Cyclic Lambda Calculi

Zena M. Ariola and Stefan Blom

1 Department of Computer & Information Sciences
University of Oregon, Eugene, OR 97401, USA
email: ariola@cs.uoregon.edu

2 Department of Mathematics and Computer Science
Vrije Universiteit, De Boelelaan 1081a, 1081 HV Amsterdam
email: sccblom@cs.vu.nl

Abstract. We precisely characterize a class of cyclic lambda-graphs, and then give a sound and complete axiomatization of the terms that represent a given graph. The equational axiom system is an extension of lambda calculus with the letrec construct. In contrast to current theories, which impose restrictions on where the rewriting can take place, our theory is very liberal, e.g., it allows rewriting under lambda-abstractions and on cycles. As shown previously, the reduction theory is non-confluent. We thus introduce an approximate notion of confluence. Using this notion we define the infinite normal form or Lévy-Longo tree of a cyclic term. We show that the infinite normal form defines a congruence on the set of terms. We relate our cyclic lambda calculus to the traditional lambda calculus and to the infinitary lambda calculus.

Since most implementations of non-strict functional languages rely on sharing to avoid repeating computations, we develop a variant of our calculus that enforces the sharing of computations and show that the two calculi are observationally equivalent. For reasoning about strict languages we develop a call-by-value variant of the sharing calculus. We state the difference between strict and non-strict computations in terms of different garbage collection rules. We relate the call-by-value calculus to Moggi’s computational lambda calculus and to Hasegawa’s calculus.

1 Introduction

Cyclic lambda-graphs are ubiquitous in a program development system [33]. However, previous work falls short of capturing them in an adequate way. This lack of explicit treatment of cycles results in the loss of important intensional information and in weak theories that cannot express many transformations on recursive functions. For example, consider the following term:

\[ M \equiv \text{letrec } \text{even } = \lambda x. \text{if } x = 0 \text{ then true else odd}(x-1) \]
\[ \text{odd } = \lambda x. \text{if } x = 0 \text{ then false else even}(x-1) \]
\[ \text{in even } y. \]

(A note on syntax: the construct letrec \(
\ldots \text{in } \ldots \)
stands for a collection of unordered equations and a main expression written after the keyword in.) At
compile time it might make sense to *unfold* or *inline* odd in the definition of even, triggering the *constant folding* and *unused lambda expression* transformations obtaining the term below:

\[ N \equiv \text{letrec } \text{even } = \lambda x.\text{if } x = 0 \text{ then true else if } x = 1 \text{ then false else even}(x-2) \]

in even y.

We can express terms \( M \) and \( N \) in the lambda calculus extended with pairs (denoted by \( (, ) \) with destructors Fst and Snd) and the \( \mu \)-operator (rendered by the \( \mu \)-rule \( \mu x.M \rightarrow M[x := \mu x.M] \)) as follows (we denote the translation by \( [ ]_\mu \)):

\[ [M]_\mu \equiv \text{let even_odd } = \mu z.(\lambda x.\text{if } x = 0 \text{ then true else Snd } z (x-1), \lambda x.\text{if } x = 0 \text{ then false else Fst } z (x-1)) \]

in Fst even_odd y

\[ [N]_\mu \equiv \text{let even } = \mu y.\lambda x.\text{if } x = 0 \text{ then true else if } x = 1 \text{ then false else } y(x-2) \]

in even y.

However, note that \([M]_\mu\) does not rewrite to \([N]_\mu\) in \( \lambda \mu \). The two terms are not even provably equal. This means that these simple inlining optimizations are not expressible as source-to-source transformations in \( \lambda \mu \); thus, one cannot use the calculus to reason about their correctness or to study the efficiency of different application strategies.

Cycles are also important for reasoning about run-time issues. For example, the execution of \( M \) will involve a substitution of even in the main expression, followed by a \( \beta \)-reduction, obtaining:

\[
\text{letrec even } = \lambda x.\text{if } x = 0 \text{ then true else odd}(x-1) \\
\quad \text{odd } = \lambda x.\text{if } x = 0 \text{ then false else even}(x-1) \\
\text{in if } y = 0 \text{ then true else odd}(y-1).
\]

Let us consider \([M]_\mu\). We first apply the \( \mu \)-rule to expose the lambda-abstraction, and then, as before, perform one substitution followed by a \( \beta \)-reduction, obtaining the following term in which we have denoted the \( \mu \)-expression occurring in \([M]_\mu\) by \( P \):

\[
\text{let even_odd } = (\lambda z.\text{if } x = 0 \text{ then true else Snd } P (x-1), \lambda x.\text{if } x = 0 \text{ then false else Fst } P (x-1)) \\
\text{in if } y = 0 \text{ then true else Snd } P (y-1).
\]

The unsuitability of a calculus such as \( \lambda \mu \) for reasoning about execution now comes to the surface. While the execution of \( M \) has made only one copy of even, the execution of \([M]_\mu\) has created four copies of even and three copies of odd.

Interestingly enough, a theory of cycles turns out to be useful also for defining a parser. As described by Tomita [38] and Billot et al. [16], a compact representation of all possible parse trees (that could be an infinite number) associated