Sharing in the Weak Lambda-Calculus

Tomasz Blanc, Jean-Jacques Lévy, and Luc Maranget

INRIA - Rocquencourt
{Tomasz.Blanc, Luc.Maranget, Jean-Jacques.Levy}@inria.fr
http://moscova.inria.fr/~{tblanc, levy, maranget}

Abstract. Despite decades of research in the λ-calculus, the syntactic properties of the weak λ-calculus did not receive great attention. However, this theory is more relevant for the implementation of programming languages than the usual theory of the strong λ-calculus. In fact, the frameworks of weak explicit substitutions, or computational monads, or λ-calculus with a `let` statement, or super-combinators, were developed for adhoc purposes related to programming language implementation. In this paper, we concentrate on sharing of subterms in a confluent variant of the weak λ-calculus. We introduce a labeling of this calculus that expresses a confluent theory of reductions with sharing, independent of the reduction strategy. We finally state that Wadsworth's evaluation technique with sharing of subterms corresponds to our formal setting.

1 Introduction

In the terminology of the λ-calculus, a *strong* calculus validates the following ξ-rule

$$(\xi)\ \frac{M \to N}{\lambda x.M \to \lambda x.N}$$

A *weak* calculus does not validate this rule. One easily shows that the weak λ-calculus is not confluent. In [18], an extension of the weak λ-calculus was introduced. It is strongly inspired from the one of Çağman and Hindley [6] for Combinatory Logic. In this calculus, a restricted version of the ξ-rule is valid; this new ξ'-rule is intuitively defined by

$$(\xi')\ \frac{M \xrightarrow{R} N \quad x \notin R}{\lambda x.M \xrightarrow{R} \lambda x.N}$$

meaning that the ξ-rule is valid when the bound variable x is not free in the redex R contracted between M and N (This rule will be presented in Section 2 in a form slightly different from — but equivalent to — the σ-rule used in [18]). The resulting new weak λ-calculus is confluent as shown in [18].

The theory of optimal reductions in the λ-calculus [5] has been extensively studied by Abadi, Asperti, Coppola, Gonthier, Guerrini, Lamping, Lawall, Lévy, Mairson, Martini et al [3, 4, 9, 13, 15, 17]. These authors represent λ-terms as graphs with shared subcontexts. For instance, in $(\lambda x.xa(xb))(\lambda y.Iy)$ where $I = \lambda x.x$, it is necessary to share the redex Iy independently of the value

A. Middeldorp et al. (Eds.): Processes... (Klop Festschrift), LNCS 3838, pp. 70–87, 2005.

of y. Therefore the subcontext $I[\,]$ has to be shared. But after reduction of the external redex, we get $(\lambda y.Iy)a((\lambda y.Iy)b)$ and further $Ia((\lambda y.Iy)b)$, where the shared subcontext $I[\,]$ is instantiated with two different terms. Technically, in these graphs, a shared subcontext can be referenced through a *fan-in* node to multiplex incoming arcs from terms using it; and context holes are filled through *fan-out* nodes to demultiplex outgoing arcs pointing to terms filling these holes. For instance, Lamping's graphs [13] operate on fans and *brackets*, which can be decomposed into more elementary fans, brackets and *croissants* obeying to reduction rules directed by the *context semantics* defined in [9].

In the weak λ-calculus, reductions are not performed under λ-abstractions. In the above example, the subterm Iy in $(\lambda x.xa(xb))(\lambda y.Iy)$ cannot be reduced since inside the abstraction $\lambda y.Iy$. Thus the subcontext $I[\,]$ needs not be shared. The λ-terms with sharing can be represented by directed acyclic graphs (*dags*) instead of the (cyclic) Lamping's graph structures required to implement shared subcontexts. In this paper, we present a weak labeled λ-calculus that expresses a confluent theory of sharing within the weak λ-calculus corresponding to the shared evaluation strategy by Wadsworth [25] defined with dags in 1971!

The weak λ-calculus corresponds to runtime systems in functional languages, since runtimes just pass arguments to functions and never compute function bodies, i.e. under λ-abstractions. At compile-time, inlining or partial evaluation are feasible; but the weak λ-calculus just corresponds to the execution phase. However a runtime of a functional language usually implements a particular reduction strategy such as call-by-name, call-by-need or call-by-value. We prefer to model these runtimes by a confluent calculus which allows to consider mixed strategies alternating call-by-need and call-by-value. Moreover, a confluent calculus makes independent sharing and reduction strategy, which are two independent concepts. In runtimes of lazy functional languages (Haskell, LML, G-machine) [20, 22], the call-by-need strategy will correspond to a (weak) leftmost outermost reduction with some amount of sharing.

In most functional runtime systems (See e.g. [16]), functions are implemented by closures (See e.g. [16]), i.e. a pairs of a λ-abstraction (program) and a substitution (environment). The theory of explicit substitutions is related to the notion of closure but does not restrict reduction strategies [1]. This theory is not simple. It uses de Bruijn indices and is not confluent for open terms: Klop's counterexample [11] for surjective pairing can be adapted to the calculus of explicit substitutions [1]. However, confluence (on open terms) can be recovered at the price of either considering a much more complex calculus of explicit substitutions [7], or a theory of weak explicit substitutions as in [8, 18]. In the latter case, a theory of sharing was sketched, through the definition of a weak labeled calculus of explicit substitutions.

Closures and explicit substitutions are not necessary to express sharing of λ-terms. For instance, call-by-need strategies by Launchbury, Odersky or Ariola et al. [14, 19, 2] use a λ-calculus with a new `let` construct. However, these calculi have often critical pairs (in the sense of term rewriting) and extra rules to handle the `let` construct. Term rewriting systems (TRS) can also be used by lifting free