From categorical combinators to lambda-sygma-calculi, a quest for confluence

Thérèse Hardin

UNITÉ DE RECHERCHE
INRIA-ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France
Tél. (1) 39 63 55 11

Rapports de Recherche

1992

25ème anniversaire

N° 1777

Programme 2
Calcul Symbolique, Programmation
et Génie logiciel

FROM CATEGORICAL COMBINATORS TO λσ-CALCULI, A QUEST FOR CONFLUENCE

Thérèse HARDIN

Octobre 1992

RR_1777

From Categorical Combinators to $\lambda\sigma$-calculi, a quest for confluence

Thérèse Hardin

Des combinateurs Catégoriques aux $\lambda\sigma$-calculs : une quête de la confluence

Thérèse Hardin[1]

Résumé

Le λ-calcul est souvent considéré comme le modèle même des langages de programmation fonctionnelle. Cependant, l'opération de substitution, qui décrit le passage des paramètres, n'y apparaît que dans le méta-langage, ce qui se révèle très gênant lorsqu'on souhaite décrire les mécanismes de compilation. Il semble donc nécessaire de trouver des extensions du λ-calcul, qui puissent manipuler explicitement des substitutions, tout en conservant les propriétés de confluence de ce langage. La Logique Combinatoire Catégorique, introduite en 1983, puis les $\lambda\sigma$-calculs, arrivés en 1988 et 1989, sont des théories qui répondent à cette requête. Nous en donnons ici une présentation unifiée, en montrant que le passage de l'un à l'autre peut se comprendre comme une amélioration des propriétés de confluence. Ce rapport ne contient aucun résultat nouveau, les théories sont présentées de manière assez informelle : les aspects trop techniques n'ont pas été, dans la mesure du possible, détaillés.

From Categorical Combinators to $\lambda\sigma$-calculi, a quest for confluence

Thérèse Hardin

Abstract

The λ-calculus is known to be the theoretical base of functional programming languages. But, the substitution, which describes the parameters'passage, belongs only to the meta-language and this is a major drawback when dealing with compilation. Therefore, extensions of λ-calculus, able to manipulate explicitly substitutions, and still confluent, are required. The categorical Combinatory Logic, introduced in 1983, and the $\lambda\sigma$-calculi, presented in 1988 and 1989, are responses to this question. We give here a survey of these theories, explaining their evolution from a confluence point of view. This report does not contain new results and remains rather informal, avoiding too technical details.

[1]INRIA Rocquencourt et LITP, Université Paris 6

Introduction

It is well-known that λ-calculus is the theoretical base of functional programming languages since, in essence, a functional program consists in a set of applications of functions to arguments. The replacement of the formal parameters of functions by the actual ones is exactly a step of β-reduction and it is not so simple: clashes of names have to be avoided and we have to take care of the scopes of arguments of functions. These problems are hidden in λ-calculus because the substitution of variables by λ-terms is a built-in operation, defined at the level of the meta-language. For theoretical purposes, it suffices to consider β-conversions modulo renaming of bound variables (α-conversion). But this is not realistic in practise, where moreover, substitutions are delayed and recorded in environments. So, when modeling implementations of functional languages, it is necessary to fully describe the substitution process inside the language in order to be able to study their properties. Therefore the problem is twofold: design a language, extending λ-calculus, such that the substitution is a first-order citizen, and such that names'management is explicitely handled.

This problem is not new and several solutions have been proposed, since the early seventies. The important steps were the design of a name-free notation for λ-calculus by de Bruijn[dBr72], then the introduction of Categorical Combinatory Logic[Cur83] CCL by Curien and, during the last years, the definitions of $\lambda\sigma$-calculi [ACCL89, HLe89]. We shall give a panorama of these different theories, from a rewriting point of view. The classical λ-calculus is a Combinatory Reduction System (see [Klo82]), that is, roughly, a term rewriting system with bound variables. CCL and the $\lambda\sigma$-calculi are very classical Term Rewriting Systems. The most important property that an extension of λ-calculus has to verify is the Church-Rosser property. CCL is not weakly confluent but a restriction of the sytem, sufficient to compute β-reduction, is confluent on a subset of combinators containing λ-terms. The first version, called $\lambda\sigma$, of $\lambda\sigma$-calculus is not weakly confluent but is ground confluent. The second one, called $\lambda\sigma_{\Uparrow}$ (but also $\lambda\sigma$ when no confusion is possible), is fully confluent. We may untersdand the evolution from CCL to $\lambda\sigma_{\Uparrow}$ as a sequence of improvements to obtain Church-Rosser property and we shall present these theories following this guide line and focusing on confluence. We restrict ourselves to the untyped versions of these three systems; typed versions of CCL are given in [Cur83] and [ACCL89] contains several type sytems for $\lambda\sigma$. We shall not detail either weak $\lambda\sigma$-calculi, which are studied in [CHL91]. In classical λ-calculus, the weak β-reduction is a restriction of β, which prevents reduction of β-redexes "under" the λ's: it is the rule used to evaluate programs. This restriction is not sufficient to avoid captures ($(\lambda x.\lambda y.x)y$ reduces on $\lambda z.y$ and α-conversion $\lambda y.x =_\alpha \lambda z.x$ has to be performed before the reduction). Moreover, weak β-reduction is not confluent. In $\lambda\sigma$, this is possible, on one hand, to forbid substitutions under the λ's (a simple way to avoid names'clashes), and on the other hand, to prevent applications of the rule (Beta) under the λ's. The weak conditional $\lambda\sigma$-calculus autorizes no (Beta) step and no substitutions under the λ's. The weak $\lambda\sigma$-calculus prevents only substitution under the λ's and is given in two versions: the first one uses de Bruijn numbers, the second one keeps the variables'names. These weak calculi are confluent and compute weak head normal forms of λ-terms.

1 λ-calculi

1.1 The classical λ-calculus

Definition 1 *The pure λ-calculus on a set V of variables, denoted by Λ_V, is defined inductively as follows:*

$$a ::= x \mid \lambda x.a \mid aa$$

The β-rule is defined by:

$$(\lambda\, x.a)b \longrightarrow a\{\, x \leftarrow\, b\}$$

We suppose familiarity with this theory. For further details, see [Bar84, Klo82, Lev78]. We only give the complete definition of the substitution, in order to recall how intricate this definition is: all the following details are needed to avoid names'capture. Moreover, this definition belongs only to the meta-language of λ-calculus.

Definition 2 *Let a and $b \in \Lambda_V$. The set of free (resp. bound) variables of a is denoted by $FV(a)$ (resp. $BV(a)$). The substitution of b at all the free occurrences of x in a, denoted by $a\{x \leftarrow b\}$, is defined as follows:*

1. $x\{x \leftarrow b\} = b$ *and* $y\{x \leftarrow b\} = y$ *if* $y \not\equiv x$
2. $(a_1\, a_2)\{x \leftarrow b\} = (a_1\{x \leftarrow b\})\,(a_2\{x \leftarrow b\})$
3. $(\lambda\, x.a)\{x \leftarrow b\} = \lambda\, x.a$
4. $(\lambda\, y.a)\{x \leftarrow b\} = \lambda\, y.(a\,\{x \leftarrow b\})$ *if* $y \not\equiv x$ *and* ($y \notin FV(b)$ *or* $x \notin FV(a)$)
5. $(\lambda\, y.a)\{x \leftarrow b\} = \lambda\, z.(a\,\{y \leftarrow z\}\,\{x \leftarrow b\})$ *if* $y \not\equiv x$ *and* ($y \in FV(b)$ *and* $x \in FV(a)$) ; $z \notin V(ab)$.

The complicated points d. and e. are needed to prevent names'clashes: α-conversion has to be done before the substitution under a λ. If these substitutions are forbidden, they are no conflicts of names (see [CHL91]).

The theoretical properties of Λ_V may be studied modulo α-conversion. This solution is unreasonable in implementations. But there is a simple way to avoid such names'conflicts, which was introduced by de Bruijn[dBr72, dBr78, Bal86, Ned92]: it is to suppress the names! To correctly perform β-reduction, it suffices to know what are the occurrences which have to be replaced. In Λ_V, this is indicated by the identity between the name of the variable at a given occurrence and the name of the λ's variable of the β-redex. It may also be indicated by the *binding height* of an occurrence, that is, by the number of λ's one has to cross between this occurrence and its binder. If we decide to suppress also names of free variables $\{x_0, \ldots, x_n\}$ of a given term a, it suffices to consider a as a subterm of $\lambda.x_0 \ldots x_n.a$. We recall this notation below, because it is underlying all the theories which manage explicitly the substitution.

1.2 λ-calculus in de Bruijn notation

Definition 3 *The set of λ-terms, Λ, in de Bruijn's notation, is defined inductively as follows, where* n *is an integer greater or equal to 1*[1]*:*

$$a ::= \mathtt{n} \mid \lambda(a) \mid aa$$

The β-reduction is defined by:

$$\lambda(a)b \longrightarrow a\{1 \leftarrow b\}$$

In this formalism, names of variables have disappeared. A given occurrence u of a variable, say x, is replaced by the number n of symbols λ, whose occurences are "between" the binder of this x and u. For example, $\lambda x\ y.x(\lambda z.zx)y$ is written $\lambda(\lambda 2(\lambda 13)1$. Therefore explicit α-conversion is avoided but, as described below, numbers have to be adjusted when a substitution is performed. The *λ-height* of an occurrence u in a term a is the number of λ's at prefix occurrences of u. It is denoted by $(|u|, a)$. A number p, at occurrence u, in a term a is bound iff $p \leq (|u|, a)$. So when reducing the redex $\lambda(a)b$, a number p at the occurrence u in a is affected by the substitution of b iff $p = (|u|, a) + 1$. Before replacing this p, the numbers of b have to be modified, to take care of the λ "above" u. This is done by the lift operation. Substitution, in de Bruijn's formalism, is defined as follows:

Definition 4 *The substitution of b at the λ-height* $(n-1)$ *in a, denoted by* $a\{\mathtt{n} \leftarrow b\}$*, and the lift with n at the λ-height i, denoted by* $t_i^n(a)$*, are defined by induction as follows:*

$$\begin{array}{rcll}
(a_1a_2)\{\mathtt{n} \leftarrow b\} & = & a_1\{\mathtt{n} \leftarrow b\}\ a_2\{\mathtt{n} \leftarrow b\} & \\
\lambda a\{\mathtt{n} \leftarrow b\} & = & \lambda(a\{\mathtt{n+1} \leftarrow b\}) & \\
\mathtt{m}\{\mathtt{n} \leftarrow b\} & = & \mathtt{m}-1 \ \text{ if } \ \mathtt{m} > \mathtt{n} & (\in FV(a)) \\
 & & t_0^{\mathtt{n}}(b) \ \text{ if } \ \mathtt{m} = \mathtt{n} & \text{(bound by the } \lambda \text{ of the (Beta)-redex)} \\
 & & \mathtt{m} \ \text{ if } \ \mathtt{m} < \mathtt{n} & (\in BV(a))
\end{array}$$

where:

$$\begin{array}{l}
t_i^n(a_1a_2) = t_i^n(a)t_i^n(a_2) \\
t_i^n(\lambda a) = \lambda(t_{i+1}^n(a)) \\
t_i^n(\mathtt{m}) = \begin{array}{ll} \mathtt{m}+\mathtt{n}-1 & \text{if } \ \mathtt{m} \geq \mathtt{i}+1 \\ \mathtt{m} & \text{if } \ \mathtt{m} \leq \mathtt{i} \end{array}
\end{array}$$

Let us give an example: the term $\lambda x.((\lambda y.xy)x)$ is written $\lambda((\lambda(21))1)$ in de Bruijn notation. It reduces to $\lambda x.xx$ or to $\lambda((21)\{1 \leftarrow 1\}) = \lambda(11)$ in de Bruijn notation.

Note that, in this de Bruijn setting, the substitution operation is still only defined in the meta-language. The only operation of this language remains the β-reduction. This is an important drawback when dealing with implementations. We need a more precise language accepting substitution as a first-class citizen for practical purposes.

[1] Some authors use 0 has the first de Bruijn number, see [Cur83] for example.

1.3 λ-calculus with couples

As is well-known, couples and projections may be encoded by λ-terms, say C, *Fst* and *Snd*. The projection rules are then simply implemented by steps of β-reduction: $Fst\,(C\,a\,b) \xrightarrow{\beta^\star} a$. But H. Barendregt[Bar74] has shown that it is impossible to prove the uniqueness of the construction inside the λ-calculus itself. So, if we want to ensure it, we are led to explicitly add three constants C, *Fst*, *Snd*, projection rules and a rule giving this uniqueness of the couple operation, which is called (SP): $C(Fst\,a)\,(Snd\,a) \longrightarrow a$. Remark that this rule is not linear, and such non-linear rules are known to create some difficulties in confluence problems. Indeed, in an untyped setting, this rule is very bad: as proved by J. W. Klop[Klo82](see also [Har89] and [CHL91]), it destroys the confluence property. Note that this problem disappears when typing rules are added.

1.4 η-reduction

Classically, the λ-calculus is also endowed with another rewriting rule, the η-rule: $\lambda x.ax \rightarrow_\eta a$, if $x \notin FV(a)$. This rule η is an important tool, both from practical and theoretical points of view because it expresses extensionality: remark that $(\lambda x.a\,x)b \rightarrow_\beta a\,b$ so this abstraction of x in $a\,x$ is useless from a computational view point. To translate this rule in de Bruijn notation, we have to express the condition "$x \notin FV(a)$" and to do the update of numbers in the reduct a because the reduction removes the λ.

Definition 5 *$a \in \Lambda$ verifies the* condition *$C(\eta)$ if and only if, for any occurrence u of a number p in a, $p \neq (|u|, a) + 1$.*

The decrementation *operation may be applied to any term a verifying $C(\eta)$. Its result is denoted by $a^\downarrow$. The term $a^\downarrow$ is obtained from a by replacing any number p at occurrence u in a by the number $(p-1)$ provided that p verifies $p > (|u|, M) + 1$.*

The η-reduction in Λ is the rewriting relation defined by the rule:

$$\lambda.(a\,1) \longrightarrow a^\downarrow \text{ if } a \text{ verifies } C(\eta)$$

Definition 6 *The system $\beta\eta SP$ is the rewriting system defined on λ-calculus with couples by the rules β, η, (Fst), (Snd), and (SP).*

2 The Categorical Combinatory Logic

The Categorical Combinatory Logic, CCL, was introduced by P.-L. Curien[Cur83] in 1983 as a syntactical model of Cartesian Closed Categories (CCC). It is well-known that these CCCs serve also as models of functional programming languages. CCL is a first-order theory, its operators come directly from categories: they are the composition "$\circ$", the identity *id*, the pairing $<,>$ and the projections *Fst* and *Snd*, the abstraction Λ and the applicator *App*. This algebra is endowed with a set of rules, called $CCL/\beta\eta SP$, composed of four sub-systems: a system containing only the rule (Beta) which starts the process of β-reduction, a system *SL*

which serves to compute the substitution step by step, a system containing two rules (FSI) and (SP) ensuring uniqueness of pairing and a system containing the rules (AI) and (SΛ), which gives the uniqueness of the exponentiation and which is closed to the η-reduction rule of λ-calculus. The subsystem *Subst* is obtained by adding (FSI) and (SP) to *SL*, in order to obtain a weakly confluent system.

	The system SL		
(Ass)	$(x \circ y) \circ z$	$=$	$x \circ (y \circ z)$
(IdL)	$id \circ x$	$=$	x
(IdR)	$x \circ id$	$=$	x
(Fst)	$Fst \circ < x, y >$	$=$	x
(Snd)	$Snd \circ < x, y >$	$=$	y
(Dpair)	$< x, y > \circ z$	$=$	$< x \circ z, y \circ z >$
(DΛ)	$\Lambda(x) \circ y$	$=$	$\Lambda(x \circ < y \circ Fst, Snd >)$
(FSI)	$< Fst, Snd >$	$=$	id
(SP)	$< Fst \circ x, Snd \circ x >$	$=$	x
(Beta)	$App \circ < \Lambda(x), y >$	$=$	$x \circ < id, y >$
(AI)	$\Lambda(App)$	$=$	id
(SΛ)	$\Lambda(App \circ < x \circ Fst, Snd >)$	$=$	x

The λ-calculus with couples, endowed with $\beta\eta SP$ and CCL, endowed with $CCL\beta\eta SP$, are two equivalent equational theories (Curien[Cur83]). The translation of a λ-term, written in de Bruijn notation, into a CCL combinator is straightforward: $1 \Rightarrow Snd$, $n+1 \Rightarrow Snd \circ Fst^n$, $\lambda.a \Rightarrow \Lambda(a)$, $(a\ b) \Rightarrow App \circ < a, b >$ and pairing and projections are translated upon the corresponding operations. For example, the term $\lambda y.(\lambda x.xy)y$, written in De Bruijn notation $\lambda((\lambda.1\ 2)\ 1)$, is translated in the combinator $\Lambda(App \circ < \Lambda(App \circ < Snd, Snd \circ Fst >), Snd >$). Therefore, the translation of a λ-term is a ground term of CCL, that is, a combinator.

An application of the rule (Beta) suppresses the redex, say $App \circ < \Lambda(a), b >$, build a new environment $< id, b >$ which retains the term b to be substituted (id plays the role of nil) and starts the substitution by composing a with this new environment. As environments are represented by pairs, λ-variables, translated in compositions of projections, may access in these environments. The rule (DΛ) pushes an environment y "under" a Λ: the λ-variables bound by this Λ are not concerned by the substitutions recorded in y (if $x = Snd$, then $x \circ < \_, Snd >$ rewrites on Snd), the numbers "inside" y are to be put under a new Λ so they have to be updated and this is done by the composition $y \circ Fst$.

Clearly, a β-step is computed in CCL by a (Beta)+SL derivation. But SL is not weakly confluent and we have to add (FSI) and (SP) to obtain weak confluence so we are led to consider (Beta)+*Subst*. Is this system confluent? The answer is no, due to the (SP) rule. However, it is possible to define a subsytem $\mathcal{E}$ of *Subst* and a subset, say $\mathcal{D}$, of CCL terms, closed under (Beta)+$\mathcal{E}$ derivations, such that this system remains confluent when restricted to $\mathcal{D}$. Moreover, translations of λ-terms belong to $\mathcal{D}$ and (Beta)+$\mathcal{E}$ is sufficient to compute substitutions. To obtain these results, a simple method was introduced by Hardin[Har89], it is called the *interpretation method*[2]. We recall it now.

2.1 The interpretation method

It is based upon the following simple lemma:

Proposition 1 *Let R and $\mathcal{E}$ be reduction relations on a set X such that $\mathcal{E} \subseteq R^*$ and $\mathcal{E}$ is canonical (confluent and terminating). Let $\mathcal{E}(M)$ denote the $\mathcal{E}$-normal form of M.*

If there exists a relation $\mathcal{E}(R)$ on the set of $\mathcal{E}$-normal forms, such that, if $M \longrightarrow_R N$, then $\mathcal{E}(M) \longrightarrow_{\mathcal{E}(R)} \mathcal{E}(N)$ and such that $\mathcal{E}(R) \subseteq R^*$, then $\mathcal{E}(R)$ is confluent iff R is confluent.

The proof is as follows. Suppose that $M \longrightarrow_{R^*} N$ and $M \longrightarrow_{R^*} P$. We deduce by interpretation a $\mathcal{E}(R)$-derivation from $\mathcal{E}(M)$ to $\mathcal{E}(N)$ and another one from $\mathcal{E}(M)$ to $\mathcal{E}(P)$. Now, $\mathcal{E}(N)$ and $\mathcal{E}(P)$ have a common $\mathcal{E}(R)$-reduct. As $\mathcal{E} \subseteq R^*$ and $\mathcal{E}(R) \subseteq R^*$, these terms have a common R^*-reduct. The other way is obtained similarly.

2.2 How to design $\mathcal{D}$

The sub-system *Subst* computes the substitution. It is weakly confluent and terminating. This last, difficult, result was first obtained by Hardin and Laville[HLa86] and has been then reproved by different methods (see [CHR92] and [Zan92]). Now, CCL contains λ-calculus with couples and $CCL\beta\eta SP$ may simulate $\beta\eta SP$. So, in view of the non confluence of $\beta\eta SP$, we may suppose that *Subst*+ (Beta) is not confluent. To obtain a positive confluence result, we have to remove the (SP) and (FSI) rules, which were added to SL to obtain weak confluence and to modify SL in such a way that substitutions remain completely computed by a canonical system. A careful examination of critical pairs of SL on one hand and of the substitution process on the other hand leads to remove (SP), (FSI) and (IdR), because this last rule has a critical pair with (DA) which needs (FSI) to be solved. To compute completely substitution, we have to add two instances of (IdR), the rules ($Fst \circ id \longrightarrow Fst$) and ($Snd \circ id \longrightarrow Snd$). Let $\mathcal{E}$ be the system obtained from *Subst* in this way: it is still canonical so we may define $\mathcal{E}$-interpretation. The next step is to add (Beta): bad step, $\mathcal{E}$+(Beta) is not even weakly confluent, due to a critical pair between (Ass) and (Beta), which would compel us to reintroduce (IdR). The solution is to restrict ourselves to a subset of ground terms such that this critical pair disappears when putting the terms in $\mathcal{E}$-normal

[2]The method has been used independently elsewhere, e.g. in [BrT88]

form: this is the definition of $\mathcal{D}$. We omit the formal definition of this subset, which is quite technical, and refer to [Har89].

Proposition 2 *Let $M \in \mathcal{D}$ be an (IdR)-redex and N its reduct. Then, $\mathcal{E}(M) \equiv \mathcal{E}(N)$.*

The proof may be found in [Har89]. Now, we build the interpretation of (Beta) on $\mathcal{D}$, it is called $sim\beta$ and defined by: $M \longrightarrow_{sim\beta} N$ if $M \longrightarrow_{(Beta)} P$ and $\mathcal{E}(P) = N$.

Proposition 3 *If $M \in \mathcal{D}$ and $M \longrightarrow_{(Beta)} N$, then $\mathcal{E}(M) \longrightarrow_{sim\beta}{}^{*} \mathcal{E}(N)$.*

It remains to prove the confluence of $sim\beta$: it is easy because this interpretation looks like the classical β and we can use Tait-Martin Löf method. From this result, we deduce the confluence of $\mathcal{E}+$ (Beta).

Proposition 4 *1. $\mathcal{E}+$(Beta) is not weakly confluent.*

2. *Restricted to the subset $\mathcal{D}$, $\mathcal{E}+$(Beta) is confluent.*

3. *Subst+ (Beta) is weakly confluent but not confluent.*

4. *Restricted to the subset of translations of λ-terms, Subst+(Beta) is confluent. Moreover, on translations of λ-terms, $sim\beta$ coincides with β-reduction.*

The proof is done by the interpretation method and may be found in [Har89].

3 The $\lambda\sigma$-calculus

CCL is the first known term rewriting sytem, which fully implements substitution but it is not completely satisfying: to study derivations of λ-terms, we have to work with the restriction to $\mathcal{D}$ and this is rather complicated. Moreover, functions and environments are both represented by combinators but their semantic is quite different. To respond to these problems, Curien designed first a calculus, called $\lambda\rho$ [Cur88], containing two sorts: `term` and `substitution` and then, extended it to obtain $\lambda\sigma$-calculus[ACCL89]. $\lambda\sigma$ is a term rewriting system with these two sorts, it is composed of a rule (Beta), which starts the substitution and of a sub-system, called σ, which performs the replacement. This system was introduced without references to CCL, but it turns out that, if we remove the sorts, this system is exactly $\mathcal{E}+$(Beta). So we explain now how this introduction of sorts leads to better confluence results.

Definition 7 *The $\lambda\sigma$-terms are inductively defined by:*

$$\begin{array}{ll} \textit{terms} & a ::= \mathtt{X} \mid \mathtt{1} \mid ab \mid \lambda a \mid a[s] \\ \textit{substitutions} & s ::= \mathtt{x} \mid id \mid \uparrow \mid a \cdot s \mid s \circ t \end{array}$$

where `X` *et* `x` *are metavariables of sort* `term` *and* `substitution`.

A λ-term ab is translated onto the CCL combinator $App \circ < a, b >$. This combinator App is interesting from a categorical view point but it complicates the translation. If we accept to go away from the categorical world, then we may simply keep the application. Now, the CCL composition plays two roles: first, it serves to push the substitution (see rules (Beta),(DΛ)(Dpair), etc.), and so acts between a term and an environment, secondly, it serves to make computations between several substitutions (see the rule (Ass) for example). These uses are separated by the sorts: pushing an environment in a term is done by the operator $[\ ]$[3] while the composition of substitutions is still denoted by " ∘ ". Then the rule (Ass) is duplicated on the rules (Clos), which introduces the composition of substitutions, and (Ass).

$$\text{(Clos)}\quad a[s][t] \longrightarrow a[s \circ t] \qquad \text{(AssEnv)}\quad (s_1 \circ s_2) \circ s_3 \longrightarrow s_1 \circ (s_2 \circ s_3)$$

The pairing in CCL was used to code the λ-application and the environments. The rule (Beta) is now:

$$\text{(Beta)}\quad (\lambda a)\ b \longrightarrow a[b \cdot id]$$

The term a has to be evaluated in the environment defined by the substitution $b \cdot id$. Note that id is the right son of the binary operator $\cdot$ (called *cons*), it was the left son of the pairing in the CCL version: this explains some transpositions in the rules. It is just a matter of notational convention. Remark that the pairing of two terms is no longer a primitive operation: λσ does not contain λ-calculus with couples (but, as explained below, there is still a surjective pairing problem). Now, the access: 1 was coded by *Snd*, here we keep 1; n+1 became, in CCL, $Snd \circ Fst^n$ and is translated here on $1[\uparrow^n]$. The role of ↑ is given by the following rule:

$$\text{(ShiftCons)}\quad \uparrow \circ (a \cdot s) \longrightarrow s$$

The rule (Dpair) is replaced by two rules:

$$\text{(App)}\quad (ab)[s] \longrightarrow (a[s]\ b[s]) \qquad \text{(MapEnv)}\quad (a \cdot s) \circ t \longrightarrow a[t] \cdot (s \circ t)$$

The two instances of (IdR) used in $\mathcal{E}$ become the following rules:

$$\text{(ShiftId)}\quad \uparrow \circ id \longrightarrow \uparrow \qquad \text{(VarId)}\quad 1[id] \longrightarrow 1$$

The rule (IdL) is now defined only for substitutions:

$$\text{(IdL)}\quad id \circ s \longrightarrow s$$

The rule (DΛ) is called here (Abs):

$$\text{(Abs)}\quad (\lambda a)[s] \longrightarrow \lambda(a[1 \cdot (s \circ \uparrow)])$$

So, the λσ-system is defined by:

[3] One may read the term $a[s]$ as the term a in the environment defined by the substitution s.

(Beta)	$(\lambda a)\ b \longrightarrow a[b \cdot id]$
(App)	$(ab)[s] \longrightarrow (a[s]\ b[s])$
(VarId)	$\mathtt{1}[id] \longrightarrow \mathtt{1}$
(VarCons)	$\mathtt{1}[a \cdot s] \longrightarrow a$
(Clos)	$a[s][t] \longrightarrow a[s \circ t]$
(Abs)	$(\lambda a)[s] \longrightarrow \lambda(a[\mathtt{1} \cdot (s \circ \uparrow)])$
(IdL)	$id \circ s \longrightarrow s$
(ShiftId)	$\uparrow \circ id \longrightarrow \uparrow$
(ShiftCons)	$\uparrow \circ (a \cdot s) \longrightarrow s$
(AssEnv)	$(s_1 \circ s_2) \circ s_3 \longrightarrow s_1 \circ (s_2 \circ s_3)$
(MapEnv)	$(a \cdot s) \circ t \longrightarrow a[t] \cdot (s \circ t)$

The sytem $\lambda\sigma$ is ground confluent: the proof uses the interpretation method, see [ACCL89]. But, like $\mathcal{E}+$ (Beta), it is not weakly confluent, due to the lack of the rule (IdR).

Comparing $\lambda\sigma$ and CCL, the sorted version has two important avantages: first, it allows a clean semantic separation between terms and substitutions, secondly the $\lambda\sigma$-ground terms are exactly the λ-terms, and so the sorts avoid the restriction to $\mathcal{D}$. Moreover, the interpretation of $\lambda\sigma$ by σ is exactly the β-reduction.

To obtain weak confluence, we have to add the following rules, getting the system $\lambda\sigma_{SP}$:

(Id)	$a[id] \longrightarrow a$
(IdR)	$s \circ id \longrightarrow s$
(VarShift)	$\mathtt{1} \cdot \uparrow \longrightarrow id$
(SCons)	$\mathtt{1}[\mathtt{s}] \cdot (\uparrow \circ s) \longrightarrow s$

The rule (Scons) is a non-linear one and we encounter the same obstacle.

Proposition 5 *1. $\lambda\sigma_{SP}$ is not confluent.*

2. When restricted to $\lambda\sigma$-terms containing no metavariables of sort `substitution`*, $\lambda\sigma_{SP}$ is confluent.*

The proof of the first point is given in [CHL91], it is done with the interpretation method. Our term used for the counterexample in CCL (see [Har89]) was the translation of a term of λ-calculus with couples, it is no longer the case for the one we use here. However,

this last term comes rather directly from a new counterexample in λ-calculus with couples (see [CH90]). The second point has been proved recently by Rios[Rio92], also with the interpretation method.

4 The $\lambda\sigma_{\Uparrow}$-calculus

The next step is to define a term rewriting system, as good as $\lambda\sigma$ for λ-calculus, which is confluent. The rule (IdR) is needed to solve the critical pair between (Beta) and (AssEnv) (or (Ass)). The rule (Scons) (or (SP)), which destroys the confluence property, is needed to solve a critical pair induced by the one between (Abs) (or (DΛ)) and (IdR):

$$(\lambda(a)[id] \longrightarrow \lambda(a[1 \cdot (id \circ \uparrow)]) \longrightarrow \lambda(a[1 \cdot |])$$

We need the rule (Varshift) to solve this critical pair. But, (Varshift) has a critical pair with (MapEnv) which needs (Scons).

In fact, we get these troubles because the rule (Abs) introduces the binary operator *Cons*. This is to ensure correct computations of numbers: the first component is $\mathtt{1}$, ensuring that bound numbers in a will remain unchanged and the second component is $s \circ \uparrow$, preparing the update of numbers in s. There is no use to anticipate updates of numbers, when crossing a λ. Instead, we may only memorising the fact that a λ has just been crossed. This needs the introduction of a new unary operator $\Uparrow$, called *lift*.

Here is the syntax of the $\lambda\sigma_{\Uparrow}$-calculus, introduced in [HLe89]. We use de Bruijn numbers instead of encoding them by $\mathtt{1}[|^n]$.

$$\begin{array}{ll} \textit{terms} & a ::= \mathtt{X} \mid \mathtt{n} \mid ab \mid \lambda a \mid a[s] \\ \textit{substitutions} & s ::= \mathtt{x} \mid id \mid \uparrow \mid a \cdot s \mid s \circ t \mid \Uparrow(s) \end{array}$$

The constants $\mathtt{n}$ are integers greater than 1. The rule (Abs) is replaced by the following rule, called (Lambda):

$$\text{(Lambda)} \quad (\lambda a)[s] \longrightarrow \lambda(a[\Uparrow s])$$

We need rules to do updates of de Bruijn numbers when they are evaluated in an environment defined by a substitution $\Uparrow(s)$:

$$\text{(FVarLift1)} \quad \mathtt{1}[\Uparrow(s)] \longrightarrow \mathtt{1}$$

$$\text{(RVarLift1)} \quad \mathtt{n+1}[\Uparrow(s)] \longrightarrow \mathtt{n}[s \circ \uparrow]$$

and we rephrase the rules for *cons* with de Bruijn numbers:

$$\text{(FVarCons)} \quad \mathtt{1}[a \cdot s] \longrightarrow a$$

$$\text{(RVarCons)} \quad \mathtt{n+1}[a \cdot s] \longrightarrow \mathtt{n}[s]$$

A number is incremented with the following rule :

$$(\text{VarShift1}) \quad \mathtt{n}[\uparrow] \longrightarrow \mathtt{n{+}1}$$

The constant $\uparrow$ may suppress a $\Uparrow$ as follows:

$$(\text{ShiftLift1}) \quad \uparrow \circ \Uparrow(s) \longrightarrow s \circ \uparrow$$

Two substitutions having a $\Uparrow$ as top-symbol may be composed:

$$(\text{Lift1}) \quad \Uparrow(s) \circ \Uparrow(t) \longrightarrow \Uparrow(s \circ t)$$

Follows the complete system of rules of $\lambda\sigma_{\Uparrow}$. Remark that this system includes the rule (IdR) on substitutions and its version (Id) on the terms. If we keep the representation of de Bruijn numbers by $\mathtt{1}[\uparrow^n]$, then rules (VarShift) and (RVar) may be removed.

(Beta)	$(\lambda a)b$	$\longrightarrow$	$a[b \cdot id]$
(App)	$(ab)[s]$	$\longrightarrow$	$a[s]b[s]$
(Lambda)	$(\lambda a)[s]$	$\longrightarrow$	$\lambda(a[\Uparrow s])$
(Clos)	$(a[s])[t]$	$\longrightarrow$	$a[s \circ t]$
(VarShift1)	$\mathtt{n}[\uparrow]$	$\longrightarrow$	$\mathtt{n{+}1}$
(VarShift2)	$\mathtt{n}[\uparrow \circ s]$	$\longrightarrow$	$\mathtt{n{+}1}[s]$
(FVarCons)	$\mathtt{1}[a \cdot s]$	$\longrightarrow$	a
(FVarLift1)	$\mathtt{1}[\Uparrow(s)]$	$\longrightarrow$	$\mathtt{1}$
(FVarLift2)	$\mathtt{1}[\Uparrow(s) \circ t]$	$\longrightarrow$	$\mathtt{1}[t]$
(RVarCons)	$\mathtt{n{+}1}[a \cdot s]$	$\longrightarrow$	$\mathtt{n}[s]$
(RVarLift1)	$\mathtt{n{+}1}[\Uparrow(s)]$	$\longrightarrow$	$\mathtt{n}[s \circ \uparrow]$
(RVarLift2)	$\mathtt{n{+}1}[\Uparrow(s) \circ t]$	$\longrightarrow$	$\mathtt{n}[s \circ (\uparrow \circ t)]$
(AssEnv)	$(s \circ t) \circ u$	$\longrightarrow$	$s \circ (t \circ u)$
(MapEnv)	$(a \cdot s) \circ t$	$\longrightarrow$	$a[t] \cdot (s \circ t)$
(ShiftCons)	$\uparrow \circ (a \cdot s)$	$\longrightarrow$	s
(ShiftLift1)	$\uparrow \circ \Uparrow(s)$	$\longrightarrow$	$s \circ \uparrow$
(ShiftLift2)	$\uparrow \circ (\Uparrow(s) \circ t)$	$\longrightarrow$	$s \circ (\uparrow \circ t)$
(Lift1)	$\Uparrow(s) \circ \Uparrow(t)$	$\longrightarrow$	$\Uparrow(s \circ t)$
(Lift2)	$\Uparrow(s) \circ (\Uparrow(t) \circ u)$	$\longrightarrow$	$\Uparrow(s \circ t) \circ u$
(LiftEnv)	$\Uparrow(s) \circ (a \cdot t)$	$\longrightarrow$	$a \cdot (s \circ t)$
(IdL)	$id \circ s$	$\longrightarrow$	s
(IdR)	$s \circ id$	$\longrightarrow$	s
(LiftId)	$\Uparrow(id)$	$\longrightarrow$	id
(Id)	$a[id]$	$\longrightarrow$	a

The sytem $\sigma_{\Uparrow}$ is obtained by removing (Beta) from $\lambda\sigma_{\Uparrow}$ and is the system computing substitution. This system is canonical but the proof of its termination cannot be deduced from the one of *Subst*, which is, as we said previously, seriously difficult. Here, fortunately, the termination of $\sigma_{\Uparrow}$ can be etablished via a lexicographic ordering with two polynomial components (see [HLe89] for details).

Proposition 6 *1. The system $\lambda\sigma_{\Uparrow}$ is confluent.*

2. The ground terms of sort `term` *are exactly the λ-terms.*

3. The interpretation of $\lambda\sigma$ by $\sigma_{\Uparrow}$ is simβ, which coincides on λ-terms with the classical β-reduction.

The point a. was first obtained with the following lemma, coined in [Yok89] (see also [Oos92]):

Lemma 1 *Let R and S be two relations defined on a same set X, R being canonical and S being strongly confluent. Suppose that if $f \longrightarrow_S g$ and $\longrightarrow_R h$, then $g \longrightarrow_{R^*} k$ and $h \longrightarrow_{R^*SR^*}$. Then, the relation R^*SR^* is confluent.*

We use this lemma with $S = \sigma_{\Uparrow}$ and a classical parallelisation of (Beta) as R (see [HLe89, CHL91] for details). The proof of a. can also be done with the interpretation method (see [Har92]). This is more complicated but gives the point c. of this proposition.

The system $\lambda\sigma_{\Uparrow}$ may be extended with data constructors and the corresponding rules. For example, one may add two projections *Fst* and *Snd*, and a pairing operation $<,>$ together with the following rules. The termination and confluence properties remain valid.

(Fst)	$Fst(<a,b>)$	$\longrightarrow$	a
(Snd)	$Snd(<a,b>)$	$\longrightarrow$	b
(FstEnv)	$Fst(a)[s]$	$\longrightarrow$	$Fst(a[s])$
(SndEnv)	$Snd(a)[s]$	$\longrightarrow$	$Snd(a[s])$
(DPair)	$<a,b>[s]$	$\longrightarrow$	$<a[s],b[s]>$

5 η-reduction in $\lambda\sigma$-calculi

It is interesting to extend calculi of substitutions with a notion of η-reduction: this is useful for optimisations of code and also for computations of types in higher-order λ-calculi, for example. The theory $\beta\eta$ is confluent. So we expect to find a confluent extension. In CCC, the extensionality is expressed by the uniqueness of the exponentiation and is given, in CCL, by the rules (AI) and (SΛ). But $\mathcal{E}$+(Beta)+ (SΛ) +(AI) is not weakly confluent, even when restricted to $\mathcal{D}$. The subterm $M \circ Fst$ of the left member of (SΛ) makes critical pairs with

several rules of *Subst* and an infinity of rules have to be added to ensure weak confluence. Instead, we may define the relation $c\eta$ by:

$$\Lambda(App \circ <M, Snd>) \longrightarrow_{c\eta} N \quad \text{if} \quad N \circ Fst =_{\mathcal{E}} M$$

Clearly, $c\eta$ contains the (SA) reduction. We may replace the reduction defined by (AI) and (SA) by the transitive closure of the relation (AI) $\cup c\eta$. We call it also $c\eta$.

Proposition 7 *$\mathcal{E} + (Beta) + c\eta$ is confluent on $\mathcal{D}$.*

The proof is done by the interpretation method. The interpretation of $c\eta$ is a relation which coincides exactly with η-reduction on translations of λ-terms (see [Har87, Har89] for details).

The definition of $c\eta$ extends immediately to $\lambda\sigma$-calculi. Here, we have no longer to deal with rule (AI). The rule (SA) is translated onto $\lambda(a[\uparrow]1) \longrightarrow a$ and we encounter all the same difficulties, due to critical pairs. Therefore, the relation $c\eta$ is defined as follows:

$$\lambda(a1) \xrightarrow{c\eta} b \text{ if } a =_{\sigma} b[\uparrow]$$

Proposition 8 *1. $\lambda\sigma_{\Uparrow} + c\eta$ is confluent.*

2. $\lambda\sigma + c\eta$ is confluent.

3. the interpretation of $c\eta$, defined on $\sigma_{\Uparrow}(\sigma)$-normal forms by a step of $c\eta$-reduction followed by a derivation to $\sigma_{\Uparrow}(\sigma)$-normal forms, coincides on λ-terms with η-reduction.

The proof is done by interpretation and needs the development of some technical lemmas like the following one, which ensures the correctness of the definition:

Lemma 2 *Let a and b be two terms in $\sigma_{\Uparrow}$-n.f. Let $p \geq 0$. If $a[\Uparrow^p(\uparrow)] =_{\sigma_{\Uparrow}} b[\Uparrow^p(\uparrow)]$ then $a \equiv b$.*

Let s and t be two substitutions in $\sigma_{\Uparrow}$-n.f. If $s \circ \Uparrow^p(\uparrow) =_{\sigma_{\Uparrow}} t \circ \Uparrow^p(\uparrow)$, then $s \equiv t$.

A rather detailled proof is given in [Har92].

6 Conclusion

The $\lambda\sigma_{\Uparrow}\eta$-calculus is a first complete response to our request, which was to find a proper confluent extension of λ-calculus, computing explicitly substitutions and solving the names'problem. From a rewriting view point, we have with the development leading from CCL to $\lambda\sigma_{\Uparrow}$ a real word example of how to improve confluence properties by modifications of the term rewriting system itself. Also, the interpretation method, which is a very simple trick, appears very powerful: it permits to reduce proofs on "complicated" terms to proofs on "regularized" terms.

From a λ-calculus view point, the two systems are very interesting: they are used to describe and compare some abstract machines[ACCL89, Ler90], which are based on classical

β-strategies. We have still to design new strategies, using their full power, that is, strategies which contain computations between substitutions. This is needed for optimisations (see [Bal86, Har87, Ned92]). These calculi are also a good framework to discuss about optimality and sharing. We know that this is not possible to obtain Lévy's optimality with these systems[Fie90] because they are term rewriting systems so subterms, which code function's bodies, have to be unshared before β-reduction. But, we may discuss about a new kind of optimality, which refer to the number of elementary steps needed to compute the normal form of a given term, whenever it exists. We have also to fully unterstand what are the possible uses of the metavariables of $\lambda\sigma_{\Uparrow}$, which bring a really new capability of calculus: $\lambda\sigma_{\Uparrow}$ is a calculus of contexts. For example, if metavariables represent names of macros in a given text, then the preprocessing of a text is simply described by a first-order substitution applied to these metavariables.

Some open problems involve the rewriting and λ-calculus aspects: the most important one is to prove the termination of $\lambda\sigma$-calculi in a typed framework (see [ACCL89], which contains type systems for $\lambda\sigma$ and may be extended straigthforward to type $\lambda\sigma_{\Uparrow}$). This problem seems rather difficult: to prove the termination of typed λ-calculus is not so easy, this is the same for the termination of *Subst*, and these two points are subgoals of the desired proof.

References

[ACCL89] M. Abadi, L. Cardelli, P.-L. Curien, J.-J. Lévy, *Explicit Substitutions*, Journal of Functional Programming, 1(4), 375-416, 1991.

[Bal86] H. Balsters, *Lambda Calculus extended with segments*. Dissertation, Eindhoven University, 1986.

[Bar84] H. P. Barendregt, *The Lambda-Calculus*, vol 103, Elsevier Science Publishing Company, Amsterdam, 1984.

[Bar74] H. P. Barendregt, *Pairing without conventional restraints*, Zeitschr. J. Math And Logik und Grundlagen p Math, 20, pp 289–306, 1974.

[BrT88] V. Breazu-Tannen, *A combining algebra and higher-order types*, Proc. LICS 88, Edinburgh.

[dBr72] N. de Bruijn, *Lambda-Calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser Theorem*, Indag. Math., 34(5), pp 381–392, 1972.

[dBr78] N. de Bruijn, *Lambda-Calculus notation with namefree formulas involing symbols that represent reference transforming mappings*, Indag. Math., 40, pp 348–356, 1978.

[Cur83] P.-L. Curien, *Categorical Combinators, Sequential Algorithms and Functional Programming*, Research Notes in Theoretical Computer Science, Pitman, London, 1986. Second revised version to appear, Birhkauser, (1993).

[Cur88] P.-L. Curien, *Environment machines*, Note, TCS 82 (2), 389-402, 1991.

[CH90] P.-L. Curien, T. Hardin, *Yet Yet another counterexample for λ-calculus with surjective pairing*, Communication in forum Types (1990).

[CHL91] P.-L. Curien, T. Hardin, J-J Lévy, *Confluence properties of weak and strong calculi of explicit substitutions*, INRIA Report 1617.

[CHR92] P.-L. Curien, T. Hardin, A. Rios, *Strong normalisation of Substitutions*, MFCS92, LNCS Vol 629.

[Fie90] J. Field, *On Laziness and Optimality in Lambda Interpreters*, ACM Conference on Principle of Programming Languages, San Francisco, 1990.

[Har87] T. Hardin, *Résultats de confluence pour les Règles fortes de la Logique Combinatoire Catégorique et Liens avec les Lambda-calculs*, thèse de Doctorat, Université de Paris 7, 1987.

[Har89] T. Hardin, *Confluence Results for the Pure Strong Categorical Logic CCL, λ-calculi as subsystems of CCL*, TCS, 65, pp 291–342, 1989.

[Har91] T. Hardin, *Explicit substitutions and η-reduction on ground $\lambda\sigma$- calculus.* Symposium SemaGraph'91, Nimeguen 1991.

[Har92] T. Hardin, *η-reduction for explicit substitutions*, Algebraic and Logic Programming'92, LNCS 632.

[HLa86] T. Hardin and A. Laville, *Proof of Termination of The Rewriting System Subst on C.C.L.*, Theoretical Computer Sc., 46, pp 305–312, 1986.

[HLe89] T. Hardin, J.J. Lévy, *A confluent calculus of substitutions*, France-Japan Artificial Intelligence and Computer Science Symposium, Izu (Japan) 1989.

[Hin86] R. Hindley and J. Seldin, *Introduction to Combinators and λ- calculus*, Volume 1 of *London Mathematical Society Student texts*, Cambridge University Press, 1986.

[Klo82] J. W. Klop, *Combinatory Reduction Systems*, PhD, Mathematisch Centrum Amsterdam, 1982.

[Ler90] X. Leroy, *The ZINC experiment: an economical implementation of the ML language.* INRIA report 117, 1990.

[Lev78] J.-J. Lévy, *Réductions correctes and optimales dans le Lambda-Calcul*, Thèse d'Etat, Université de Paris 7, 1978.

[Ned92] R.P. Nederpelt, *The fine-structure of lambda calculus*, Eindhoven University of Technology, Computer Science Note 92/07.

[Rio92] A. Ríos, *Variations sur le calcul explicite de la substitution*, Thèse, Université Paris 7, 1992.

[Oos92] V. Van Oostrom, *Confluence by decreasing diagrams*, Vrije Universiteit, de Boelelaan 1081a, 1081 HV Amsterdam. Preprint, 1992.

[Yok89] H. Yokouchi, *Relationship between λ-calculus and Rewriting Systems for Categorical Combinators*, Theoretical Computer Sc. 65, 1989.

[Zan92] H. Zantema, *Termination of term rewriting by interpretation*, Utrecht University, RUU-CS-92-14 report.

ISSN 0249-6399