Label-Selective λ-Calculus Syntax and Confluence†

Hassan Aït-Kaci[1] and Jacques Garrigue[2]

[1] Digital Equipment Corporation, Paris Research Laboratory
85 Avenue Victor Hugo, 92500 Rueil-Malmaison, France
[2] Department of Information Science, The University of Tokyo
7-3-1 Hongo, Bunkyo-ku, Tokyo 113, Japan

Abstract. We introduce an extension of λ-calculus, called *label-selective λ-calculus*, in which arguments of functions are selected by labels. The set of labels includes numeric positions as well as symbolic keywords. While the latter enjoy free commutation, the former must comply with relative precedence in order to preserve currying. This extension of λ-calculus is conservative in the sense that when the set of labels is the singleton $\{1\}$, it coincides with λ-calculus. The main result of this paper is that the label-selective λ-calculus is confluent. In other words, argument selection and reduction commute.

Keywords. λ-Calculus, record calculus, concurrency, communication.

1 Synopsis

Many modern programming languages allow specifying arguments of functions and procedures by symbolic keywords as well as using the traditional and natural numeric positions [16, 12, 4]. Symbolic keywords are usually handled as syntactic sugar and "compiled away" as numeric positions. This is made easy if the language does not support currying (like Common LISP or ADA). Even if currying is supported and the situation reduced to numeric positions, it is allowed strictly in a left-to-right order so that the first argument is "consumed" before the second. In general, if a function f is defined on two arguments and it is desired that the second be consumed before the first, one must resort to using an explicit closure of form $\lambda x.\lambda y.f(y,x)$ and curry that one. However, the cost incurred (the closure construction and ensuing weight of handling in terms of depth of stack, *etc.*) is undue since out-of-order currying simply amounts to commutation of stack offsets.

More precisely, currying is possible thanks to the following natural isomorphism: $A \times B \to C \;\simeq\; A \to (B \to C)$ for any set A, B and C. However, there is another obvious natural isomorphism that could also be useful; namely, $A \times B \;\simeq\; B \times A$. Hence we should be able to exploit this directly in the form: $A \to (B \to C) \;\simeq\; B \to (A \to C)$. One way to do that is to use a style of Cartesian product more of a *category-theoretic*, as opposed to *set-theoretic*, flavor. By this we mean that if projections π_1 and π_2 were

† This is a short version of [2]. We have systematically omitted all proofs. Please refer there for details of all the proofs.

used explicitly instead of the implicit 1*st* and 2*nd* of the $\times$ notation, then instead of $A \times B$ we would write $\pi_1 \Rightarrow A \times \pi_2 \Rightarrow B$. Thus, allowing this explicit product expression makes Cartesian product commutative explicitly, as opposed to "up to isomorphism." Indeed, it becomes obvious that:[3] $\pi_1 \Rightarrow A \times \pi_2 \Rightarrow B \simeq \pi_2 \Rightarrow B \times \pi_1 \Rightarrow A$, and thus that: $\pi_1 \Rightarrow A \to (\pi_2 \Rightarrow B \to C) \simeq \pi_2 \Rightarrow B \to (\pi_1 \Rightarrow A \to C)$.

The advantage of explicit projections is clear: one can account directly for symbolic keywords since these play precisely the role of projections. The other benefit is the aforementioned permutativity of currying which allows out-of-order partial application of function to its arguments. For example, an out-of-order application like $f(2 \Rightarrow a)$ can be readily used when there is a need to consume the second argument before the first, as opposed to the more complex and costly $(\lambda x.\lambda y.f(y,x))(a)$.

The drawback of explicit projections, however, is also obvious: implicit argument positions as numeric offset is lost, and the notation is more cumbersome. It is indeed much easier to write $f(x,y)$ instead of $f(1 \Rightarrow x, 2 \Rightarrow y)$ every time we need to apply f to two arguments.

So the question is: can we allow freely mixing implicit and explicit argument selectors safely? In other words, can we allow the notation $f(x,y)$ to be syntactic sugar for explicitly selecting $f(1 \Rightarrow x, 2 \Rightarrow y)$? If we do, the least we should require is that the "all-functions-are-unary" paradigm of λ-calculus be retained. This means that the equation $f(x,y) = f(x)(y)$ should hold for any such expression. However, the syntactic sugaring gives, on one hand, $f(x,y) = f(1 \Rightarrow x, 2 \Rightarrow y)$, and on the other hand, $f(x)(y) = f(1 \Rightarrow x)(1 \Rightarrow y)$. Therefore the free syntax should guarantee that $f(1 \Rightarrow x, 2 \Rightarrow y) = f(1 \Rightarrow x)(1 \Rightarrow y)$. In other words, stack offset permutation must be built into the rule of application at numeric positions. This is essentially what is performed in the extension of λ-calculus that we propose here.

Relation to other work There is an immediate relation between our calculus and the notation with offsets introduced by de Bruijn [7] and used for the compilation of λ-calculus in the style of the SECD machine [11]. In fact, our calculus enforces commutativity of these indices and therefore extends the use of de Bruijn offsets for that model of implementation to include label-selective argument passing. In that way, selective currying can be statically compiled into direct stack access by generating simple arithmetic code involving de Bruijn offsets and selector numbers. Hence, our work is a simple and natural generalization of de Bruijn's idea. We have already adapted the calculus of explicit substitutions [1], and are currently working on a compiling scheme for label-selective λ-calculus based on it.

Another, albeit remote since unexplored, potential connection may be with the recent work of Ohori in compiling extensible records for functional programming [15]. Indeed, records are essentially labeled Cartesian products. Since that style of records allows extensions and out-of-order labels, it is possible to use them in a way similar to ours for passing arguments. At this time, the potential connection is a simple speculation and begs for deeper study.

An intuitive, but accurate, explanation of label-selective λ-calculus can be given as extracting implicit concurrency from λ-calculus. It is well-known that λ-calculus is a

[3] Parse the following with '$\Rightarrow$' binding tighter than '$\times$' or '$\to$'.

sequential calculus and for a clear reason: function application is not commutative. This inherent sequentiality is exacerbated all the more by the strict syntactic left-associativity of application adopted by λ-calculus. Hence, our idea is to reveal the inherent concurrency lost in λ-calculus; namely, commutation of arguments in applications. The syntax and operational semantics that we propose are precisely meant to expose, explicate, and exploit this implicit concurrency. This concurrency is inherent in λ-calculus in the sense that it does not interfere with the confluence of the calculus. This would not be the case with a fully concurrent extension of λ-calculus using parallel composition, a commutative monoid. Thus does our calculus differ from the known calculi for communication of concurrent processes [5, 14, 13].

In [5], Gérard Boudol proposes γ-calculus, an extension of λ-calculus based on realizing that β-reduction is *communication* between a receiving λ-abstraction and a sending operand along one single channel called λ. Thus, the argument of a β-redex is implicitly prefixed with $\bar{\lambda}$. This idea is taken to its full extent by Robin Milner in [14] where, rather than λ alone, there are (countably) many channel *names*. In both Milner's and Boudol's calculi, parallel composition is used to achieve full concurrency and thus, naturally, confluence is lost. By contrast, label-selective λ-calculus is *not* a fully concurrent calculus. Indeed, our calculus is a confluent one. It explicates the fine interaction between functional application as process communication along channel names that are identified, not as λ's as in [14, 5], but as explicit *position* names. This is a wholly different insight. In addition, the availability of *numeric* channels and their laws of relative commutation allows also to speak of *relatively* numbered channels, as opposed to *absolutely* named channels only.

We are also developing, and will report later [3], our label-selective calculus as a true calculus of communication and concurrency. We plan to extend the calculus along the lines of Robin Milner's π-calculus, adding, for example, process operators, such as parallel composition and non-deterministic choice, as well as exploring other directions, for example, by allowing computable channel names. One of the gains expected is that λ-calculus will need not be *encoded* as in [13], but directly embedded as syntactic identity.

Very recently,[4] we became aware of a yet unpublished paper by Laurent Dami [6]. In that work, Dami develops a complex calculus of record objects with a part dealing with functions with named arguments. At a first glance, because it allows named arguments and out-of-order application, Dami's calculus looks reminiscent of the part of our calculus dealing with symbolic positions. It is in fact fundamentally different. In Dami's calculus, a functional abstraction is seen as a pair consisting of a name and an expression (its body)—the name (the "inlet" [*sic*]) designates the *result* of applying the function to arguments. Arguments are identified as the "unprotected" [*sic*] (*i.e.*, free) names in the expression. To avoid confusion in name references, a "protection" level in the form of a string of backslashes may prefix a name's occurrence—if a name's occurrence is prefixed with *n* backslashes in an expression, that reference occurs nested *n* levels deep from its free scope.[5] Accordingly, an application is also a pair made of a name and an expression. The name designates the unprotected reference (the

[4] After this paper had been accepted to appear in these proceedings.

[5] In other words, the number of slashes is the reference's De Bruijn index.

"outlet" [*sic*]) in the applied function's body to substitute for the expression. Hence, Dami's application is really instantiation through a first-order substitution that associates expressions to names.[6] Clearly, this view allows out-of-order argument consumption. In fact, Dami's functions are one-fielded records from which to extract the result (by field-name selection). Complex records are contructed by two operations—a product (called combination) and a coproduct (called alternation)—and are objects behaving like function products and sums, respectively. Besides its intriguing back-to-front view of functions, an essential difference between Dami's calculus and ours is that his calculus does not distinguish variable names and position names—the two are confused. Although quite original and interesting, his calculus departs, unlike ours, in some rather basic way from λ-calculus.[7]

In summary, what we recount in this paper, has not, to our knowledge, been studied as such.

Organization of paper We have organized this paper as follows. In Section 2.1 we introduce our language of selective λ-terms. In Section 2.2 we present reduction systems for these terms. The core of the paper lies in Section 3 where we give the proof of confluence of selective λ-calculus.[8] Section 4 is a reflection on the link between symbolic and numeric labels. Finally, we close the paper with some conclusion and a brief discussion of further work to follow this idea in Section 6.

2 Selective λ-terms

2.1 Syntax

Selective λ-terms are formed by variables taken from a set $\mathcal{V}$, and two labeled constructions: abstraction and application. The labeling is done with labels taken from a set of *position labels* $\mathcal{L}$. This set is the disjoint union of two sets: the set $\mathcal{N} = \mathbb{N} - \{0\}$ of *numeric labels*, and the set $\mathcal{S}$ of *symbolic labels*. Each of the three sets $\mathcal{N}$, $\mathcal{S}$, and $\mathcal{L}$, is totally ordered. Namely, $\mathcal{N}$ is ordered with the natural number ordering, that we shall write $<_{\mathcal{N}}$; $\mathcal{S}$ is ordered with a linear order that we write $<_{\mathcal{S}}$; and, $\mathcal{L}$ is ordered by the order $<_{\mathcal{L}}$ such that $<_{\mathcal{L}} = <_{\mathcal{N}}$ on $\mathcal{N}$, $<_{\mathcal{L}} = <_{\mathcal{S}}$ on $\mathcal{S}$, and $\forall (n,p) \in \mathcal{N} \times \mathcal{S},\ n <_{\mathcal{L}} p$. In other words, all numeric labels are less than all symbolic labels.

We will denote variables by x, y, labels in $\mathcal{L}$ by p, q, reserving m, n to numbers in $\mathcal{N}$, and λ-expressions by capitals.

We can define the syntax of λ-terms as: $M ::= x \mid \lambda_p x.M \mid M \mathbin{\widehat{}}_p M$; respectively: variables, abstractions, and applications. We will say of a term $\lambda_p x.M$ that it *"abstracts x at p in M,"*, and of the term $M \mathbin{\widehat{}}_p N$, that it *"applies M to N through p."*

It will often be convenient to break the atomicity of an abstraction or an application. In the abstraction $\lambda_p x.M$, the part $\lambda_p x$ will be called its *abstractor*, and M its *body*. In

[6] In a sense, plugging these expressions into the free "outlets!"

[7] Quoting Dami's own words: *"an abstraction (...) is somewhat similar to a λ-abstraction (...). However, there are perhaps more differences than similarities."* Indeed!

[8] More precisely, it is the outline of the full proof since we omitted the proofs of all theorems due to space constraints. See [2] for details.

the application $M \mathbin{\widehat{p}} N$, the part $\mathbin{\widehat{p}} N$ will be called the *applicator*. By *entity*, we will mean either an abstractor or an applicator (in which case we speak of a *labeled entity*), or simply a variable.

Relative and absolute positions Before we delve into the technicalities of reduction, let us give some intuition to justify this syntax.

Symbolic labels are what we referred to as "keywords" in the introduction. A useful way of thinking of these symbols is to see them as *channel names* used for process communication [14]. Here, a process is a λ-term, where *sending* is performed by applicators and *receiving* by abstractors. If an application is performed ("sends arguments") through two different channels p and q, then clearly there cannot be any ambiguity as far as which abstractor will "receive" them. Hence, these reductions ("communications") may be done in any order, with the same end result. However, if that situation arises with $p = q$, then clearly the order in which they are performed will matter. In this case, the rules will insure that reduction will respect the order specified syntactically. In other words, several arguments sent through the same channel are "buffered" in sequence.[9]

If numeric labels are always kept explicit, then the above view applies to them as well. Indeed, recall from the introduction that the free syntax of function application to several arguments at a time uses their positions as Cartesian projections; *e.g.*, $f(a_1, \ldots, a_n)$ may be seen as the more explicit $f(1 \Rightarrow a_1, \ldots, n \Rightarrow a_n)$. However, numeric labels do not quite behave like symbolic labels in that a number is always *implicitly* seen as the *first* position *relatively* to the form on its left. More precisely, currying works by seeing each argument as the first one relatively to the form on its left. This has the benefit of simplifying the rule of functional reduction to be a *local* rule never needing to consider more than a single argument at a time. So, clearly, we do want to allow using relative argument positions.

Nevertheless, it is more natural to use absolute positions "packaged" as labeled Cartesian tuples. For instance, it is easier to write $(\lambda(1 \Rightarrow x, 2 \Rightarrow y, 4 \Rightarrow z).M) \frown (1 \Rightarrow a, 4 \Rightarrow b)$ rather than $(\lambda_1 x.\lambda_1 y.\lambda_2 z.M) \mathbin{\widehat{1}} a \mathbin{\widehat{3}} b$. However, the latter fully curried form is needed to express reduction with local rules. Fortunately, translation from the notation with absolute labels to a fully curried one with relative labels is in fact systematic: one need simply subtract from each numeric label the number of numeric-labeled components to its left in the labeled Cartesian product. Namely, for any $i_k, j_\ell \in \mathcal{N}$ such that $i_k < i_{k+1}, j_\ell < j_{\ell+1}$, $(1 \leq k \leq n, 1 \leq \ell \leq m)$:

$$(\lambda(i_1 \Rightarrow x_1, \ldots, i_k \Rightarrow x_k, \ldots, i_n \Rightarrow x_n)\,.\,M) \frown (j_1 \Rightarrow N_1, \ldots, j_\ell \Rightarrow N_\ell, \ldots, j_m \Rightarrow N_m)$$

translates into: $(\lambda_{i_1} x_1 \ldots \lambda_{i_k - k+1} x_k \ldots \lambda_{i_n - n+1} x_n\,.\,M) \mathbin{\widehat{j_1}} N_1 \ldots \mathbin{\widehat{j_\ell - \ell+1}} N_\ell \ldots \mathbin{\widehat{j_m - m+1}} N_m$. With this, we are justified to limit our syntax to that of relative-labeling lending itself

[9] In fact, we are also considering a possible variation of our calculus where this sequential buffering is not guaranteed. Rather, several arguments received on a given channel are chosen non-deterministically. This interesting twist yields essentially the functionality of asynchronous process communication, at the expense, of course, of confluence. That work is the object of our current study and will be reported later [3].

to simpler local reduction rules, while still keeping the freedom of a flexible surface syntax with Cartesian tuples using absolute position labeling.

Now, a reasonable question that one may have is whether we could not also treat symbolic labels as we do numeric labels. That is, we could envisage using a function associating each symbol to its predecessor in the linear order of symbols, thus doing away with names altogether.[10] This, however, would be possible only if the order on $\mathcal{S}$ were not dense. Since, in practice, $\mathcal{L}$ is the free monoid, generated by a subset of the ASCII alphabet, and is densely ordered by lexicographic ordering, this is ruled out. Hence, symbolic labels always designate *absolute* positions of arguments. In other words, packaging symbolic-labeled arguments in labeled Cartesian tuples is always safe since they are not concerned with relative positioning. In fact, the ordering on symbols is only necessary as a trick to avert non-termination so that rules may perform well-founded label commutation.

Substitutions Substitution of variables by λ-expressions needs the same precautions as in λ-calculus and obeys exactly the same rules. As usual, we use the equal sign $(=)$ to mean syntactic equality modulo α-conversion, defining α-conversion as for classical λ-calculus.

Let $\mathrm{FV}(M)$ be the set of free variables in M; that is, variables that are not abstracted anywhere in M. The expression $[N/x]M$ denotes the term obtained by replacing all the free occurrences of variable x by N in (an appropriate α-renaming of) M. That is,

$$
\begin{array}{lll}
[N/x]x & = N & \\
[N/x]y & = y & \text{if } y \in \mathcal{V},\ y \neq x \\
[N/x](M_1 \mathbin{\widehat{}}_p M_2) & = ([N/x]M_1) \mathbin{\widehat{}}_p ([N/x]M_2) & \\
[N/x](\lambda_p x.M) & = \lambda_p x.M & \\
[N/x](\lambda_p y.M) & = \lambda_p y.[N/x]M & \text{if } y \neq x \text{ and } y \notin \mathrm{FV}(N) \\
[N/x](\lambda_p y.M) & = \lambda_p z.[N/x][z/y]M & \text{if } y \neq x \text{ and } y \in \mathrm{FV}(N), \\
 & & \text{and } z \notin \mathrm{FV}(N) \cup \mathrm{FV}(M).
\end{array}
$$

2.2 Reduction systems

We introduce three distinct groups of reduction rules: β-reduction and two reordering systems. What we shall eventually call label-selective λ-calculus is the system freely combining β-reduction and reordering.

β-Reduction Intuitively, β-reduction for labeled terms can be performed as soon as an abstraction at position p is applied through the same position p to a term.

$$(\beta)\ (\lambda_p x.M) \mathbin{\widehat{}}_p N \to [N/x]M$$

[10] This would amount to "compiling them away" as alluded to in the introduction.

Reordering rules Clearly, some reordering of abstractors and applicators must be performed in order to make β-reduction possible. There are two sets or reordering rules to consider: those dealing with at least one symbolic label and those dealing only with numeric labels. The difference, as explained above, lies in the fact that symbolic labels, being always explicit, can commute freely with others, symbolic or numeric, as long as they are distinct. On the other hand, numeric labels need to be kept in relative coherence so that they are implicitly always the first argument of the form on their left.

Symbolic labels There are three rules to consider for swapping two adjacent labeled entities p and q when at least one among p or q is a symbolic label. Rule (1) commutes order of abstractors; Rule (2) commutes order of applicators; and Rule (3) moves an applicator into the body of an abstraction when the expression looks "almost" like a β-redex, were it not for $p \neq q$.

(1) $\lambda_p x.\lambda_q y.M \to \lambda_q y.\lambda_p x.M \qquad (\text{if } p > q)$
(2) $M \mathbin{\widehat{}}_p N \mathbin{\widehat{}}_q P \to M \mathbin{\widehat{}}_q P \mathbin{\widehat{}}_p N \qquad (\text{if } p > q)$
(3) $(\lambda_p x.M) \mathbin{\widehat{}}_q N \to \lambda_p x.(M \mathbin{\widehat{}}_q N) \quad (\text{if } p \neq q)$

Rule (3) must be performed modulo appropriate α-renaming in order to avoid capture.

Numeric labels When two adjacent labeled entities are both numeric, the three above cases must be considered as well. Similar commutations can also take place, except that swapping must preserve relative coherence of implicit positions. This is simply done by decrementing the greater of the two positions.

Let m and n be two positive integers.

(4) $\lambda_m x.\lambda_n y.M \to \lambda_n y.\lambda_{m-1} x.M \qquad (\text{if } m > n)$
(5) $M \mathbin{\widehat{}}_m N \mathbin{\widehat{}}_n P \to M \mathbin{\widehat{}}_n P \mathbin{\widehat{}}_{m-1} N \qquad (\text{if } m > n)$
(6) $(\lambda_m x.M) \mathbin{\widehat{}}_n N \to \lambda_{m-1} x.(M \mathbin{\widehat{}}_n N) \quad (\text{if } m > n)$
(7) $(\lambda_m x.M) \mathbin{\widehat{}}_n N \to \lambda_m x.(M \mathbin{\widehat{}}_{n-1} N) \quad (\text{if } m < n)$

Rules (6) and (7) must be performed modulo appropriate α-renaming in order to avoid capture. Of course, there is direct correspondence between the sets of rules for symbolic and numeric labels. Rules (1) and (2) are directly translated into (4) and (5), with the appropriate changes in labels. Rule (3) must be split into (6) and (7) in order to distinguish cases where $m < n$ and $m > n$.

Examples of reductions

Symbolic labels We suppose that $p < q < r < s$,

$$
\begin{aligned}
&(\lambda_p x.\lambda_q y.\lambda_r z.M) \mathbin{\widehat{}}_r N \mathbin{\widehat{}}_s P \mathbin{\widehat{}}_p Q && \to_3 (\lambda_p x.((\lambda_q y.\lambda_r z.M) \mathbin{\widehat{}}_r N)) \mathbin{\widehat{}}_s P \mathbin{\widehat{}}_p Q \\
\to_2 (\lambda_p x.((\lambda_q y.\lambda_r z.M) \mathbin{\widehat{}}_r N)) \mathbin{\widehat{}}_p Q \mathbin{\widehat{}}_s P && \to_\beta (\lambda_q y.\lambda_r z.[Q/x]M) \mathbin{\widehat{}}_r [Q/x]N \mathbin{\widehat{}}_s P \\
\to_3 (\lambda_q y.((\lambda_r z.[Q/x]M) \mathbin{\widehat{}}_r [Q/x]N)) \mathbin{\widehat{}}_s P && \to_\beta (\lambda_q y.[Q/x][N/z]M) \mathbin{\widehat{}}_s P \\
\to_3 \lambda_q y.([Q/x][N/z]M) \mathbin{\widehat{}}_s P &&
\end{aligned}
$$

Numeric labels

$$
\begin{array}{ll}
(\lambda_2 x.\lambda_1 y.\lambda_2 z.M)\ \widehat{4}\ N\ \widehat{5}\ P\ \widehat{2}\ Q & \to_4\ (\lambda_1 y.\lambda_1 x.\lambda_2 z.M)\ \widehat{4}\ N\ \widehat{5}\ P\ \widehat{2}\ Q \\
\to_7\ (\lambda_1 y.((\lambda_1 x.\lambda_2 z.M)\ \widehat{3}\ N))\ \widehat{5}\ P\ \widehat{2}\ Q & \to_5\ (\lambda_1 y.((\lambda_1 x.\lambda_2 z.M)\ \widehat{3}\ N))\ \widehat{2}\ Q\ \widehat{4}\ P \\
\to_7\ (\lambda_1 y.\lambda_1 x.((\lambda_2 z.M)\ \widehat{2}\ N))\ \widehat{2}\ Q\ \widehat{4}\ P & \to_\beta\ (\lambda_1 y.\lambda_1 x.[N/z]M)\ \widehat{2}\ Q\ \widehat{4}\ P \\
\to_7\ (\lambda_1 y.((\lambda_1 x.[N/z]M)\ \widehat{1}\ Q))\ \widehat{4}\ P & \to_\beta\ (\lambda_1 y.[Q/x][N/z]M)\ \widehat{4}\ P \\
\to_7\ \lambda_1 y.([Q/x][N/z]M\ \widehat{3}\ P) &
\end{array}
$$

3 Proof of confluence

3.1 Pseudo-reduction

Rules Pseudo-reduction rules are intended to make reordering systems confluent in the absence of β-reduction. They promote the formation of new reordering redexes by commutation over β-redexes. The idea is that, without β-reduction, β-redexes just sit there, presenting "obstacles" to the formation of reordering redexes. Hence, we need pseudo-reduction rules to simulate the promotion of reordering redexes that would appear if the β-reduction had been performed. We simulate that effect by having a labeled entity "jump" into, or out of, the body of the abstraction part of a β-redex through its "λ-membrane" and seek reordering on the other side of that membrane. There are two cases: (a) one corresponding to having an applicator jump "into" the body of the abstraction part of a β-redex, and (b) the other corresponding to having an abstractor jump "out of" it. Namely, for *any* labels p, q:

$$
\begin{array}{l}
(a)\ (\lambda_p x.M)\ \widehat{p}\ N\ \widehat{q}\ P \to (\lambda_p x.(M\ \widehat{q}\ P))\ \widehat{p}\ N \\
(b)\ (\lambda_p x.\lambda_q y.M)\ \widehat{p}\ N \to \lambda_q y.\big((\lambda_p x.M)\ \widehat{p}\ N\big)
\end{array}
$$

These two rules must be performed modulo appropriate α-renaming in order to avoid capture. Note that neither rule actually destroys the occurrence of the β-redex which stays there.

Example 1. If we use only reordering rules, $(\lambda_1 x.\lambda_2 y.x\ \widehat{1}\ y)\ \widehat{2}\ a\ \widehat{1}\ b$ can be reduced by Rules (7) and (6) yielding $A = (\lambda_1 x.\lambda_1 y.x\ \widehat{1}\ y\ \widehat{1}\ a)\ \widehat{1}\ b$. It can also be reduced by Rule (5) to $B = (\lambda_1 x.\lambda_2 y.x\ \widehat{1}\ y)\ \widehat{1}\ b\ \widehat{1}\ a$. Both terms A and B are normal forms with respect to reordering. We need pseudo-reduction to recover confluence. Namely, applying Rule (a) to B promotes the appearance of a redex for Rule (6), which yields A.

We will *not* use pseudo-reduction as a reduction system. Rather, we use it to define an equivalence relation modulo which confluence will be defined for partial reduction systems that we are to consider.

Pseudo-reduced form equivalence Since some critical pairs appear between pseudo-reduction and reordering rules, we introduce an equivalence relation that combines their effects. We define two pseudo-reduced form (PRF) equivalences: (I) and (II). PRF equivalence (I) inverts the nesting of two β-redexes, with the necessary precautions. Namely, if $x \notin \mathrm{FV}(N)$, and with appropriate renaming:

(I) $(\lambda_p x.((\lambda_q y.M)\ \widehat{}_q\ N))\ \widehat{}_p\ P\ \rightarrow\ (\lambda_q y.((\lambda_p x.M)\ \widehat{}_p\ P))\ \widehat{}_q\ N$

This defines an equivalence by transitive closure, since applying twice this rule at the same occurrences is the identity. Note that it does not bother requiring that $y \notin \mathrm{FV}(P)$. Indeed, this condition is always satisfied because P is external in the original form, and had it contained y, it would have been renamed to make the terms distinct.

PRF equivalence (II) is a renaming of labeling for any β-redex. Namely,

(II) $(\lambda_p x.M)\ \widehat{}_p\ N\ \leftrightarrow\ (\lambda_q x.M)\ \widehat{}_q\ N$

This last equivalence will be essential to identify normal forms for reordering that differ only by the labeling of some β-redexes. Such situations would not occur if only symbolic labels were considered. However, it can arise for numeric labels that may, or may not, have been decremented through reordering.

3.2 Combined systems and restricted reductions

Using the rules above, there are in fact several reductions systems that we can consider depending on various combinations of the groups of rules. The full system that we will aim for is the free combination of β-reduction and reordering rules: System (β) + (1)-(7). We will call it *selective λ-calculus*. However, we will first consider the following partial reduction systems.

We will first focus on reordering rules alone, and prove a few useful properties. To do this we have to add some rules to preserve confluence: pseudo-reductions (a) and (b). So the system we are really interested in is System (1)-(7) + (a)-(b). We will still call it the *reordering system*. An *order-normal form* is a normal form for the reordering system. For this reordering system we will consider a special class of reductions, called *standard reorderings*, which are innermost reorderings.

We will also consider a partial reduction system called *β-reordering*. It is β-reduction on order-normal forms, where the result of each step is normalized by the reordering system.

The last system, or *label-parallel system*, which makes the link between selective λ-calculus and β-reordering, is the combination of all three reduction systems: System (β) + (1)-(7) + (a)-(b).

For each of these reduction systems, we shall use the symbol $\rightarrow$ to indicate a single reduction step using any of system's rules, and $\rightarrow_r$ if the rule uses Rule (r). When unconcerned by termination, we shall accept the (possibly subscripted) step $(M \rightarrow M)$ in this relation. As usual, $\stackrel{*}{\rightarrow}$ is the reflexive and transitive closure, also possibly subscripted. Given a reduction strategy ϱ, we will use the symbol $\triangleright_\varrho$ to denote the subrelation of $\stackrel{*}{\rightarrow}$ using only ϱ-reduction steps. For example, $\triangleright_{std}$ for standard reorderings, $\triangleright_{mcd}$ for minimal complete developments, *etc.*

Confluence modulo an equivalence relation is defined as follows ([10]):

Definition 1. A relation $\rightarrow$ is *confluent modulo* an equivalence $\sim$ iff $\forall x, y, x', y'$ $\big((x \sim y \text{ and } x \stackrel{*}{\rightarrow} x' \text{ and } y \stackrel{*}{\rightarrow} y')\ \Rightarrow\ \exists x'', y''\ (x' \stackrel{*}{\rightarrow} x'' \text{ and } y' \stackrel{*}{\rightarrow} y'' \text{ and } x'' \sim y'')\big)$.

Confluence will always mean at least modulo α-conversion. Furthermore, for several of our reduction systems, confluence will also be considered modulo *pseudo-*

reduced form equivalence (PRF), that is the composition of the two equivalences *PRF* (I) and *PRF* (II) of 3.1.

3.3 Confluence of the reordering system

We can see a λ-term as a tree. An abstraction $\lambda_p x.M$ corresponds to a node labeled with the abstractor $\lambda_p x$ and it has one son corresponding to the body M. An application $M \mathbin{\widehat{}}_p N$ corresponds to a node labeled $\mathbin{\widehat{}}_p$ having two sons: a left one corresponding to M and a right one corresponding to N. A leaf node corresponds to a variable. Given a λ-term M, we will be interested in a particular sequence of entities of M, called its *spine*. Roughly, a term's spine is what is left of the tree after clipping all its right branches leaving only stubs of depth one. More precisely, the spine of a term M is the sequence of its entities, starting at the root, of the *rightmost* depth-first traversal of the tree obtained from the term's tree by replacing all the right expressions of applications by new variable names, each uniquely identifying the omitted subterm.

Definition 2 Spine. The *spine* of an expression is the sequence of its entities obtained inductively as follows:

$$\begin{aligned} \mathit{spine}\,(\lambda_p x.M) &= (\lambda_p x), \mathit{spine}\,(M) \\ \mathit{spine}\,(M \mathbin{\widehat{}}_p N) &= (\mathbin{\widehat{}}_p z_N), \mathit{spine}\,(M) \quad \text{where } z_N \text{ is new} \\ \mathit{spine}\,(x) &= (x) \qquad\qquad\qquad \text{if } x \in \mathcal{V}. \end{aligned}$$

Example 2. Let $M = \lambda_p x.\big((\lambda_q y.\big(y \mathbin{\widehat{}}_r (\lambda_s u.u)\big)\big) \mathbin{\widehat{}}_p (x \mathbin{\widehat{}}_t v)\big)$. Its spine is given by: $\mathit{spine}\,(M) = (\lambda_p x), (\mathbin{\widehat{}}_p z_1), (\lambda_q y), (\mathbin{\widehat{}}_r z_2), (y)$, where z_1 and z_2 are new variables standing respectively for M's subterms $x \mathbin{\widehat{}}_t v$ and $\lambda_s u.u$.

Note that the spine of a λ-term's subterm is *not* necessarily a subsequence of the term's own spine. Given a term M, by *set of spines* of M, we mean the set of spines of all its subterms (including itself).

The *size* of a spine is the number of entities it contains. We shall speak of the *index* of an entity in a spine as its position in the spine, starting from the root. The deeper a subterm occurs in a term, the higher its index in that term's spine will be.

Lemma 3 Stability of entities. *The reordering rules* (1)–(7) *do not produce any labeled entities nor do they destroy any. Moreover, a labeled entity stays on the same spine after any reordering rule application.*

Innermost reordering of a term is performed following the following strategy.

Definition 4 Standard reordering. A *standard reordering* is a reordering which rewrites entities of lower spine index first.

Definition 5 Locally dependent entities. An abstractor and an applicator are said to be *locally dependent* in a term whenever they constitute a β-redex. They are *locally independent* if they do not.

Definition 6 Dependent entities. An abstractor and an applicator are said to be *dependent* in a term if they can become *locally dependent* by a standard reordering. Otherwise they are *independent*.

Note that a consequence of Lemma 3 is that two dependent entities are always on the same spine.

Lemma 7. *An abstractor cannot* cross *(exchange indexes with) an applicator of higher index in the spine. Or, equivalently, an applicator cannot cross an abstractor of lower index.*

We call a *masked step* during reordering an application of (a) replaced by two applications of System (1)-(7)—namely, (2) then (3), or (5) then (7)—or the similar case for (b). Entities on different spines being independent, a step is masked only relatively to one spine, and disappears in the next reduction along its spine.

Lemma 8. *During a standard reordering locally dependent entities stay locally dependent, except temporarily during* masked steps *along their spine.*

Theorem 9 Termination of standard reordering. *Standard reordering is Noetherian.*

Theorem 10 Confluence of standard reordering. *Standard reordering is confluent modulo PRF equivalence.*

Proposition 11. *Standard order-normal form and order-normal form are equal modulo PRF equivalence.*

Theorem 12. *The reordering system is confluent modulo PRF equivalence.*

Lemma 13. *Locally dependent pairs stay locally dependent in the order-normal form.*

Proposition 14. *The reordering system is Noetherian.*

3.4 Confluence of β-reordering

Definition 15 β-Reordering. A *β-reordering* step is a β-reduction step immediately followed by a reduction to order-normal form.

Since reordering is Noetherian and confluent modulo PRF-equivalence, this completely defines a reduction rule on the quotient of order-normal λ-terms modulo PRF-equivalence. The one-step β-reordering relation is denoted as $\rightarrow_{\beta\downarrow}$.

Not to worry about renaming problems during reordering, we will assume that all free variables and abstraction variables have distinct names.

For Definition 15 to stand we have to prove that PRF-equivalent order-normal expressions are still equivalent after β-reordering. PRF equivalence (II) is not a problem: either the modified entity dependence is reduced, and then, the label does not matter, or it is not, and then it is changed back to what it was. As for, PRF equivalence (I), we can assume that the reduced dependent pair is always in the least

possible index in the spine. If it is not, it does not matter since this equivalence obviously preserves entity dependencies.

This justifies us in the rest of this section, to consider no longer terms, but their equivalence classes modulo PRF equivalence.

For any label-selective λ-term M, we will note $M\downarrow$ its order-normal form. To lighten notation, but only in this section, we will write a term M when we actually mean $\downarrow$ implicitly. We shall do that everywhere in the section, except of course when we prove properties about it. That is, everywhere except Lemmas 17, 18 and 21. That means that generally $M = N$ should be read as $M\downarrow = N\downarrow$, $M \rightarrow_{\beta\downarrow} N$ as $M\downarrow \rightarrow_{\beta\downarrow} N\downarrow$, and $M \triangleright_{mcd} N$ as $M\downarrow \triangleright_{mcd} N\downarrow$.

From here on, the proof of β-reordering confluence follows the Martin-Löf-Tait scheme as in [9]. By *contracting* a β-redex, we mean applying the corresponding step of β-reduction.

Definition 16 Residuals. Let R, S be β-redexes in an order-normal λ-term P. When R is contracted, let P change to P'. The *residuals* of S with respect to R are redexes in P', defined as follows:

- R, S are *non-overlapping parts* of P. Then contracting R leaves S unchanged. This unchanged S in P' is called the residual of S.
- $R = S$. Then contracting R is the same as contracting S. We say S has no residual in P'.
- *R is part of S and $R \neq S$.* Then S has form $(\lambda_p x.M) \widehat{}_p N$ and R is in M or in N. Contracting R changes M to M' or N to N', and S to $(\lambda_{p'} x.M') \widehat{}_{p'} N$ or $(\lambda_p x.M) \widehat{}_p N'$; this is the residual of S.
- *S is part of R and $S \neq R$.* This case will not happen in our proof.

Note that in this definition, the meaning of *non-overlapping* is taken in a large sense: it means that there is a configuration of entity dependencies such that R and S are not overlapping. Note also that in these three cases S has at most one residual.

Lemma 17 Substitution. $[N/x]M = [N/x]M\downarrow$; i.e., *reordering before or after a substitution does not change the result.*

Lemma 18 Induction.
$$M \rightarrow_{\beta\downarrow} M' \Rightarrow \lambda_p x.M \rightarrow_{\beta\downarrow} \lambda_p x.M'$$
$$M \rightarrow_{\beta\downarrow} M' \Rightarrow M \widehat{}_p N \rightarrow_{\beta\downarrow} M' \widehat{}_p N$$
$$N \rightarrow_{\beta\downarrow} N' \Rightarrow M \widehat{}_p N \rightarrow_{\beta\downarrow} M \widehat{}_p N'$$

Let $R_1, \ldots, R_n$ $(n \geq 0)$ be redexes in a term P. An R_i is called *minimal* iff there is a configuration of entity dependencies in which it properly contains no other R_j. That is, there is a configuration of entity dependencies where it is the most internal redex and there is no redex in applications of spine index higher than or equal to its own.

A *minimal complete development* (MCD) of $\{R_1, \ldots, R_n\}$ in P is a sequence of contractions on P performed as follows:

- First, contract any minimal R_i (say $i = 1$ for convenience). This leaves at most $n-1$ residual $R'_2, \ldots, R'_n$, of $R_2, \ldots, R_n$.

- Then, contract any minimal R'_j. This leaves at most $n-2$ residuals.
- Repeat the above two steps until no residuals are left.

Note that this process is non-deterministic, and thus there are more than one such sequence of contractions.

Definition 19 MCD. Let P be a term as above, and Q a term. We write $P \rhd_{mcd} Q$ iff Q is obtained from P by *minimal complete development* of the set $\{R_1, \ldots, R_n\}$.

Note that if $M \rhd_{mcd} M'$ and $N \rhd_{mcd} N'$, then $M \mathbin{\widehat{p}} N \rhd_{mcd} M' \mathbin{\widehat{p}} N'$. (*cf.*, Lemma 18)

Lemma 20. *If $M \rhd_{mcd} M'$ and $N \rhd_{mcd} N'$, then*

$[N/x]M \rhd_{mcd} [N'/x]M'$.

Lemma 21 Proof induction. *If there is an MCD*

$$\begin{aligned} P \;=\; (\lambda_p x.M) \mathbin{\widehat{p}} N &\xrightarrow{*}_{\beta\downarrow} (\lambda_p x.M') \mathbin{\widehat{p}} N' \\ &\to_{\beta\downarrow} [N'/x]M' \quad = \; Q \\ &\xrightarrow{*}_{\beta\downarrow} Q' \end{aligned}$$

then there is an MCD

$$\begin{aligned} P \;=\; (\lambda_p x.M) \mathbin{\widehat{p}} N &\xrightarrow{*}_{\beta\downarrow} (\lambda_p x.M'') \mathbin{\widehat{p}} N'' \\ &\to_{\beta\downarrow} [N''/x]M'' \quad = \; Q' \end{aligned}$$

Lemma 22. *If $P \rhd_{mcd} A$ and $P \rhd_{mcd} B$, then there exists T such that $A \rhd_{mcd} T$ and $B \rhd_{mcd} T$.*

Theorem 23. *β-reordering is confluent modulo PRF equivalence.*

$P \xrightarrow{*}_{\beta\downarrow} M,\ P \xrightarrow{*}_{\beta\downarrow} N \Rightarrow (\exists T)\ M \xrightarrow{*}_{\beta\downarrow} T,\ N \xrightarrow{*}_{\beta\downarrow} T.$

3.5 Confluence of selective λ-calculus

In this section, $\to$ (or $\to_\lambda$) denotes the union of β-reduction and ordering rules (label-selective λ-calculus), and $\to_\omega$ is the union of all rules (label-parallel system). We will now *no longer* consider terms modulo PRF equivalence, except in the β-reordering diamond of Figure 1.

Definition 24 Normalized reduction. For each label-parallel reduction $M_0 \to_\omega M_1 \to_\omega \ldots \to_\omega M_n$ we define its *normalized reduction* $N_0 \to_{\beta\downarrow} N \to_{\beta\downarrow} \ldots \to_{\beta\downarrow} N_n$ by taking for each N_i the order-normal form $M_i{\downarrow}$.

Proposition 25. *Normalized reduction is a β-reordering.*

Theorem 26 Confluence of label-parallel reduction. *The label-parallel system is confluent. That is,*

$P \xrightarrow{*}_\omega M,\ P \xrightarrow{*}_\omega N \ \Rightarrow\ (\exists T)\ M \xrightarrow{*}_\omega T,\ N \xrightarrow{*}_\omega T.$

Theorem 27 Confluence of label-selective λ-calculus. *The label-selective λ-calculus is confluent. That is,*

$P \xrightarrow{*} M,\ P \xrightarrow{*} N \ \Rightarrow\ (\exists T)\ M \xrightarrow{*} T,\ N \xrightarrow{*} T.$

Figure 1 shows a schematic diagram of the process.

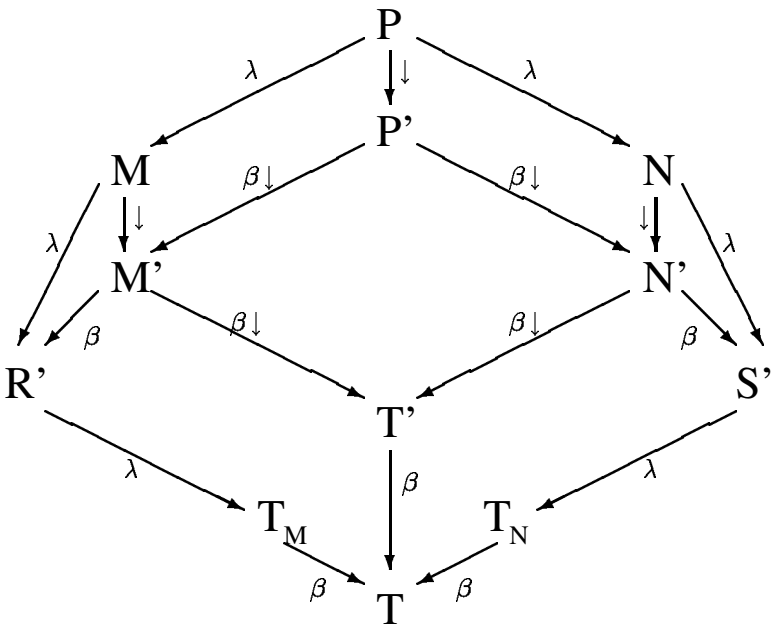

Fig. 1. Schematic confluence of label-selective λ-calculus

4 Extended systems

These proofs are valid in fact for a whole range of systems. The most immediate ones are the two systems in which only numeric or only symbolic labels are accepted. Intuitively they represent two different way of introducing some commutativity in λ-calculus. Selective λ-calculus is their sum. Namely, it is obtained as the calculus over the set of labels which is their disjoint sum: $\mathcal{L} = \mathcal{S} + \mathcal{N}$.

4.1 Product system

Since we have a sum, it is natural to think of a product. That is, the calculus over the set of labels which is the Cartesian product of symbolic and numeric label. Namely, a label in now a pair symbol-numeral in $\mathcal{L} = \mathcal{S} \times \mathcal{N}$. This system corresponds to a calculus of named channels on which only numeric positions are considered. Per channel name, relative numeric position arguments can be done in that relative reordering of distinct positions is allowed. In a sense, it is orthogonal to the sum system, in that it separates names and numbers, although it provides each with the functionality of label-selective λ-reduction.

β-Reduction for product, as for the sum, requires identically labeled abstractor and applicator: $(\beta)\ \ (\lambda_{pm}x.M)\ \widehat{}_{pm}\ N\ \rightarrow\ [N/x]M$. As for reordering, we need only slightly alter reordering rules of the sum system as follows:

$$
\begin{array}{lll}
(1)\ \lambda_{pm}x.\lambda_{qn}y.M & \to \lambda_{qn}y.\lambda_{pm}x.M & p > q \\
(2)\ M\ \widehat{}_{pm}\ N\ \widehat{}_{qn}\ P & \to M\ \widehat{}_{qn}\ P\ \widehat{}_{pm}\ N & p > q \\
(3)\ (\lambda_{pm}x.M)\ \widehat{}_{qn}\ N & \to \lambda_{pm}x.(M\ \widehat{}_{qn}\ N) & p \neq q \\
(4)\ \lambda_{pm}x.\lambda_{pn}y.M & \to \lambda_{pn}y.\lambda_{p(m-1)}x.M & m > n \\
(5)\ M\ \widehat{}_{pm}\ N\ \widehat{}_{pn}\ P & \to M\ \widehat{}_{pn}\ P\ \widehat{}_{p(m-1)}\ N & m > n \\
(6)\ (\lambda_{pm}x.M)\ \widehat{}_{pn}\ N & \to \lambda_{p(m-1)}x.(M\ \widehat{}_{pn}\ N) & m > n \\
(7)\ (\lambda_{pm}x.M)\ \widehat{}_{pn}\ N & \to \lambda_{pm}x.(M\ \widehat{}_{p(n-1)}\ N) & m < n
\end{array}
$$

Clearly, all proofs done above on selective λ-calculus are still valid for this system: we take care of numbers only when symbols are identical, and have then the same rules.

4.2 Polynomial systems

If we have sums and products, we should be able to combine them. A *polynomial system* is defined by two mutually exclusive sets of names $\mathcal{S}$ and $\mathcal{T}$, and taking the set of labels to be $\mathcal{L} = \mathcal{S} \cup (\mathcal{T} \times \mathcal{N})$. The syntax is: $M ::= x \mid \lambda_p x.M \mid \lambda_{qn}x.M \mid M\ \widehat{}_p\ M' \mid M\ \widehat{}_{qn}\ M'$, with p in $\mathcal{S}$ and q in $\mathcal{T}$.

We just take Rules (1)–(3) of selective λ-calculus applied on $\mathcal{S} \cup (\mathcal{T} \times \mathcal{N})$ and Rules (4)–(7) of product systems, applied on $\mathcal{T}$.

Selective λ-calculus is a polynomial system where $\mathcal{T}$ is restricted to an unique constructor ϵ, abbreviated.

It is only for clarity that proofs where given for selective λ-calculus, rather than for any polynomial system, where they are anyway valid. We should always keep in mind that when we prove or build something on selective λ-calculus it will be nearly always generalizable to any polynomial system, for which we are just using an abbreviated notation.

5 Towards a transformation calculus

We will just give here a new notation, and explain in what way this notation suggests another extension of selective λ-calculus.

The intuition behind selective λ-calculus is no longer functions but functions over labeled arguments which behave like communicating processes through named or (relatively) numbered channels. Application corresponds to process communication. What we called entities are seen as *actions*: abstractors correspond to receiving and applicators to sending. But what about composition? It is easily defined for functions as $f \circ g = \lambda f.\lambda g.\lambda x.f(gx)$ in the classical calculus. We could of course think of a labeled composition one for each label. But this is rather weak. Particularly when we think of the powerful out-of-order currying power of our system.

Rather, we will just change notations, and define: $M\ \overset{\vee}{}_p\ x \overset{def}{=} \lambda_p x.M$. All terms now take the form $x \cdot P$, where P is a sequence of entities, or actions, and $\cdot$ the syntactic juxtaposition. Then we can obtain a more interesting composition with $M \circ P = M \cdot P$, where we do not specify anything about labels, and may have created more than one connection at once. Here again P is a sequence of actions, and we would like

to manipulate them as such. The potential of this notation is such that it simplifies considerably many proofs. For instance we could prove easily Böhm's expansion theorem using it. Since our extension is conservative, our proof is valid for classical λ-calculus as well.

Of course, all this starts to be strongly reminiscent of a calculus for process communication [14], although still a direct and conservative extension of classical λ-calculus. We believe that this is not coincidental. This idea is the object of our current research and we are actively exploring the deep connections with the various existing communication calculi. But, we shall say no more on this for now.

6 Conclusion and further work

Label-selective λ-calculus offers the advantage of realizing directly a more complete isomorphism of Cartesian products and function applications. An immediate consequence is a more convenient notation, and a more efficient, indeed concurrent, manner to extract arguments out of order.

Beyond the bare calculus, we have started studying a typed version of our calculus [8]. There, we propose a simply typed version of this calculus, and show that it extends to second order and polymorphic typing. For this last one there exists a most generic type, and we give the algorithm to find it.

A topic for further work along this idea is, of course compilation. As mentioned in the first section, we plan to extend the stack-based model of execution of λ-calculus with our label-selection scheme to realize efficient access to arguments regardless of position label. We have already adapted the calculus of explicit substitutions [1], as are currently working on a compiling scheme for label-selective λ-calculus based on it.

Also, we plan to study the work of Ohori [15] to elucidate the gains that this may have in the compilation of records. As for semantics, we have initiated work on a typed version of label-selective λ-calculus and a framework of models for it.

We have formulated and are studying several concurrent calculi extending label-selective λ-calculus towards full concurrency [3], including the provision for computable channel names. One of the gains expected is that λ-calculus will need not be *encoded* as in [13], but directly embedded as syntactic identity.

Finally, the real goal that has motivated our working out this calculus has been to use it for a useful generalization of object-oriented style of message passing. Method invocation based on the type of the first argument of a call can be elegantly explained by seeing a method definition in a class as a curried form with respect to the object instance of the class. Label-selective currying can thus reinstate the lost symmetry by distributing one partially-applied form for each arguments of the class of a method. As a result, message-passing can be used on any argument of a call, making labels act as channels. Our confluence result guarantees that the choice of channel does not matter. We plan to pursue this insight and investigate all its ramifications.

Acknowledgements The authors wish to thank Kathleen Milsted, Atsushi Ohori, Mitsuhiro Okada, and Andreas Podelski for their comments and suggestions.

References

1. Martin Abadi, Luca Cardelli, Pierre-Louis Curien, and Jean-Jacques Lévy. Explicit substitutions. In *Proceedings of the Seventeenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (1990).
2. Hassan Aït-Kaci and Jacques Garrigue. Label-selective λ-calculus. PRL Research Report 31, Digital Equipment Corporation, Paris Research Laboratory, Rueil-Mailmaison, France (May 1993).
3. Hassan Aït-Kaci and Kathleen Milsted. Concurrent label-selective λ-calculus. PRL research report, Digital Equipment Corporation, Paris Research Laboratory, Rueil-Malmaison, France (forthcoming).
4. Hassan Aït-Kaci and Andreas Podelski. Towards a meaning of LIFE. PRL Research Report 11, Digital Equipment Corporation, Paris Research Laboratory, Rueil-Malmaison, France (1991). (Revised, October 1992; to appear in the *Journal of Logic Programming*).
5. Gérard Boudol. Towards a lambda-calculus for concurrent and communicating systems. In *Proceedings of TAPSOFT'89*, pages 149–161, Berlin, Germany (1989). Springer-Verlag. LNCS 351.
6. Laurent Dami. A name-based calculus for extensible objects and functions. Draft manuscript (1993). Centre Universitaire d'Informatique, Genève, Switzerland.
7. N. G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation. *Indag. Math.*, 34:381–392 (1972).
8. Jacques Garrigue and Hassan Aït-Kaci. Typed label-selective λ-calculus. PRL research report, Digital Equipment Corporation, Paris Research Laboratory, Rueil-Mailmaison, France (forthcoming).
9. J. Roger Hindley and Jonathan P. Seldin. *Introduction to Combinators and λ-Calculus*, volume 1 of *London Mathematical Society Student Texts*. Cambridge University Press, Cambridge, UK (1986).
10. Gérard Huet. Confluent reductions: Abstract properties and applications to term rewriting systems. *Journal of the ACM*, 27(4):797–821 (October 1980).
11. Peter J. Landin. The mechanical evaluation of expressions. *Computer Journal*, 6(4):308–320 (1965).
12. Henry Ledgard. *ADA: An Introduction, Ada Reference Manual (July 1980)*. Springer-Verlag, New York, NY (1981).
13. Robin Milner. Functions as processes. Rapport de Recherche 1154, INRIA, Le Chesnay, France (February 1990).
14. Robin Milner. The polyadic π-calculus: A tutorial. LFCS Report ECS-LFCS-91-180, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, Edinburgh, UK (October 1991).
15. Atsushi Ohori. A compilation method for ML-style polymorphic records. In *Proceedings of the Ninetheenth ACM Symposium on Principles of Programming Languages*, pages 154–165 (January 1992).
16. Guy L. Steele. *Common LISP: The Language*. Digital Press (1984).

This article was processed using the LaTeX macro package with LLNCS style