in an efficient way [Oka89, BT88, GBT89, JO97].

Pattern abstractions generalize lambda abstractions by binding structured expressions instead of variables, and are commonly used to compile case-expressions in functional programming languages [Pey87] and to provide term calculi for sequent calculi [KPT96]. For example, the pattern abstractions $\lambda 0.0$ and $\lambda \mathtt{succ}(\mathcal{X}).\mathcal{X}$ are used to compile the predecessor function $\lambda\mathcal{X}.\ \mathsf{case}\ \mathcal{X}\ \mathsf{of}\{0 \twoheadrightarrow 0 \mid \mathtt{succ}(\mathcal{X}) \twoheadrightarrow \mathcal{X}\}$, whereas the pattern abstraction $\lambda\langle\mathcal{X},\mathcal{Y}\rangle.\ \mathcal{X}$ is used to encode the sequent derivation

$$\dfrac{\dfrac{\sigma, \tau \vdash \sigma}{\sigma \wedge \tau \vdash \sigma}}{\vdash (\sigma \wedge \tau) \to \sigma}$$

Rule abstractions generalize in turn pattern abstractions by binding arbitrary expressions instead of patterns, and are used in the Rewriting Calculus to provide a first-class account of rewrite rules and rewriting strategies. For example, rule abstractions can be used to encode innermost rewriting strategies for term

[1] *i.e.*, each computation ends

rewriting systems. Furthermore, rule abstractions correspond to a form of higher-order natural deduction, where (parts of) proof trees are discharged instead of assumptions. Although such rule abstractions are a firmly grounded artifact both in logic and in programming language design and implementation, they lack established foundations.

The Rewriting Calculus, also called Rho-Calculus, makes all the basic ingredients of rewriting explicit objects, in particular the notions of rule formation (*abstraction*), rule *application* and *result*. It has nice properties and encompasses in a uniform setting λ-calculus and first-order rewriting. Thanks to its matching power, *i.e.* the ability to discriminate directly between patterns, *and* thanks to the first class status of rewrite rules, it also has the capability to represent in a simple and natural way object calculi [CKL01a, DK00]. In the Rewriting Calculus, the usual lambda abstraction $\lambda\mathcal{X}.\mathcal{T}$ is replaced by a rule abstraction $A \twoheadrightarrow B$, where A is an arbitrary term and B is the argument to be fired, where the free variables of A are bound (via pattern matching) in B.

The matching power of the Rho-Calculus can be regulated using arbitrary theories. In classical rewriting, this leads to non-deterministic behavior but since "results" are first class citizens in the Rho-Calculus, we can represent all possible results. The way to represent these results is also a parameter of the calculus since different possibilities are represented *w.r.t.* a structure which can have many different theories – see the Examples 1.13 and 1.14. Typically we recover the initial semantics of sets of result (presented in [CK01, Cir00]) by giving an associative-commutative and idempotent semantics to the structure. If one prefers lists or multisets results, then the corresponding formalization of "," should be specified.

Different extensions/variations of the Rho-Calculus are now available: in [LS03] we deal with an imperative version of the calculus and in [FK02] with exceptions in the rewriting calculus. The latter extensively uses the explicit management of errors in the calculus. Unlike λ-calculus with patterns, the Rho-Calculus uses matching in a full way, *i.e.*, matching can succeed or fail and matching failures are explicitly expressed in the Rho-Calculus.

Matching failures are not trivial and causes problems as for the confluence of the calculus [Cir00]. This is why in [CKL02] there is major evolution in the syntax and capability of the calculus: *delayed matching constraints* become explicit part of the calculus. So, the application of an abstraction $A \twoheadrightarrow B$ to a term C - denoted $(A \twoheadrightarrow B)C$" - always "fires" and produces - via a rule called (ρ) - as a result the term $[A \ll B]C$ which represents a term where the matching constraint is "put on the stack". The matching constraint will be (self) evaluated (if a matching solution exists) or delayed (if no solution exists). If a solution σ exists, the delayed matching constraint is self-evaluated to $\sigma(B)$ - via the σ rule.

Thanks to its abstraction mechanism, the calculus allows to design extremely powerful type systems generalizing the λ-cube [CKL01b] and the PTS [BCKL03]. In [CLW03], the first-order calculus *à la Church* is extensively studied. The calculs enjoys subject reduction, type uniqueness and decidability of typing (linear algorithm). The type system allows one to type fix points, recursion operators, ensuring computational expressiveness of the calculus.

The ability to parametrise the Rho-Calculus with the matching theory opens new possibilities and leads to a very expressive calculus. Nevertheless, it seems surprising that all the computations - that can be really important in some matching theories - related to the considered matching theory belong to a meta level. The same problem arises in Rogue [SDK$^+$03]. Rogue is a new programming language, based on an untyped version of the Rho-Calculus ([CK01]), to implement decision procedure. This new work appears since efficient decision procedures require a substantial engineering effort to implement in mainstream languages. The operational semantics given in this paper, as the rules of the Rho-Calculus, uses helper functions or in other words there are computations that we do not explicit.

In all the explicit substitution calculus [ACCL91, Les94, CHL96, Ros96], substitutions can be delayed thanks to the $Beta$ rule that transforms a β redex $(\lambda x.a)b$ into the *explicit* application on a of the substitution that replaces x by b. In the Rho-Calculus, matching constraints can be delayed too, thanks to the ρ rule. This rule does not compute anything but only transforms the application of a rewrite rule into the application of a matching constraint:

$$(\rho) \quad (A \twoheadrightarrow B)C \quad \mapsto_{\rho} \quad [A \ll C]B$$

It is really powerful, for implementation and control issues, to give the opportunity to decide when you really

want to start the computations needed to apply a rewrite rule. The σ rule computes in *one* reduction the substitution from the matching constraints and apply it:

$$(\sigma) \quad [A \ll B]C \quad \rightarrow \quad \sigma_{(A \ll B)}(C)$$

Of course, in implementations these operations should be separated and we should interact with other computations, *i.e.*, we want computations on constraints and applications of constraints to be explicit. This leads to the $\rho_{\mathtt{x}}$-calculus. We can think of an implementation of the Rho-Calculus with explicit computations and applications of constraints and, with a scheduler that switches regularly between computations on constraints, applications of substitutions and the ρ, δ rules. In such an implementation, computations can be done only when needed (and we can compose constraints as much as wanted).

The contributions of this training are twofold. First, we will give the intuition of how programs can be represented in the Rho-Calculus. We will compare with the λ-calculus and take many examples of O'CAML programs. Secondly, we propose, study and exemplify a rewriting calculus with explicit constraint handling. In this calculus, *matching* and *application* of substitutions become explicit. The approach is really modular, allowing extension to arbitrary matching theories. We will also prove that the calculus is powerful to deal with errors. We will afterwards prove the confluence of the calculus and the termination of the "explicit" part of the calculus, some non trivial proofs. We conclude by an other extension of the calculus to handle composition of substitutions.

In the first chapter, we present smoothly the Rho-Calculus giving many examples to show the expressiveness and the behavior of the calculus. In chapter two, we study programming aspects of the Rho-Calculus from the handling of data structures to the encoding of calculi of objects. The last chapter is devoted to the presentation and the study of the $\rho_{\mathtt{x}}$-calculus and its extension.

Chapter 1

The Rho-Calculus

Some sections of the two first chapters are strongly inspired from [LS03, CLW03]. I want to sincerly thanks those authors to allow me to use their sources.

As in any calculus involving binders, in all this report, we work modulo the "*α-conversion*" of Church [Chu41], and modulo the "*hygiene-convention*" of Barendregt [Bar84], *i.e.*, free and bound variables have different names. The symbol = denotes syntactic identity of objects like terms or substitutions.

1.1 Syntax - Examples

The syntax of the calculus is given in Figure 1.1. We refer the reader to the introduction and to the next section for the motivation and the use of matching constraints.

Terms	A, B, C	$::=$	$\mathcal{X}$	(Variables)
		$\mid$	$\mathcal{K}$	(Constants)
		$\mid$	$A \twoheadrightarrow B$	(Abstraction)
		$\mid$	$A\ B$	(Fonctional application)
		$\mid$	A, B	(Structure)
		$\mid$	$[A \ll B]C$	(Delayed matching constraint)

Figure 1.1: Syntax of the Rho-Calculus

The symbols $X, Y, Z \ldots$ range over the set $\mathcal{X}$ of variables, the symbols $a, b, c, \ldots, f, g, \ldots$ range over the set $\mathcal{K}$ of constants. The symbols $A, B, C \ldots$ range over the set of ρ-terms. All the symbols can be indexed.

We also assume that the (hidden) application operator (often denoted by "$\bullet$") associates to the left, while the other operators associate to the right. The priority of "$\bullet$" is higher than that of "[ ]", which is higher than that of "$\twoheadrightarrow$", which is, in turn, of higher priority than the ",". The symbol $;$ stands for "," or "$\bullet$".

One can encode the λ-calculus in the Rho-Calculus. The binder "λ" is replaced by a rule abstraction "$\twoheadrightarrow$". The variables are denoted by $X, Y, Z \ldots$

Example 1.1 (Encoding of the λ-terms)

λ-calculus	$Rho - Calculus$
$\lambda X.X$	$X \twoheadrightarrow X$
$\lambda X \lambda Y.X$	$X \twoheadrightarrow (Y \twoheadrightarrow X)$
$\lambda X.(XX)$	$X \twoheadrightarrow XX$

The application is denoted as in the λ-calculus using the hidden operator "$\bullet$". Let n denote the arity of the constant $\mathtt{f}$ that can be obtained either from the $\mathtt{f}$ type or that can be a data, as in classical rewriting. In the rewriting, the function application is denoted by $\mathtt{f}(A_1, \ldots, A_n)$ whereas in the Rho-Calculus one uses the constant $\mathtt{f}$ and n applications: $(\ldots((\mathtt{f}\ A_1)A_2)\ldots)A_n)$. This changes the head symbol, something quite significant *w.r.t.* matching concerns. The notation $(\ldots((\mathtt{f}\ A_1)A_2)\ldots)A_n)$ is tedious to read. We will often use the following syntactic sugar:

$$\mathtt{f}(A_1 \ldots A_n) \triangleq (\ldots((\mathtt{f}\ A_1)A_2)\ldots)A_n)$$

It is well-known that the λ-calculus is Turing-complete. One aim of the Rho-Calculus is to model important features of programming languages - such as matching and errors - not present in the λ-calculus. In the next chapter, we will give more examples comparing the λ-calculus and the Rho-Calculus.

Example 1.2 (Encoding of first-order terms; functional application) *Using the constants* $\mathfrak{tt}$ *- to denote the boolean value true - the constants* $\mathfrak{ff}$ $\mathfrak{not}, \mathfrak{and}, \mathfrak{or}, \mathfrak{xor}$*, we can define the following first-order terms:*
$\mathfrak{and}(X\ \mathfrak{tt})$
$\mathfrak{or}(\mathfrak{not}(X)\ \mathfrak{not}(Y))$.

Thanks to rule abstraction, we can bind arbitrary expressions.

Example 1.3 (Rule abstraction) *Here are some rules to compute in Boolean algebra.*

$\mathfrak{and}(X\ \mathfrak{tt}) \twoheadrightarrow X$*; the free variable of the pattern - here* $\mathfrak{and}(X\ \mathfrak{tt})$ *- are bound in the body of the abstraction - here* X*.*

$\mathfrak{not}(\mathfrak{and}(X\ Y)) \twoheadrightarrow \mathit{or}(\mathfrak{not}(X)\ \mathfrak{not}(Y))$*; this more elaborated rule bounds the two variables* X *and* Y*.*

$\mathfrak{and}(X\ X) \twoheadrightarrow X$*; a non-linear rule.*

Most of the time, in programming language, non-linearity must be done "by hand", by checking the equality of two symbols. The definition of non-linear functions, *e.g.* in O'CAML [WL96, CMP00], leads to quite amazing behaviors (note that no warning is issued).

Example 1.4 (Non-Linearity in O'CAML)

```
# let f x x = x;;
val f : 'a -> 'b -> 'b = <fun>
# f 2 3;;
- : int = 3
```

For λ-calculists, this result is natural since in the λ-calculus $(\lambda X.(\lambda X.X))2\ 3$, representing the given O'CAML function, can be reduced to 3. But we may also want programming languages to be used by non λ-calculists. We refer to Example 1.7 for the handling of a non-linear rule in the Rho-Calculus.

The rule application is also denoted using the (often hidden) symbol "$\bullet$".

Example 1.5 (Rule Application) *The application of the rewrite rule* $\mathfrak{and}(X\ X) \twoheadrightarrow X$ *to* $\mathfrak{and}(\mathfrak{ff}\ \mathfrak{ff})$ *is denoted by the term* $\big(\mathfrak{and}\,(X\ X) \twoheadrightarrow X\big)\ \mathfrak{and}(\mathfrak{ff}\ \mathfrak{ff})$.

To give a first class status to results, we need to have a "data" structure to represent different choices. For example, non-determinism in the application of a rewrite system - due to critical pairs - can be easily expressed. See Example 1.8. The semantics given to the structure constructor is described by an appropriate theory that depends on the kind of result one wants to formalize. Typically we recover the first semantics of sets of result [Cir00, CK98] by giving an associative-commutative and idempotent semantics to ",". If one prefers lists or mulisets, then the corresponding formalization of "," should be specified.

The ability to use structure of terms is a very powerful way to have straightforward encoding. A whole rewrite system can be represented by a single term, *e.g.*, " $\mathfrak{if}\,(\mathfrak{tt}\ X\ Y) \twoheadrightarrow X$, $\mathfrak{if}\,(\mathfrak{ff}\ X\ Y) \twoheadrightarrow Y$". The syntax of the Rho-Calculus allows arbitrary terms on the left hand side of the abstraction. One nice and useful example is the ability to discriminate between structures. For example, the "structure projection" can be written as $X, Y \twoheadrightarrow X$. As we will see in the next section, if we apply this term to a, b we obtain a.

$$
\begin{array}{llll}
(\rho) & (A \twoheadrightarrow B)C & \to_\rho & [A \ll C]B \\
(\sigma) & [A \ll C]B & \to_\sigma & \sigma_{(A \ll C)}(B) \\
(\delta) & (A, B)\, C & \to_\delta & A\, C, B\, C
\end{array}
$$

Figure 1.2: Small-step reduction semantics

1.2 Semantics - Examples

The small-step reduction semantics is defined by the reduction rules presented in Figure 1.2. The central idea of the (ρ) rule of the calculus is that the application of a term $A \twoheadrightarrow B$ to a term C, reduces to the delayed matching constraint $[A \ll_{\mathbb{T}} C]B$ where we assume we are given a theory $\mathbb{T}$ over terms having a decidable matching problem. The (σ) rule consists in solving the matching equation $A \ll_{\mathbb{T}} C$, and applying the obtained result to the the term B. The theory $\mathbb{T}$ denotes the matching theory used. The rule (δ) deals with the distributivity of the application on the structures built with the "," constructor.

As usual, we define the define the one step $\mapsto_{\mathcal{R}}$ and the many steps $\mapsto\!\!\!\!\to_{\mathcal{R}}$ relations of a relation $\to_{\mathcal{R}}$. For example, we will denote $\mapsto\!\!\!\!\to_{\rho\sigma\delta}$ the many step relation of $\to_{\rho\sigma\delta}=\to_\rho \cup \to_\sigma \cup \to_\delta$.

It is important to remark that:

- If A is a variable, then the subsequent combination of (ρ) and (σ) rules corresponds exactly to the (β) rule of the λ-calculus.
- Variable manipulations in substitutions are handled externally, using α-conversion and Barendregt's hygiene convention if necessary.

Unless otherwise stated, the empty matching theory will be used, *i.e.*, rules are matched syntacticly. To have a better understanding of the behavior of the calculus, let us give some examples of reductions.

Example 1.6 (Application of a rewrite rule) *We fully describe on a very simple example how the reduction semantics computes the rewrite rule application.*

$\mathfrak{not}(\mathfrak{and}(X\ Y)) \twoheadrightarrow \mathfrak{or}(\mathfrak{not}(X)\ \ \mathfrak{not}(Y)) \quad \mathfrak{not}(\mathfrak{and}(\mathfrak{tt}\ \mathfrak{ff}))$
$\quad \mapsto_\rho \quad [\mathfrak{not}(\mathfrak{and}(X\ Y)) \ll \mathfrak{not}(\mathfrak{and}(\mathfrak{tt}\ \mathfrak{ff}))]\,(\mathfrak{or}(\mathfrak{not}(X)\ \ \mathfrak{not}(Y)))$
$\quad \mapsto_\sigma \quad (\mathfrak{or}(\mathfrak{not}(\mathfrak{tt})\ \ \mathfrak{not}(\mathfrak{ff})))$

Let us comment on the σ step. First, the substitution solution of the matching constraint $\mathfrak{not}(\mathfrak{and}(X\ Y)) \ll \mathfrak{not}(\mathfrak{and}(\mathfrak{tt}\ \mathfrak{ff}))$ is computed. In this example, we obtain the substitution $\sigma = \{X \mapsto \mathfrak{tt}, Y \mapsto \mathfrak{ff}\}$. The matching computations are done in a meta-level using a classical algorithm. If the matching does not have any solution, then the σ rule cannot be applied. Secondly, the substitution is applied, also at the meta level. In this example $\sigma\,((\mathfrak{or}(\mathfrak{not}(X)\ \ \mathfrak{not}(Y)))) = (\mathfrak{or}(\mathfrak{not}(\mathfrak{tt})\ \ \mathfrak{not}(\mathfrak{ff})))$.

The application of a non-linear rewrite rule is as simple as that of a linear one.

Example 1.7 (Application of a non-linear rule)

$(\mathfrak{xor}(X\ X) \twoheadrightarrow X)\ \mathfrak{xor}(\mathfrak{tt}\ \mathfrak{tt})$
$\quad \mapsto_\rho \quad [\mathfrak{xor}(X\ X) \ll \mathfrak{xor}(\mathfrak{tt}\ \mathfrak{tt})]X$
$\quad \mapsto_\sigma \quad \mathfrak{tt}$

Example 1.8 (Application of a rewrite system) *We show how a rewrite system applies to a term. To keep the example easily readable, the rewrite system is taken as simple as possible. The non determinism of classical rewriting is here explicitly dealt with.*

$\big(f(X\ Y) \twoheadrightarrow X,\ f(X\ a) \twoheadrightarrow b\big)\ \ f(a\ a)$
$\quad \mapsto_\delta \quad (f(X\ Y) \twoheadrightarrow X)\ f(a\ a),\ (f(X\ a) \twoheadrightarrow b)\ f(a\ a)$
$\quad \mapsto_\rho \quad [f(X\ Y) \ll f(a\ a)]X,\ (f(X\ a) \twoheadrightarrow b)\ f(a\ a)$
$\quad \mapsto_\rho \quad [f(X\ Y) \ll f(a\ a)]X,\ [f(X\ a) \ll f(a\ a)]b$
$\quad \mapsto_\sigma \quad a,\ [f(X\ a) \ll f(a\ a)]b$
$\quad \mapsto_\sigma \quad a,\ b$

Example 1.9 (Run-time error: matching failure) *When we apply a rewrite system there are some rules that do not match the term we have to evaluate. This leads to a lot of matching failures. We deal with them by an appropriate theory for the ",". See Example 1.14. We give an example of a matching failure representing a run-time error as shown in Section 2.1.3.*

$\big(\mathfrak{not}(\mathfrak{not}(X)) \twoheadrightarrow X\big)\ \mathfrak{not}(\mathfrak{ff})$
$\quad \mapsto_\rho \quad [\mathfrak{not}(\mathfrak{not}(X)) \ll \mathfrak{not}(\mathfrak{ff})]X$

The result can be read as follows: "the result would be an instance of X but at run-time one tried to match $\mathfrak{not}(\mathfrak{not}(X))$ against $\mathfrak{not}(\mathfrak{ff})$, yielding the (dirty) result $[\mathfrak{not}(\mathfrak{not}(X)) \ll \mathfrak{not}(\mathfrak{ff})]X$".

Example 1.10 (Delayed matching constraint) *We illustrate the powerful use of delayed matching constraints. The same term led in previous versions of the Rho-Calculus to non-confluent reductions. But now, thanks to delayed matching constraints the problem is solved. The motivation for constraints can be summed up as follows:* the delay compensates for the local lack of information.

$(X \twoheadrightarrow (a \twoheadrightarrow b)\ X)\ a$
$\quad \mapsto_\rho \quad (X \twoheadrightarrow [a \ll X]b)\ a$

The term $[a \ll X]b$ cannot be reduced since the matching constraint has no solution. The instanciation of X by a a will make the constraint solvable.

$\quad \mapsto_\rho \quad [X \ll a]\big([a \ll X]b\big)$
$\quad \mapsto_\sigma \quad [a \ll a]b$
$\quad \mapsto_\sigma \quad b$

Example 1.11 (Run-time error: non-linearity) *In Example 1.9, we have seen that matching failure can be caused by clashes on symbols. In this example, we show that matching failures can also be caused by non-linearity.*

$\big(\mathfrak{xor}(X\ X) \twoheadrightarrow \mathfrak{ff}\big)\ \mathfrak{xor}(\mathfrak{tt}\ \mathfrak{ff})$
$\quad \mapsto_\rho \quad [\mathfrak{xor}(X\ X) \ll \mathfrak{xor}(\mathfrak{tt}\ \mathfrak{ff})]\mathfrak{ff}$

This term is in normal form, as a dirty result.

Example 1.12 (Application of a rewrite rule in an commutative theory) *We said before that the matching theory is a parameter of the calculus. We present here a reduction where the symbol $\mathfrak{and}$ is assumed to be commutative. The other symbols remain syntactic.*

$\big(\mathfrak{and}(X\ \mathfrak{or}(Y\ Z)) \twoheadrightarrow \mathfrak{or}(\mathfrak{and}(X\ Y)\ \mathfrak{and}(X\ Z))\big)\quad \mathfrak{and}(\mathfrak{or}(\mathfrak{tt}\ \mathfrak{ff})\ \mathfrak{or}(\mathfrak{ff}\ \mathfrak{ff}))$
$\mapsto_\rho \quad [\mathfrak{and}\,(X\ \mathfrak{or}\,(Y\ Z)) \ll \mathfrak{and}\,(\mathfrak{or}\,(\mathfrak{tt}\ \mathfrak{ff})\ \ \mathfrak{or}\,(\mathfrak{ff}\ \mathfrak{ff}))]\,(\mathfrak{or}\,(\mathfrak{and}\,(X\ Y)\ \ \mathfrak{and}\,(X\ Z)))$
$\mapsto_\sigma \quad \mathfrak{or}\,(\mathfrak{and}\,(\mathfrak{or}\,(\mathfrak{tt}\ \mathfrak{ff})\ \ \mathfrak{ff})\ \ \mathfrak{and}\,(\mathfrak{or}\,(\mathfrak{tt}\ \mathfrak{ff})\ \ \mathfrak{ff}))\,,$
$\qquad\quad \mathfrak{or}\,(\mathfrak{and}\,(\mathfrak{or}\,(\mathfrak{ff}\ \mathfrak{ff})\ \ \mathfrak{tt})\ \ \mathfrak{and}\,(\mathfrak{or}\,(\mathfrak{ff}\ \mathfrak{ff})\ \ \mathfrak{ff}))$

The matching is not unitary, so we obtain two results represented thanks to the structure ",".

Example 1.13 (Application of a rewrite rule in empty, A, C, AC theories) *We look at the evaluation of the same term but in different matching theories for the $\oplus$ operator (here in an infix notation). This operator is successively syntactic, associative, commutative, and finally associative-commutative. A reduction in the empty theory is denoted $\mapsto_\emptyset$, in the associative $\mapsto_A \ldots$*

$(X \oplus Y \twoheadrightarrow X)\ (a \oplus (b \oplus c)) \quad \mapsto_\emptyset \quad a$
$(X \oplus Y \twoheadrightarrow X)\ (a \oplus (b \oplus c)) \quad \mapsto_A \quad a, a \oplus b$
$(X \oplus Y \twoheadrightarrow X)\ (a \oplus (b \oplus c)) \quad \mapsto_C \quad a, b \oplus c$
$(X \oplus Y \twoheadrightarrow X)\ (a \oplus (b \oplus c)) \quad \mapsto_{AC} \quad a, b, c, a \oplus b, b \oplus c, a \oplus c$

We have seen that the theory of the structure "," can be chosen. When we deal with the application of a rewrite system there are a lot of rewrite rules that do not apply to the head of the term. This causes a lot of matching failures. So, we can define a theory for the structure operator "," to approximate the class of matching equations that will not be solvable. This theory is denoted by $\mathbb{T}_{\mathsf{nostuck}}$ and defined as follows:

$$\frac{A_1 \not\sqsubseteq B_1}{[A_1 \ll B_1]C, T =^{\mathsf{no}}_{\mathsf{stuck}} T} \ (=^{\mathsf{no}}_{\mathsf{stuck}})$$

The relation $A \not\sqsubseteq B$ reads "*A does not potentially superpose to B*". For example, $3 \not\sqsubseteq 4$, or $g(X) \not\sqsubseteq h(3)$, or $f(X\ Y) \not\sqsubseteq h(3)$, *etc.* Note that $g(3) \not\sqsubseteq g(X)$ does not hold (*i.e.* $g(3)$ potentially superposes $g(X)$), since X is a variable and the matching equation can succeed if X is instantiated with 3 later on. The more precise definition and study of this theory can be found in [CLW03]. This theory will be extensively used in the next chapter.

Example 1.14 (Use of $\mathbb{T}_{\mathsf{nostuck}}$) *We consider the application of the rewrite system* $\mathcal{R} = \{g(g(X)) \to h(g(X)),\ h(h(X)) \to g(X)\}$. *The application of* $\mathcal{R}$ *to* $g(g(a))$ *is described in the Rho-Calculus by:* $(g(g(X)) \twoheadrightarrow h(g(X)),\ h(h(X)) \twoheadrightarrow g(X))\ g(g(a))$

$$\begin{array}{ll}
\mapsto_{\delta} & \big(g(g(X)) \twoheadrightarrow h(g(X))\big)\ g(g(a)),\ \big(h(h(X)) \twoheadrightarrow g(X)\big)\ g(g(a)) \\
\mapsto_{p} & \big[g(g(X)) \ll g(g(a))\big]\ h(g(X)),\ \big[h(h(X)) \ll g(g(a))\big] g(X) \\
\mapsto_{\sigma} & h(g(a)),\ \big[h(h(X)) \ll g(g(a))\big] g(X) \\
=^{\mathsf{no}}_{\mathsf{stuck}} & h(g(a))
\end{array}$$

But actually the example chosen is simple since we do not need to apply the rewrite system twice as for $g(g(g(a))) \mapsto\!\!\!\!\twoheadrightarrow_{\mathcal{R}} h(g(g(a)))$, *i.e.*, we need to encode normalization. Of course to program in the Rho-Calculus, we need such encodings, *e.g.*, to encode recursion. This is the main topic of the next chapter.

Chapter 2

Programming in the Rho-Calculus

As the λ-calculus, the Rho-Calculus is not at all a programming language but it should be a nice back-end to construct a new one. The purpose of this chapter is definitively not to explain how to build a programming language from the Rho-Calculus. We want to give an intuition about the way to represent a program in the framework. The first section deals with data structures taking the examples of lists. We will afterwards present a type system for the Rho-Calculus and explain why all the examples are well typed - from the encoding of calculi of objects to recursive functions.

2.1 Dealing with data structures

We will see in this section how a data structure can be defined and used in the Rho-Calculus. As usual, we will describe the constructors, the destructors, the observers and the combiners. We deal with the example of lists.

2.1.1 Constructors

In the Rho-Calculus, the constructors are defined using constants. Here, we will use the constants "`Empty`" and "`Cons`". This corresponds in O'CAML to the use of "`[]`" and "`::`".

2.1.2 Destructors

In O'CAML the list destructors can be written:

```
(*Destructors*)
 let car l = match l with
   |x::m -> x;;

 let cdr l = match l with
   | x::m -> m;;
```

These are partial function since we cannot apply them to `[]` without raising the exception:

```
# car [];;
Exception: Match_failure ("", 12, 42).
```

In the Rho-Calculus, the list destructors are written:

$$car \triangleq \texttt{Cons}(X\ M) \multimap X$$
$$cdr \triangleq \texttt{Cons}(X\ M) \multimap M$$

In the Rho-Calculus, when we apply *car* on `Empty` we obtain, as in O'CAML a run time error but here represented by a matching constraint without solutions:

$$(\texttt{Cons}(X\ M) \twoheadrightarrow X)\ \texttt{Empty} \qquad \mapsto_p \quad [\texttt{Cons}(X\ M) \ll \texttt{Empty}]X$$

Unlike in O'CAML, the matching failure is explicit and the programmer can have a better understanding of the error. Actually, the O'CAML interpreter answers to the previous definitions of *car* and *cdr* with:

```
Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched: []
```

So, the list destructors must be written:

```
(*Destructors*)
 let car l = match l with
     []  -> failwith "car"
   |x::m -> x;;

 let cdr l = match l with
     []    -> failwith "cdr"
   | x::m -> m;;
```

Since the handling of errors is not explicit in O'CAML - and more generally not explicit in the paradigm on which O'CAML is based - we have to *handle by hand* the matching failures, whereas this is done automatically in the Rho-Calculus thanks to the reduction semantics.

The table below sums up the definition of list accessors and recalls the encoding of lists in the λ-calculus.

	λ-calculus	Rho-Calculus
`cons`	$\lambda XYZ.ZXY$	$X \twoheadrightarrow Y \twoheadrightarrow \texttt{Cons}(X\ Y)$
`car`	$\lambda Z.Z(\lambda XY.X)$	$\texttt{Cons}(X\ Y) \twoheadrightarrow X$
`cdr`	$\lambda Z.Z(\lambda XY.Y)$	$\texttt{Cons}(X\ Y) \twoheadrightarrow Y$

2.1.3 Encoding fix points in small step

Before studying observers on lists we will give the simple addition example of the encoding of (object oriented flavored) fix-points in small step semantics. Let us assume that *rec* and *add* are two constants[1]. We will use Kamin's self application [Kam88] which is written in the Rho-Calculus: $S.rec \triangleq S\ rec(S)$

$$plus \triangleq rec(S) \twoheadrightarrow \left(\begin{array}{l} add(0\ Y) \twoheadrightarrow Y, \\ add(suc(X)\ Y) \twoheadrightarrow suc\,(S.rec\ \ add(XY)) \end{array} \right)$$

$$addition \triangleq X \twoheadrightarrow Y \twoheadrightarrow plus.rec\ add(X\ Y)$$

Then, this term computes indeed the addition over Peano integers, as illustrated in the following example, where the expressions "$\overline{m}$", and "$\overline{m+n}$", and "$\overline{m-n}$" are just aliases for the Peano representations of these numbers as sequences of $suc(\ldots suc(0)\ldots)$. To ease the reading, within the failed matching constraints, we keep only the sub-terms which do lead to a failure (for instance, in $[add(0\ Y) \ll add(\overline{n}, \overline{m})]$, we only keep

[1]Type concerns can be found in the next section.

$[0 \ll \overline{n}]$).

$$
\begin{array}{ll}
addition(n\ m) \triangleq plus.rec\ \ add(\overline{n}\ \overline{m}) & \\
\mapsto_{\rho\sigma\delta} & (add(0\ Y) \twoheadrightarrow Y)\ \ add(\overline{n}\ \overline{m}) \\
 & (add(suc(X)\ Y) \twoheadrightarrow suc\,(plus.rec\ \ add(X\ Y)))\ \ add(\overline{n}\ \overline{m}) \\
 & \cdots \\
\mapsto_{\rho\sigma\delta} & [0 \ll \overline{n}]\overline{m} \quad [0 \ll \overline{n-1}](\overline{m+1})\ \cdots \\
 & [0 \ll 0](\overline{m+n}) \\
 & [suc(X) \ll 0](suc\,(plus.rec\ \ add(X\ Y))) \\
=^{\mathsf{no}}_{\mathsf{stuck}} & [0 \ll 0](\overline{m+n}) \\
\mapsto_{\sigma} & \overline{m+n}
\end{array}
$$

Worth noticing is that all the stuck results are dropped by $=^{\mathsf{no}}_{\mathsf{stuck}}$; the only interesting member of the structure is $[0 \ll 0](\overline{m+n})$. Before this term, all the terms get stuck because we try to match 0 against $suc(\overline{n})$; after too because we try to match $suc(X)$ against 0. Notice that if we erase all the "administrative" sub-terms which encode the recursive machinery, we get back a TRS computing addition:

$$
\begin{array}{lll}
add(0\ Y) & \twoheadrightarrow & Y \\
add(suc(X)\ Y) & \twoheadrightarrow & suc\,(add(X\ Y))
\end{array}
$$

This mechanical encoding works for many TRS[2] as well: one just has to put the sub-term "$S.rec$" before all the defined constants - *i.e.*, at top level in the left hand side of a rewrite rule - so that the whole rewrite system can be re-applied to any of the corresponding sub-terms.

2.1.4 Observers

We will study how to encode observers on lists in the Rho-Calculus. We will take the example of the computation of the length and the test of equality of two lists.

Length of a list

This function can be easily written in O'CAML.

```
let rec length l = match l with
     []   -> 0
  | x::m  -> 1 + (length m);;
```

To encode this function in the Rho-Calculus, we will use - as in O'CAML - the following rewriting system:

$$
\begin{cases}
\texttt{length}(\texttt{Empty}) \twoheadrightarrow 0 \\
\texttt{length}(\texttt{Cons}(X\ M)) \twoheadrightarrow 1 + \texttt{length}(M)
\end{cases}
$$

Of course, we have to encode - as for the addition - the recursion (or in a rewriting approach the normalization *w.r.t.* the rewriting system).

$$
len \triangleq rec(S) \twoheadrightarrow \left(\begin{array}{l} \texttt{Length}(\texttt{Empty}) \twoheadrightarrow 0, \\ \texttt{Length}(\texttt{Cons}(X\ M)) \twoheadrightarrow 1 + S.rec\ (\texttt{Length}\ M) \end{array} \right)
$$

$$
\texttt{length} \triangleq L \twoheadrightarrow len.rec\ (\texttt{Length}\ L)
$$

It must be said again that we can transform the rewriting system to the ρ-term just by adding "administrative" sub-terms that can be easily automatically generated.

[2]provided it is well typed, convergent and ground reducible.

Equality of lists

We want to test the equality of two lists. The straightforward method is to test the equality of each element. In O'CAML, this can be written:

```
let rec eqlist l1 l2 = match (l1,l2) with
    (x1::m1,x2::m2) -> if x1 = x2 then (eqlist m1 m2) else false
 |  ([],[]) -> true
 |  (_, _) -> false;;
```

In this example, we explicitly used the convention concerning the order of evaluation of the *match* cases. Since strategies can be encoded in the Rho-Calculus [CKLW03], we will use one of them: the `first`. The `first` selects between its arguments the first term such that the application of this term to a given ρ-term is not a failure. One application of the `first` is to model the previous convention.

Unlike in O'CAML, we explicitly deal with non-linearity in the Rho-Calculus[3]. Thus the *eqlist* function can be written:

$$eq \triangleq rec(S) \twoheadrightarrow \texttt{first} \left(\begin{array}{l} Eqlist(\texttt{Cons}(X_1\ M_1)\ \texttt{Cons}(X_1\ M_2)) \twoheadrightarrow S.rec\ Eqlist(M_1\ M_2), \\ Eqlist(\texttt{Empty Empty}) \twoheadrightarrow true, \\ Eqlist(X\ Y) \twoheadrightarrow false\ \end{array} \right)$$

$$eqlist \triangleq L_1 \twoheadrightarrow L_2 \twoheadrightarrow eq.rec\ Eqlist(L_1\ L_2)$$

Looking for an element

The following function looks for one element in a list. It returns $\mathfrak{tt}$ if the element is in the list, $\mathfrak{ff}$ otherwise.

```
let rec search a l = match l with
    [] -> false
  | x::m -> if x = a then true else search a m;;
```

The encoding in the Rho-Calculus:

$$sear \triangleq rec(S) \twoheadrightarrow \texttt{first} \left(\begin{array}{l} Search(X\ \texttt{Empty}) \twoheadrightarrow false, \\ Search(X_1\ (\texttt{Cons}(X_1\ M)) \twoheadrightarrow true, \\ Search(X_1\ (\texttt{Cons}(X\ M) \twoheadrightarrow S.rec\ Search(X_1\ M) \end{array} \right)$$

$$search \triangleq X \twoheadrightarrow L \twoheadrightarrow sear.rec\ Search(X\ L)$$

Combiners

Reversing

The first combiner we will program is the function which reverses a list. We will use the well-known method by accumulator. We first give the definition in O'CAML.

```
let rec reverseAux l m = match l with
    [] -> m
  |x::xl -> reverseAux xl (x::m);;
let reverse l= (reverseAux l []);;
```

The definition in the Rho-Calculus is as simple as in O'CAML.

[3]The equality used to compute non-linearity in the Rho-Calculus relies on the equality modulo the theory $\mathbb{T}$, modulo which matching is performed.

$$rev \triangleq rec(S) \twoheadrightarrow \left(\begin{array}{l} ReverseAux(\texttt{Empty}\ L_2) \twoheadrightarrow L_2 \\ ReverseAux(\texttt{Cons}(X\ M)\ L_2) \twoheadrightarrow S.rec\ ReverseAux(M\ (\texttt{Cons}(X\ L_2)) \end{array} \right)$$

$$reverse \triangleq L \twoheadrightarrow rev.rec\ ReverseAux(L\ \texttt{Empty})$$

Mapping

The last function is the usual map function on lists that applies a given function to each element of the list.

```
let rec map f l = match l with
    [] -> []
  |x::xl -> (f x)::(map f xl);;
```

We will give the purified version of the definition, erasing all the administrative sub-terms. The addition of them is an easy exercise. The map function does not cause any trouble in the Rho-Calculus since higher-order functionality is allowed.

$$Map(F\ \texttt{Empty}) \twoheadrightarrow \texttt{Empty}$$
$$Map(F\ (\texttt{Cons}(X_2\ M)) \twoheadrightarrow \texttt{Cons}((F\ X)\ \ (Map(X\ M))$$

2.2 Higher-order programming

2.2.1 Using continuations

To illustrate the higher-order capabilities of the Rho-Calculus, we give an example of a function written in the continuation passing style. We recall that each function has a new parameter corresponding to the "future" of the computation result. Of course thanks to this mechanism each recursive function can be rewritten in a tail recursive version. We give here two examples. The first deals with the function *append* on list which is here tail recursive. The second example shows how to write a simple recursive function simulating a *do while*.

```
let rec append l m k = match l with
    []    -> (k m)
  |x::l  -> append l m (fun r -> k (cons x r));;

let rec do_while  test body = if test() then body  (fun r -> (do_while
test body)) else ();;
```

The *append* function can, of course, be easily written in the Rho-Calculus.

$$Append(\texttt{Empty}\ M\ K) \twoheadrightarrow (K\ M)$$
$$Append(\texttt{Cons}(X\ L)\ M\ K) \twoheadrightarrow Append(L\ M\ (R \twoheadrightarrow K\ (\texttt{Cons}(X\ R))))$$

The *do while* function makes sense only when side-effects are allowed. In the presented Rho-Calculus, we do not dispose of side effects. But in [LS03], one studies a pattern-matching based calculus with side-effects. Its presentation is not in the scope of this report but the encoding of the *do while* O'CAML function in this calculus is easy.

2.2.2 Using strategies

As such, the notion of rewrite strategy is in everyday use: from normalization strategies to optimal strategies, from leftmost-innermost to lazy ones. Typically, programming languages use call by value or lazy strategies. Automated theorem provers use depth-first or breadth-first proof search strategies. In many cases these strategies are built-in and the users of the programming language or of the prover have no way to adapt them to their specific use.

Since strategies allow us to control the schematically and locally described transformations, it is conceptually and practically fundamental to permit the users to define their own ones. This is what proof assistants like LCF, Coq, ELF permit under the name of tactics and tacticals. In programming languages, such a control can be reached using reflexivity like in LISP and Maude or explicitly using a dedicated language like in ELAN or Stratego.

It is quite surprising to see the emergence of alternatives in order to go through the defined strategy of the considered programming language. A nice example is the strategy Cut in Prolog.

Suppose we want to deal with logic formulae. The common characteristics of most logic formula handling programs (LFHP):

- **symbolic computation** Most LFHP involve some forms of manipulation of expressions. This often takes the form of rewriting combining with strategies. For example, one often wants to transform logic formulae in normal form (disjunctive or conjunctive). In language like O'CAML, Java, ... it is boring to express these computations because one needs to encode rewriting, *e.g.*, traversal operators. Moreover, in such languages, we only have a syntactic matching whereas it is very handy to use an associative commutative matching for $\mathfrak{and}, \mathfrak{or} \ldots$. If such a matching is used we can write a rewrite systems with 4 rules to compute the disjunctive normal form:

$$\begin{array}{lcl} \mathfrak{not}(\mathfrak{not}(X)) & \to & X \\ \mathfrak{not}(\mathfrak{or}(X,Y)) & \to & \mathfrak{and}(\mathfrak{not}(X),\mathfrak{not}(Y)) \\ \mathfrak{not}(\mathfrak{and}(X,Y)) & \to & \mathfrak{or}(\mathfrak{not}(X),\mathfrak{not}(Y)) \\ \mathfrak{and}(X,\mathfrak{or}(Y,Z))) & \to & \mathfrak{or}(\mathfrak{and}(X,Y),\mathfrak{and}(X,Z)) \end{array}$$

- **imperative data structures** LFHP usually employ special-purpose data structures for efficiency, *e.g.*, hash tables. Implementing such data structures in a programming language is most natural if the language has mutable references or some other imperative construct.

To the knowledge of the author, it does not exist any programming language meeting the two needs. The Rho-Calculus should be a paradigm for new programming languages involving these two components since we have a imperative version of the Rho-Calculus [LS03] and since strategies can be encoded in the Rho-Calculus [CKLW03].

2.3 Type Inference for the Rho-Calculus

Various typed versions of the Rho-Calculus have been or are being developed, for different purposes: normalization, automated deduction, typed programming disciplines...

Here we present a simple (first-order) type inference system *à la* Curry which affects types to some terms of the Rho-Calculus. The rules are adapted from the polymorphic type system for the Rho-Calculus [CKL02], and from a first-order type system *à la* Church for the Rho-Calculus [CLW03]. As far as programming is concerned, the main interest of type systems for the Rho-Calculus is to forbid errors as the addition of an integer with a string.

In a nutshell, monomorphic (called $\rho_{\to}$) and polymorphic (called $\rho 2$) type systems for the Rho-Calculus have the "nice" property that they do not guarantee termination since it allows the encoding of many fixpoints. Therefore, they are suitable to be taken as a foundational basis of realistic rewriting- and functional-based programming languages. The type system is adapted from the simply typed λ-calculus by generalizing

the abstraction rule for patterns:

$$\frac{\begin{array}{l}\Gamma,\Delta \vdash A : \varphi \\ \Gamma,\Delta \vdash B : \psi \qquad \mathcal{D}om(\Delta) = \mathcal{FV}ar(A)\end{array}}{\Gamma \vdash A \twoheadrightarrow B : \varphi \twoheadrightarrow \psi}\ (Abs) \qquad \frac{\Gamma \vdash B_1 : \varphi \twoheadrightarrow \psi \quad \Gamma \vdash B_2 : \varphi}{\Gamma \vdash B_1\ B_2 : \psi}\ (Appl)$$

The full type system can be found in [CLW03] with all its properties and proofs. As a simple example, we present here a term inspired by the famous $\omega\ \omega$ term of the untyped λ-calculus. The encoding is obtained using the constant f whose type is $(\alpha \twoheadrightarrow \alpha) \twoheadrightarrow \alpha$. We define:

$$\omega_f \triangleq f(X) \twoheadrightarrow X\ f(X)$$

and we can type check it the following way (we suppose the constant type is given implicitly in a suitable signature (here omitted)):

$$\frac{\dfrac{(1)}{\vdash \omega_f : \alpha \twoheadrightarrow \alpha} \qquad \dfrac{\vdash f : (\alpha \twoheadrightarrow \alpha) \twoheadrightarrow \alpha \quad \vdash \omega_f : \alpha \twoheadrightarrow \alpha}{\vdash f(\omega_f) : \alpha}}{\vdash \omega_f\ f(\omega_f) : \alpha}$$

where (1) is

$$\frac{\dfrac{\begin{array}{l}X{:}\alpha \twoheadrightarrow \alpha \vdash f : (\alpha \twoheadrightarrow \alpha) \twoheadrightarrow \alpha \\ X{:}\alpha \twoheadrightarrow \alpha \vdash X : \alpha \twoheadrightarrow \alpha\end{array}}{X{:}\alpha \twoheadrightarrow \alpha \vdash f(X) : \alpha} \qquad \dfrac{\begin{array}{l}X{:}\alpha \twoheadrightarrow \alpha \vdash X : \alpha \twoheadrightarrow \alpha \\ X{:}\alpha \twoheadrightarrow \alpha \vdash f(X) : \alpha\end{array}}{X{:}\alpha \twoheadrightarrow \alpha \vdash X\ f(X) : \alpha}}{\vdash \omega_f : \alpha \twoheadrightarrow \alpha}$$

Finally, the only reduction path from $\omega_f\ f(\omega_f)$ does not end:

$$\begin{array}{rcl}\omega_f\ f(\omega_f) & = & (f(X) \twoheadrightarrow X\ f(X))\ f(\omega_f) \\ & \mapsto_\rho & [f(X) \ll f(\omega_f)].(X\ f(X)) \\ & \mapsto_\sigma & (X\ f(X))[\omega_f/X] \equiv \omega_f\ f(\omega_f) \\ & \mapsto_\rho & \ldots\end{array}$$

This kind of typed fix-point is made possible because any well-formed type can be chosen for the constants of the calculus. Here, f has type $(\alpha \twoheadrightarrow \alpha) \twoheadrightarrow \alpha$, so at the type level it makes a function over the set α look like an object in α.

A similar method can be used in ML in order to build a fix-point without using `let rec`. Instead of defining a constant with an arbitrary type, we just have to define a particular inductive type whose unique constructor behaves like f:

Example 2.1 (Fix points in O'CAML - without `let rec`)
```
type t = F of (t -> t);;
let omega x = match x with (F y) -> y (F y);;
```
Then the evaluation of `omega (F omega)` *does not terminate.*

2.4 Encoding Abadi and Cardelli's Object-Calculus

As fully detailed in [CKL01a], the (untyped) Rho-Calculus allows one to encode some classical object-calculi like $\varsigma\mathcal{O}bj$ [AC96]. We will show that this is still true for the typed $\rho_{\twoheadrightarrow}$ and we will use a more concise encoding by better exploiting the pattern matching facilities of the Rho-Calculus. A method is encoded as $m(S) \twoheadrightarrow B_m$[4], where the constant m is the name of the method, the variable S will play the role of the meta-variable this and B_m is a term encoding the body of the method. An object *obj* is then a structure

[4] In [CKL01a], the original encoding was $m \twoheadrightarrow S \twoheadrightarrow B_m$, needing two reduction steps where one is enough with our enhanced encoding.

filled with methods. The method $meth$ is then called by Kamin's self application - as for the encoding of fix-points.

$$\begin{array}{rcl} obj\ meth(obj) & \equiv & (\ldots, meth(S) \twoheadrightarrow B_{meth}, \ldots)\ meth(obj) \\ & \mapsto_{\beta\delta} & [meth(S) \ll meth(obj)].B_{meth} \\ & \mapsto_{\beta\delta} & B_{meth}[obj/S] \end{array}$$

Observe that here other methods fail because the equation $(m(S) \ll meth(obj))$ has no solution for all $m \not\equiv meth$. These "dirty" results are all "dropped away" by the equivalence $=_{\mathsf{stuck}}^{\mathsf{no}}$ whose task is to eliminate all definitive matching failures. The variable S is indeed instantiated with obj in the body of the method, allowing all the usual operations on the meta-variable this.

As such, the previous example can be typed in the Rho-Calculus as follows: lab is the constant type of labels, S has type $lab \twoheadrightarrow \varphi$, and φ is the type of B_{meth}. For the sake of simplicity, we suppose obj has just one method triggered by the constant $meth^{(lab \twoheadrightarrow \varphi) \twoheadrightarrow lab}$. For lack of space, we do not give the entire type derivations. Type decorations are omitted.

$$\frac{\vdash meth(S) : lab \quad \vdash B_{meth} : \varphi}{\vdash meth(S) \twoheadrightarrow B_{meth} : lab \twoheadrightarrow \varphi}\ (Abs)$$

Considering the meaning we want for S, it is sound that obj and S have the same type. Then $obj.meth \triangleq obj\ meth(obj)$ can be typed as follows:

$$\frac{\vdash obj : lab \twoheadrightarrow \varphi \quad \vdash meth(obj) : lab}{\vdash obj\ meth(obj) : \varphi}$$

It is worth noting that our type system is not "orthodox" in the object-oriented sense; more precisely its main goal is not to capture message-not-understood run-time errors (they are directly translated into run-time matching failures), but to guarantee data-flow consistency. The notion of this can be easily captured in our dynamic semantics but it is totally lost in the static semantics. Moreover, the type rule for structures seems rather restrictive: it forces all the methods within a given object to yield results with the same type, which is obviously not the case in practical object-oriented programming. To have more general objects, we would like to enhance the type system with a more liberal rule for typing structures with a different types, like:

$$\frac{\vdash A : \varphi \quad \vdash B : \psi}{\vdash A, B : \varphi \wedge \psi}\ (Struct_{\wedge})$$

It must be noticed that all previous examples can be easily encoded. For example, for the addition we must define: $rec^{(lab \twoheadrightarrow \iota \twoheadrightarrow \iota) \twoheadrightarrow \iota}$ and $add^{\iota \twoheadrightarrow \iota \twoheadrightarrow \iota}$.

Chapter 3

The $\rho_{\mathtt{X}}$-calculus

When we reduce a term using the σ rule, in one step we go from the explicit application of a matching constraint to the meta application of the corresponding substitution. This means that, in one step, we compute the substitution from the matching constraint and apply it. Actually, all these computations can be decomposed into two steps, *i.e.*, two reduction rules. We need to extend the syntax to deal with substitutions. In an informal way, we define here the explicit applications of substitutions by $\sigma.C$ and the meta application $\sigma(C)$.

$$\begin{array}{lllll} (Matching) & [A \ll B]C & \rightarrow & \sigma.C & \\ & & & \texttt{if } \sigma = \mathcal{S}ol(A \ll B) \\ (Apply) & \sigma.C & \rightarrow & \sigma(C) \end{array}$$

This decomposition does not mean that the matching computations to go from constraints to substitutions and to apply a substitution are explicit but this means that they are separated. Depending on the matching theory, these computations can be really significant and we want to go further on, *i.e.*, we want to make explicit all the computations, to use the Rho-Calculus as a theoretical back-end for an implementation. Such an implementation with explicit computations and applications of constraints should be based on a scheduler that switches regularly between computations on constraints, applications of substitutions and the ρ, δ rules.

In [Cir00], Cirstea proposes a rewriting calculus with explicit substitutions. This calculus is mainly an extension of the $\lambda\sigma$-calculus to the studied framework. It is called the $\rho\sigma$-calculus. In the $\rho\sigma$-calculus matching constraints are not an explicitly part of the calculus and thus the computation of the substitution from the matching constraints is not part of the syntax. Thus the problem is quite different since this calculus makes explicit the substitution application but not the computations from constraints to substitutions. In [Ngu01], Nguyen studied a cooperation Coq-ELAN to automate proof assistants where the $\rho\sigma$-calculus have been intensively used to represent proof terms of rewrite derivations.

The explicit substitutions have been widely studied. We can cite for example [ACCL91, Les94, CHL96, Ros96]. This framework is a nice tool to deal with higher order unification [DHK00, DHKP96, ARK00] or to represent incomplete proofs in type theory [Muñ97].

In the [BKK98a], a new calculus called the PSA-calculus is introduced. The explicit application of a rewrite rule - and thus the explicit matching - coined for the first time in this ancestor of the Rho-Calculus. Nevertheless, it was a first approach to make explicit the rewriting and thus the calculus is really less powerful than the current Rho-Calculus. For example, the PSA-Calculus is not enough powerful to allow strategies as explicit objects and thus there is a hierarchy between rules and strategies.

From constraints to substitutions

As we said before, the Rho-Calculus is well-suited to deal with errors, represented by constraints without any solutions (*e.g.* $+ \ll_{AC} *$). This means that there exist constraints that do not represent substitutions. In our wish to make explicit the application of constraints, it is natural to ask if it makes sense to propagate

constraints without any solutions. Of course, in the λ-calculus, such questions do not arise since we only consider substitutions and, to propagate a substitution (*i.e.*, to apply it step by step) always makes sense. The author believes that *we must not propagate constraints without any solutions.* Otherwise, we would lose the error's location and we would obtain *in fine* a term with constraints without any solutions applied on each leaf of the term. The information contained in such a term seems useless to analyze the error and, for debugging reasons, we do not want to lose the error's location. Moreover it seems to be quite difficult to compose constraints if we do not know if they are satisfiable. For example, what should be the result of the composition of the constraint $X \ll a \wedge X \ll b$ with $Z \ll X$?

We simplify an arbitrary constraint until we possibly find a $\mathbf{g}$ ("good") sub-part of the constraint (*i.e.* a part of the constraint which can be transformed into a substitution) which is independent of the other part of the constraint. This part of the constraint can now be applied. The remaining part of the constraint is unchanged. For example, in the constraint $X \ll A \wedge Y \ll B$ the subpart $X \ll A$ is $\mathbf{g}$ and is independent of $Y \ll B$ since $X \neq Y$ and $X \notin B$ by α-conversion. In the constraint $X \ll A \wedge X \ll B$, the subpart $X \ll A$ is $\mathbf{g}$ but is not independent of $X \ll B$. An example of a subpart which is not $\mathbf{g}$ is $g(X) \ll h(a)$.

The question is now to find a clever way to identify $\mathbf{g}$ constraints. We can mainly distinguish between two approaches. First, a $\mathbf{g}$ part of a constraint is a matching constraint $X \ll A$ and such a constraint is independent of the remaining part $\mathcal{C}$ of the initial constraint if $X \notin \mathcal{D}om(\mathcal{C})$. This leads to a calculus *à la* $\lambda_{\mathtt{x}}$-calculus quite simple but without substitution compositions. We will naturally call it the $\rho_{\mathtt{x}}$-calculus. In the $\rho_{\mathtt{x}}$-calculus, the constraint $X \ll A \wedge Y \ll B$ is decomposed into two $\mathbf{g}$ parts: $X \ll A$ and $Y \ll B$. Thus to apply the constraint to a term, we will need to visit the whole term twice.

Secondly, we will take a more general approach allowing more sophisticated $\mathbf{g}$ constraints. We will obtain a calculus near from the λ_σ-calculus[1], *i.e.*, a calculus with explicit constraint handling and allowing substitution composition. We will call it the $\rho_{\mathtt{xC}}$-calculus. In this calculus, the constraint $X \ll A \wedge Y \ll B$ is identified as a $\mathbf{g}$ part. Now, we can apply it in one traverse of the term.

In our wish to make explicit the application of constraints, the symbol "[ ]" should not be in the syntax of the calculus and we would rather overload the functional application "$\bullet$". However, we need a special symbol "{}" to denote the application of substitutions. Otherwise, a terminating and confluent rule system for the explicit application of substitutions seems difficult to achieve. For example, the application of the constraint $f(X\ Y) \ll f(a\ b)$ to X is denoted $(f(X\ Y) \ll f(a\ b))\ X$. The application of the substitution $X \ll A$ to X is written $\{X \ll A\}X$.

The matching theory

In our calculus, we will mainly distinguish between three different kinds of reduction:

- Reductions corresponding to applications of the ρ and δ rules.
- Reductions needed to go from constraints to substitutions. The first part of these reductions simplify the constraint until getting a "normal form". For example, $f(X\ Y) \ll f(a\ b) \mapsto X \ll a\ \wedge\ Y \ll b$ using a decomposition rule. The second part of the rules decide whether a constraint is a substitution or not, going from constraint applications to substitution applications.
- Reductions related to the application of substitutions. For example, the application of the substitution $X \ll A$ to X denoted $\{X \ll A\}X$ is reduced to A.

The second set is strongly related to the matching theory, parameter of the Rho-Calculus. It does not exist any generic algorithm to decide/solve matching constraints. Thus, we need to restrict ourselves to a given class of theories. We could present the $\rho_{\mathtt{x}}$-calculus for syntactic theories [Kla92, Kir85] using the generic matching algorithm. But such a presentation should be rather tedious to read. So, we will present the $\rho_{\mathtt{x}}$-calculus for the empty theory[2].

[1] But we will not deal with variable names, α-conversion...

[2] The reader familiar with these notions can generalize very easily our presentation replacing the rules corresponding to the decomposition to the generic rules for the syntactic theories recovering C, AC, ACI ... matching theories.

3.1 Syntax

The syntax of the $\rho_{\mathbf{x}}$-calculus is given is Figure 3.1. The symbol $\wedge$ is supposed to be ACI (associative

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

Figure 3.1: Syntax of the $\rho_{\mathbf{x}}$-calculus

commutative and idempotent).

3.2 Semantics

We will present the semantics of the calculus. We refer the reader to all the examples given in the next section.

We want to make explicit the matching. One nice starting point is to look at one of the matching algorithms one can find in rewriting books (modulo different notations). We give one of us in Figure 3.2. Of

$Decomposition$	$f(A_1, \ldots, A_n) \ll f(A'_1, \ldots, A'_n) \wedge \mathcal{C}$	$\rightarrow$	$A_1 \ll A'_1 \wedge \ldots \wedge A_n \ll A'_n \wedge \mathcal{C}$
$SymbolClash$	$f(A_1, \ldots, A_n) \ll g(A'_1, \ldots, A'_m) \wedge \mathcal{C}$	$\rightarrow$	$\mathbb{F}$ if $f \neq g$
$SymbolVariableClash$	$f(A_1, \ldots, A_n) \ll X \wedge \mathcal{C}$	$\rightarrow$	$\mathbb{F}$ if $X \in \mathcal{X}$
$MergingClash$	$X \ll A \wedge X \ll B \wedge \mathcal{C}$	$\rightarrow$	$\mathbb{F}$ if $A \neq B$

Figure 3.2: Matching rules in the empty theory

course, we must modify these rules to adapt them to our context.

For a given term, deciding whether a constraint is solvable is, in general, an undecidable problem. We will propose "careful" algorithms: A decomposition will be done iff no reduction at a different position would influence the reduction. We can take the example of the constraint $a \ll X$ that is not solvable but that might become solvable if the variable X is instantiated by a. This appends when in the term $(X \twoheadrightarrow (a \twoheadrightarrow b)X)a$ we reduce first the inner ρ redex. We obtain the term $(X \twoheadrightarrow (a \ll X)\ b)\ a$. Then we must wait for the possible information on X and we cannot reduce the sub-term $(a \ll X)b$ yet. So, it does not make sense to reduce the constraint $a \ll X$ to a failure. Thus, the $(SymbolVariableClash)$ rule does not make any sense in our context.

To simplify the constraint $f\ a \ \ll\ (x \twoheadrightarrow fx)a$, using a decomposition rule, in the conjunction $(f \ll x \twoheadrightarrow fx) \wedge (a \ll a)$ does not seem to be relevant. In fact, if we reduce $(x \twoheadrightarrow f\ x)\ a$ on $f\ a$, the initial constraint

becomes equivalent to the identity although the simplified constraint is definitively not solvable. Thus, we will not decompose the symbol • but we would rather decompose, as in classical rewriting, $f(A_1 \dots A_n)$[3]. This leads to the rule $(Decompose_{\mathcal{F}})$. Neither will we decompose the "$\rightarrowtriangle$" since a nice decomposition of such a symbol is still an open question. However, we will decompose the ",". The theory associated to the structure operator is also a parameter of the Rho-Calculus. In our seek to be easy to read we will deal with an empty theory for ",". The same remarks, as for the matching theory, concerning a possible generalization to other theories can be done here. In the case of the empty theory we obtain straightforwardly the rule $(Decompose_{,})$.

One question still remains: how to deal with non-linearity? There are many answers but the simplest one is: wait until the problem becomes linear. Surprising as it may seem, this is the best way to deal with this difficulty. To be more precise, when we find a constraint looking like $X \ll A \ \wedge \ X \ll B$ we will try to reduce A and/or B until we obtain two equal terms and thus, *in fine* we have the constraint $X \ll A'$ - since $\wedge$ is assumed to be idempotent. We can look at Example 3.3.

As said before, we simplify an arbitrary constraint by using the decomposition rules until we possibly find a sub-part of the constraint in "solved form" - that is looking like $X \ll A$ - which is independent of the remaining part $\mathcal{C}$ of the initial constraint *i.e.*, $X \notin \mathcal{D}om(\mathcal{C})$. We obtain the rule $(ToSubst_{\wedge})$. The two other simplifying rules $(ToSubst_{\ll})$ and $(SimplifyId)$ play the same rôle but they deal with atomic constraints - *i.e.*, constraints equal to $X \ll A$ - and "identity" constraints - *i.e.*, constraints reduced to the same constant matching: $a \ll a$.

The application of a substitution is denoted by $\{X \ll A\}B$ and is defined by straightforward rules. When we apply a substitution to an abstraction then the substitution does not affect the left hand side of the abstraction (by α-conversion). The same remark can be done for the application of a substitution to a matching equation. The small-step reduction semantics of the $\rho_{\mathbf{x}}$-calculus is given in Figure 3.3.

3.3 Examples

In this section we will give many examples of the behavior of our calculus. First, we take a very simple example. This example corresponds to the translation in the $\rho_{\mathbf{x}}$-calculus of the following derivation of the λ_x-calculus. In Section 3.5, we will recall very briefly the definition of the λ_x-calculus and we will do a more precise comparison between the λ_x-calculus and the $\rho_{\mathbf{x}}$-calculus.

$(\lambda X.X)Y$
$\mapsto_{\beta} \quad X\langle X := Y\rangle$
$\mapsto_{\beta} \quad Y$

Example 3.1 (Derivation from the λ_x-calculus)
$(X \rightarrowtriangle X)Y$
$\mapsto_{\rho} \quad (X \ll Y) \ X$
$\mapsto_{ToSubst_{\ll}} \quad \{X \ll Y\}X$
$\mapsto_{Eliminate_{\mathcal{X}}} \quad Y$

In our calculus the reduction is longer because of the step from constraints to substitutions. The next example illustrates the atomicity of the application of a rewrite rule.

Example 3.2 (Application of a rewrite rule)
$(\pi_1(X \ Y) \rightarrowtriangle X) \ \pi_1(a \ b)$
$\mapsto_{\rho} \quad (\pi_1(X \ Y) \ll \pi_1(a \ b)) \ X$

[3]Where n is the arity of f.

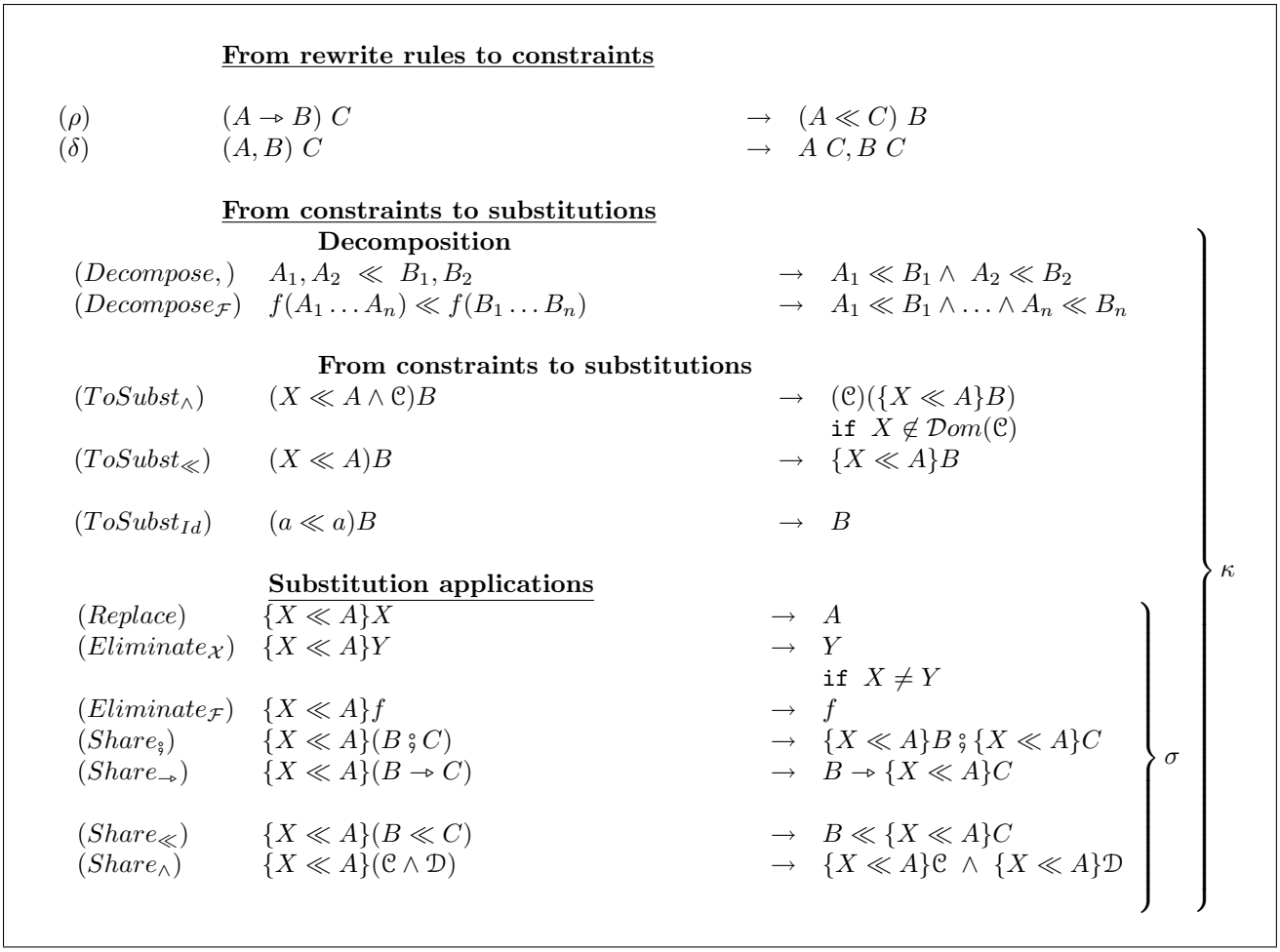
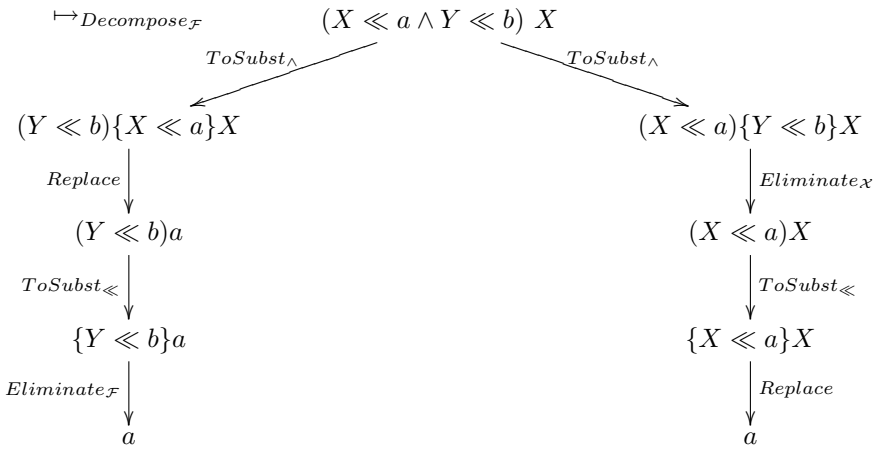


Figure 3.3: Small-step reduction semantics of the $\rho_{\mathtt{x}}$-calculus



In the previous example, the computations to go from constraints to substitutions are very simple. We will see later that the computation can be really important *e.g.*, if the commutative theory is used - see Example 3.29. We can apply a rewrite system as in the plain Rho-Calculus. First we distribute the rewrite rules to the term and secondly we apply each rule as in the previous example.

Example 3.3 (Application of a non-linear rewrite rule)
$(\mathfrak{xor}(X\ X) \twoheadrightarrow \mathfrak{ff})\ \mathfrak{xor}(\mathfrak{tt}\ \mathfrak{tt})$
$\mapsto_{\rho}$ $\big(\mathfrak{xor}(X\ X) \ll \mathfrak{xor}(\mathfrak{tt}\ \mathfrak{tt})\big)\ \mathfrak{ff}$
$\mapsto_{Decompose_{\mathcal{F}}}$ $\big(X \ll \mathfrak{tt} \wedge X \ll \mathfrak{tt}\big)\ \mathfrak{ff} = \big(X \ll \mathfrak{tt}\big)\ \mathfrak{ff}$ since $\wedge$ is idempotent
$\mapsto_{ToSubst_{\ll}}$ $\{X \ll \mathfrak{tt}\}\mathfrak{ff}$
$\mapsto_{Eliminate_{\mathcal{F}}}$ $\mathfrak{ff}$

As in the plain Rho-Calculus, a run time error can be caused by a clash on symbols. Nevertheless in the plain Rho-Calculus since the matching is not explicit, when we deal with big terms, it can be difficult to find the error if it occurs at a deep position. Thanks to the explicit decomposition of matching equations the term is decomposed until an error is obtained. One nice example is the following.

Example 3.4 (Run-time error: matching failure)
Let us consider the following term $f(g(h(X)\ i(a)\ k(l(Y)))\ c) \twoheadrightarrow \pi(X\ Y) \quad f(g(h(a)\ i(b)\ k(l(b)))\ c)$. *First, we give the reduction of this term in the plain Rho-Calculus and afterwards in the* $\rho_{\mathfrak{x}}$*-calculus.*

$$\left.\begin{array}{l} f(g(h(X)\ i(a)\ k(l(Y)))\ c) \twoheadrightarrow \pi(X\ Y) \quad f(g(h(a)\ i(b)\ k(l(b)))\ c) \\ \mapsto_{\rho} [f(g(h(X)\ i(a)\ k(l(Y)))\ c) \ll f(g(h(a)\ i(b)\ k(l(b)))\ c)]\ \pi(X\ Y) \end{array}\right\} \text{In the plain Rho-Calculus}$$

$$\left.\begin{array}{l} f(g(h(X)\ i(a)\ k(l(Y)))\ c) \twoheadrightarrow \pi(X\ Y) \quad f(g(h(a)\ i(b)\ k(l(b)))\ c) \\ \mapsto_{\rho} \quad f(g(h(X)\ i(a)\ k(l(Y)))\ c) \ll f(g(h(a)\ i(b)\ k(l(b)))\ c) \quad \pi(X\ Y) \\ \mapsto\!\!\!\!\mapsto_{Decompose_{\mathcal{F}}} (X \ll a \wedge \underline{a \ll b} \wedge Y \ll b \wedge c \ll c) \quad \pi(X\ Y) \end{array}\right\} \text{In the } \rho_{\mathfrak{x}}\text{-calculus}$$

The matching fails because we tried to match a over b. This error is really easy to read in the $\rho_{\mathfrak{x}}$-calculus, unlike in the Rho-Calculus. This nice way to deal with errors is again shown in the following example where the run-time error is due to non-linearity.

Example 3.5 (Run-time error: non-linearity)
$\big(\mathfrak{xor}(X\ X) \twoheadrightarrow \mathfrak{ff}\big)\ \mathfrak{xor}(\mathfrak{tt}\ \mathfrak{ff})$
$\mapsto_{\rho}$ $\big(\mathfrak{xor}(X\ X) \ll \mathfrak{xor}(\mathfrak{tt}\ \mathfrak{ff})\big)\ \mathfrak{ff}$
$\mapsto_{Decompose_{\mathcal{F}}}$ $(X \ll \mathfrak{tt} \wedge X \ll \mathfrak{ff})\ \mathfrak{ff}$
In the $\rho_{\mathfrak{x}}$*-calculus, the non-linearity is easy to detect.*

In the following example, we see how a substitution applies to a delayed matching constraint. We follow the examples presented in Section 1.2.

Example 3.6 (Delayed matching constraint)

$(X \twoheadrightarrow (a \twoheadrightarrow b)\ X)\ a$

$\mapsto_{p}$ $(X \twoheadrightarrow (a \ll X)\ b)\ a$

$\mapsto_{p}$ $(X \ll a)\ \big((a \ll X)\ b\big)$

$\mapsto_{ToSubst_{\wedge}}$ $\{X \ll a\}\big((a \ll X)\ b\big)$

$\mapsto_{Share_{\wr}}$ $\{X \ll a\}(a \ll X)\ \{X \ll a\}b$

$\mapsto_{Share_{\ll}}$ $(a \ll \{X \ll a\}a)\ \{X \ll a\}b$

$\mapsto\!\!\!\!\mapsto_{Eliminate_{\mathcal{F}}}$ $(a \ll a)\ b$

$\mapsto\!\!\!\!\mapsto_{ToSubst_{Id}}$ b

Example 3.7 (A more elaborated reduction)

$(\mathfrak{xor}(Z\ Z) \twoheadrightarrow \mathfrak{ff})\ \mathfrak{xor}((\mathfrak{if}(\mathfrak{tt}\ X\ Y) \twoheadrightarrow X)\ \mathfrak{if}(\mathfrak{tt}\ \mathfrak{tt}\ \mathfrak{ff})\ \mathfrak{tt})$

$\mapsto_{p}$ $\big(\mathfrak{xor}(Z\ Z) \ll \mathfrak{xor}((\mathfrak{if}(\mathfrak{tt}\ X\ Y) \twoheadrightarrow X)\ \mathfrak{if}(\mathfrak{tt}\ \mathfrak{tt}\ \mathfrak{ff})\ \mathfrak{tt})\big)\ \mathfrak{ff}$ The outermost redex

$\mapsto_{Decompose_{\mathcal{F}}}$ $\big(Z \ll (\mathfrak{if}(\mathfrak{tt}\ X\ Y) \twoheadrightarrow X)\ \mathfrak{if}(\mathfrak{tt}\ \mathfrak{tt}\ \mathfrak{ff})\ \wedge\ Z \ll \mathfrak{tt}\big)\ \mathfrak{ff}$

$\mapsto_{p}$ $\Big(Z \ll \big(\mathfrak{if}(\mathfrak{tt}\ X\ Y \ll \mathfrak{if}(\mathfrak{tt}\ \mathfrak{tt}\ \mathfrak{ff})\big)\ X \wedge\ Z \ll \mathfrak{tt}\Big)\ \mathfrak{ff}$ The only possible reduction

$\mapsto_{Decompose_{\mathcal{F}}}$ $\big(Z \ll (\mathfrak{tt} \ll \mathfrak{tt}\ \wedge\ X \ll \mathfrak{tt}\ \wedge\ Y \ll \mathfrak{ff})\ X \wedge Z \ll \mathfrak{tt}\big)\mathfrak{ff}$

$\mapsto_{ToSubst_{\wedge}}$ $(Z \ll (\mathfrak{tt} \ll \mathfrak{tt}\ \wedge\ X \ll \mathfrak{tt})\{Y \ll \mathfrak{ff}\}X \wedge Z \ll \mathfrak{tt})\mathfrak{ff}$

$\mapsto_{ToSubst_{\wedge}}$ $\Big(Z \ll (\mathfrak{tt} \ll \mathfrak{tt})\big(\{X \ll \mathfrak{tt}\}(\{Y \ll \mathfrak{ff}\}X)\big) \wedge Z \ll \mathfrak{tt}\Big)\mathfrak{ff}$

$\mapsto_{SimplifyId}$ $\Big(Z \ll \{X \ll \mathfrak{tt}\}(\{Y \ll \mathfrak{ff}\}X) \wedge Z \ll \mathfrak{tt}\Big)\mathfrak{ff}$

$\mapsto_{Eliminate_{\mathcal{X}}+Replace}$ $\Big(Z \ll \mathfrak{tt} \wedge Z \ll \mathfrak{tt}\Big)\mathfrak{ff} = (Z \ll \mathfrak{tt})\ \mathfrak{ff}$

$\mapsto_{Eliminate_{\mathcal{F}}}$ $\mathfrak{ff}$

3.4 Properties of the $\rho_{\mathfrak{x}}$-calculus

The main theorem we will show is the confluence of the calculus. It will lead us to show that we have a sound substitution application and a well-behaved calculus. It must be noticed that the $\rho_{\mathfrak{x}}$-calculus is the only Rho-Calculus we currently know which is confluent without specific strategies nor restrictions on terms. This means that regulating the matching algorithm is a nice alternative way to obtain conflence.

A nice example is given by extending the decomposition rules for active variables, *i.e.*, we add the rule:

$$(AV) \quad X(A_1 \dots A_n) \ll f(B_1 \dots B_n) \quad \rightarrow \quad X \ll f \wedge A_1 \ll B_1 \wedge \dots \wedge A_n \ll B_n$$

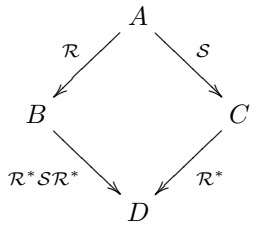
Adding this rule to the $\rho_{\mathfrak{x}}$-calculus does not affect termination and confluence. We have not given this rule in the initial presentation for pedagogic reasons and to show that we can regulate the matching as needed. This higher-order matching should be really powerful in languages like λ-Prolog [Mil03] or ELAN [BKK+98b]. Actually, in these languages we only have first order matching and when we need higher-order matching, we use preprocessing to go from one to another. This preprocessing instantiates all the higher-order rules for all symbols of the given signature. It is not an efficient method and we should rather use a matching handling active variables. One nice example is when one wants to compute commutativity defined with the rule $X(x\ y) \twoheadrightarrow X(y\ x)$.

With the (AV) rule we can have: $(X(A_1\ A_2) \twoheadrightarrow X(A_1\ A_1))\ f(a\ b) \mapsto f(a\ a)$. In the Rho-Calculus "on the market", we do not allow active variables in the left hand side of a rewrite rule[4].

To show the confluence, we will use the Yokouchi's lemma. A proof of this lemma can be found in [CHL96].

Lemma 3.8 (Yokouchi) [YH90]) *Let $\mathcal{R}$ and $\mathcal{S}$ be two relations defined on the same set X, $\mathcal{R}$ being confluent and strongly normalizing, and $\mathcal{S}$ verifying the diamond property. Suppose moreover that the following diagram holds:*

[4]Since in these calculi, we decompose the application operator. If active variables were allowed, we would be able to reduce $(X\ Y \twoheadrightarrow X)\ \ (a \twoheadrightarrow g(b))\ a$ to $a \twoheadrightarrow g(b)$ or to b. The calculus would then be no longer confluent.



Then the relation $\mathcal{R}^*\mathcal{S}\mathcal{R}^*$ is confluent.

We will apply the lemma for $\mathcal{R} = \kappa$ and for $\mathcal{S}$ equal to the parallelization of $\rho\delta$. Let us show the hypotheses of the lemma.

3.4.1 Termination of $\longrightarrow_\kappa$

First of all, we will show that $\longrightarrow_\kappa$ is strongly normalizing.

Lemma 3.9 *$\longrightarrow_\kappa$ is strongly normalising*

Proof : We need to define a measure ζ on terms representing the number of different variables in a left-hand side of constraint matching equations - but not of substitutions. For example, X is not in a left hand side of a constraint matching equation in $\{X \ll A\}B$ whereas it is in $(X \ll A)B$. More precisely, ζ is the cardinality of m defined by:

$m(X) = \emptyset$ for all $X \in \mathcal{X}$

$m(k) = \emptyset$ for all $k \in \mathcal{K}$

$m(A \twoheadrightarrow B) = m(A) \cup m(B)$

$m(A \fatsemi B) = m(A) \cup m(B)$

$m(\mathcal{C}\ A) = m(\mathcal{C}) \cup m(A)$

$m(\{X \ll A\}B = m(A) \cup m(B)$

$m(\{X \ll A\}\mathcal{C}) = m(A) \cup m(\mathcal{C})$

$m(A \ll B) = \mathcal{FV}ar(A) \cup m(A) \cup m(B)$

$m(\mathcal{C} \wedge \mathcal{D}) = m(\mathcal{C}) \cup m(\mathcal{D})$

We show that $\longrightarrow_\kappa$ is strongly normalizing using the lexicographic product of ζ and $\succ$ where $\succ$ is the recursive path ordering obtain from the precedence $\ll\ >\ \wedge \qquad \{\}\ >\ \fatsemi \qquad \{\}\ >\ \twoheadrightarrow \qquad \{\}\ >\ \ll$ $\{\}\ >\ \wedge$ and with the status "multiset" for the symbol $\{\}$.

	ζ	$\succ$
$Decompose_,$	$=$	$>$
$Decompose_{\mathcal{F}}$	$=$	$>$
$ToSubst_\wedge$	$>$	
$ToSubst_\ll$	$>$	
$ToSubst_{Id}$	$=$	$>$
$Replace$	$=$	$>$
$Eliminate_{\mathcal{X}}$	$\geq$	$>$
$Eliminate_{\mathcal{K}}$	$\geq$	$>$
$Share_\fatsemi$	$=$	$>$
$Share_\twoheadrightarrow$	$=$	$>$
$Share_\ll$	$=$	$>$
$Share_\wedge$	$=$	$>$

□

3.4.2 Well-behaved properties of the $\rho_{\mathtt{x}}$-calculus

We will establish the relationship between explicit and pure substitutions: we show that the behavior of the substitutions $\{X \ll A\}B$ relates well to the usual meta-substitution $(X := A)B$. We first prove the result for pure terms[5].

Lemma 3.10 *Let A, B be pure terms and $\mathcal{C}$ be a pure constraint. Then:*

- $\{X \ll A\}B \longrightarrow_{\sigma}^{*} (X := A)B$
- $\{X \ll A\}\mathcal{C} \longrightarrow_{\sigma}^{*} (X := A)\mathcal{C}$

Proof : We prove the statements together. Let U stands for either B or $\mathcal{C}$. We proceed by induction on $size(U)$ where $size(U)$ is the size of U, that is, the number of symbols occurring in it.

We distinguish cases according to the structure of U. We start with terms.

1. $U = X$ then $\{X \ll A\}X \longrightarrow_{\sigma} A \equiv (X := A)X$
2. $U = B_1 \S B_2$ then $\{X \ll A\}(B_1 \S B_2) \longrightarrow_{\sigma} \{X \ll A\}B_1 \S \{X \ll A\}B_2$. By IH, $\{X \ll A\}B_1 \longrightarrow_{\sigma}^{*} (X := A)B_1$ since of course B_1 is pure. In the same way, $\{X \ll A\}B_2 \longrightarrow_{\sigma}^{*} (X := A)B_2$. So, $\{X \ll A\}(B_1 \S B_2) \longrightarrow_{\sigma}^{*} (X := A)B_1 \S (X := A)B_2 \ = (X := A)(B_1 \S B_2)$
3. The other cases are similar to the previous ones. □

Definition 3.11 ($\longrightarrow_{\sigma}$) *The rewrite system corresponding to the substitution applications is denoted by σ. See Figure 3.3.*

Lemma 3.12 (Termination and confluence of $\longrightarrow_{\sigma}$) *The relation $\longrightarrow_{\sigma}$ is strongly normalizing and confluent.*

Proof : The termination follows from Lemma 3.9. The system is orthogonal thus confluent. □

Definition 3.13 ($\downarrow_{\sigma}$) *We will denote $\downarrow_{\sigma}(A)$ the normal form of A w.r.t. σ.*

Lemma 3.14 (Representation lemma) *For all terms A, B and variable X,*
$\{X \ll A\}B \longrightarrow_{\sigma}^{*} (X :=\downarrow_{\sigma}(A)) \downarrow_{\sigma}(B)$

Proof : This is an easy consequence of the previous lemmas. Since $\{X \ll A\}B \longrightarrow_{\sigma}^{*} \{X \ll\downarrow_{\sigma}(A)\} \downarrow_{\sigma}(B)$ and since $\downarrow_{\sigma}(A)$ and $\downarrow_{\sigma}(B)$ are pure terms then we can apply the Lemma 3.10 thus $\{X \ll\downarrow_{\sigma}(A)\} \downarrow_{\sigma}(B) \longrightarrow_{\sigma}^{*} [X :=\downarrow_{\sigma}(A)] \downarrow_{\sigma}(B)$. □

Conservativity

One important property of explicit calculus is the conservativity. This property shows that a reduction in the Rho-Calculus can be simulated in the $\rho_{\mathtt{x}}$-calculus. This property is sometimes called simulation.This is formulated by the following lemma:

Lemma 3.15 (Conservativity)

If $A \rightarrow_{\rho} B$ then $A \rightarrow_{\rho_x}^{*} B$

If $\mathcal{C} \rightarrow_{\rho} \mathcal{D}$ then $\mathcal{C} \rightarrow_{\rho_x}^{*} \mathcal{D}$

Proof : We impose the Rigid Pattern Condition on terms[6]. We prove the statements together as in the previous lemmas. The proof is easy. □

[5] As usual in explicit substitution calculi, the set of pure terms is the subset of terms without explicit substitution "{}".

[6] One main difference between the plain Rho-Calculus and the $\rho_{\mathtt{x}}$-calculus is that we do not decompose, strictly speaking, the application operator. If one allows to decompose the application, of course, one allows more reductions like

3.4.3 Confluence of $\longrightarrow_\kappa$

We first show that $\longrightarrow_\kappa$ is locally confluent by analyzing all critical pairs. To do this, we need a substitution lemma.

Lemma 3.16 (Substitution lemma) *For all terms A, B, C such that $Y \notin \mathcal{FV}ar(A)$ we have:*

$$\{X \ll A\}(\{Y \ll B\}C) \downarrow_\sigma \{Y \ll \{X \ll A\}B\}(\{X \ll A\}C)$$

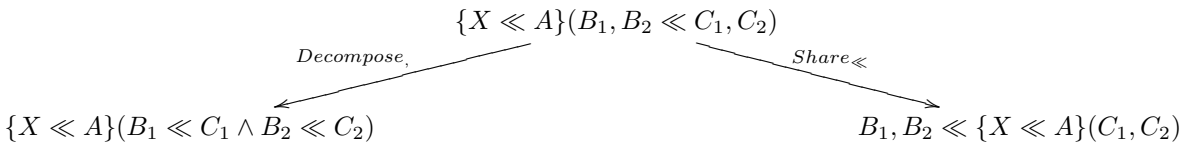
Proof : From the representation lemma by the substitution lemma of the Rho-Calculus.

□

Lemma 3.17 $\longrightarrow_\kappa$ *is locally confluent*

Proof : We analyze all critical pairs.

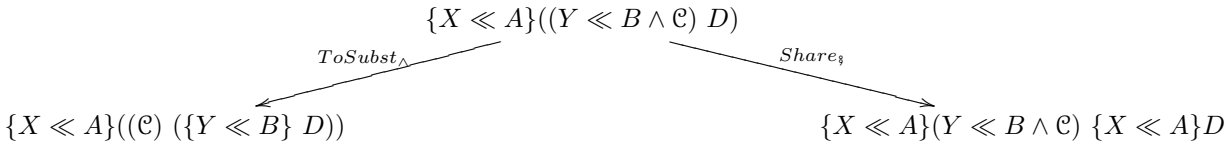
- $(Decompose_,) + (Share_\ll)$



  These terms can both be reduced to $B_1 \ll \{X \ll A\}C_2 \wedge B_1 \ll \{X \ll A\}C_2$.

- $(Deompose_\mathcal{F})+ (Share_\ll)$ Similar to the previous case.

- $(ToSubst_\wedge) + (Share_?)$



  Then $\{X \ll A\}((\mathcal{C})\ (\{Y \ll B\}\ D))$

$$\longrightarrow_\sigma \quad \{X \ll A\}\mathcal{C} \quad \{X \ll A\}(\{Y \ll B\}D)$$

  And $\{X \ll A\}(Y \ll B \wedge \mathcal{C})\ \{X \ll A\}D$

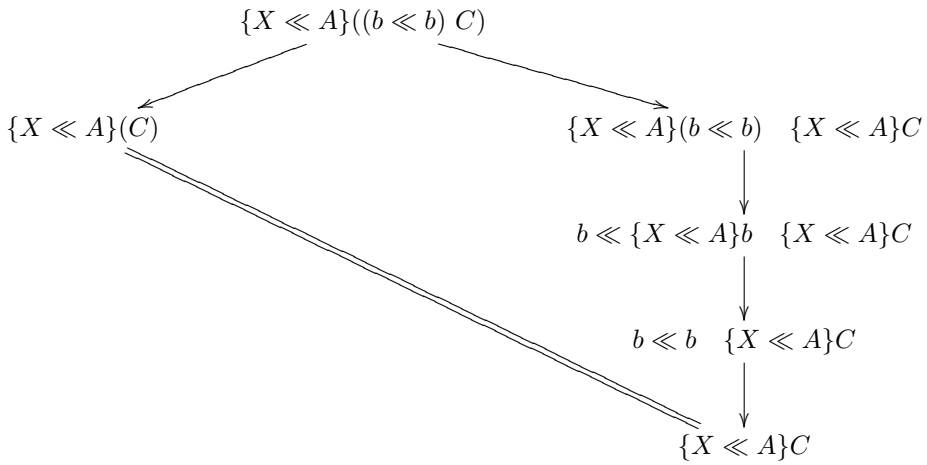
$$\begin{array}{ll} \longrightarrow_\sigma & \{X \ll A\}(Y \ll B) \wedge \{X \ll A\}\mathcal{C}) \quad \{X \ll A\}D \\ \longrightarrow_\sigma & (Y \ll \{X \ll A\}B \wedge \{X \ll A\}\mathcal{C}) \quad \{X \ll A\}D \\ \longrightarrow_\sigma & \{X \ll A\}\mathcal{C} \quad \{Y \ll \{X \ll A\}B\}(\{X \ll A\}D) \end{array}$$

  The substitution lemma concludes the proof since by α-conversion we can suppose that $Y \notin \mathcal{FV}ar(A)$.

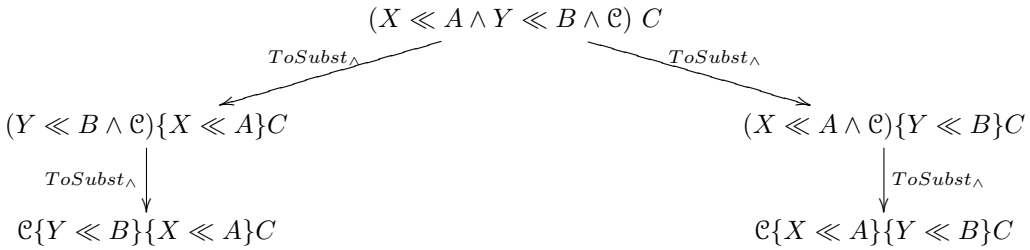
- $(ToSubst_\ll) + (Share_?)$ Similar to the previous case.

$(X\ Y \twoheadrightarrow X)\ \ (a \twoheadrightarrow g(b))\ a \ \mapsto\ a \twoheadrightarrow g(b)$. The author does not know what the decomposition of "•" means in a semantics point of view. We thus restrict ourselves to cases when the decomposition of "•" is not needed and when the decomposition of functions f is sufficient. The Rigid Pattern Condition is the chosen restriction. It is actually always used to obtain confluence Rho-Calculus.

- $(ToSubst_{Id}) + (Share_{\fatsemi})$



- $(ToSubst_{\wedge}) + (ToSubst_{\wedge})$ This critical pair is due to the AC theory of $\wedge$. Actually:



and thanks to well-behaved lemmas, the two terms are joinable.

□

Corollary 3.18 $\longrightarrow_{\kappa}$ *is confluent*

Proof : By Newman's lemma and Lemma 3.17. □

3.4.4 Parallel version of the ρ, δ rules

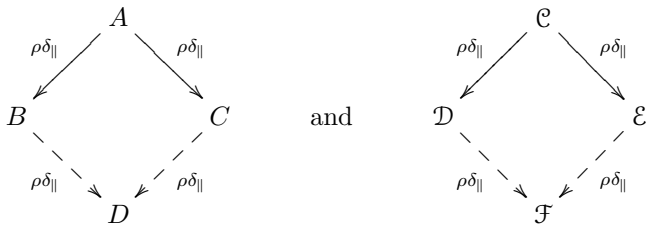
Definition 3.19 (Parallelization of $\rho\delta$) *We define the obvious parallelization of the $\rho\delta$ denoted $\longrightarrow_{\rho\delta_{\|}}$ by:*

$$\frac{}{A \longrightarrow_{\rho\delta_{\|}} A} \qquad \frac{A \longrightarrow_{\rho\delta_{\|}} A' \quad B \longrightarrow_{\rho\delta_{\|}} B'}{A \star B \longrightarrow_{\rho\delta_{\|}} A' \star B'}\ \star \in \{\fatsemi, \ll, \{\}, \twoheadrightarrow, \wedge, \}$$

$$\frac{A \longrightarrow_{\rho\delta_{\|}} A' \quad B \longrightarrow_{\rho\delta_{\|}} B' \quad C \longrightarrow_{\rho\delta_{\|}} C'}{(A \twoheadrightarrow B)\ C \longrightarrow_{\rho\delta_{\|}} (A' \ll C')\ B'} \qquad \frac{A \longrightarrow_{\rho\delta_{\|}} A' \quad B \longrightarrow_{\rho\delta_{\|}} B' \quad C \longrightarrow_{\rho\delta_{\|}} C'}{(A, B)\ C \longrightarrow_{\rho\delta_{\|}} A'\ C', B'\ C'}$$

Lemma 3.20 (Diamond Property of $\longrightarrow_{\rho\delta_{\|}}$) *For all terms A, B, C resp. constraints $\mathcal{C}, \mathcal{D}, \mathcal{E}$ there exists*

a term D resp. a constraint $\mathcal{F}$ such that the corresponding diagram commutes:

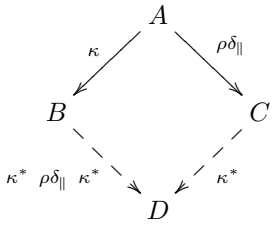


Proof : The proof is straightforward using always the same approach. See for example Lemma 3.10.

□

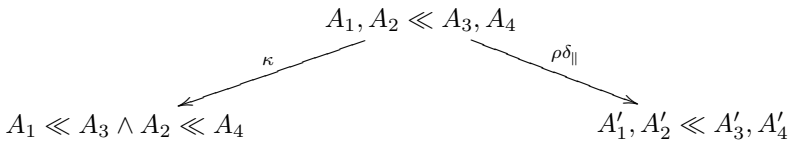
3.4.5 Yokouchi's diagram and the confluence of the $\rho_{\mathbf{x}}$-calculus

Lemma 3.21 (Yokouchi's diagram)



Proof : When the two steps from A to B and from A to C do not overlap, the lemma is easy since the system is left linear[7]. So we have to inspect every critical pair[8]. Since a strict subexpression of a $\rho\delta_{||}$ redex can never overlap with a κ redex, it is sufficient to work by cases on the derivation from A to B.

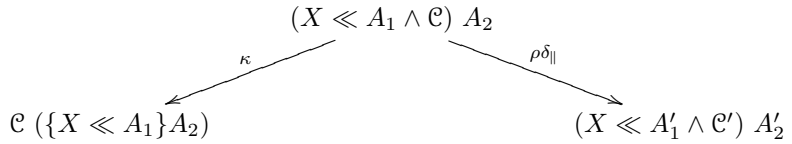
1. ($Decompose_{,}$) The reduction must be of the from:



   with $A_i \longrightarrow_{\rho\delta_{\|}} A'_i$. This is easy to find the wanted $D \triangleq A'_1 \ll A'_3 \wedge A'_2 \ll A'_4$

2. ($Decompose_{\mathcal{F}}$) This case is similar to the previous one since $f(A_1\ \ldots\ A_n)$ can only be reduced by $\rho\delta_{\|}$ to $f(A'_1\ \ldots\ A'_n)$ with $A_i \longrightarrow_{\rho\delta_{\|}} A'_i$.

3. ($ToSubst_{\wedge}$) Then if $X \notin \mathcal{D}om(\mathcal{C})$



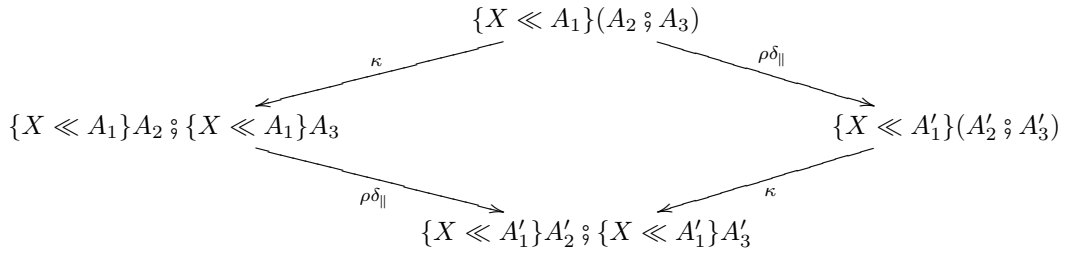
   with $A_i \longrightarrow_{\rho\delta_{\|}} A'_i$ and $\mathcal{C} \longrightarrow_{\rho\delta_{\|}} \mathcal{C}'$. The diagram can be completed with the term $\mathcal{C}'\ (\{X \ll A'_1\}A'_2)$ since of course if $X \notin \mathcal{D}om(\mathcal{C})$ then $X \notin \mathcal{D}om(\mathcal{C}')$.

[7] Actually it is sufficient to know that $\longrightarrow_{\kappa}$ is left-linear.

[8] As in the $\lambda\sigma_{\Uparrow}$-calculus a critical pair as a sense slightly different from the standard one because of the parallel reduction.

4. ($Share_{\S}$) There are three cases:

   (a)



   (b) $A = \{X \ll A_1\}\,((A_2 \multimap A_3)\ A_4)$, $B = \{X \ll A_1\}(A_2 \multimap A_3)\ \{X \ll A_1\}A_4$ and $C = \{X \ll A_1'\}\,((A_2' \ll A_4')\,A_3')$. We can choose $D = (A_2' \ll \{X \ll A_1'\}A_4')\ \{X \ll A_1'\}A_3'$.

   (c) If $A = \{X \ll A_1\}((A_2, A_3)A_4)$ and $B = \{X \ll A_1\}(A_2, A_3)\ \ \{X \ll A_1\}A_4$ and $C = \{X \ll A_1'\}(A_2'\ A_4',\ A_3'\ A_4')$. Similar to the previous case.

5. The other cases are simple and similar to the previous ones. They are left as a possible exercise for the reader.

□

Corollary 3.22 $\xrightarrow{*}_{\kappa} \longrightarrow_{\rho\delta_{\|}} \xrightarrow{*}_{\kappa}$ *is confluent*

Proof : By application of Yokouchi's lemma.

Theorem 3.23 *The $\rho_{\mathtt{x}}$-calculus is confluent.*

Proof : The result follows from Corollary 3.22 and from $\longrightarrow_{\rho_{\mathtt{x}}} \subseteq \xrightarrow{*}_{\kappa} \longrightarrow_{\rho\delta_{\|}} \xrightarrow{*}_{\kappa} \subseteq \xrightarrow{*}_{\rho_{\mathtt{x}}}$

3.5 Comparing the $\rho_{\mathtt{x}}$-calculus with the λ_x-calculus

In this section, we will see that the $\rho_{\mathtt{x}}$-calculus calculus embeds the λ_x-calculus[Ros96]. First, we recall the syntax and the semantics of the λ_x-calculus.

3.5.1 The λ_x-calculus

Definition 3.24 ($\boldsymbol{\lambda_{\mathtt{x}}}$-terms) *The set of $\lambda_{\mathtt{x}}$-terms, denoted $\Lambda_{\mathtt{x}}$, is the extension of the λ-terms defined inductively by:*

$$M \ ::= \ x \ \mid \ \lambda x.M \ \mid \ MN \ \mid \ M\langle x := N\rangle$$

Definition 3.25 (Reduction semantics of $\boldsymbol{\lambda_{\mathtt{x}}}$-calculus) *The reduction semantics of the λ_x-calculus is given by the following rules:*

$$\begin{array}{rcll}
(\lambda x.M)N & \to & M\,\langle x := N\rangle & (b) \\
x\,\langle x := N\rangle & \to & N & (xv) \\
x\,\langle y := N\rangle & \to & x \ \ if\ x \neq y & (xvgc) \\
(\lambda x.M)\,\langle y := N\rangle & \to & \lambda x.M\,\langle y := N\rangle & (xab) \\
(M_1M_2)\,\langle y := N\rangle & \to & M_1\,\langle y := N\rangle\ M_2\,\langle y := N\rangle & (xap)
\end{array}$$

In [Ros96] one can find a detailed presentation and study of the λ_x-calculus.

3.5.2 From λ_x-calculus to $\rho_{\mathtt{x}}$-calculus

We define a translation function from $\lambda_{\mathtt{x}}$-terms to $\rho_{\mathtt{x}}$-terms.

Definition 3.26 ($\mathcal{T}_{\lambda_{\mathtt{x}}\to\rho_{\mathtt{x}}}$) *We define $\mathcal{T}_{\lambda_x\to\rho_x}$, often denotes $\mathcal{T}$, by induction on the structure of $\lambda_{\mathtt{x}}$-terms.*

- $\mathcal{T}(x) = X$
- $\mathcal{T}(\lambda x.M) = X \twoheadrightarrow \mathcal{T}(M)$
- $\mathcal{T}(MN) = \mathcal{T}(M)\ \mathcal{T}(N)$
- $\mathcal{T}(M\langle x := N\rangle) = \{X \ll \mathcal{T}(M)\}\mathcal{T}(N)$

Lemma 3.27 *If $M \longrightarrow_{\lambda_x} N$ then $\mathcal{T}(M) \longrightarrow^*_{\rho_x} \mathcal{T}(N)$*

Proof : The proof is by induction on M.

1. $M = \lambda x.M_1$ then $N = \lambda x.M_1'$ with $M_1 \longrightarrow_{\lambda_{\mathtt{x}}} M_1'$. By IH we obtain $\mathcal{T}(M_1) \longrightarrow_{\rho_{\mathtt{x}}} \mathcal{T}(M_1')$. Thus $\mathcal{T}(M) = X \twoheadrightarrow \mathcal{T}(M_1) \quad \longrightarrow_{\rho_{\mathtt{x}}} \quad X \twoheadrightarrow \mathcal{T}(M_1') = \mathcal{T}(N)$.
2. $M = M_1M_2$ and $N = M_1'M_2$ (or $N = M_1M_2'$ but the case is similar) this case is similar to the previous one.
3. $M = (\lambda x.M_1)M_2$ and $N = M_1\langle x := M_2\rangle$. By definition $\mathcal{T}(M) = (X \twoheadrightarrow \mathcal{T}(M_1))\mathcal{T}(M_2)$ and $\mathcal{T}(N) = \{X \ll \mathcal{T}(M_2)\}\mathcal{T}(M_1)$. The result follows from the application of the (ρ) rule followed by the application of the $(ToSubst_{\ll})$ rule.
4. $M = M_1\langle x := M_2\rangle$. If the reduction from M to N does not occur at the root position then the case is similar to the previous one. Otherwise, we distinguish cases according to the possible value for M_1:
   (a) $M = x\langle x := M_2\rangle$
   (b) $M = y\langle x := M_2\rangle$
   (c) $M = (\lambda y.M_3)\langle x := M_2\rangle$
   (d) $M = (M_3M_4)\langle x := M_2\rangle$

   Each case is easy to deal since each reduction rule of the substitution application of the λ_x-calculus is a particular case - modulo notations - of a rule of the $\rho_{\mathtt{x}}$-calculus:

$$\begin{array}{lcl} (xv) & \hookrightarrow & (Replace) \\ (xvgc) & \hookrightarrow & (Eliminate_{\mathcal{X}}) \\ (xab) & \hookrightarrow & (Share_{\twoheadrightarrow}) \\ (xap) & \hookrightarrow & (Share_{\S}) \end{array}$$

□

Lemma 3.28 (Simulation) *If $\mathcal{T}(M) \longrightarrow_{\rho_x} T$ then $T \longrightarrow^{0,1}_{\lambda_x} T'$ with $T' = \mathcal{T}(N)$ and $M \longrightarrow_{\lambda_x} N$*

Proof : The notation $\longrightarrow^{0,1}$ denotes a zero or one step reduction. To understand the need of T' see Example 3.1. The proof is easy. □

3.6 Extensions of the $\rho_{\mathtt{x}}$-calculus

There are many ways to extend the $\rho_{\mathtt{x}}$-calculus from the explicit matching algorithm to the management of substitutions. For example, we have seen that we can deal with active variables easily. In this section, we will present an example of extension of the $\rho_{\mathtt{x}}$-calculus to other matching theories and we will propose a version of the calculus with substitution compositions.

3.6.1 How to extend the $\rho_{\mathtt{x}}$-calculus for syntactic theories

The ($Decompose_{\mathcal{F}}$) rule can be regulated according to the theory one wants to deal with. For example, if we want to deal with commutative symbols[9], we obtain the ($Decompose^{C}_{\mathcal{F}}$) - where $\mathfrak{S}_n$ denotes the permutation of $\{1,\ldots,n\}$:

$$Decompose^{C}_{\mathcal{F}} \quad f(A_1 \ldots A_n) \ll_C f(A'_1 \ldots A'_n) \quad \rightarrow \quad \underset{\varphi\in\mathfrak{S}_n}{,} \left(\bigwedge_{i=1}^{n} A_i \ll_C A'_{\varphi(i)} \right)$$

One can do the same for every syntactic theories [Kla92, Kir85] using the generic decomposition rule. We give an example of computation in the commutative theory.

Example 3.29 (Application of a rewrite rule in a commutative theory) *We suppose the symbol* $\mathfrak{and}$ *commutative whereas the other ones remain syntactic.*

$\big(\mathfrak{and}(X\ \mathfrak{or}(Y\ Z)) \twoheadrightarrow \mathfrak{or}(\mathfrak{and}(X\ Y)\ \mathfrak{and}(X\ Z))\big) \quad \mathfrak{and}(\mathfrak{or}(\mathfrak{tt}\ \mathfrak{ff})\ \mathfrak{or}(\mathfrak{ff}\ \mathfrak{ff}))$

$\mapsto_{\rho}$ $(\mathfrak{and}\,(X\ \mathfrak{or}\,(Y\ Z)) \ll \mathfrak{and}\,(\mathfrak{or}\,(\mathfrak{tt}\ \mathfrak{ff})\ \mathfrak{or}\,(\mathfrak{ff}\ \mathfrak{ff})))\,(\mathfrak{or}\,(\mathfrak{and}\,(X\ Y)\ \mathfrak{and}\,(X\ Z)))$

$\mapsto_{Decompose^{C}_{\mathcal{F}}}$ $\Big((X \ll \mathfrak{or}(\mathfrak{tt}\ \mathfrak{ff}) \wedge \mathfrak{or}(Y\ Z) \ll \mathfrak{or}(\mathfrak{ff}\ \mathfrak{ff})),\ (X \ll \mathfrak{or}(\mathfrak{ff}\ \mathfrak{ff}) \wedge \mathfrak{or}(Y\ Z) \ll \mathfrak{or}(\mathfrak{tt}\ \mathfrak{ff})\Big) \quad \mathfrak{and}(X\ Z)$

$\mapsto_{Decompose^{C}_{\mathcal{F}}}$ $\Big((X \ll \mathfrak{or}(\mathfrak{tt}\ \mathfrak{ff}) \wedge Y \ll \mathfrak{ff} \wedge Z \ll \mathfrak{ff}),\ (X \ll \mathfrak{or}(\mathfrak{ff}\ \mathfrak{ff}) \wedge \mathfrak{or}(Y\ Z) \ll \mathfrak{or}(\mathfrak{tt}\ \mathfrak{ff})\Big) \quad \mathfrak{and}(X\ Z)$

$\mapsto_{Decompose^{C}_{\mathcal{F}}}$ $\Big((X \ll \mathfrak{or}(\mathfrak{tt}\ \mathfrak{ff}) \wedge Y \ll \mathfrak{ff} \wedge Z \ll \mathfrak{ff}),\ (X \ll \mathfrak{or}(\mathfrak{ff}\ \mathfrak{ff}) \wedge Y \ll \mathfrak{tt} \wedge Z \ll \mathfrak{ff})\Big) \quad \mathfrak{and}(X\ Z)$

$\mapsto_{\delta}$ $(X \ll \mathfrak{or}(\mathfrak{tt}\ \mathfrak{ff}) \wedge Y \ll \mathfrak{ff} \wedge Z \ll \mathfrak{ff}) \quad \mathfrak{and}(X\ Z),$
$(X \ll \mathfrak{or}(\mathfrak{ff}\ \mathfrak{ff}) \wedge Y \ll \mathfrak{tt} \wedge Z \ll \mathfrak{ff}) \quad \mathfrak{and}(X\ Z)$

Afterwards, the substitution applies as in the empty theory since of course the application of the substitution is not dependent of the matching theory we are dealing with.

The same remark can be formulated about the theory of the structure operator ",".

3.6.2 How to extend the $\rho_{\mathtt{x}}$-calculus with the composition of substitutions: the $\rho_{\mathtt{xc}}$-calculus

Syntax of the $\rho_{\mathtt{xc}}$-calculus

We have seen in the introduction of this chapter that there are actually two ways to identify constraints that can be applied - *i.e.*, constraints that can be transformed to substitutions. The first method chosen leads to a simple calculus but where the composition of substitutions is not allowed because substitions are restricted to $X \ll A$. Because of this restriction, the number of reduction steps to apply a single substitution is very important because we need to visit the term for every atomic matching equation. For example,

$(X \ll A \wedge Y \ll B)\ g^n(X)$ where $g^n(X) = g(g(\ldots g(X)))$

$\mapsto_{\rho_{\mathtt{x}}}$ $(X \ll A)\{Y \ll B\}g^n(X)$

$\mapsto^{2n+2}_{\rho_{\mathtt{x}}}$ $(X \ll A)g^n(X)$

$\mapsto^{2n+2}_{\rho_{\mathtt{x}}}$ $g^n(A)$

So we need $4n+5$ steps to apply this very simple substitution. We should want to visit the term not twice but only one time. If we had continued the Example 3.29 in the $\rho_{\mathtt{x}}$-calculus, it would have taken about one page to give the entire reduction.

So, the question is now to propose a nice way to label part of constraints which are solvable and independent of the remaining constraints. Thanks to this label we will be able to compose substitutions and thus to give a powerful calculus. We should straightforwardly encapsulate identified constraints using special constants let us say $\mathbf{g}$ ("good") for solvable constraints. For example, the constraint $X \ll a$ will be labeled $\mathbf{g}(X \ll a)$. In this approach, labeling rules will have the following shape: $C \rightarrow \mathbf{g}(C)$. This solution seems quite natural but the label information is not strictly speaking put onto the constraint but over it (*i.e.*, we modify the context in which the constraints appears but not the constraint). For a rewriting approach,

[9] In general, commutative symbols are binary. We present commutativity for n-ary symbols.

this causes many problems. For example, the constraints $\mathbf{g}(C)$ can be rewritten using the previous rules at position 1 in $\mathbf{g}(\mathbf{g}(C))$ and so on...

So, it is clear that we must not put labels in the context of an identified constraint but we must rather modify the constraint itself. The $\mathbf{g}$ label needs to be a global attribute. It seems that the best way to identify solvable constraints is to label the head symbol of such constraints. For example, the solvable constraints $X \ll a \ \wedge \ Y \ll b$ (with $X \neq Y$) is solvable and should be reduced to $X \ll a \ \wedge_{\mathbf{g}} \ Y \ll b$. Choosing the symbol $\wedge_{\mathbf{g}}$ AC1[10] (the identity will be denoted $\mathtt{id}$), we identify $X \ll A$ and $X \ll A \wedge_{\mathbf{g}} \mathtt{id}$ and thus we implicitly consider atomic constraints as solvable recovering the $\rho_{\mathtt{x}}$-calculus. The symbol $\wedge$ may be chosen with a AC1 theory since the idempotency follows from the $(Merge)$ rule. Since $\wedge_{\mathbf{g}}$ is AC1, we can label every constraint with $\wedge_{\mathbf{g}}$ using $\mathtt{id}$. For example, $a \ll b =_{AC1} a \ll b \wedge_{\mathbf{g}} \mathtt{id}$. So a $\mathbf{g}$ constraint is a constraint looking like: $\mathcal{C} \wedge_{\mathbf{g}} \mathcal{D}$ with $\mathcal{C}, \mathcal{D} \neq \mathtt{id}$ or $\mathcal{C} = \mathtt{id}$ and $\mathcal{D} = X \ll A$. In short, we will denote a $\mathbf{g}$ constraint by $\mathcal{C}^{\mathbf{g}}, \mathcal{D}^{\mathbf{g}}$.

The syntax is presented in Figure 3.4. In the following the symbol $\wedge_?$ will stand for either $\wedge$ or $\wedge_{\mathbf{g}}$. The only difference with the syntax of the $\rho_{\mathtt{x}}$-calculus is the improvement of the set of substitutions. Of

Terms	A, B, C, T	::=	$\mathcal{X}$	(Variables)
		|	$\mathcal{K}$	(Term constants)
		|	$A \twoheadrightarrow B$	(Abstraction)
		|	$A\ B$	(Fonctional application)
		|	$\mathcal{C}\ B$	(Constraint application)
		|	A, B	(Structure)
		|	$\{\vartheta\}A$	(Substitution application on terms)
Constraints	$\mathcal{C}, \mathcal{D}, \mathcal{E}$	::=	$A \ll B$	(Match-equation)
		|	$\mathcal{C} \wedge \mathcal{D}$	(Conjonction of contraints)
		|	$\{\vartheta\}\mathcal{C}$	(Substitution application on constraints)
		|	ϑ	(Substitutions)
Substitutions	ϑ, φ, ξ	::=	$X \ll A$	(Atomic match-equation)
		|	$\vartheta \wedge_{\mathbf{g}} \varphi$	(Solvable conjonctions of substitutions)
		|	$\{\vartheta\}\varphi$	(Substitution application on substitutions)
		|	$\mathtt{id}$	(The identity subsitution)

Figure 3.4: Syntax of the $\rho_{\mathtt{xc}}$-calculus

course, the constraint $X \ll A \wedge_{\mathbf{g}} X \ll B$ is not a substitution since it is not solvable. In the empty theory, the condition to form solvable conjunctions of substitutions ϑ and φ should be replaced by the condition $\mathcal{D}om(\vartheta) \cap Dom(\varphi) = \emptyset$. It must be noticed that all substitutions are constraints. It was exactly the same in the $\rho_{\mathtt{x}}$-calculus since the matching constraint $X \ll A$ can be seen either as a constraint - when the $\bullet$ is used - or as a substitution - when the $\{\}$ operator is used.

Semantics of the $\rho_{\mathtt{xc}}$-calculus

We deal now with $\mathbf{g}$ labeling of constraints. We must recall that since the symbol $\wedge_{\mathbf{g}}$ is AC1, an atomic constraint is always implicitly labeled as a $\mathbf{g}$ constraint since, in the theory AC1, $X \ll A$ and $X \ll A \wedge_{\mathbf{g}} \mathtt{id}$ belong to the same equivalent class. We label a solvable constraint step by step. A part of a constraint is identify as $\mathbf{g}$ if it is a conjunction of two atomic constraints with different variables *i.e.* we consider the (GI) rule (for "$\mathbf{g}$ init"):

[10] *i.e.*, associative commutative and identity.

$$(GI) \quad X \ll A \wedge Y \ll B \quad \rightarrow \quad X \ll A \wedge_{\mathbf{g}} Y \ll B \quad \texttt{if } X \neq Y$$

When we want to merge two $\mathbf{g}$ constraints, we must be careful because their domains may intersect. The problem appears when one wants to deal with the constraint $(X \ll A \wedge_{\mathbf{g}} Y \ll B) \wedge X \ll C$. As in the $\rho_{\mathtt{x}}$-calculus, the nicest way to deal with non-linearity is to wait until the problem becomes linear. Thus the merge rule will be

$$(Merge) \quad (X \ll A \wedge_{\mathbf{g}} \mathcal{C}) \wedge (X \ll A \wedge_{\mathbf{g}} \mathcal{D}) \quad \rightarrow \quad (X \ll A \wedge_{\mathbf{g}} \mathcal{C}) \wedge \mathcal{D}$$

And we merge two $\mathbf{g}$ constraints with domains that do not intersect. The rule (GI) becomes an instantiation of the $(Good)$ rule.

$$(Good) \quad \mathcal{C}^{\mathbf{g}} \wedge \mathcal{D}^{\mathbf{g}} \quad \rightarrow \quad \mathcal{C}^{\mathbf{g}} \wedge \mathcal{D}^{\mathbf{g}} \quad \texttt{if } \mathcal{D}om(\mathcal{C}^{\mathbf{g}}) \cap \mathcal{D}om(\mathcal{D}^{\mathbf{g}}) = \emptyset$$

The question now is to find a nice way to compose substitutions. One would like to write:

$$(Compose) \quad \{\vartheta\}(\{\varphi\}A) \quad \rightarrow \quad \{\{\vartheta\}\varphi\}\ \{\vartheta\}A$$

This rule is quite nice but actually it does not terminate if one replace both substitutions by atomic matching equations. The solution is to define from the two substitutions the composed substitutions and applying it. This leads to the rule:

$$(Compose) \quad \{\vartheta\}(\{\varphi\}A) \quad \rightarrow \quad \{\vartheta \wedge_{\mathbf{g}} \{\vartheta\}\varphi\}\ A$$

Since we apply ϑ on φ we are sure that the domain of ϑ and the codomain of $\{\vartheta\}\varphi$ are disjoint. The domains are of course disjoint by α-conversion.

We have chosen the $\wedge_{\mathbf{g}}$ operator with an AC1 theory because the $\mathtt{id}$ substitution appears naturally as in most of the explicit substitution calculus dealing with composition. A nice way to see this is to have a look at the $(Eliminate_{\mathcal{X}})$ rule which allows to reduce either $\{X \ll A\}Y$ to $\{\mathtt{id}\}Y$ or $\{X \ll A \wedge_{\mathbf{g}} Y \ll B\}Y$ to $\{Y \ll B\}Y$.

Since the symbols $\wedge$ are AC1, the following rule is admissible since we can derive it from the $(ToSubst_{\wedge})$ rule.

$$(ToSubst_{\ll}) \quad (X \ll A)B \quad \rightarrow \quad \{X \ll A\}B$$

Moreover, unlike in the $\rho_{\mathtt{x}}$-calculus, the $(Decompose_{\mathcal{F}})$ rule can also be applied for $n = 0$ and thus allows to reduce $a \ll a$ to $\mathtt{id}$. The semantics of the $\rho_{\mathtt{xc}}$-calculus is presented in Figure 3.5. We do not comment on the other reduction rules since they are similar to those of the $\rho_{\mathtt{x}}$-calculus where the notion of substitution is improved.

Properties of the $\rho_{\mathtt{xc}}$-calculus

First of all, we will show that $\longrightarrow_{\kappa}$ is strongly normalizing.

Lemma 3.30 *$\longrightarrow_{\kappa}$ is strongly normalizing*

Proof : We need to define a measure ζ on terms representing the numbers of different variables in a left hand side of constraint matching equations - but not of substitutions. So, X is not in a left hand side of a constraint matching equation in $\{X \ll A\}B$ whereas it is in $(X \ll A)B$. More precisely, ζ is the cardinality of m defined by:

$m(X) = \emptyset$ for all $X \in \mathcal{X}$

$m(k) = \emptyset$ for all $k \in \mathcal{K}$

$m(\mathtt{id}) = \emptyset$

$m(A \twoheadrightarrow B) = m(A) \cup m(B)$

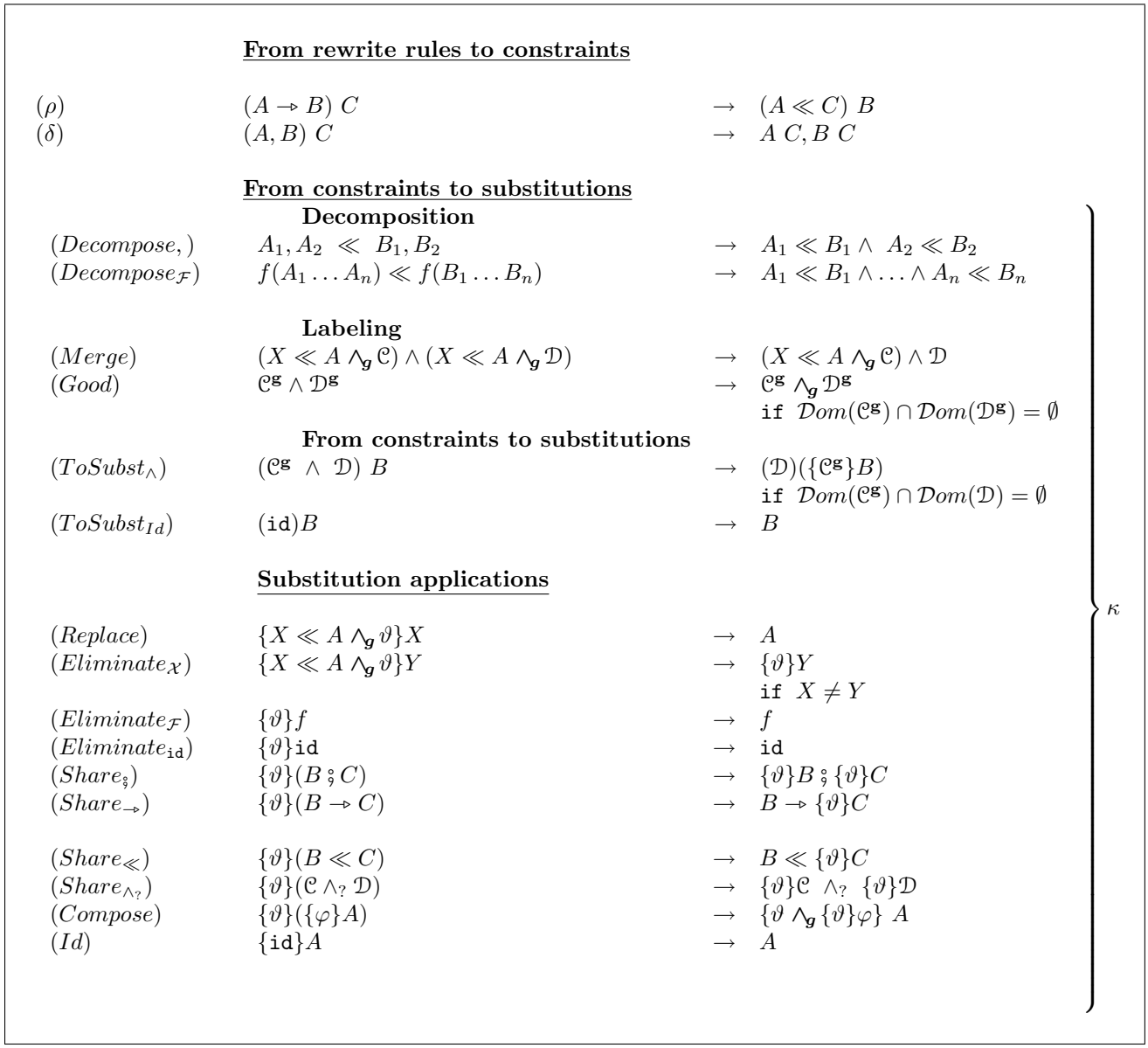


Figure 3.5: Small-step reduction semantics of the $\rho_{\mathtt{xc}}$-calculus

$m(A \S B) = m(A) \cup m(B)$

$m(\mathcal{C}\ A) = m(\mathcal{C}) \cup m(A)$

$m(A \ll B) = \mathcal{FV}ar(A) \cup m(A) \cup m(B)$

$m(\mathcal{C} \wedge_? \mathcal{D}) = m(\mathcal{C}) \cup m(\mathcal{D})$

$m(\{\vartheta\}B = m(\vartheta) \cup m(B) \backslash \mathcal{D}om(\vartheta)$

$m(\{\vartheta\}\mathcal{C}) = m(\vartheta) \cup m(\mathcal{C}) \backslash \mathcal{D}om(\vartheta)$

$m(\{\vartheta\}\varphi) = m(\vartheta) \cup m(\varphi) \backslash \mathcal{D}om(\vartheta)$

We also need to define a polynomial interpretation on terms.

$P(X) = P(k) = P(\mathtt{id}) = 3$ for all $x \in \mathcal{X}$ and $k \in \mathcal{K}$

$P(A \twoheadrightarrow B) = P(A) + P(B)$

$P(A \S B) = P(A) + P(B)$

$P(\mathcal{C}\ A) = P(\mathcal{C}) + P(A)$

$P(A \ll B) = P(A) + P(B)$

$P(\mathcal{C} \wedge_? \mathcal{D}) = P(\mathcal{C}) + P(\mathcal{D})$

$P(\{\vartheta\}B) = (P(\vartheta) + 2) \times P(B)$

$P(\{\vartheta\}\mathcal{C}) = (P(\vartheta) + 2) \times P(\mathcal{C})$

$P(\{\vartheta\}\varphi) = (P(\vartheta) + 2) \times P(\varphi)$

We show that $\longrightarrow_\kappa$ is strongly normalizing using the lexicographic product of ζ , P and $\succ$ where $\succ$ is the recursive path ordering obtain from the precedence $\ll\ >_\Sigma\ \wedge\quad \{\}\ >_\Sigma\ \S\quad \{\}\ >_\Sigma\ \wedge_?\quad \wedge\ >_\Sigma\ \bigwedge_{\mathtt{g}}$ and with the status "multiset" for the symbol $\{\}$. □

All the properties shown in the previous section should generalize to the $\rho_{\mathtt{xc}}$-calculus. For example, the $\rho_{\mathtt{xc}}$-calculus may include the λ_σ-calculus[11].

Conclusion and future work

Conclusion

We have proposed a new Rho-Calculus to deal explicitly with constraints, embedding most of λ-calculus with explicit substitutions. We have proved that the calculus enjoys nice properties such as the confluence of the calculus and the termination of the constraint handling part. We have seen that the calculus is really modular and can be adapted to many matching theories as well as many theories for the structure operator ",". We can either chose to be atomic and give a simple definition of substitution but forbidding compositions. Or we can be more general and efficient and deal with a calculus with composition.

Rho-Calculi and especially the $\rho_{\mathtt{x}}$-calculus, are nice frameworks to encode programs and their behaviors and especially dealing with errors.

[11] Without the handling of variable names.

Future work

After this training, two questions arise: First, by taking advantage of the very general management of errors it should be interesting to "label" errors so has to propose an named exception mechanism. Secondly, one should propose an extension to deal with variable names and α-conversion. One possible approach is to follow the works about the λ_l-calculus with names.

More generally, we want to understand what an interpreter/compiler for the Rho-Calculus could mean and how to implement them. This question is strongly related to the wish to propose an environment where calculi and deductions are uniformly integrated, *i.e.*, to unify ML-languages, Coq-languages and tactics languages since programming a function or a proof should be done in the same language.

Bibliography

[AC96] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer Verlag, 1996.

[ACCL91] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy. Explicit substitutions. *Journal of Functional Programming*, 1(4):375–416, October 1991.

[ARK00] M. Ayala-Rincón and F. Kamareddine. Unification via λs_e-Style of Explicit Substitution. In *Second International Conference on Principles and Practice of Declarative Programming*, pages 163–174, Montreal, Canada, September 2000.

[Bar84] H. P. Barendregt. *The Lambda-Calculus, its syntax and semantics*. Studies in Logic and the Foundation of Mathematics. North Holland, 1984. Second edition.

[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.

[BKK98a] P. Borovanský, C. Kirchner, and H. Kirchner. A functional view of rewriting and strategies for a semantics of ELAN. In M. Sato and Y. Toyama, editors, *The Third Fuji International Symposium on Functional and Logic Programming*, pages 143–167, Kyoto, April 1998. World Scientific.

[BKK+98b] P. Borovanský, C. Kirchner, H. Kirchner, P.-E. Moreau, and C. Ringeissen. An overview of ELAN. In C. Kirchner and H. Kirchner, editors, *Proceedings of the second International Workshop on Rewriting Logic and Applications*, volume 15, `http://www.elsevier.nl/locate/entcs/volume15.html`, Pont-à-Mousson (France), September 1998. Electronic Notes in Theoretical Computer Science. Report LORIA 98-R-316.

[BT88] V. Breazu-Tannen. Combining Algebra and Higher-order Types. In *Proc. of LICS*, pages 82–90, 1988.

[CHL96] P.-L. Curien, T. Hardin, and J.-J. Lévy. Confluence properties of weak and strong calculi of explicit substitutions. *Journal of the ACM (JACM)*, 43(2):362–397, 1996.

[Chu41] A. Church. 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 1, France, October 2000.

[CK98] H. Cirstea and C. Kirchner. ρ-calculus. Its Syntax and Basic Properties. In *Proc. of Workshop CCL*, 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, 2001.

[CKL01a] H. Cirstea, C. Kirchner, and L. Liquori. Matching Power. In *Proc. of RTA*, volume 2051 of *LNCS*, pages 77–92. Springer-Verlag, 2001.

[CKL01b] H. Cirstea, C. Kirchner, and L. Liquori. The Rho Cube. In *Proc. of FOSSACS*, volume 2030 of *LNCS*, pages 166–180, 2001.

[CKL02] H. Cirstea, C. Kirchner, and L. Liquori. Rewriting calculus with(out) types. In F. Gadducci and U. Montanari, editors, *Proceedings of the fourth workshop on rewriting logic and applications*, Pisa (Italy), September 2002. Electronic Notes in Theoretical Computer Science.

[CKLW03] H. Cirstea, C. Kirchner, L. Liquori, and B. Wack. Rewrite strategies in the rewriting calculus. In B. Gramlich and S. Lucas, editors, *Proceedings of the Third International Workshop on Reduction Strategies in Rewriting and Programming* , Valencia, Spain, June 2003. Electronic Notes in Theoretical Computer Science.

[CLW03] H. Cirstea, L. Liquori, and B. Wack. Typed recursive rewriting calculus. *Manuscript*, 2003.

[CMP00] E. Chailloux, P. Manoury, and B. Pagano. *Développement d'applications avec Objective CAML*. O'REILLY, 2000.

[DHK00] G. Dowek, T. Hardin, and C. Kirchner. Higher order unification via explicit substitutions. *Information and Computation*, 157(1/2):183–235, 2000.

[DHKP96] G. Dowek, T. Hardin, C. Kirchner, and F. Pfenning. Unification via explicit substitutions: The case of higher-order patterns. In M. Maher, editor, *Proceedings of JICSLP'96*, Bonn (Germany), September 1996. The MIT press.

[DK00] H. Dubois and H. Kirchner. Objects, rules and strategies in `ELAN`. In *Proceedings of the second AMAST workshop on Algebraic Methods in Language Processing, Iowa City, Iowa, USA*, May 2000.

[FK02] G. Faure and C. Kirchner. Exceptions in the rewriting calculus. In S. Tison, editor, *Proceedings of the RTA conference*, volume 2378 of *Lecture Notes in Computer Science*, pages 66–82, Copenhagen, July 2002. Springer-Verlag.

[For02] J. Forest. A weak calculus with explicit operators for pattern matching and substitution. pages 174–191, 2002.

[GBT89] J. Gallier and V. Breazu-Tannen. Polymorphic Rewriting Conserves Algebraic Strong Normalization and Confluence. In *Proc. of ICALP*, volume 372 of *LNCS*, pages 137–150. Springer-Verlag, 1989.

[JO97] J. Jouannaud and M. Okada. Abstract Data Type Systems. *Theoretical Computer Science*, 173(2):349–391, 1997.

[Kam88] S. N. Kamin. Inheritance in Smalltalk-80: A Denotational Definition. In T. A. press, editor, *Proc. of POPL*, pages 80–87, 1988.

[Kir85] C. Kirchner. *Méthodes et outils de conception systématique d'algorithmes d'unification dans les théories équationnelles*. Thèse de Doctorat d'Etat, Université Henri Poincaré – Nancy 1, 1985.

[Kla92] F. Klay. *Unification dans les Théories Syntaxiques*. Thèse de Doctorat d'Université, Université Nancy 1, 1992.

[KOR93] J. Klop, V. v. Oostrom, and F. v. Raamsdonk. Combinatory Reduction Systems: Introduction and Survey. *Theoretical Computer Science*, 121:279–308, 1993. Special issue in honour of Corrado Böhm.

[KPT96] D. Kesner, L. Puel, and V. Tannen. A Typed Pattern Calculus. *Information and Computation*, 124(1):32–61, 10 1996.

[Les94] P. Lescanne. From $\lambda\sigma$ to λv a journey through calculi of explicit substitutions. In *Conference Record of POPL '94: 21ST ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, Oregon*, pages 60–69, New York, NY, January 1994. ACM.

[LS03] L. Liquori and B. Serpette. An imperative rewriting calculus. *Manuscript*, 2003.

[Mil91] D. Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. In *Proc. of ELP*, volume 475 of *LNCS*, pages 253–281. Springer-Verlag, 1991.

[Mil03] D. Miller. *lambda-Prolog: An Introduction to the Language and its Logic.* 2003.

[Muñ97] C. Muñoz. *Un calcul de substitutions pour la représentation de preuves partielles en théorie de types*. Thèse de doctorat, Université Paris 7, 1997. English version available as INRIA research report RR-3309.

[Ngu01] Q.-H. Nguyen. Certifying Term Rewriting Proof in `ELAN`. In M. van den Brand and R. Verma, editors, *Proc. of RULE'01*, volume 59. Elsevier Science Publishers B. V. (North-Holland), September 2001.

[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.

[Oka89] M. Okada. Strong Normalizability for the Combined System of the Typed λ Calculus and an Arbitrary Convergent Term Rewrite System. In *Proc. of ISSAC*, pages 357–363. ACM Press, 1989.

[OT97] P. W. O'Hearn and R. D. Tennent, editors. *Algol-Like Languages, Vols I and II.* Progress in Theoretical Computer Science. Birkhauser, 1997.

[Pey87] S. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice Hall, 1987.

[Ros96] K. H. Rose. *Operational Reduction Models for Functional Programming Languages*. PhD thesis, DIKU, University of Copenhagen, Denmark, Universitetsparken 1, DK-2100 København Ø, February 1996. Available as DIKU report 96/1.

[SDK+03] A. Stump, A. Deivanayagam, S. Kathol, D. Lingelbach, and D. Schobel. Rogue decision procedures at the 1st international workshop on pragmatics of decision procedures in automated reasoning. 2003.

[SW01] D. Sangiorgi and D. Walker, editors. *The pi-calculus: a theory for mobile process*. Cambridge University Press, 2001.

[vO90] V. van Oostrom. Lambda Calculus with Patterns. Technical Report IR-228, Faculteit der Wiskunde en Informatica, Vrije Universiteit Amsterdam, 1990.

[WL96] Weiss and Leroy. *Le langage Caml*. InterEditions, 1996.

[Wol93] D. A. Wolfram. *The Clausal Theory of Types*, volume 21 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1993.

[YH90] H. Yokouchi and T. Hikita. A rewriting system for categorical combinators with multiple arguments. *SIAM Journal on Computing*, 19(1):78–97, February 1990.

Contents