Label-Selective λ-Calculus
Syntax and Confluence†

Hassan Aït-Kaci¹ and Jacques Garrigue²

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 λx.λy.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 x B \rightarrow C \simeq A \rightarrow (B \rightarrow C) for any set A, B and C. However, there is another obvious natural isomorphism that could also be useful; namely, A x B \simeq B x A. Hence we should be able to exploit this directly in the form: A \rightarrow (B \rightarrow C) \simeq B \rightarrow (A \rightarrow 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 \pi₁ and \pi₂ 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 1st and 2nd 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 \( \pi_1 \Rightarrow A \times \pi_2 \Rightarrow B \cong \pi_2 \Rightarrow B \times \pi_1 \Rightarrow A \), and thus that: \( \pi_1 \Rightarrow A \rightarrow (\pi_2 \Rightarrow B \rightarrow C) \cong \pi_2 \Rightarrow B \rightarrow (\pi_1 \Rightarrow A \rightarrow C) \).

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 \( \lambda \)-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 \( \lambda \)-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 \( \lambda \)-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 \( \lambda \)-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 \( \lambda \)-calculus can be given as extracting implicit concurrency from \( \lambda \)-calculus. It is well-known that \( \lambda \)-calculus is a

\[ \text{Parse the following with '⇒' binding tighter than '×' or '→'.} \]