Combining first order algebraic rewriting systems, recursion and extensional lambda calculi

Roberto Di Cosmo* Delia Kesner §

Abstract

It is well known that confluence and strong normalization are preserved when combining left-linear *algebraic rewriting systems* with the simply typed lambda calculus. It is equally well known that confluence fails when adding either the usual extensional rule for η, or recursion together with the usual contraction rule for surjective pairing.

We show that *confluence* and *normalization* are *modular* properties for the combination of *left-linear* algebraic rewriting systems with typed lambda calculi enriched with *expansive* extensional rules for η and surjective pairing. For that, we use a translation technique allowing to simulate expansions without expansion rules. We also show that confluence is maintained in a modular way when adding *fixpoints.* This result is also obtained by a simple translation technique allowing to simulate bounded recursion with β reduction.

1 Introduction

Confluence and strong normalization for the combination of lambda calculus and algebraic rewriting systems have been the object of many researches [BT88, JO91, BTG94, HM90], where the *modularity* of these properties is studied. However, the lambda calculus under consideration cannot be equipped with extensional rules (notably η) since confluence immediately fails as the following example shows:

Example 1.1 Take a single algebraic rule $fx \xrightarrow{alg} a$ where f and a are algebraic symbols. Then the following diagram cannot be closed:

$$f \xleftarrow{\eta} \lambda x.fx \xrightarrow{alg} \lambda x.a$$

On the other side, there is some recent increasing interest on a different computational interpretation of the extensional axioms, which are turned into *conditional expansion rules* instead of contractions (see [Aka93, Dou93, DCK94b, Cub92, JG92]). For example, the η equality is turned into the rule

$$M \xrightarrow{\eta} \lambda x.Mx \quad \text{if } M : A \to B,\ M \text{ is not a } \lambda\text{-abstraction and } M \text{ is not applied}$$

With this expansionary interpretation the previous counterexample simply goes away as it becomes $f \longrightarrow \lambda x.fx \longrightarrow \lambda x.a$. It is then quite legitimate to ask if it is possible to recover the confluence property when combining *confluent* algebraic rewriting systems with the typed lambda calculus and *expansive extensional rules*, and moreover to ask whether this property can be derived in a

*DMI-LIENS (CNRS URA 1347) Ecole Normale Supérieure - 45, Rue d'Ulm - 75230 Paris France
E-mail:`dicosmo@ens.fr`
§CNRS and LRI - Bât 490, Université de Paris-Sud - 91405 Orsay Cedex, France
E-mail:`kesner@lri.fr`

modular way. Notice that these conditional expansion rules do *not* fit even into the very general framework of [JO91], where higher-order variables are not allowed as left-hand sides of rules. It is to be noticed that, if we just want to get rid of the specific counterexample above, without using expansion rules, there is also the possibility to state that an algebraic term f alone is not a well formed term if it has not a base type. While this approach can also lead to a confluent system, as shown by [**?**], it is important to consider that the expansional reading of extensional rules was not born just for handling the counterexample above, but to solve many different problems (like the ones posed by the combination with the terminal type, or the fixpoint operators): all work done using η as a contraction will inevitably be confronted with the same unsormontable difficulties as soon as

Another natural question also arises when taking into consideration fixpoint operators: whenever we want to show confluence in the presence of fixpoints, we usually resort to the very same technique of labeled reductions originated from Lévy's work on the untyped lambda-calculus [Lév76], that is really not modular. Furthermore, it has already been shown that fixpoint are compatible with *expansion* rules for surjective pairing [DCK94b, Dou93], while recursion together with the (SP)-axiom oriented as a *contraction* rule cause confluence to fail [Nes89]. It is quite reasonable to ask again for a more friendly proof technique based on modular properties, possibly capable of handling conditional expansion rules.

In this paper we answer positively to these questions: confluence and normalization are modular properties when combining *left-linear* algebraic rewriting systems with typed lambda calculi featuring extensional rules, and confluence can be modularly derived when adding fixpoints, even in the presence of conditional expansion rules. We adapt the simulation technique developed in [DCK94b] for an extensional typed lambda calculus with expansion rules, in such a way that confluence and strong normalization in presence of extensionality are both reduced to the already known confluence and strong normalization properties of the system without extensionality.

For the fixpoint combinators we adopt also a similar technique: the confluence property of any *left-linear reduction system* with fixpoints is reduced to the confluence of the system without fixpoints. We also show how to extend this result to the expansive interpretation of extensional rules: they do not fit into the general definition of a left-linear reduction system because they are conditional rules, and must be handled separately.

The paper is logically divided into two main sections: we first show how the combination of left-linear algebraic rewriting systems with the typed lambda-calculus preserves strong normalization *and* confluence even with expansion rules for η and surjective pairing, then we present a general technique for handling fixpoint combinators, even in the presence of expansion rules. These two results will give us the full picture: left-linear algebraic rewriting systems and fixpoint combinators can be added preserving confluence to the extensional simply typed lambda calculus, while fixpoint can be added preserving confluence to any left-linear reduction system containing β-reduction and possibly expansion rules.

These results provide a simple, clean and powerful way of incorporating extensionality into a higher order language with algebraic data types, and have clearly immediate application in the field of automated theorem proving, by providing a simple way of deciding extensional equality that does not rely on ad-hoc normalization strategies as it is done in previous works.

2 Basic definitions

We first recall the standard definitions concerned with algebraic rewriting systems and extensional typed lambda calculus with pairs. We also fix the notations for the different reduction relations.

Definition 2.1 (Signature) A signature $\Sigma = \langle \mathcal{T}, \mathcal{F}, \mathcal{D} \rangle$ consists of

- A set $\mathcal{T}$ of *base types.*
- A set $\mathcal{F}$ of *function symbols*
- A set $\mathcal{D}$ of *declarations* of the form $f : \alpha_1 \to \ldots \to \alpha_n \to \alpha$, where $f \in \mathcal{F}$, $\alpha_1, \ldots, \alpha_n, \alpha \in \mathcal{T}$ and $n \geq 0$. We say that n is the *arity* of f.

We assume the sets $\mathcal{F}$ and $\mathcal{T}$ to be disjoint and we require every functional symbol in $\mathcal{F}$ to have exactly one declaration in $\mathcal{D}$. From now on, we suppose the signature Σ to be fixed.
Given a signature Σ, we define the set of types of our calculus by the following grammar:

$$A \quad ::= \quad \xi \ \mid \ A \times A \ \mid \ A \to A$$

where ξ ranges over the set of base types.
Variables and constants are typed by the following axioms:

$$\begin{array}{lll} x_1 : A_1, \ldots, x_n : A_n \vdash & x_i : A_i & (1 \leq i \leq n) \\ x_1 : A_1, \ldots, x_n : A_n \vdash & f : A & \text{if } f : A \text{ is in the signature} \end{array}$$

where the x_j's are pairwise distinct.

And terms are typed by the following rules:

$$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x : A.M : A \to B} \qquad \frac{\Gamma \vdash M : A \to B \quad \Gamma \vdash N : A}{\Gamma \vdash (MN) : B}$$

$$\frac{\Gamma \vdash M_1 : A_1 \quad \Gamma \vdash M_2 : A_2}{\Gamma \vdash \langle M_1, M_2 \rangle : A_1 \times A_2} \qquad \frac{\Gamma \vdash M : B_1 \times B_2}{\Gamma \vdash \pi_1(M) : B_1} \qquad \frac{\Gamma \vdash M : B_1 \times B_2}{\Gamma \vdash \pi_2(M) : B_2}$$

Definition 2.2 (Algebraic Term) A term is *algebraic* if it is either a variable of base type or has the form $f\ T_1 \ldots T_n$, where $f \in \mathcal{F}$ has arity n, and every T_i is an algebraic term. Note that an algebraic term is always of base type according to our definition.

Definition 2.3 (Algebraic rewriting system) An *algebraic rewriting rule* is an ordered pair (L, R) of algebraic terms such that L is not a variable, and every variable of R appears also in L. An *algebraic rewriting system* $\mathcal{R}$ is a finite set $\{(L_i, R_i)\}_{i=1}^n$ of algebraic rewriting rules.

Definition 2.4 (Left-linear rewriting system) A rewriting system is *left linear* if no variable occurs in the left-hand side of the same rule more then once.

We note $FV(M)$ the set of *free variables* of the term M. We write $[\overline{N}/\overline{x}]$ for the typed substitution mapping each variable $x_i : A_i$ to a term $N_i : A_i$ and $M[\overline{N}/\overline{x}]$ for the term M where each variable x_i free in M is replaced by N_i.

Definition 2.5 (Algebraic reduction) The algebraic reduction between terms is a binary relation $\xrightarrow{\mathcal{R}}$, defined in such a way that $C[T] \xrightarrow{\mathcal{R}} C[H]$ if and only if there exists a substitution θ and a rule $(L, R) \in \mathcal{R}$ such that $T = \theta(L)$ and $H = \theta(R)$.

We identify terms up to α-conversion and we consider the following set of elementary rules:

$$
\begin{array}{lll}
(\beta) & (\lambda x : A.M)N \xrightarrow{\beta} M[N/x] & \\
(\pi_i) & \pi_i\langle M_1, M_2\rangle \xrightarrow{\pi_i} M_i, & \text{for } i = 1, 2 \\
(\delta) & M \xrightarrow{\delta} \langle \pi_1(M), \pi_2(M)\rangle, & \text{if } \begin{cases} M : A \times B \\ M \text{ is not a pair} \end{cases} \\
(\eta) & M \xrightarrow{\eta} \lambda x : A.Mx, & \text{if } \begin{cases} x \notin FV(M) \\ M : A \to C \\ M \text{ is not a } \lambda - \text{abstraction} \end{cases}
\end{array}
$$

Definition 2.6 (Relation $\Longrightarrow$) The one-step reduction relation between terms, denoted $\Longrightarrow$, is defined as the closure of the reduction rules $\beta, \eta, \pi_1, \pi_2, \delta$ for all the contexts *except* application and projection, *i.e*:

- If $M \Longrightarrow M'$, then $MN \Longrightarrow M'N$ except in the case $M \xrightarrow{\eta} M'$
- If $M \Longrightarrow M'$, then $\pi_i(M) \Longrightarrow \pi_i(M')$ except in the case $M \xrightarrow{\delta} M'$

Notation 2.7 *The transitive and the reflexive transitive closure of* $\Longrightarrow$ *are noted* $\Longrightarrow^+$ *and* $\Longrightarrow^*$ *respectively.*

We use $\overset{B}{\Longrightarrow}$ to denote the reduction relation $\Longrightarrow$ *without* extensional rules, and $\overset{E}{\Longrightarrow}$ to denote the relation $\Longrightarrow$ *with only* extensional rules.

Definition 2.8 We denote by $\rightsquigarrow$ the reduction relation $\Longrightarrow \cup \xrightarrow{\mathcal{R}}$, by $\rightsquigarrow_B$ the relation $\overset{B}{\Longrightarrow} \cup \xrightarrow{\mathcal{R}}$ and by $\rightsquigarrow_E$ the relation $\overset{E}{\Longrightarrow} \cup \xrightarrow{\mathcal{R}}$.

We use the standard notions of substitutions, reduction, confluence, normalization, etc from the theory of rewriting systems.

3 Modularity of confluence and strong normalization

We know from [BT88, BTG94] that combining the non-extensional simply typed lambda calculus (even with pairs) with a left-linear confluent algebraic rewriting system preserves confluence. On the other hand, this combination yields a strongly normalizing system when the algebraic one is.

We use here these very same results to show that the restriction to non-extensional lambda calculus can be raised when using expansion rules.

3.1 An overview of the proof technique

To do so, we use a technique originally developed in [DCK94b] for a specific typed lambda calculus with expansion rules, that is general enough to be applied in this context. For that, we define a translation $\_^\circ$ from our calculus into itself satisfying the following property:

- Any reduction step $M \rightsquigarrow N$ can be simulated by a reduction sequence $M^\circ \rightsquigarrow_B^+ N^\circ$
- The translation is the identity on the (η, δ)-normal forms

The first property (called *simulation*) is sufficient to show that strong normalization is preserved when adding expansion rules, while both properties allow to derive the preservation of confluence.

Proposition 3.1 (Normalization via translation) *Given two reduction relations R, S on a given set of terms, and a translation s.t. any reduction step $\xrightarrow{R\cup S}$ can be simulated by a reduction $\xrightarrow{S}^{+}$, then if S is strongly normalizing, also $R \cup S$ is strongly normalizing.*

Proof. Straightforward since any infinite reduction sequence of $R \cup S$ can be turned via simulation into an infinite reduction of S, leading to a contradiction with the hypothesis. □

Proposition 3.2 (Confluence via translation) *Given two reduction relations R, S on a given set of terms, and a translation $\_^\circ$ s.t.*

- *Any reduction step $M \xrightarrow{R\cup S} N$ can be simulated by a reduction $M^\circ \xrightarrow{S}^{+} N^\circ$*
- *The translation is the identity on the R normal-forms*

then if S is confluent and R is weakly normalizing, $R \cup S$ is confluent.

Proof. The following picture shows how to close any diagram of $R \cup S$:

$$\begin{tikzcd}
 & P_1 \arrow[r, "R*"] & \overline{P_1} \arrow[r, phantom, "\equiv"] & {} & \overline{P_1}^\circ \arrow[drr, "S*"] & & \\
M \arrow[ur, "R\cup S*"] \arrow[dr, "R\cup S*"'] \arrow[rr, dotted, no head] & & M^\circ \arrow[urr, "S*"] \arrow[drr, "S*"'] & & & & Q \\
 & P_2 \arrow[r, "R*"'] & \overline{P_2} \arrow[r, phantom, "\equiv"] & {} & \overline{P_2}^\circ \arrow[urr, "S*"'] & &
\end{tikzcd}$$

The dotted line pinpoints the first translation step from M to M°, while $\overline{P_i}$ is any R-normal form of P_i.

The weak normalization property for R ensures the existence of the R normal forms $\overline{P_1}$ and $\overline{P_2}$. Of course, in case S is strongly normalizing such hypothesis on R can be derived from the simulation property, and is then superfluous. The link between the outer and inner diagrams can be done by the second property since translation does not affect R normal forms. Finally, the inner diagram can be closed by the confluence property of the reduction relation S. □

We exhibit now a translation with these properties from $\rightsquigarrow$ into $\rightsquigarrow_B$ so that the desired modularity result can be derived from the above proposition.

3.2 Translating the extensional rules

In this section we show that confluence and strong normalization are modular properties of the combination of left-linear algebraic rewriting systems with the extensional typed lambda calculus with pairing based on *expansion rules*. We first define a translation of terms mapping our calculus into itself such that for every possible reduction in the original system from a term M to another term N, there is a reduction sequence from the translation of M to the translation of N, that is *non empty* and *does not* contain any expansion. We show in this way how the calculus without expansions can be used to simulate the calculus with expansions. For a detailed discussion about the following translation we refer the reader to [DCK94b].

Definition 3.3 (Translation for expansions) To every type C we associate a term, called the *expansor of type* C and denoted Δ_C, defined by induction as follows:

$$\begin{array}{ll} \Delta_{A\to B} = & \lambda x : A \to B.\lambda z : A.\Delta_B(x(\Delta_A z)) \\ \Delta_{A\times B} = & \lambda x : A \times B.\langle \Delta_A(\pi_1(x)), \Delta_B(\pi_2(x))\rangle \\ \Delta_A & \textit{is empty}, \text{ in any other case} \end{array}$$

We then define a translation M° for a term $M : A$ as follows[1]:

$$M^\circ = \begin{cases} M^{\circ\circ} & \text{if } M \text{ is a } \lambda\text{-abstraction or a pair} \\ \Delta_A^k M^{\circ\circ} \quad \text{for any } k > 0 & \text{otherwise} \end{cases}$$

where $\Delta_A^k M$ denotes the term $\underbrace{(\Delta_A \ldots (\Delta_A}_{k \quad times} M) \ldots)$ and $M^{\circ\circ}$ is defined by induction as:

$$\begin{array}{llll} x^{\circ\circ} & = x & f^{\circ\circ} & = f \\ \langle M, N\rangle^{\circ\circ} & = \langle M^\circ, N^\circ\rangle & (\lambda x : B.M)^{\circ\circ} & = \lambda x : B.M^\circ \\ \pi_i(M)^{\circ\circ} & = \pi_i(M^{\circ\circ}) & (MN)^{\circ\circ} & = (M^{\circ\circ} N^\circ) \end{array}$$

We can show by induction on the structure of terms that translation preserves types and leave unchanged terms when expansions are not possible.

Lemma 3.4 *If* $\Gamma \vdash M : A$*, then* $\Gamma \vdash M^\circ : A$*.*

Proof. By induction on the structure of M; see [DCK94b] for details. □

Lemma 3.5 (Translation and normal forms) *If* M *is in* (η, δ)*-normal form, then* $M^\circ = M$*.*

Proof. By induction on the structure of M; see [DCK94b] for details. □

The following property is essential to show that every time we perform a β-reduction on a term M in the original system, any translation of M reduces to a translation of the term we have obtained via $\xrightarrow{\beta}$ from M. Take for example the reduction $(\lambda x : A.M)N \xrightarrow{\beta} M[N/x]$. We know that $((\lambda x : A.M)N)^\circ = \Delta_A^k((\lambda x : A.M^\circ)N^\circ)$ and we want to show that there is a *non empty* reduction sequence leading to $M[N/x]^\circ$. Since $\Delta_A^k((\lambda x : A.M^\circ)N^\circ) \xrightarrow{\beta} \Delta_A^k M^\circ[N^\circ/x]$, we have now to check that the term $(M[N/x])^\circ$ can be reached. We state the property as follows:

Lemma 3.6 *If* $\Gamma \vdash M : A$*, then* $\forall k \geq 0,\ \Delta_A^k M^\circ[\overline{N^\circ}/\overline{x}] \rightsquigarrow^*_B (M[\overline{N}/\overline{x}])^\circ$*.*

Proof. Exactly as in [DCK94b], as algebraic constants behave as free variables. □

Lemma 3.7 *Let* M *be an algebraic* linear *term such that* $\theta(M) = N$*. Then, there is a substitution* φ *such that* $\varphi(M) = N^\circ$ *and* $\forall x \in FV(M)$ *we have* $\varphi(x) = \theta(x)^\circ$*.*

Proof. By induction on the structure of M.

- $M \equiv x$. Then we define $\varphi(x) = \theta(x)^\circ = N^\circ$.
- $M \equiv f\ M_1 \ldots M_n$. Then M is of base type and $N \equiv f\ N_1 \ldots N_n$, where $\theta(M_i) = N_i$ for $i = 1 \ldots n$. By i.h. there are substitutions $\varphi_1, \ldots, \varphi_n$ verifying the hypothesis of the lemma such that $\varphi_i(M_i) = N_i^\circ$ for $i = 1 \ldots n$. Let $\varphi = \varphi_1 \cup \ldots \cup \varphi_n$. Since M is linear $\varphi(f\ M_1 \ldots M_n) = f\ \varphi_1(M_1) \ldots \varphi_n(M_2) = f\ N_1^\circ \ldots N_n^\circ = (f\ N_1 \ldots N_n)^\circ = N^\circ$. As every x in $FV(M)$ is a variable in $FV(M_i)$ for some $1 \leq i \leq n$, then $\varphi(x)$ is uniquely determined by $\varphi_i(x)$ (so $\varphi(x) = \varphi_i(x)$) and by i.h. $\varphi_i(x) = \theta(x)^\circ$. □

[1] Actually, this is really a *family* of translations, but this does not affect the proofs.

Lemma 3.8 *Let M be an algebraic term, θ a substitution with $FV(M) \subseteq Dom(\theta)$ and φ the substitution defined by $\varphi(x) = \theta(x)^\circ$, for every $x \in FV(M)$. Then $\varphi(M) = \theta(M)^\circ$.*

Proof. By induction on M. □

Theorem 3.9 (Simulation for expansions) *If $\Gamma \vdash M : A$ and $M \leadsto N$, then $M^\circ \leadsto^+{}_B N^\circ$.*

Proof. The proof proceeds by induction on the structure of M and then by case-analysis. We consider here only the case $M \equiv f\ P_1 \ldots P_n$ and we refer the reader to [DCK94b] for all the other details.

First of all remark that $f\ P_1 \ldots P_n$ is not reducible by η nor by δ because it is of base type. Therefore, there are just two cases to consider:

- If $N = f\ P_1 \ldots N_i \ldots P_n$, where $P_i \leadsto N_i$, then

$$\begin{array}{lclcl} (f\ P_1 \ldots P_i \ldots P_n)^\circ & = & f\ P_1^\circ \ldots P_i^\circ \ldots P_n^\circ & \leadsto^+{}_B & \text{by i.h.} \\ f\ P_1^\circ \ldots N_i^\circ \ldots P_n^\circ & = & (f\ P_1 \ldots N_i \ldots P_n)^\circ & = & N^\circ \end{array}$$

- If $N = \theta(R)$, where $(L, R) \in \mathcal{R}$ and $M = \theta(L)$. By lemma 3.7 there is a substitution φ such that $\varphi(L) = M^\circ$ and for every $x \in FV(L)$ we have $\varphi(x) = \theta(x)^\circ$. Since every x in $FV(R)$ is also in $FV(L)$, then $\varphi(R) = \theta(R)^\circ$ by lemma 3.8. Hence $M^\circ = \varphi(L) \xrightarrow{\mathcal{R}} \varphi(R) = N^\circ$. □

Theorem 3.10 (Modularity of the strong normalization property) *Let R be any strongly normalizing left-linear algebraic rewriting system. Then R plus the simply typed lambda calculus with pairing and expansion rules for η and δ is strongly normalizing.*

Proof. We know by [HM90, BTG94] that the combination of a strongly normalizing left-linear algebraic rewriting system with the lambda calculus with pairing yields a strongly normalizing reduction. By proposition 3.1 and theorem 3.9, our reduction $\leadsto$ is strongly normalizing. □

Corollary 3.11 *The relation $\stackrel{E}{\Longrightarrow}$ is strongly normalizing.*

Proof. A trivial consequence of the previous theorem. Independent proofs can be found in [Min77, Kes93]. □

Theorem 3.12 (Modularity of the confluence property) *Let R be any confluent left-linear algebraic rewriting system. Then R plus the simply typed lambda calculus with pairing and expansion rules for η and δ is confluent.*

Proof. We know by [BT88, HM90] that the combination of a confluent left-linear algebraic rewriting system with the lambda calculus with pairing yields a confluent reduction. So, our reduction relation $\leadsto_B$ is confluent. In view of proposition 3.2, using theorem 3.9 and lemma 3.5 this is enough to deduce that $\leadsto$ is also confluent. □

4 Adding recursion

We focus now on adding fixpoint operators to a rewriting system S. We assume a new constant $fix_A : (A \to A) \to A$ for each type A, with the reduction rule[2]

$$(fix) \qquad fix_A \xrightarrow{fix} \lambda f.f(fix_A\ f)$$

We also assume that the fix constant does not occur in any rule of S. To simplify notations, we will drop the type suffix in the rest of this section.

[2]We choose to work with this rule instead of the more common one $fix_A\ M \xrightarrow{fix} M(fix_A\ M)$, because it is computationally equivalent but allows to simplify the proofs a bit. This version is also used for example in [HM90].

4.1 The traditional approach

Let us recall here a proof technique essentially due to Lévy (see [Lév76]) that is used quite often to prove that a specific confluent calculus stays confluent when combined with a fixpoint operator. Usually, one considers an auxiliary reduction relation with *bounded* fixpoint operators fix^n and the more restrictive reduction rule[3]

$$(\overline{fix}) \qquad fix^n \xrightarrow{\overline{fix}} \lambda f.f(fix^{n-1}\,f) \quad n>0$$

Essentially, this puts a bound on the depth of any recursive call, so to preserve strong normalization of the original reduction relation S. If it happens that local confluence also still holds, then by Newman's Lemma we have confluence of this auxiliary reduction relation $\xrightarrow{S\cup\overline{fix}}$. Then for the specific system under consideration one has to show the following facts:

(Erasing) If $M\xrightarrow{S\cup\overline{fix}}N$, then $|M|\xrightarrow{S\cup fix}|N|$, where $|M|$ is obtained from M by removing all the indices from the fix terms.

(Lifting) For any reduction sequence $M_0\xrightarrow{S\cup fix}M_1\xrightarrow{S\cup fix}\dots\xrightarrow{S\cup fix}M_n$, there exists an indexed computation $N_0\xrightarrow{S\cup\overline{fix}}N_1\xrightarrow{S\cup\overline{fix}}\dots\xrightarrow{S\cup\overline{fix}}N_n$ such that $|N_i|=M_i$, for $i=0\dots n$ (usually it suffices to index all the fix constructors in M_0 by a number $k\geq n$).

Finally, using just the confluence property for $\xrightarrow{S\cup\overline{fix}}$, the confluence of the reduction relation $\xrightarrow{S\cup fix}$ can be derived using the following general proposition.

Proposition 4.1 *If* $\xrightarrow{S\cup fix}$ *and* $\xrightarrow{S\cup\overline{fix}}$ *satisfy the erasing and lifting properties, then confluence of* $\xrightarrow{S\cup\overline{fix}}$ *implies confluence of* $\xrightarrow{S\cup fix}$.

Proof. Take any diagram $M'\ {}^*\!\xleftarrow{S\cup fix}M\xrightarrow{S\cup fix}{}^*\ M''$. By lifting we get $N'\ {}^*\!\xleftarrow{S\cup\overline{fix}}N\xrightarrow{S\cup\overline{fix}}{}^*\ N''$ with $M=|N|$, $M'=|N'|$ and $M''=|N''|$. Then by confluence of $\xrightarrow{S\cup\overline{fix}}$ there exists a Q s.t. $N'\xrightarrow{S\cup\overline{fix}}{}^*\ Q\ {}^*\!\xleftarrow{S\cup\overline{fix}}N''$ and finally by erasing we get that $M'\xrightarrow{S\cup fix}{}^*\ |Q|\ {}^*\!\xleftarrow{S\cup fix}M''$. □

4.2 Our approach

Only two points of the traditional approach are really needed to prove confluence with unbounded recursion :

1. The erasing and lifting properties
2. Bounded fixpoints preserve confluence of the original specific reduction system

Indeed, the strong normalization property of $\xrightarrow{\overline{fix}}$ is used only as a technical tool to prove confluence of $\xrightarrow{\overline{fix}}$, and has no real interest: we show that bounded fixpoints preserve confluence for *any* reduction relation, as soon as it includes the usual β-rule, and we do not need at all the strong normalization property in the proof. For the sake of completeness, though, we will briefly also show that strong normalization is preserved for *any* reduction relation including the β-rule.

Surprisingly enough, only the lifting property puts some constraint on the reduction system under consideration. But let's start with strong normalization.

---

[3]The corresponding bounded rule for the more common fixpoint rule is $fix^n\ M\xrightarrow{\overline{fix}}M(fix^{n-1}\ M)\quad n>0$.

Instead of adapting an existing normalization proof to take into account the bounded fixpoint reduction rule, as is usually done, it is better to look for a more modular approach.

If one starts with a system that includes the usual β-rule for the lambda calculus, such an approach exists indeed, and is based on a very simple observation, already present for example in [Dou93], where it is used for a specific calculus:

Remark 4.2 (Bounded recursion is just iteration) *What n-bounded recursion does is just n-fold iteration of a function over an argument $fix^0\ f$. Such a behavior can be quite easily simulated in the simply typed lambda calculus by means of the terms $\lambda f.f^n(yf)$ where y is a fresh variable.*

This suggests a very simple and natural translation of the fix^n constants into simply typed λ-terms as follows[4]:

Definition 4.3 (Translation for fixpoints)

$$\begin{array}{rcl} [\![\ fix^0\ ]\!] & = & y \quad (y \text{ fresh}) \\ [\![\ fix^n\ ]\!] & = & \lambda f.f(\lambda h.([\![\ fix^{n-1}\ ]\!]h)f) \end{array}$$

We extend $[\![\ ]\!]$ to all the terms in the calculus by just taking the identity on all terms that are not one of the fix^n constants.

This translation has the property that

$$[\![\ fix^n\ ]\!] = \lambda f.f(\lambda h.([\![\ fix^{n-1}\ ]\!]h)f) \xrightarrow{\beta} \lambda f.f([\![\ fix^{n-1}\ ]\!]f)$$

so that every step of fixpoint reduction in the bounded recursion relation can be simulated by a β step in the relation without recursion[5].

Using this property, we can prove the following

Proposition 4.4 (Simulation for bounded fixpoints) *Given any reduction system S that contains β, if $M \xrightarrow{S\cup\overline{fix}} N$, then $[\![\ M\ ]\!] \xrightarrow{S} [\![\ N\ ]\!]$.*

Proof. We have already shown that a $\overline{fix}$ step can be simulated by a β step. As for S-steps, notice that this translation preserves equality, so all possible S-reductions on a term M are still possible on $[\![\ M\ ]\!]$, even the non linear ones. □

By proposition 3.1, we can conclude that:

Theorem 4.5 (Bounded recursion preserves strong normalization) *Let S be* any *reduction system including the β rule for the simply typed lambda calculus. If S is strongly normalizing, then so is S plus bounded recursion.*

As we already remarked, this very simple and general result has no real interest because, as we will show shortly, we can prove directly that confluence is a modular property with respect to fixpoints. We considered anyway that it was worth presenting here in view of the fact that it does not seem to be very well known: in many works normalization with bounded fixpoints is proven again and again without noticing that it is a fairly general property [PV87, HM90, DCK93].

---

[4] This translation is different from the one suggested in [Dou93], and is crucial to be able to show the modularity of the confluence property.

[5] Notice that we can reason in the same way if we directly take here a given fresh variable y for fix^0.

The traditional approach to confluence with bounded fixpoints would require here to go through any (usually old) already available proof of (local) confluence for the reduction relation without fixpoints, and find the spots where to fit the slight modifications necessary to accommodate the presence of fix^n in the calculus. Once this is done, if the calculus with bounded fixpoints is strongly normalizing (that is now a much easier task using the result above), one gets easily confluence using Newman's lemma.

But we can do much better: indeed, we do not need at all to assume that the calculus without fixpoint is confluent *and* strongly normalizing, because we can show immediately that confluence is a modular property with respect to bounded fixpoints.

Theorem 4.6 (Bounded recursion preserves confluence) *Let L be any* left-linear *reduction system including the β rule for the simply typed lambda calculus. If L is confluent, then so is L plus bounded recursion.*

Proof. We apply proposition 3.2 taking $\overline{fix}$ as R and L as S using our simple translation of the fix^n constants (definition 4.3). We already verified the simulation property. The normalization of $\overline{fix}$ alone trivially comes from the simulation property of the translation applied to the simply typed lambda calculus, that is strongly normalizing. Finally, with our reduction rules for bounded fixpoints, in a $\overline{fix}$ normal form there are no occurrences of fix^n constants with $n > 0$, and we can take fix^0 to be a fixed fresh variable y, so that the translation on these normal forms is just the identity. We can then conclude that L plus bounded recursion is confluent. $\square$

When we turn to the unbounded fixpoint operator, though, we face a difficulty that limits these results: for a general reduction relation $\xrightarrow{S}$, the confluence (with or without normalization) of $\xrightarrow{S\cup\overline{fix}}$ does not implies confluence for $\xrightarrow{S\cup fix}$. The reason is that if there is some rule (like the contractive version of surjective pairing) where some metavariable appears more than once, it is easy to build counterexamples like the following one to the lifting property that is crucial for Lévy's trick:

Example 4.7 Let $P = \lambda x.x$, and consider the reduction sequence $(\lambda p.\langle\pi_1(p), \pi_2(p)\rangle) fix\ P \xrightarrow{\beta}$ $\langle\pi_1(fix\ P), \pi_2(fix\ P)\rangle \xrightarrow{fix} \langle\pi_1(P(fix\ P)), \pi_2(fix\ P)\rangle \xrightarrow{\beta} \langle\pi_1(fix\ P), \pi_2(fix\ P)\rangle \xrightarrow{SP} fix\ P$.

Whatever index n we associate to the original fix operator, there is no way to simulate the SP reduction step in the labeled calculus as required by proposition 4.1, because the occurrences of fix in the first component of the pair $\langle\pi_1(fix\ P), \pi_2(fix\ P)\rangle$ and the occurrences of fix in the second component of the same pair will have labels differing by 1.

Indeed, we know from [Nes89] that Klop's counterexample [Klo80] can be adapted to show that the contractive version of surjective pairing is not confluent in the presence of a fixpoint operator. $\square$

Nevertheless, if the reduction system S is left-linear, then it is quite trivial to verify that $\xrightarrow{S\cup\overline{fix}}$ can simulate $\xrightarrow{S\cup fix}$, so we can conclude that

Theorem 4.8 (Recursion preserves confluence of left-linear systems) *Let L be any* left-linear *reduction system including the β rule for the simply typed lambda calculus. If L is confluent, then so is L plus unbounded recursion.*

This result can be extended to conditional rewriting systems as soon as we can guarantee that our coding of fixpoint operators satisfies the simulation property. Of course, the expansion rules for the extensional axioms for arrow and product types are interesting conditional rewrite rules to consider, and our result still holds for left-linear systems extended with these expansion rules.

The proof technique still relies on a translation: let S be our given left-linear rewriting system extended with the expansion rules for η and surjective pairing, and let R be the bounded fixpoint reduction rule alone. We cannot proceed as we did in the absence of expansion rules, because now the conditional nature of the expansions can prevent the translation of an expandable term from being expandable: for example, fix^n can be expanded to $\lambda g.fix^n g$, but its translation $\lambda f.f(\lambda h.([\![fix^{n-1}]\!]h)f)$ cannot be expanded to $\lambda g.(\lambda f.f(\lambda h.([\![fix^{n-1}]\!]h)f))g$ as it is already a λ-abstraction.

Fortunately, it is possible to modify slightly the original translation to avoid this pitfall:

Definition 4.9 (Expansion compatible translation)

$$
\begin{array}{rcl}
\ll fix^0 \gg & = & y \quad (y \text{ fresh}) \\
\ll fix^n \gg & = & \left\{ \begin{array}{l} \lambda f.f(\lambda h.(\ll fix^{n-1} \gg h)f) \text{ if } fix^n \text{ is applied} \\ \\ (\lambda r.\lambda x.(rx))\lambda f.f(\lambda h.(\ll fix^{n-1} \gg h)f) \text{ if } fix^n \text{ is not applied} \end{array} \right.
\end{array}
$$

We extend the translation $\ll \gg$ to all the terms in the calculus by just taking the identity on all terms that are not one of the fix^n constants.

For this translation, η expansions are no longer problematic: indeed, if fix^n occurs not applied, then it can undergo an η expansion to $\lambda x.(fix^n x)$, but then

$$\ll fix^n \gg \; = (\lambda r.\lambda x.(rx))\lambda f.f(\lambda h.(\ll fix^{n-1} \gg h)f) \xrightarrow{\beta} \lambda x.(\lambda f.f(\lambda h.(\ll fix^{n-1} \gg h)f)x) = \; \ll \lambda x.(fix^n x) \gg$$

Notice that we do not need to modify the translation in order to accommodate surjective pairing expansions, because the fix^n constants have always functional types, so they can never undergo SP expansions.

Anyway, we can be forced, with this definition, to give different translations to different occurrences of a same constant fix^n, depending on the fact that they occur applied or not. This fact has the negative consequences that:

- the translation no longer preserves equality, so it is wiser to restrict our attention to left linear rewriting systems, that are the only system we are really interested in, in view of the Theorem 4.11 for unbounded fixpoints;
- it is no longer trivial that if $M \xrightarrow{S \cup \overline{fix}} N$, then $\ll M \gg \xrightarrow{S} \ll N \gg$.
  Indeed, in the presence of a rule of the form:

  $$xy \xrightarrow{extract} x$$

  the term $fix^n N$ would reduce to fix^n, while on the translations we have

  $$\ll fix^n \; N \gg \equiv (\lambda f.f(\lambda h.(\ll fix^{n-1} \gg h)f))N \xrightarrow{extract} \lambda f.f(\lambda h.(\ll fix^{n-1} \gg h)f)$$

  but this last term does not reduce to $\ll fix^n \gg \equiv (\lambda r.\lambda x.(rx))\lambda f.f(\lambda h.(\ll fix^{n-1} \gg h)f)$ as would be required. What is happening here is that an applied subterm is extracted and put into a non applied position. We will call *extractive* a rule capable of doing this.[6]
  For this reason, we will focus only on non-extractive rules, in view of the following

[6]This can only be a higher-order rule, but we will stay more general by forbidding only extractive rules and not all the higher order rules.

Proposition 4.10 (Simulation for fixpoints with expansions) *Let S be any left-linear rewriting system containing β and the expansion rules, s.t. no rule is extractive.*
If $M \overset{S\cup\overline{fix}}{\longrightarrow} N$, then $\ll M \gg \overset{S}{\longrightarrow} \ll N \gg$.

Proof. We first remark that β, η and SP expansions are non-extractive rules. This is evident for η and SP, while for β it comes from the fact that substitution can put a non applied term into applied position, but not vice-versa.

As for the simulation, we have already seen that there is no problem with η and SP expansions. Also,we can still simulate fix^n reductions (applied or not) using β:

$$\ll fix^n \gg = \begin{cases} \lambda f.f(\lambda h.(\ll fix^{n-1} \gg h)f) \overset{\beta}{\longrightarrow} \lambda f.f(\ll fix^{n-1} \gg f) = \ll \lambda f.f(fix^{n-1} f) \gg \\ \\ (\lambda r.\lambda x.(rx))\lambda f.f(\lambda h.(\ll fix^{n-1} \gg h)f) \overset{\beta}{\longrightarrow} \lambda f.f(\ll fix^{n-1} \gg f) = \ll \lambda f.f(fix^{n-1} f) \gg \end{cases}$$

Finally, notice that a nonextractive rule cannot extract an applied subterm and put it in a non applied position, but it can put a non-applied subterm in an applied position, like in the case of the following β reduction:

$$(\lambda x.xM)fix \overset{\beta}{\longrightarrow} fix\ M$$

This kind of situation is very easy to handle, because the translation of a non-applied fix^n constant reduces (via β) to the translation of an applied fix^n constant, so we just need to perform some additional β steps. On the example above:

$$\begin{array}{rcl} \ll (\lambda x.xM)fix \gg & \equiv & (\lambda x.xM)((\lambda r.\lambda x.(rx))\lambda f.f(\lambda h.(\ll fix^{n-1} \gg h)f)) \\ & \overset{\beta}{\longrightarrow} & ((\lambda r.\lambda x.(rx))\lambda f.f(\lambda h.(\ll fix^{n-1} \gg h)f))\ M \\ & \overset{\beta}{\longrightarrow}{}^{*} & (\lambda f.f(\lambda h.(\ll fix^{n-1} \gg h)f))\ M \equiv \ll fix\ M \gg \end{array}$$

□

Then, we can apply again proposition 3.2 and proposition 4.1 and conclude that

Theorem 4.11 (Modularity of confluence with fix and expansions) *Let L be any left-linear non-extractive rewriting system containing β and extended with expansion rules for η and surjective pairing. If L is confluent, then so is L plus unbounded recursion.*

5 Conclusions and future work

We have shown that extensional equalities can be easily incorporated in higher order programming languages with algebraic data-types: the problems encountered in previous work to accomplish this goal where not due to extensional equalities themselves, but to the wrong choice of orientation of the associated rewrite rules. The well known modularity results for confluence and normalization extend naturally once we choose expansion rules instead of the traditional contractive ones, thus providing a fully satisfactory solution to this long standing problem. In particular, this proves a simple way of deciding extensional equality that does not rely on ad-hoc normalization strategies as is done in previous works. We believe that this new approach can be successful also in the framework of polymorphic calculi, where the contractive interpretation of extensional equalities poses similar problems.

We have also shown how to deal in full generality with fixpoint combinators: confluence is preserved under very permissive hypotheses, so that for many interesting calculi it is now possible to focus on the recursion-free fragment.

Based on an earlier presentation of these results which appeared in [DCK94a], Oostrom showed that the same results can be obtained using a different proof technique that makes use of developments **??**.

These observations have another nice consequence: it is a long time that many people (including the authors) was looking for a confluent rewriting system for the lambda calculus with extensional (categorical) sums. We know that categorical sums are incompatible with arbitrary fixpoints on terms, so the goal was set to find such a system without general recursion, but with weaker notions, like integer iteration or no recursion at all. The last theorem above narrows sharply the region of this quest: it tells us that any confluent (and consistent) left-linear rewriting system with just β and extensional sums would yield a confluent (and consistent) rewriting system with unbounded recursion, which we know is impossible. *So, if any confluent reduction system ever exists for extensional sums, it cannot be left-linear.*

This fact is stated in a weaker form in [Dou93]: there it is noticed that an equivalent of theorem 4.5 used in conjunction with Lévy's trick rules out any left-linear confluent *and* strongly normalizing rewriting system (but this simple observation does not hold in the presence of conditional rules). Our more general technique can be used to rule out even the conditional rule (!+) suggested in [Dou93].

References

[Aka93] Yohji Akama. On Mints' reductions for ccc-Calculus. In *Typed Lambda Calculus and Applications*, number 664 in LNCS, pages 1–12. Springer Verlag, 1993.

[BT88] Val Breazu-Tannen. Combining algebra and higher order types. In IEEE, editor, *Proceedings of the Symposium on Logic in Computer Science (LICS)*, pages 82–90, July 1988.

[BTG94] Val Breazu-Tannen and Jean Gallier. Polymorphic rewiting preserves algebraic confluence. *Information and Computation*, 1994. To appear.

[Cub92] Djordje Cubric. On free ccc. Distributed on the `types` mailing list, 1992.

[DCK93] Roberto Di Cosmo and Delia Kesner. A confluent reduction for the extensional typed $\lambda-$calculus with pairs, sums, recursion and terminal object. In Andrzej Lingas, editor, *Intern. Conf. on Automata, Languages and Programming (ICALP)*, volume 700 of *Lecture Notes in Computer Science*, pages 645–656. Springer-Verlag, 1993.

[DCK94a] Roberto Di Cosmo and Delia Kesner. Combining first order algebraic rewriting systems, recursion and extensional lambda calculi. In Serge Abiteboul and Eli Shamir, editors, *Intern. Conf. on Automata, Languages and Programming (ICALP)21*, number 820 in Lecture Notes in Computer Science, pages 462–472. Springer-Verlag, July 1994.

[DCK94b] Roberto Di Cosmo and Delia Kesner. Simulating expansions without expansions. *Mathematical Structures in Computer Science*, 4:1–48, 1994. A preliminary version is available as Technical Report LIENS-93-11/INRIA 1911.

[Dou93] Daniel J. Dougherty. Some lambda calculi with categorical sums and products. In *Proc. of the Fifth International Conference on Rewriting Techniques and Applications (RTA)*, 1993.

[HM90] Brian Howard and John Mitchell. Operational and axiomatic semantics of pcf. In *Proceedings of the LISP and Functional Programming Conference*, pages 298–306. ACM, 1990.

[JG92] Colin Barry Jay and Neil Ghani. The virtues of eta-expansion. Technical Report ECS-LFCS-92-243, LFCS, 1992. University of Edimburgh, to appear inJournal of Functional Programming.

[JO91] Jean-Pierre Jouannaud and Mitsuhiro Okada. A computation model for executable higher-order algebraic specification languages. In *Proceedings, Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 350–361, Amsterdam, The Netherlands, July 15–18 1991. IEEE Computer Society Press.

[Kes93] Delia Kesner. *La définition de fonctions par cas à l'aide de motifs dans des langages applicatifs.* Thèse de doctorat, Université de Paris XI, Orsay, december 1993. To appear.

[Klo80] Jan Wilhelm Klop. Combinatory reduction systems. *Mathematical Center Tracts*, 27, 1980.

[Lév76] Jean-Jaques Lévy. An algebraic interpretation of the $\lambda\beta\kappa$-calculus and a labelled λ-calculus. *Theoretical Computer Science*, 2:97–114, 1976.

[Min77] Gregory Mints. Closed categories and the theory of proofs. *Zapiski Nauchnykh Seminarov Leningradskogo Otdeleniya Matematicheskogo Instituta im. V.A. Steklova AN SSSR*, 68:83–114, 1977.

[Nes89] Dan Nesmith. An application of klop's counterexample to a higher-order rewrite system. Draft Paper, 1989.

[PV87] Axel Poigné and Josef Voss. On the implementation of abstract data types by programming language constructs. *Journal of Computer and System Science*, 34(2-3):340–376, April/June 1987.