Cyclic Lambda Calculi

Article · November 2000

DOI: 10.1007/BFb0014548 · Source: CiteSeer

3 authors, including:

Zena Ariola
University of Oregon
80 PUBLICATIONS **1,597** CITATIONS

Cyclic Lambda Calculi

Zena M. Ariola[1] and Stefan Blom[2]

[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:

$$\begin{aligned} M \equiv\ & \texttt{letrec even } = \lambda x.\texttt{if } x = 0 \texttt{ then true else odd}(x\text{-}1) \\ & \texttt{odd } = \lambda x.\texttt{if } x = 0 \texttt{ then false else even}(x\text{-}1) \\ & \texttt{in even } y\ . \end{aligned}$$

(A note on syntax: the construct `letrec` $\cdots$ `in` $\cdots$ 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:

$$N \equiv \texttt{letrec even} = \lambda x.\texttt{if } x = 0 \texttt{ then true} \\ \texttt{else if } x = 1 \texttt{ then false else even}(x\text{-}2) \\ \texttt{in even } y \ .$$

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

$$[\![M]\!]_\mu \equiv \texttt{let even\_odd} = \mu z.\langle \lambda x.\texttt{if } x = 0 \texttt{ then true else Snd } z\ (x\text{-}1), \\ \lambda x.\texttt{if } x = 0 \texttt{ then false else Fst } z\ (x\text{-}1)\rangle \\ \texttt{in Fst even\_odd } y \\ [\![N]\!]_\mu \equiv \texttt{let even} = \mu y.\lambda x.\texttt{if } x = 0 \texttt{ then true} \\ \texttt{else if } x = 1 \texttt{ then false else } y(x\text{-}2) \\ \texttt{in even } y \ .$$

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 β-reduction, obtaining:

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

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

$$\texttt{let even\_odd} = \langle \lambda x.\texttt{if } x = 0 \texttt{ then true else Snd } P\ (x\text{-}1), \\ \lambda x.\texttt{if } x = 0 \texttt{ then false else Fst } P\ (x\text{-}1)\rangle \\ \texttt{in if } y = 0 \texttt{ then true else Snd } P\ (y\text{-}1) \ .$$

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

with a string is a cyclic graph, called a *parse forest*. The lack of a theory regarding cyclic objects has forced some research projects investigating automatic programming environment generators, such as the ASF+SDF developed by Klint [22], to apply a disambiguation process to remove the cycles [23] and so retrieving a tree object. Familiar rewriting can then be applied on that object. This disambiguation process would not be required if cycles were part of the rewriting technology.

We conclude that a theory of cycles is necessary if one wants to reason about compilation, optimization and execution of programs. Presentation of such a theory is the goal of this paper.

What makes a theory of cycles difficult to develop is that confluence is lost once lambda-abstraction and cycles are admitted, unless the theory is powerful enough to represent irregular structures as shown in [6, 7]. To regain confluence, current formulations of cycles either impose restrictions, such as disallowing reduction under a lambda-abstraction or on a cycle [6, 7, 15, 31, 35], or adopt a framework based on interaction nets [24]. As discussed in [28] and [12], cycles do not destroy confluence in the context of interaction nets, but only at the expense of greater complexity.

In this paper, we limit our attention to cyclic lambda-graphs that occur in current implementations of strict and non-strict functional languages (we are not interested in optimality); thus, we will only consider cyclic lambda-graphs that naturally unwind to regular trees. In contrast to current approaches, we do not restrict the selection of redexes but introduce an alternative way of guaranteeing the consistency of the calculus. This consists of an approximate notion of confluence - *confluence up to information content*. This new notion is somewhat reminiscent of another 'approximate' notion of rewriting and confluence, namely, rewriting modulo an equivalence relation and confluence modulo equivalence [20]. However, unlike these notions, we do not combine rewriting with an equivalence relation, but with an equally fundamental notion, namely a quasi order, expressing a comparison between the 'information content' of the objects in question. Explicit studies of such a combination of a rewrite relation with a quasi order are not abundant; the only study that we are aware of is [37]. In the context of process algebra, Sangiorgi and Milner [36] consider equivalences of processes up to an asymmetric relation, such as a quasi order, as a technique to prove bisimulation.

The paper is organized as follows: We start in Sect. 2 by introducing cyclic lambda-graphs. As in Wadsworth [39], we do not deal with all possible lambda-graphs, but restrict our focus to the set of well-formed lambda-graphs. To define this class we introduce in Sect. 2.1 the notion of a scoped lambda-graph. The well-formed lambda-graphs are those that have associated scoped lambda-graphs. In Sect. 2.2, we introduce the syntactic formalism used to represent scoped lambda-graphs. Throughout this paper these syntactic objects are referred to as cyclic lambda terms. In Sect. 2.3, we present a mapping from cyclic lambda terms to scoped lambda-graphs. Since different cyclic terms can represent the same scoped lambda-graph, we introduce in Sect. 2.4 the *representational calculus*

$\mathcal{R}_0$. $\mathcal{R}_0$ equates all distinct representations of a scoped lambda-graph. Since we would also like to equate different representations of the same well-formed graph, we extend $\mathcal{R}_0$ with two other axioms in Sect. 2.5. We call the resulting system $\mathcal{R}_1$. In Sect. 2.6, we further extend $\mathcal{R}_1$ to make terms that unwind to the same tree provably equal. We call the calculus $\mathcal{R}_2$. $\mathcal{R}_2$ combined with a notion of β-reduction constitutes our cyclic extension of lambda calculus, which is presented in Sect. 3. In Sect. 4, we introduce the notions of confluence up to a quasi order, completeness up to a quasi order and infinite normal form. We present some sufficient conditions that guarantee soundness of the infinite normal form with respect to reduction. In Sect. 5, we show confluence up to information content of the cyclic lambda calculus. In Sect. 6, we prove that the infinite normal form defines a congruence with respect to the term formation rules, guaranteeing observational equivalence. In Sect. 7, we relate our cyclic lambda calculus to the traditional lambda calculus and to the infinitary lambda calculus of Kennaway *et al.* [21]. In Sect. 8, we add the notion of sharing to our cyclic calculus for reasoning about current implementations of non-strict functional languages. In the call-by-name calculus every term is substitutable, while in the sharing calculus substitution is restricted to values, thus avoiding duplication of work. We show that this restriction does not change the infinite normal form of a cyclic term. For reasoning about strict languages, in Sect. 9, we introduce the cyclic call-by-value calculus, which is obtained by restricting the garbage collection axiom of the sharing calculus to collect values only. This expresses the fact that strict and non-strict computations capture the same amount of sharing. The call-by-value calculus is then equipped with a term model, which allows us to relate our calculus to the commutative version of Moggi's computational lambda calculus [30] and to the recently developed calculus of Hasegawa [18]. The reader can refer to [3] for a detailed exposition and the proofs.

2 Graphs as Terms and Terms as Graphs

In this section we establish an isomorphism between cyclic lambda-graphs and their syntactic representations. We start by introducing a basic formalism for cyclic lambda-graphs in a format similar to the one used for first-order term graphs in [14].

2.1 Cyclic Lambda-graphs and Scoped Lambda-graphs

Following an idea used by Bourbaki in *Eléments de Théorie des Ensembles* to deal with quantifiers, an occurrence of a bound variable in a lambda-graph is represented by a back-pointer to the corresponding binding lambda-node. This implies we will not be able to represent the lambda-graph of Fig. 1, which Wadsworth [39] calls a non-admissible lambda-graph. Each argument of a node is either a normal pointer to some node, a back-pointer to a lambda-node, or is a free variable from the set of variables $\mathcal{V}$. A normal pointer is denoted by v, w, a back-pointer by $\overline{v}, \overline{w}$ and a variable by x, y, z. We let $A(v)_i$ denote the

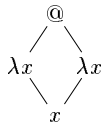


Fig. 1. Wadsworth's non-admissible lambda-graph

i^{th} argument of node v. A graph has a root r, which can be anything that an argument can be. If the label of a node is $\bullet$ then that node is called a black hole. The black hole denotes provable non-termination. It was already introduced in the first-order case to be able to reduce a cyclic graph in the presence of collapsing rules, *i.e.,* rules of the form $Ix \rightarrow x$. A more thorough discussion of the black hole is given in [9].

Notational conventions: given sets S and T, $\mathcal{P}(S)$ stands for the powerset of S, $S \setminus T$ stands for set difference, Σ^* stands for strings over the alphabet Σ, $|w|$ stands for the length of string w, and $V \oplus W$ stands for the disjoint union of sets V and W. From now on, we will sometimes write graph for cyclic lambda-graph.

Definition 1. A lambda-graph is a tuple (V, L, A, r) where

- V is a set (possibly infinite) of nodes.
- $L : V \rightarrow \{\lambda, @, \bullet\}$ is a labeling function.
- $A : V \rightarrow (V \oplus \{\overline{v} \mid v \in V, L(v) = \lambda\} \oplus \mathcal{V})^*$ is a successor function such that if $L(v) = \bullet$ then $|A(v)| = 0$, if $L(v) = \lambda$ then $|A(v)| = 1$ and if $L(v) = @$ then $|A(v)| = 2$.
- $r \in (V \oplus \{\overline{v} \mid v \in V, L(v) = \lambda\} \oplus \mathcal{V})$.

In addition to the usual conventions on how to give a graphical representation of a cyclic lambda-graph, we adopt the convention that if an arrow enters a lambda-node from above it represents a normal pointer and when it enters from below it represents a back-pointer. Thus, for example, we distinguish between the two lambda-graphs of Fig. 2. We draw a free variable as a labeled line and not as a node labeled x. For simplicity, we draw the label of the line at the end of the line. We picture the root as a pointer or a labeled line that has no starting node. Fig. 3 shows some examples of lambda-graphs. The lambda-graph on the left corresponds to a free variable x with a lambda-subgraph not accessible from the root. Note the difference between an arrow and a labeled line. In the following, when we talk about a path in a lambda-graph, we mean a directed path using only the arrows representing normal pointers. We refer to a graph in which all

Fig. 2. Pointers versus back-pointers

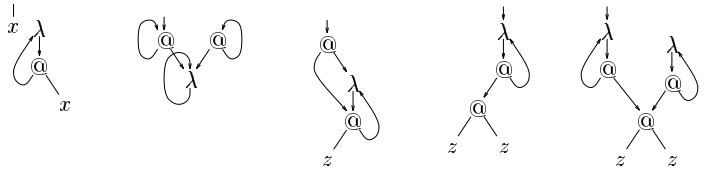


Fig. 3. Examples of lambda-graphs

nodes are reachable by a path starting at the root as a garbage free graph. We assume that equality between graphs stands for graph isomorphism.

In this paper, we will only deal with a subset of all possible lambda-graphs. For example, we will not consider the third lambda-graph of Fig. 3. To characterize this subset, referred to as the subset of the *well-formed lambda-graphs*, we introduce the notion of scope, which associates a set of nodes to a lambda-node. Intuitively, as it will be discussed in Sect. 3, the scope of a lambda-node v corresponds to that part of the graph that is copied when a β-reduction is performed with v denoting the function part of the application. This notion points out the main difference between our approach and that based on interaction nets [24]: interaction nets come equipped with certain operators that allow partial copying, that is, the copying can occur on a node-to-node basis whereas for us it occurs all at once.

Definition 2. A scoped lambda-graph is a tuple (V, L, A, S, r), where

- (V, L, A, r) is a lambda-graph
- $S : \{v \in V | L(v) = \lambda\} \rightarrow \mathcal{P}(V)$ is a function such that for every lambda-node v the following axioms apply.
  - *auto:* $v \in S(v)$
  - *bind:* $\forall w : \overline{v}$ is an argument of w then $w \in S(v)$
  - *upward-closure:* If $w_1 \notin S(v)$, $w_2 \in S(v)$ and w_2 is an argument of w_1 then $w_2 \equiv v$
  - *nesting:* $\forall$ lambda-nodes $w : S(w) \cap S(v) = \{\}$ or $S(w) \subseteq S(v) \backslash \{v\}$ or $S(v) \subseteq S(w) \backslash \{w\}$
- *root condition:* $r \in \mathcal{V}$ or $r \in V$ such that $\forall v : r \notin S(v) \backslash \{v\}$.

We depict the scope of v by drawing a line starting at one side of the lambda-node around all other nodes that are a member of the scope ending at the other side of the lambda-node.

Example 1. The graphs of Fig. 4 and 5 are examples of scoped and ill-formed scoped graphs, respectively. Since the scope of a lambda-node involves nodes and not edges, we can take the liberty to draw the label of an edge either inside or outside the scope. See the second graph of Fig. 4 in which the left label is outside the scope and the right one is inside. However, we will follow the convention that labels are drawn inside the scope.

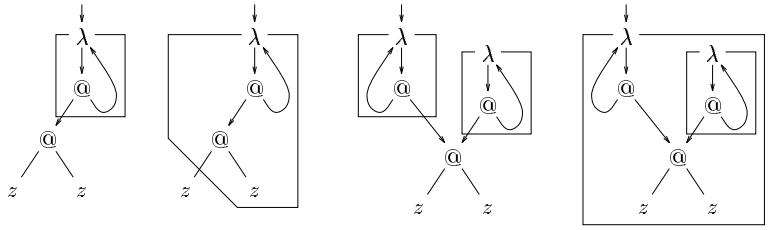


Fig. 4. Examples of scoped graphs

The first graph of Fig. 5 violates the *bind* scope axiom because the first application node has a back-pointer to the lambda-node v and is not a member of the scope of v. The second graph violates the *upward-closure* scope axiom because the scope of the upper lambda-node v is entered from a node different than v. The third graph violates the *nesting* scope axiom because the intersection of the two scopes is non-empty without one being a proper subset of the other. The fourth graph violates the *root* condition because the root points to a node inside a scope.

By definition, we get a lambda-graph when we drop the scope function from the tuple. However, not every lambda-graph is obtained by stripping the scope function from the tuple.

Definition 3. A lambda-graph (V, L, A, r) is well-formed if there exists a scope function S such that (V, L, A, S, r) is a scoped graph.

From now on we will assume that a graph consists of a finite set of nodes unless stated otherwise.

For garbage free graphs there is a necessary and sufficient condition to ensure that they are well-formed.

Proposition 4. *Given a garbage free graph g. Then g is well-formed iff for every node v with a back-pointer to w, w is on each path from the root to v.*

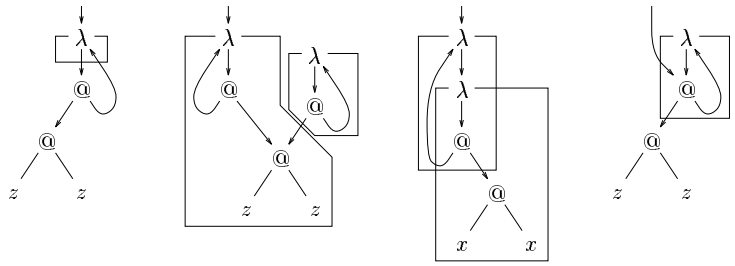


Fig. 5. Examples of ill-formed scoped graphs

Note that the third graph in Fig. 3 is not well-formed since it does not meet the condition stated in the proposition above.

2.2 Cyclic Lambda Terms: a Syntactic Representation of Scoped Lambda-graphs

We now introduce the syntactic formalism used to represent scoped graphs. The same formalism was already introduced in [8] for the first-order case, and extended with lambda-abstraction in [6, 7]. However, in that work a precise connection between terms and lambda-graphs was not established.

Definition 5. The following clauses define the syntax of cyclic lambda terms:

$$\begin{array}{ll} \text{Terms } (\Lambda\circ) & M ::= x \mid \lambda x.M \mid M\ N \mid \langle M \mid D \rangle \\ \text{Declarations} & D ::= x_1 = M_1, \ldots, x_n = M_n \end{array}$$

where the *recursion variables* x_i, $1 \leq i \leq n$, are distinct from each other.

In other words, the set of cyclic lambda terms consists of the lambda calculus terms (*i.e.,* variables, abstractions and applications) and the `letrec` construct: $\langle M \mid x_1 = M_1, \cdots, x_n = M_n \rangle$. We sometimes refer to D and M as the internal and external part of $\langle M \mid D \rangle$. Terms that differ in the order of the equations are identified. We adopt the following notation: A context C is a term with a hole, $\Box$, in the place of one subterm. The operation of filling the context C with a term M yields the term $C[M]$, possibly capturing some free variables of M in the process. By convention, bound and free variables are distinct from each other. $M[x := N]$ stands for the substitution of N for each free occurrence of x in M, without capturing any free variables in N. $\equiv$ stands for syntactic equivalence up to α-renaming, applied to both lambda-bound variables and recursion variables. If D_1 and D_2 are the lists of declarations $x_1 = M_1, \cdots, x_m = M_m$ and $y_1 = N_1, \cdots, y_n = N_n$, respectively, such that $\forall i, j : x_i \not\equiv y_j$ then we denote the list of declarations $x_1 = M_1, \cdots, x_m = M_m, y_1 = N_1, \cdots, y_n = N_n$ by D_1, D_2. When it is convenient to do so we sometimes denote a list of declarations as a set, *e.g.,* $D_1 = \{x_1 = M_1, \cdots, x_m = M_m\}$.

2.3 Mapping Cyclic Lambda Terms to Scoped Lambda-graphs

We define a mapping from cyclic terms to scoped graphs to give graph-semantics to cyclic terms. To simplify the definition of this mapping we introduce the notion of a *(scoped) pre-graph*, in which the condition on the arity of a black hole is relaxed.

Definition 6. A (scoped) pre-graph is a (scoped) graph where a node labeled with $\bullet$ may have 0 or 1 argument(s). If such a node has arity 0 we still call it a black hole but if it has arity 1 we call it an indirection node.

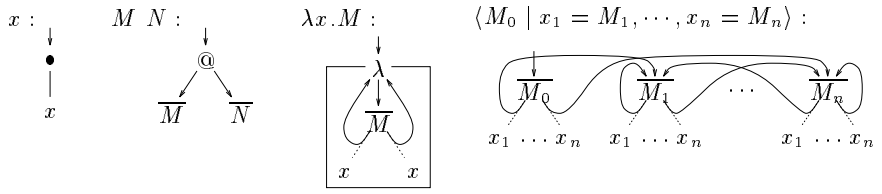


Fig. 6. Pictorial definition of ρ_{pre}

We have chosen the same symbol for indirection nodes and black holes because a black hole can be seen as a special case of an indirection node: an indirection node that refers to itself and therefore cannot be removed.

We map cyclic terms to scoped graphs via the mappings ρ_{pre} and Sim:

$$\begin{aligned} &\rho_{\mathrm{pre}} : \Lambda\circ && \rightarrow \text{scoped pre-graphs} \\ &\mathrm{Sim} : \text{scoped pre-graphs} && \rightarrow \text{scoped graphs} \ . \end{aligned}$$

The mapping ρ_{pre} transforms every lambda, every application and every occurrence of a variable that is not on the left-hand side of an =-sign into a node of the appropriate type (lambda, application and indirection, respectively). For a term $\lambda x.M$ the scope of the lambda contains the lambda-node itself and every node corresponding to the subterm M. The definition is given pictorially in Fig. 6, in which for simplicity we denote $\rho_{\mathrm{pre}}(M)$ by $\overline{M}$. Note that in the case for a lambda-abstraction, each labeled line x is transformed into a back-pointer. Instead, in the case for a letrec expression, the labeled lines $x_1 \cdots x_n$ are transformed into pointers.

The mapping Sim transforms a scoped pre-graph into a scoped graph by removing all indirection nodes or transforming them into black holes. This is accomplished by the three rewriting rules described in Fig. 7. The first rule transforms an indirection node that has itself as argument into a black hole simply by forgetting the argument. The second rule removes the indirection node v that points to a different node w (by normal or back-pointer) and redirects every pointer to v to w. The last rule removes an indirection node v that has a free variable x as an argument and changes every pointer to v to a line labeled x. The result of the normal form of these operations is well defined because we have local confluence and termination. We will refer to Sim(g) as the *simplification* of the scoped pre-graph g. The mapping ρ from cyclic terms to scoped graphs is then obtained by composing ρ_{pre} and Sim.

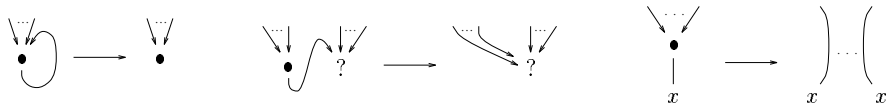


Fig. 7. Simplification of scoped pre-graphs

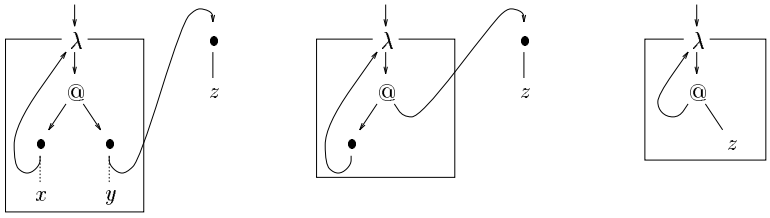


Fig. 8. Construction of $\rho(\langle \lambda x.x\ y \mid y = z \rangle)$

Definition 7. Given $M \in \Lambda\circ$. The scoped graph $\rho(M)$ is $\mathrm{Sim}(\rho_{\mathrm{pre}}(M))$.

Example 2. In Fig. 8 from left to right we have $\rho_{\mathrm{pre}}(\langle \lambda x.x\ y \mid y = z \rangle)$, the result of contracting one simplification redex and the result of contracting the remaining two simplification redexes. Note how the node labeled $\bullet$ with z as an argument is erased by giving its single argument to the application node.

Definition 8. Given $M \in \Lambda\circ$ and a scoped graph g. Then M represents g if $\rho(M) = g$.

Different cyclic terms are mapped to the same scoped graph by ρ, *e.g.*, $\rho(\langle x\ y \mid x = z, y = z \rangle) = \rho(zz)$. Thus, our next step is to characterize which cyclic lambda terms are mapped to the same scoped graph with a sound and complete set of axioms: the *representational calculus*.

2.4 Sound and Complete Axiomatization of Scoped Lambda-graphs

For all calculi developed in the paper, we assume the presence of the reflexivity axiom and the inference rules that make provable equality a congruence relation. In giving the axioms we assume that no variable capture occurs, *e.g.*, $x\langle x \mid x = z \rangle$ is not equated to $\langle xx \mid x = z \rangle$. Table 1 gives the axioms of the representational calculus $\mathcal{R}_0$.

Theorem 9. *The representational calculus is sound: If* $\mathcal{R}_0 \vdash M = N$ *then* $\rho(M) = \rho(N)$.

To show that the representational calculus is sufficient to equate all distinct representations of the same scoped graph, we introduce a mapping that associates a cyclic term to a scoped (pre)-graph:

$$\psi_{\mathrm{pre}} : \text{scoped pre-graphs} \to \Lambda\circ \ .$$

This mapping is based on the scheme of translating every node to an equation ($x = x$ for a black hole, $x = y$ for an indirection node, $x = y\ z$ for an application node and $x = \lambda y.\langle z \mid D \rangle$ for a lambda-node) and placing these equations in such a way that every equation gets placed in a subterm of a lambda-abstraction

Table 1. The axioms of $\mathcal{R}_0$

Lift:	$\langle M \mid D\rangle\ N = \langle M\ N \mid D\rangle$	
	$M\ \langle N \mid D\rangle = \langle M\ N \mid D\rangle$	
Empty box garbage collection:	$\langle M \mid\rangle = M$	
Merge:	$\langle\langle M \mid D_1\rangle \mid D_2\rangle = \langle M \mid D_1, D_2\rangle$	
	$\langle M \mid x = \langle N \mid D_1\rangle, D_2\rangle = \langle M \mid x = N, D_1, D_2\rangle$	
Variable substitution:	$\langle M \mid x = y, D\rangle = \langle M[x := y] \mid D[x := y]\rangle$	$x \not\equiv y$
Naming:	$M = \langle x \mid x = M\rangle$	x a new variable

corresponding to node v iff the equation came from a node in the scope of v except v itself.

The mappings ρ_{pre} and ψ_{pre} do not satisfy either $g = \rho_{\mathrm{pre}}(\psi_{\mathrm{pre}}(g))$ or $M \equiv \psi_{\mathrm{pre}}(\rho_{\mathrm{pre}}(M))$. For example, given $M \equiv xx$, $\psi_{\mathrm{pre}}(\rho_{\mathrm{pre}}(M))$ is $\langle x_1 \mid x_1 = x_2\ x_3, x_2 = x, x_3 = x\rangle$. We do, however, have that $\rho_{\mathrm{pre}}(\psi_{\mathrm{pre}}(g))$ simplifies to g. This means that for a scoped graph g there is a term M such that $\rho(M) = g$. Moreover, $\mathcal{R}_0 \vdash M = \psi_{\mathrm{pre}}(\rho(M))$ and if g simplifies to h then $\mathcal{R}_0 \vdash \psi_{\mathrm{pre}}(g) = \psi_{\mathrm{pre}}(h)$. This implies that the representational calculus is complete.

Theorem 10. *Given $M, N \in \Lambda\circ$. If $\rho(M) = \rho(N)$ then $\mathcal{R}_0 \vdash M = N$.*

2.5 Complete Axiomatization of Well-formed Graphs

We have now established how scoped graphs are represented by cyclic terms. Since a well-formed cyclic graph can have different scoped graphs associated with it, our next goal is to find axioms that equate the representations of these alternatively scoped graphs.

Definition 11. A scoped graph h is an alternatively scoped version of a graph $g \equiv (V, L, A, S, r)$ (written as $h \sim g$) if $h \equiv (V, L, A, S', r)$.

For example, the first two graphs of Fig. 4 are alternatively scoped versions of the same graph (*i.e.,* the fourth graph of Fig. 3). These two graphs are represented by the terms $\langle x \mid x = \lambda y.\langle w \mid w = w_1 y\rangle, w_1 = zz\rangle$ and $\langle x \mid x = \lambda y.\langle w \mid w = w_1 y, w_1 = zz\rangle\rangle$, respectively. It is then clear that if we want to equate the above two terms we need to extend the representational calculus with the following lambda lift axiom:

$$\lambda x.\langle M \mid D\rangle = \langle \lambda x.M \mid D\rangle \quad x \text{ not free in } D \ .$$

However, on graphs with garbage, the lambda lift axiom is not necessarily powerful enough. Consider Fig. 9, in which we have drawn two graphs that are alternatively scoped versions of the same graph. They are represented by the terms $\langle z \mid x_0 = \lambda y_0.\langle x \mid x_1 = \lambda y_1.\langle y \mid x_2 = y_0\ y_1\rangle\rangle\rangle$ and $\langle z \mid x_1 = \lambda y_1.\langle y \mid x_0 = \lambda y_0.\langle x \mid x_2 = y_0\ y_1\rangle\rangle\rangle$. Therefore, we introduce the garbage collection axiom:

$$\langle M \mid D\rangle = M \qquad\qquad D - M \ ,$$

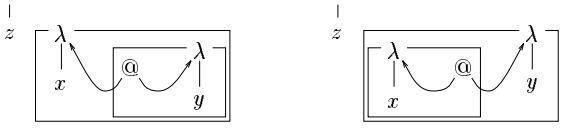


Fig. 9. Two scoped versions of the same graph

where $D - M$ means that the set of variables that occur as the left-hand side of an equation in D does not intersect with the set of free variables of M. We will also adopt the notation $D - D'$, where D' is $x_1 = M_1, \cdots, x_n = M_n$, which stands for $D - M_i, 1 \leq i \leq n$. We call $\mathcal{R}_1$ the representational calculus extended with the lambda lift and garbage collection axioms.

Theorem 12. *Given $M, N \in \Lambda\circ$. If $\rho(M) \sim \rho(N)$ then $\mathcal{R}_1 \vdash M = N$.*

2.6 Sound and Complete Axiomatization of Tree Unwinding

We now want to prove equal every two representations of graphs with the same tree unwinding. The $\mathcal{R}_1$ representational calculus is sound with respect to tree unwinding. To guarantee completeness we need another axiom, as was already pointed out in [7] for the first-order case. This axiom is the copy axiom, defined as:

$$M = N \quad \exists \sigma : \mathcal{V} \rightarrow \mathcal{V}, N^\sigma \equiv M \quad ,$$

where σ is a function from recursion variables to recursion variables, and N^σ is the term obtained by replacing all occurrences of recursion variables x by $\sigma(x)$ (leaving the free variables of N unchanged), followed by a reduction to normal form with the unification rule:

$$x = M, x = M \rightarrow x = M \quad .$$

For example, $\langle y \mid y = \lambda z.w, w = \lambda x.y \rangle = \langle y \mid y = \lambda z.w', w' = \lambda x.y', y' = \lambda z.w' \rangle$, where the mapping σ is: $w' \mapsto w$, $y \mapsto y$ and $y' \mapsto y$. This extension is called $\mathcal{R}_2$. We then have the following completeness result:

Theorem 13. *Given scoped graphs g and h with the same tree unwinding. If M and N represent g and h, respectively, then $\mathcal{R}_2 \vdash M = N$.*

We summarize the representational calculi in Table 2.

3 The Cyclic Lambda Calculus $\lambda\circ_{\mathbf{name}}$

The representational calculus $\mathcal{R}_2$ makes terms and graphs isomorphic. We now add computational power to our graphs and terms. This is done by first introducing β-reduction on scoped graphs. We will then give an axiomatic view of this reduction.

Table 2. Complete axiomatization of cyclic lambda-graphs and of tree unwinding.

$\mathcal{R}_0$	$\langle M \mid D\rangle\ N$	$= \langle M\ N \mid D\rangle$	
	$M\ \langle N \mid D\rangle$	$= \langle M\ N \mid D\rangle$	
	$\langle M \mid\rangle$	$= M$	
	$\langle\langle M \mid D_1\rangle \mid D_2\rangle$	$= \langle M \mid D_1, D_2\rangle$	
	$\langle M \mid x = \langle N \mid D_1\rangle, D_2\rangle$	$= \langle M \mid x = N, D_1, D_2\rangle$	
	M	$= \langle x \mid x = M\rangle$	x a new variable
	$\langle M \mid x = y, D\rangle$	$= \langle M[x := y] \mid D[x := y]\rangle$	$x \not\equiv y$
$\mathcal{R}_1$	$\langle M \mid D\rangle$	$= M$	$D \perp M$
	$\lambda x.\langle M \mid D\rangle$	$= \langle \lambda x.M \mid D\rangle$	x not free in D
$\mathcal{R}_2$	M	$= N$	$\exists \sigma : \mathcal{V} \to \mathcal{V}, N^\sigma \equiv M$

β-reduction on lambda-graphs

A β-redex in a lambda-graph is an application node whose first argument is a lambda-node. The contraction of a β-redex is a two-step process (see Fig. 10). In the first step we check if the reference to the lambda-node from the application node is unique and if the application node is outside the scope of the lambda-node. If one of these tests fails we copy the lambda-node and its scope in such a way that the test succeeds on the result (see the left step of Fig. 10). We then place the lambda-node (or the copy, if one has been made) and its scope in the same scope as the application node (the original if a copy has been made). The second step is a redirection of pointers consisting of: 1) replacing the application node by an indirection node whose argument is the argument of the lambda-node, 2) replacing the lambda-node by an indirection node whose argument is the former right argument of the application node. Note that all pointers to the indirection node replacing the lambda-node need to be changed from back-pointers to normal pointers. This second step is drawn on the right of Fig. 11. On the left of the same figure we have drawn the β-reduction principle used by interaction nets [24]. There the use of indirection nodes is superfluous because there is exactly one reference to the application node and exactly one back-pointer to the lambda-node.

Axiomatization of β-reduction

We now proceed by giving axioms on cyclic terms that describe β-reduction. The first step is described by the axioms introduced so far. For the second step

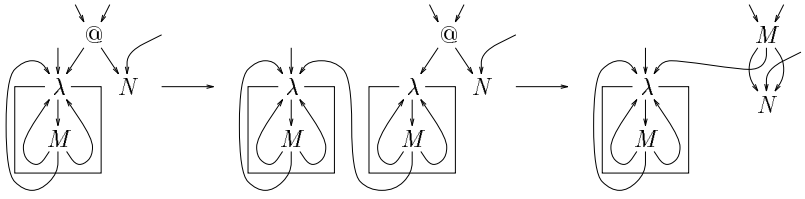


Fig. 10. β-reduction on scoped lambda-graphs

Fig. 11. Beta-reduction principles for interaction nets and lambda-graphs

we need to introduce the following $\beta\circ$-axiom:

$$(\lambda x.M)\ N = \langle M \mid x = N \rangle \ .$$

Example 3. The β-reduction of $\langle x \mid x = \lambda y.x\ y \rangle$ (see Fig. 12) is described below.

$$\begin{array}{rll}
\langle x \mid x = \lambda y.x\ y \rangle & = \langle x \mid x = \lambda y.x'\ y, x' = \lambda y'.xy' \rangle & \text{copy} \\
& = \langle x \mid x = \langle \lambda y.x'\ y \mid x' = \lambda y'.xy' \rangle \rangle & \text{merge} \\
& = \langle x \mid x = \lambda y.\langle x' \mid x' = \lambda y'.xy' \rangle y \rangle & \text{lift} \\
& = \langle x \mid x = \lambda y.(\lambda y'.xy')y \rangle & \text{naming} \\
& = \langle x \mid x = \lambda y.\langle xy' \mid y' = y \rangle \rangle & \beta\circ.
\end{array}$$

In [6] we have called the subterm xy of $\langle x \mid x = \lambda y.xy \rangle$ an *implicit β-redex* which needs to be made *explicit*, *i.e.,* of the form $(\lambda x.P)Q$, in order to be reduced. In our cyclic calculus, an implicit β-redex can be made explicit by the use of the $\mathcal{R}_2$ representational axioms. However, we would also like to make a redex explicit by applying a representational rewriting system. In that respect the rules obtained by orienting the $\mathcal{R}_2$ axioms from left to right are not sufficient. We thus introduce the following two axioms:

$$\begin{array}{c}
\langle C[y] \mid y = M, D \rangle = \langle C[M] \mid y = M, D \rangle \\
\langle N \mid x = C[y], y = M, D \rangle = \langle N \mid x = C[M], y = M, D \rangle \ ,
\end{array}$$

called external and internal substitution, respectively. These axioms capture the inlining transformations described in the introduction, and they are derivable in $\mathcal{R}_2$. The rules obtained by orienting these axioms from left to right are enough to expose implicit redexes. Referring to Example 3 we have: $\langle x \mid x = \lambda y.xy \rangle \rightarrow \langle x \mid x = \lambda y.(\lambda y'.xy')y \rangle \rightarrow \langle x \mid x = \lambda y.\langle xy' \mid y' = y \rangle \rangle$.

Definition 14. The axioms of the call-by-name cyclic calculus ($\lambda\circ_{\text{name}}$) are given in Table 3. $\xrightarrow[\lambda\circ_{\text{name}}]{}$ denotes the reduction relation obtained by orienting all axioms from left to right and by imposing the restriction $D' \neq \{\}$ to the garbage collection rule and to the lambda lift rule.

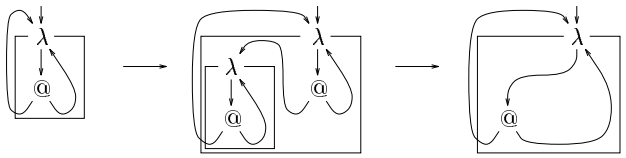


Fig. 12. β-reducing $\langle x \mid x = \lambda y.x\ y \rangle$.

Table 3. The axioms of $\lambda\circ_{\text{name}}$

$\beta\circ$:	$(\lambda x.M)N = \langle M \mid x = N\rangle$	
Substitution:	$\langle C[x] \mid x = M, D\rangle = \langle C[M] \mid x = M, D\rangle$	
	$\langle N \mid x = C[x_1], x_1 = M, D\rangle = \langle N \mid x = C[M], x_1 = M, D\rangle$	
Lift:	$\langle M \mid D\rangle N = \langle MN \mid D\rangle$	
	$M\langle N \mid D\rangle = \langle MN \mid D\rangle$	
	$\lambda x.\langle M \mid D, D'\rangle = \langle\langle \lambda x.M \mid D\rangle \mid D'\rangle$	$D \perp D'$ and x not free in D'
Merge:	$\langle M \mid x = \langle N \mid D\rangle, D_1\rangle = \langle M \mid x = N, D, D_1\rangle$	
	$\langle\langle M \mid D\rangle \mid D'\rangle = \langle M \mid D, D'\rangle$	
Garbage collection:	$\langle M \mid D, D'\rangle = \langle M \mid D\rangle$	$D' \perp \langle M \mid D\rangle$
	$\langle M \mid \rangle = M$	
Copying:	$M = N$	$\exists \sigma : \mathcal{V} \to \mathcal{V}, N^\sigma \equiv M$

In general, we denote by $\xrightarrow[R]{}\!\!\!\!\!\!\rightarrow$ the reflexive and transitive closure of a reduction relation $\xrightarrow[R]{}$. From now on, we will omit the subscript name when no confusion arises. We have omitted naming and the variable substitution axioms since they are derivable. We have also adopted a more general form of the lambda lift axiom, *i.e.,* we move subsets of equations out of a letrec. This is because we want to have a common reduct for the two reducts $\langle \lambda x.\langle y \mid y = z\ x\rangle \mid z = z\rangle$ and $\lambda x.\langle y \mid y = z\ x, z = z\rangle$ of $\lambda x.\langle\langle y \mid y = z\ x\rangle \mid z = z\rangle$. By a similar argument, we have introduced a more general form of the garbage collection axiom which allows us to remove subsets of equations.

The main difference between $\lambda\circ_{\text{name}}$ and the $\lambda\phi$-calculi [6, 7] involves the substitution and merge operations. Here, these operations may occur in any context, whereas in [6, 7] they cannot occur on a cycle. For example, the $\lambda\phi$-calculi disallow the step $\langle x \mid x = \lambda z.y(Sz), y = \lambda w.x(Sw)\rangle \to \langle x \mid x = \lambda z.(\lambda w.x(Sw))(Sz), y = \lambda w.x(Sw)\rangle$. This is an example of a cyclic substitution, since x and y lie on the same cyclic plane. Cyclic substitution is the cause of non-confluence, as shown by the following example:

$$
\begin{array}{rll}
M \equiv & \langle x \mid x = \lambda z.y(Sz), y = \lambda w.x(Sw)\rangle & \\
\to & \langle x \mid x = \lambda z.(\lambda w.x(Sw))(Sz), y = \lambda w.x(Sw)\rangle & \\
\twoheadrightarrow & \langle x \mid x = \lambda z.x(S(Sz)), y = \lambda w.x(Sw)\rangle & (*) \\
M \to & \langle x \mid x = \lambda z.y(Sz), y = \lambda w.(\lambda z.y(Sz))(Sw)\rangle & \\
\twoheadrightarrow & \langle x \mid x = \lambda z.y(Sz), y = \lambda w.y(S(Sw)) & (**)
\end{array}
$$

Now, the terms (*) and (**) have no common reduct, since in the term (*) an even number of $S's$ is reachable from the root x , while the term (**) will contain an odd number. This 'out-of-synch' phenomenon is also observed by reducing (in at most ω steps) the infinite terms that arise by unwinding the cyclic graphs. By disallowing the substitutions for x and y the counterexample disappears.

Moreover, in the $\lambda\phi$-calculi the merge rules had the proviso that only acyclic letrec's could be merged. For example, the following step is illegal: $\langle x \mid x = \langle \lambda z.y(Sz) \mid y = \lambda w.x(Sw)\rangle\rangle \to \langle x \mid x = \lambda z.y(Sz), y = \lambda w.x(Sw)\rangle$. If the above step were allowed then confluence would have been lost, since the acylic

substitution for y is turned into a cyclic substitution once the internal letrec is removed.

In summary, in [6, 7], the focus was on finding a confluent calculus that could express cyclic lambda graph rewriting. Instead, we do not take confluence as the guiding factor in designing the calculus. Thus, we do not restrict the calculus, but introduce a new way of proving the consistency of the calculus. More specifically, we introduce an approximate notion of confluence - *confluence up to information content*. This notion allows us to abstract away syntactic details.

4 Approximate Notion of Confluence

Once cycles are admitted it seems natural to consider infinite normal forms instead of normal forms. We thus introduce a new property, confluence up to a quasi order, which guarantees unicity of infinite normal forms. In the rest of this section we work with abstract rewriting systems, since we do not need the extra structure terms have to define the necessary notions.

Confluence up to a quasi order

We start by introducing a few notions about abstract rewriting system where the set of objects also has a quasi order defined on it.

Definition 15. An ordered abstract rewriting system is a structure $(A, \rightarrow, \preceq)$, where $(A, \rightarrow)$ is an ARS (abstract rewriting system) and $(A, \preceq)$ is a quasi order.

Given an ordered ARS $(A, \rightarrow, \preceq)$. Then

- $\rightarrow$ is monotonic with respect to $\preceq$ if $\forall a, b \in A : a \rightarrow b$ implies $a \preceq b$.
- $\rightarrow$ is confluent up to $\preceq$ if $\forall a, b, c \in A : a \twoheadrightarrow b, a \twoheadrightarrow c$ implies $\exists d \in A : b \twoheadrightarrow d, c \preceq d$.
- We denote by $\overset{\preceq}{\longleftrightarrow}$ the reflexive transitive closure of $((\leftarrow \cup \rightarrow) \cap \preceq)$.

Next, we give an analysis of confluence up to in terms of some simpler properties. We begin with some definitions.

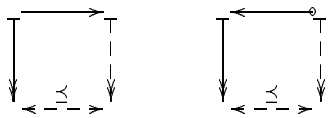
Definition 16. Given an ordered ARS $(A, \rightarrow, \preceq)$, and another reduction relation $\mapsto \subseteq \rightarrow$. Then

- $\mapsto$ is complete for $\rightarrow$ up to $\preceq$ if $\forall a, b \in A : a \twoheadrightarrow b$ implies $\exists c \in A : a \mapsto\!\!\!\!\twoheadrightarrow c, b \preceq c$.
- $\mapsto$ commutes with $\rightarrow$ up to $\preceq$ if $\forall a, b, c \in A : a \twoheadrightarrow b, a \mapsto\!\!\!\!\twoheadrightarrow c$ implies $\exists d \in A : b \mapsto\!\!\!\!\twoheadrightarrow d, c \preceq d$.
- We denote by $\circ\!\!\rightarrow$ the reduction relation $\rightarrow \setminus \mapsto$.

Lemma 17. *Given an ordered ARS $(A, \rightarrow, \preceq)$. Let $\mapsto \subseteq \rightarrow$ such that $\mapsto$ is complete for $\rightarrow$ up to $\preceq$ and $\mapsto$ commutes with $\rightarrow$ up to $\preceq$. Then $\rightarrow$ is confluent up to $\preceq$.*

To prove that the conditions stated in the lemma above hold for our cyclic calculi we will use the following lemma.

Lemma 18. *Given an ordered ARS $(A, \rightarrow, \preceq)$ and a relation $\mapsto \subseteq \rightarrow$. If the following two diagrams hold*



then $\rightarrow$ commutes with $\mapsto$ up to $\preceq$ and $\mapsto$ is complete up to $\preceq$.

Infinite normal form

We define the infinite normal form of a term as the maximum information that can be obtained by reducing that term. We model the information content of a term as a function from the set of objects of an ARS to a partial order. This function and the partial order induce a quasi order on the elements of the ARS. The intuition of information content demands that the rewrite relation is monotonic with respect to the induced order. Formally:

Definition 19. The structure $\langle (A, \rightarrow, \preceq), \omega, (B, \leq) \rangle$ is called an ARS with ordered information content if:
- $(A, \rightarrow, \preceq)$ is an ordered ARS
- $\rightarrow$ is monotonic with respect to $\preceq$
- $(B, \leq)$ is a partially order set
- ω is a function $A \rightarrow B$
- $a \preceq b$ iff $\omega(a) \leq \omega(b)$.

We often refer to $\omega(a)$ as the information content of a.

Definition 20. Given a partial order $(B, \leq)$. We define the downward closure of a set $C \subseteq B$, denoted by $\downarrow C$, as $\{b \in B \mid b \leq a \in C\}$.

Definition 21. Given an ARS $\mathcal{A}$ with ordered information content $\langle (A, \rightarrow, \preceq), \omega, (B, \leq) \rangle$.
- The infinite normal form of an $a \in A$, $\mathrm{Inf}(a)$, is defined as $\downarrow \{\omega(b) \mid a \twoheadrightarrow b\}$.
- $\mathcal{A}$ has unique infinite normal forms if $a \rightarrow b$ implies $\mathrm{Inf}(a) = \mathrm{Inf}(b)$.

Next, we show that confluence up to guarantees that the infinite normal form is an ideal. Given a partial order $(B, \leq)$, a subset I of B is an ideal iff (i) I is non-empty, (ii) $\forall a, b \in I, \exists c \in I, a \leq c$ and $b \leq c$, (iii) $\forall c \in I$, if $\exists d \in B, d \leq c$ then $d \in I$.

Proposition 22. *Given an ARS with ordered information content $\langle (A, \rightarrow, \preceq), \omega, (B, \leq) \rangle$. If $\rightarrow$ is confluent up to $\preceq$ then $\mathrm{Inf}(a)$ is an ideal.*

Confluence up to is a sufficient and necessary condition for the unicity of infinite normal forms to hold.

Proposition 23. *Given an ARS $\mathcal{A}$ with ordered information content $\langle (A, \rightarrow, \preceq), \omega, (B, \leq) \rangle$. $\rightarrow$ is confluent up to $\preceq$ iff $\mathcal{A}$ has the unique infinite normal form property.*

Table 4. Evaluation calculus: $\lambda\circ_{\text{eval}}$

$\beta\circ$:	$(\lambda x.M)N \doteq \langle M \mid x = N\rangle$
External substitution:	$\langle C[x] \mid x = M, D\rangle \doteq \langle C[M] \mid x = M, D\rangle$
Lift:	$\langle M \mid D\rangle N \doteq \langle MN \mid D\rangle$

5 Basic Properties of the Cyclic Lambda Calculus

We start by showing that $\lambda\circ$ is an abstract reduction system with ordered information content that is confluent up to information content. The information content corresponds to the *approximate normal form* of Wadsworth [39], also called *direct approximation* by Lévy [27]. In contrast to [27], we do not send all redexes and compatible redexes to Ω. Only $\beta\circ$ and the occurrences of variables that correspond to external substitution redexes are sent to Ω. As in lambda calculus, ΩM is also sent to Ω, since it could become a $\beta\circ$-redex by replacing Ω with a lambda-abstraction. The inaccessible equations are then removed.

Definition 24. Given $M, N \in \Lambda\circ$. The information content of M is given by the function ω, which given M returns the normal form of M with respect to the following rules (also called ω-rules):

$$\begin{aligned}
(\lambda x.M)N &\xrightarrow[\omega]{} \Omega \\
\langle C[x] \mid x = M, D\rangle &\xrightarrow[\omega]{} \langle C[\Omega] \mid x = M, D\rangle \\
\Omega M &\xrightarrow[\omega]{} \Omega \\
\langle M \mid D\rangle &\xrightarrow[\omega]{} M \qquad\qquad D - M
\end{aligned}$$

We define $M \preceq_{\text{name}} N$ if $\omega(M) \leq_\Omega \omega(N)$, where the order $\leq_\Omega$ is generated by the axiom $\Omega \leq_\Omega M$, for every term M. If $M \preceq_{\text{name}} N$ and $N \preceq_{\text{name}} M$ then we let $M \simeq_{\text{name}} N$.

The ω-function is well defined due to the termination and confluence of the ω-rules (due to Newman's lemma). Termination of the ω-rules follows from counting the number of non-Ω symbols in every term. Examples: $\omega(\langle \lambda x.yz \mid y = \lambda w.w\rangle) = \lambda x.\Omega$, $\omega(\langle x \mid x = x\rangle) = \Omega$, $\omega(\langle xy \mid y = \lambda w.w\rangle x) = (x\Omega)x$, and $\omega(\langle xx \mid x = \lambda w.w\rangle) = \Omega$. Note that even though $\langle xy \mid y = \lambda w.w\rangle x$ is a lift redex, its information content is not Ω.

Since $\xrightarrow[\lambda\circ]{}$ is monotonic with respect to $\preceq_{\text{name}}$ we have the following:

Proposition 25. $\langle(\Lambda\circ, \xrightarrow[\lambda\circ]{}, \preceq_{\text{name}}), \omega, (\omega(\Lambda\circ), \leq_\Omega)\rangle$ *is an ARS with ordered information content.*

Next, we present a standard reduction strategy that is complete up to information content. The idea behind standard reduction is that we take a simple subset of the rewriting system, such that only standard steps can increase information. We first restrict the system to the evaluation system of Table 4. We

need $\beta\circ$ and external substitution because those two rules potentially increase information content. We need lift to expose $\beta\circ$-redexes that are implicit. We then restrict the standard redex so that it cannot occur in the internal part D of a construct $\langle M \mid D\rangle$. We also disallow reduction of redexes that could be moved into an environment by contracting a $\beta\circ$-redex. This guarantees confluence of the standard reduction.

We illustrate the basic idea behind the formal definition of standard reduction through an example. Let us consider the term $\lambda x.(\langle x \mid y = y\rangle(\langle \lambda w.w \mid y = y\rangle z))$. We start by looking for a standard redex at the outermost position. Since we find a lambda, we look inside it and find a lift redex which we could neglect, since the redex is not obstructing a $\beta\circ$-redex. Assuming we do neglect it, we next look at the left argument of the application. Since variable x is a lambda bound variable, it can never become a redex, so we start looking in the right-hand side of the application. We again find a lift redex. This time we must reduce it because it is obstructing a $\beta\circ$-redex. This informal discussion points out how we split lift redexes into two categories: those redexes that (will in the future) obstruct a $\beta\circ$-redex (*e.g.,* $\langle \lambda x.M \mid D\rangle N$ and $\langle\langle y \mid D\rangle N \mid y = \lambda x.M\rangle$), and those that never will (*e.g.,* $\langle y \mid D\rangle M$). The redexes in the first category must be reduced and the other ones can be delayed.

Definition 26. Given $M, N \in \Lambda\circ$. M standard rewrites to N ($M \xmapsto[\lambda\circ]{} N$) if $M \equiv E[R] \xrightarrow[\lambda\circ]{} E[R'] \equiv N$, where R and R' stand for a $\lambda\circ_{\mathrm{eval}}$-redex and its contractum, and E is defined as follows:

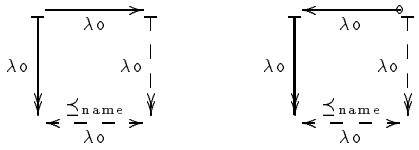
$$\begin{array}{ll} E & ::= \lambda x.E \mid \langle E \mid D\rangle \mid App[\Box, M_1, \cdots, M_n] \mid App[y, M_1, \cdots E \cdots, M_n] \\ App & ::= \Box \mid App\Box \mid \langle App \mid D\rangle \end{array}$$

where the y must be bound by a lambda or free in the final expression.

Example 4. $\lambda x.(\langle x \mid y = y\rangle(\langle \lambda w.w \mid y = y\rangle z))$ is partitioned as $E_1[(\langle x \mid y = y\rangle(\langle \lambda w.w \mid y = y\rangle z))]$, where E_1 is $\lambda x.\Box$, or as $E_2[\langle \lambda w.w \mid y = y\rangle z]$, where E_2 is $\lambda x.\langle x \mid y = y\rangle\Box$. $\langle x \mid x = y\rangle y$ is partitioned as $E_3[x]$, where E_3 is $\langle \Box \mid x = y\rangle y$. The redex $(\lambda z.z)(\lambda z.z)$ in $\langle x \mid x = y\rangle(\lambda z.z)(\lambda z.z)$ is not a standard redex, since $\langle x \mid x = y\rangle\Box \neq E[\Box]$. To make it standard, an external substitution step must first take place.

Proposition 27. *Given $M, N \in \Lambda\circ$. If $M \circ\!\xrightarrow[\lambda\circ]{} N$ then $M \simeq_{\mathrm{name}} N$.*

Lemma 28. *We have the following two diagrams:*



From the above lemma, Lemmas 17 and 18 we then have:

Theorem 29. *Given $\langle(\Lambda\circ, \xrightarrow[\lambda\circ]{}, \preceq_{\mathrm{name}}), \omega, (\omega(\Lambda\circ), \leq_\Omega)\rangle$. $\xrightarrow[\lambda\circ]{}$ is confluent up to $\preceq_{\mathrm{name}}$.*

6 Semantics of the Cyclic Lambda Calculus

Since $\xrightarrow[\lambda\circ]{}$ is confluent up to information content, by Proposition 23 the infinite normal form of any cyclic term M, written as $\mathrm{Inf}_{\lambda\circ}(M)$, is well defined and unique. Next, we want to show that the infinite normal form defines a congruence, *i.e.,* if $\mathrm{Inf}_{\lambda\circ}(M) = \mathrm{Inf}_{\lambda\circ}(N)$ then $\mathrm{Inf}_{\lambda\circ}(C[M]) = \mathrm{Inf}_{\lambda\circ}(C[N])$, for all contexts C. To that end, let us first show that the infinite normal forms of a cyclic term computed with respect to $\xrightarrow[\lambda\circ]{}$, $\overset{\cdot}{\xrightarrow[\lambda\circ]{}}$ (written as $\mathrm{Inf}_{\mathrm{std}}$) and $\xrightarrow[\lambda\circ_{\mathrm{eval}}]{}$ (written as $\mathrm{Inf}_{\mathrm{eval}}$) are the same. By the infinite normal form of a cyclic term computed with respect to $\xrightarrow[R]{}$, we mean the infinite normal form computed with respect to $\langle(\Lambda\circ, \xrightarrow[R]{}, \preceq_{\mathrm{name}}), \omega, (\omega(\Lambda\circ), \leq_\Omega)\rangle$.

Proposition 30. *Given a term $M \in \Lambda\circ$. $\mathrm{Inf}_{\mathrm{std}}(M) = \mathrm{Inf}_{\mathrm{eval}}(M) = \mathrm{Inf}_{\lambda\circ}(M)$.*

This result allows us to prove that the infinite normal form computed with respect to $\xrightarrow[\lambda\circ]{}$ is a congruence, by showing congruence of the infinite normal form computed with respect to the evaluation calculus. Congruence with respect to the evaluation calculus is easier to prove, since that calculus is confluent by using the complete development method.

As in [27], we prove some properties of reduction and of ω-reduction. However, we formulate these properties in terms of information content.

Proposition 31. *Given a term $M \in \Lambda\circ$ and a context C.*

- *If $C[M] \xrightarrow[\mathrm{eval}]{}\!\!\!\!\!\rightarrow N$ then there exists an M_1 such that $M \xrightarrow[\mathrm{eval}]{}\!\!\!\!\!\rightarrow M_1$, $C[M_1] \xrightarrow[\mathrm{eval}]{}\!\!\!\!\!\rightarrow P$ without reducing any redex inside M_1 ($\not\xrightarrow[M_1]{}\!\!\!\!\!\rightarrow$) and $N \xrightarrow[\mathrm{eval}]{}\!\!\!\!\!\rightarrow P$.*
- *If $C[M] \not\xrightarrow[M]{\mathrm{eval}}\!\!\!\!\!\rightarrow N$ then $C[\omega(M)] \xrightarrow[\mathrm{eval}]{}\!\!\!\!\!\rightarrow N_1$ with $N \simeq_{\mathrm{name}} N_1$.*
- $\mathrm{Inf}_{\mathrm{eval}}(C[\omega(M)]) \subseteq \mathrm{Inf}_{\mathrm{eval}}(C[M])$.

Theorem 32. *Given $M, N \in \Lambda\circ$. If $\mathrm{Inf}_{\lambda\circ}(M) = \mathrm{Inf}_{\lambda\circ}(N)$ then $\mathrm{Inf}_{\lambda\circ}(C[M]) = \mathrm{Inf}_{\lambda\circ}(C[N])$.*

7 The Cyclic Lambda Calculus, the Traditional Lambda Calculus and the Infinitary Lambda Calculus

As it was done in [2] for the first-order case, we can use the model to relate $\lambda\circ$ to the traditional lambda calculus. We show that cycles can be explained in terms of their expansions, which are finite lambda calculus terms. To define expansions, we introduce the notation $M \xrightarrow[\mathrm{GK(es)}]{}\!\!\!\!\!\rightarrow^n N$ which denotes n-steps of the Gross-Knuth strategy applied to the external substitution redexes occurring in M (*i.e.,* all external substitution redexes are performed). If M does not contain any external substitution redexes we will still write $M \xrightarrow[\mathrm{GK(es)}]{} N$.

Definition 33. Given $M \in \Lambda\circ$. The n^{th} expansion of M, written as M^n, is the term $strip(N)$ such that $M \xrightarrow[\text{GK(es)}]{}{}^{n} N$ and $strip(N)$ is the normal form of N with respect to the rules:

$$\begin{array}{lll} \langle C[x] \mid x = M, D\rangle & \to \langle C[\Omega] \mid x = M, D\rangle & \\ \langle M \mid D\rangle & \to M & \text{if } D - M \end{array}$$

We build towards the main result with two lemmas. The first lemma relates suitable $\beta\circ$ and external substitution sequences on graphs to β-reduction sequences on lambda calculus terms. The second lemma relates each piece of information derivable from a cyclic term to information derivable from an expansion of that term.

Lemma 34. *Given $M, M_1, M_2 \in \Lambda\circ$. If $M \xrightarrow[\beta\circ]{} M_1 \xrightarrow[\text{es}]{} M_2$ such that $M_1 \xrightarrow[\text{es}]{} M_2$ is a complete development of all the external substitution redexes created by the $\beta\circ$-steps then* $\text{strip}(M) \xrightarrow[\beta]{} \text{strip}(M_2)$.

Lemma 35. *Given $M, N \in \Lambda\circ$. If $M \xrightarrow[\lambda\circ]{} N$ then there exists an i and $P \in \Lambda$ such that $M^i \xrightarrow[\beta]{} P$ and $N \preceq_{\text{name}} P$.*

If we let $\text{Inf}_\lambda(M^i)$ denote the Lévy-Longo tree (as described in [26]) associated with a lambda calculus term, we then have the following result:

Theorem 36. *Given $M \in \Lambda\circ$. $\text{Inf}_{\lambda\circ}(M) = \bigcup\{\text{Inf}_\lambda(M^i) \mid i \geq 0\}$.*

Since cyclic terms unwind to infinite trees, it is natural to relate the cyclic calculus to the infinitary lambda calculus (λ^∞) of Kennaway *et al.* [21]. We have soundness, *i.e.*, if $M \xrightarrow[\lambda\circ]{} N$ then $M^\infty \xrightarrow[\lambda^\infty]{}{}^{\leq\omega} N^\infty$, where M^∞ is $\lim_{n\to\infty} M^n$ and $\xrightarrow[\lambda^\infty]{}{}^{\leq\omega}$ stands for a possibly infinite number of steps in λ^∞. A weak notion of completeness also holds: if $M^\infty \xrightarrow[\lambda^\infty]{} s$ then each finite prefix of s can be obtained in $\lambda\circ$ by reducing M. For this result to hold, the restricted calculus of Table 4 suffices. The semantics of $\lambda\circ$ as provided by the infinitary calculus differs from the Lévy-Longo tree model in that it distinguishes between ΩM and Ω.

We also point out that the axioms to distribute the substitution across a term in a stepwise manner, which are present in the explicit substitution calculi [1, 35], are derivable in our calculus and that they generate the same model.

8 The Cyclic Sharing Calculus $\lambda\circ_{\text{share}}$

A drawback of the cyclic lambda calculus is that it does not support sharing adequately, since it allows reductions that duplicate work. For example, in the reduction $\langle x \mid x = (\lambda y.y)(\lambda y.y)\rangle \xrightarrow[\lambda\circ]{} \langle(\lambda y.y)(\lambda y.y) \mid x = (\lambda y.y)(\lambda y.y)\rangle$, the $\beta\circ$-redex has been duplicated. Current implementations of functional languages, such as Haskell [19] and Id [32], do not allow these kinds of reductions. Therefore, we develop a variant of the cyclic calculus that takes sharing into consideration.

Table 5. The axioms of $\lambda\circ_{\mathrm{share}}$

$\beta\circ$:	$(\lambda x.M)N = \langle M \mid x = N\rangle$	
Substitution:	$\langle C[x] \mid x = V, D\rangle = \langle C[V] \mid x = V, D\rangle$	
	$\langle M \mid x = C[x_1], x_1 = V, D\rangle = \langle M \mid x = C[V], x_1 = V, D\rangle$	
Lift:	$\langle M \mid D\rangle N = \langle MN \mid D\rangle$	
	$M\langle N \mid D\rangle = \langle MN \mid D\rangle$	
	$\lambda x.\langle M \mid D, VD\rangle = \langle \lambda x.\langle M \mid D\rangle \mid VD\rangle$	$D \perp VD$ and x not free in VD
Merge:	$\langle M \mid x = \langle N \mid D\rangle, D_1\rangle = \langle M \mid x = N, D, D_1\rangle$	
	$\langle\langle M \mid D\rangle \mid D'\rangle = \langle M \mid D, D'\rangle$	
Garbage collection:	$\langle M \mid D, D'\rangle = \langle M \mid D\rangle$	$D' \perp \langle M \mid D\rangle$
	$\langle M \mid \rangle = M$	
Copying:	$M = N$	$\exists\sigma : \mathcal{V} \to \mathcal{V}, N^{\sigma} \equiv M$ and $\forall x \not\equiv x', \sigma(x) \equiv \sigma(x') : \sigma(x)$ bound to a value in M
Naming:	$M = \langle x \mid x = M\rangle$	x a new variable

We emphasize that we are only interested in capturing the sharing present in current implementations (lazy and lenient) of non-strict languages. We do not study the sharing present in optimal (in the sense of Lévy [27]) implementations of lambda calculus. Since the emphasis is on sharing and not on a specific reduction strategy, we call the calculus the sharing calculus ($\lambda\circ_{\mathrm{share}}$), because call-by-need normally implies lazy evaluation.

The sharing calculus is obtained by restricting the operations that cause duplication, such as substitution and copying, so that only values are duplicated, where a value is either a variable or a lambda-abstraction. Also, the lambda lift axiom has to be restricted to lift values only, since lifting unevaluated expressions out of a lambda-abstraction has an impact on the amount of sharing captured.

We add the following syntactic clauses of values and value declarations

$$\begin{aligned} V &::= x \mid \lambda x.M \\ VD &::= x_1 = V_1, \ldots, x_n = V_n \end{aligned}$$

to the ones of Definition 5.

Definition 37. The axioms of the cyclic sharing calculus ($\lambda\circ_{\mathrm{share}}$) are given in Table 5. $\xrightarrow[\lambda\circ_{\mathrm{share}}]{}$ denotes the rewrite relation obtained by reading all the axioms, except naming, left to right, and by imposing the restrictions $VD \neq \{\}$ and $D' \neq \{\}$ to the lambda lift rule and to the garbage collection rule. The naming axiom is introduced in the reduction theory in the following form:

$$C_{\mathrm{safe}}[M\ N] \to C_{\mathrm{safe}}[\langle x \mid x = M\ N\rangle] \quad x \text{ a new variable}$$

where C_{safe} is given by:

$$\begin{aligned} C_{\mathrm{safe}} &::= C' \mid C[\lambda x.C'] \mid C[C'\ M] \mid C[M\ C'] \\ C' &::= \Box \mid \langle C' \mid D\rangle \quad . \end{aligned}$$

The sharing calculus adds naming, since it is no longer derivable. $\lambda\circ_{\text{share}}$ extends the cyclic calculus (λ_{need}) presented in [4, 5], since reductions may occur when they are not needed. For example, λ_{need} disallows the reduction $\langle x \mid x = \lambda y.wx, w = \lambda z.z\rangle \twoheadrightarrow \langle x \mid x = \lambda y.(\lambda z.z)x\rangle \twoheadrightarrow \langle x \mid x = \lambda y.x\rangle$. Moreover, in [4, 5], the soundness and completeness of λ_{need} with respect to traditional lambda calculus were limited to the acyclic case.

8.1 Soundness and Completeness of $\lambda\circ_{\text{share}}$ with respect to $\lambda\circ_{\text{name}}$

Soundness of $\lambda\circ_{\text{share}}$ follows from the fact that the sharing theory is a subset of the call-by-name theory. In [4, 5], we proved the completeness of (acyclic) λ_{need} with respect to lambda calculus using a simple invariant: if $M \xrightarrow[name]{}\!\!\!\!\!\twoheadrightarrow N$ then $\exists P, N', M \xrightarrow[need]{}\!\!\!\!\!\twoheadrightarrow P, N \xrightarrow[name]{}\!\!\!\!\!\twoheadrightarrow N'$ and $N' \leq P$. The ordering $\leq$ was a syntactic ordering capturing the amount of sharing in a term. To show completeness of $\lambda\circ_{\text{share}}$, this invariant is too strong. We need to compare information content. Intuitively, we want to say that if M reduces to N in $\lambda\circ_{\text{name}}$, then the information contained in N can be obtained by reducing M in $\lambda\circ_{\text{share}}$. However, this does not hold. Consider the reduction $\langle x \mid x = yy\rangle \xrightarrow[\lambda\circ_{\text{name}}]{}\!\!\!\!\!\twoheadrightarrow yy$. Since yy is stable information, we would expect to get that information in $\lambda\circ_{\text{share}}$. But this information is not reachable in $\lambda\circ_{\text{share}}$ since yy is not a value. This show that if we want to compare $\lambda\circ$ and $\lambda\circ_{\text{share}}$, we need a new notion of information content for the sharing calculus which we call *printable information*. This notion, as opposed to the call-by-name information content, can be infinite. Consider the term $\langle x \mid x = \lambda y.yx\rangle$. Its information content is Ω, whereas its printing value is the sequence $\lambda y.y\Omega$,$\lambda y.y(\lambda y.y\Omega)$, $\lambda y.y(\lambda y.y(\lambda y.y\Omega)), \cdots$. Both the information content and the printable information of $\langle x \mid x = x(\lambda y.y)\rangle$ are Ω. Let $M \xrightarrow[\text{es}]{}\!\!\!\!\!\twoheadrightarrow M_1$ stand for a sequence of external substitution steps.

Definition 38. Given $M \in \Lambda\circ$.

- The printable information of M, $\text{print}(M)$, is $\downarrow \{\omega(M_1) \mid M \xrightarrow[\text{es}]{}\!\!\!\!\!\twoheadrightarrow M_1\}$, where the downward closure is with respect to $\omega(\Lambda\circ)$ and $\leq_\Omega$.
- $M \preceq_{\text{share}} N$ if $\text{print}(M) \subseteq \text{print}(N)$.
- The infinite normal form of M, $\text{Inf}_{\text{share}}(M)$, is defined as $\bigcup\{\text{print}(M_1) \mid M \xrightarrow[\lambda\circ_{\text{share}}]{}\!\!\!\!\!\twoheadrightarrow M_1\}$.

Our invariant becomes: each *finite information* obtained by reducing a term M in $\lambda\circ_{\text{name}}$ can be obtained by reducing M in $\lambda\circ_{\text{share}}$ and then printing the result.

Lemma 39. *Given $M \in \Lambda\circ$. If $M \xrightarrow[\lambda\circ_{\text{name}}]{}\!\!\!\!\!\twoheadrightarrow N$ then there exists a term P such that $M \xrightarrow[\lambda\circ_{\text{share}}]{}\!\!\!\!\!\twoheadrightarrow P$ and $\omega(N) \in \text{print}(P)$.*

Theorem 40. *Given $M \in \Lambda\circ$. $\text{Inf}_{\lambda\circ}(M) = \text{Inf}_{\text{share}}(M)$.*

$\lambda\circ_{\text{share}}$ captures the essence of lazy languages, such as Haskell [19], and of the functional core of lenient languages, such as Id [32, 11] and Parallel Haskell [10].

This is substantiated by the fact that both the lazy and lenient strategies are complete with respect to different observations. The lazy strategy, as described by Launchbury [25], only allows one to reach the top stable information. The lenient strategy allows reduction of any redex, as long as it does not occur under a lambda. Thus, it allows one to reach more information. For example, given $x(I(\lambda z.\Omega))$, the lazy strategy produces $x\Omega$ and the lenient strategy produces $x(\lambda z.\Omega)$.

Remark. Note that even though $\langle(\Lambda\circ, \xrightarrow[\lambda\circ_{\text{share}}]{}, \preceq_{\text{share}}), \text{print}, (\text{print}(\Lambda\circ), \subseteq)\rangle$ is an ARS with ordered information content, $\xrightarrow[\lambda\circ_{\text{share}}]{}$ is not confluent up to $\preceq_{\text{share}}$. Let $M \equiv \langle x \mid x = \lambda z.z\ y, y = \lambda z'.z'\ (x\ z')\rangle$. We then have $M \xrightarrow[\lambda\circ_{\text{share}}]{}\!\!\!\!\rightarrow \langle \lambda z.z\ y \mid y = \lambda z'.z'\ (z'\ y)\rangle \equiv M_1$ and $M \xrightarrow[\lambda\circ_{\text{share}}]{}\!\!\!\!\rightarrow \langle x \mid x = \lambda z.z\ (\lambda z'.z'\ (x\ z'))\rangle \equiv M_2$. However, there cannot exist M_3 such that $M_2 \xrightarrow[\lambda\circ_{\text{share}}]{}\!\!\!\!\rightarrow M_3$ and $\text{print}(M_1) \subseteq \text{print}(M_3)$ because $\text{print}(M_1)$ is infinite while the print of any reduct of M_2 is finite. The problem is that in the unwinding of M we have an infinite number of β-redexes. When we rewrite M into M_1 we do all of those redexes and when we rewrite M into M_2 we destroy the opportunity to do them in one step. Note this does not contradict Proposition 23, since the sharing infinite normal form is not defined as in Definition 21.

9 The Cyclic Call-by-Value Calculus $\lambda\circ_{\text{value}}$

Both the call-by-name cyclic calculus and the sharing calculus can be used to reason about non-strict functional languages. However, they are unsuitable to reason about strict functional languages such as SML [17]. Therefore, we develop another variant of the cyclic calculus, namely a cyclic call-by-value calculus. This calculus is derived from the sharing calculus by restricting the notion of value declaration to the set $VD_V \subset VD$, such that $VD_V \neq \{x_1 = x_2, \cdots x_n = x_1, VD\}$. The garbage collection and the lambda lift axioms are then restricted to work with VD_V instead of VD.

Definition 41. The axioms of the cyclic call-by-value calculus ($\lambda\circ_{\text{value}}$) are given in Table 6. The rewrite relation $\xrightarrow[\lambda\circ_{\text{value}}]{}$ associated to the calculus is derived in the same way as in Definition 37.

With respect to the sharing of computations, the sharing and call-by-value calculi are the same. This points out that call-by-value, call-by-need and lenient implementations support the same amount of sharing, *i.e.,* the argument of a function is not copied before it is reduced to a value. The difference between the two calculi is that in call-by-value the equations D of a term $\langle M \mid D\rangle$ do not only represent sharing, they also tell us that if one of the terms in D does not produce a value, the complete term should not either. Consider the term $\langle \lambda x.x \mid y = \Omega\rangle$. Its answer according to the sharing calculus is $\lambda x.x$. According to the call-by-value calculus, it must be Ω. This means that in call-by-value we have to be careful in eliminating inaccessible equations. In the acyclic case, a similar point

Table 6. The axioms of $\lambda\circ_{\text{value}}$

$\beta\circ$:	$(\lambda x.M)N = \langle M \mid x = N\rangle$	
Substitution:	$\langle C[x] \mid x = V, D\rangle = \langle C[V] \mid x = V, D\rangle$	
	$\langle M \mid x = C[x_1], x_1 = V, D\rangle = \langle M \mid x = C[V], x_1 = V, D\rangle$	
Lift:	$\langle M \mid D\rangle N = \langle MN \mid D\rangle$	
	$M\langle N \mid D\rangle = \langle MN \mid D\rangle$	
	$\lambda x.\langle M \mid D, VD_V\rangle = \langle\langle \lambda x.\langle M \mid D\rangle \mid VD_V\rangle$	$D \perp VD_V$ and x not free in VD_V
Merge:	$\langle M \mid x = \langle N \mid D\rangle, D_1\rangle = \langle M \mid x = N, D, D_1\rangle$	
	$\langle\langle M \mid D\rangle \mid D'\rangle = \langle M \mid D, D'\rangle$	
Garbage collection:	$\langle M \mid D, VD_V\rangle = \langle M \mid D\rangle$	$VD_V \perp \langle M \mid D\rangle$
	$\langle M \mid \rangle = M$	
Copying:	$M = N$	$\exists\sigma : \mathcal{V} \to \mathcal{V}, N^\sigma \equiv M$ and $\forall x \not\equiv x', \sigma(x) \equiv \sigma(x') : \sigma(x)$ bound to a value in M
Naming:	$M = \langle x \mid x = M\rangle$	x a new variable

was made by Maraist *et al.* [29]. In $\langle \lambda x.x \mid y = z\rangle$ and $\langle \lambda x.x \mid y = \lambda z.z\rangle$ it is safe to eliminate the binding for y, and instead in $\langle \lambda x.x \mid y = y\rangle$ and $\langle x \mid y = wz\rangle$ it is not. The proviso on the garbage collection axiom guarantees that these equations are not removed. A similar restriction is imposed on the lambda lift axiom. This is to guarantee that $\langle y \mid y = \lambda z.\langle z \mid x = x\rangle\rangle \neq \langle y \mid y = \lambda z.z, x = x\rangle$, since the first term evaluates to $\lambda z.\Omega$ while the second one evaluates to Ω.

9.1 Basic Properties of the Cyclic Call-by-Value Lambda Calculus

To compute the call-by-value information content we send $\beta\circ$-redexes to Ω. However, unlike the call-by-name calculus we do not send to Ω the substitution redexes and then remove all the inaccessible equations. This would introduce a non-confluence problem. That is, if we let N be $\langle \lambda x.y \mid y = z, z = \lambda w.w\rangle$, we then have the ω-reductions: $N \to \langle \lambda x.y \mid y = \Omega\rangle \to \Omega$ and $N \to \langle \lambda x.\Omega \mid z = \lambda w.w\rangle \to \lambda x.\Omega$. Instead, if a variable x is bound to a variable y or to a lambda-abstraction $\lambda y.P$, then each occurrence of x is replaced with y or $\lambda y.\Omega$, respectively, and then the binding gets removed. The information content of N thus becomes $\lambda x.\lambda w.\Omega$. Bindings of the form $x_1 = x_2\ M_1, \cdots x_n = x_1\ M_n$ are treated in the same way as bindings of the form $x = x$, that is, they cause the entire term to be sent to Ω. In addition, we must be able to handle terms such as $\langle \lambda x.x \mid z = \lambda x.x\rangle\ y$ and $\langle x \mid x = \langle \lambda x.x \mid z = \lambda x.x\rangle\rangle$, which contain an obstructed $\beta\circ$ and value substitution redex, respectively. Thus, we need to add the left lift and internal merge rules. Moreover, we want to equate terms such as $x\ x\ x$ and $\langle y\ x \mid y = x\ x\rangle$ because they represent the same graph. The solution is to first normalize a term with respect to a suitable subset of $\xrightarrow[\lambda\circ_{\text{value}}]{}$. We call this subset the kernelizing system ($\mathcal{K}$). $\mathcal{K}$ is given in Table 7. We let $\mathcal{K}(M)$ be the normal form of M with respect to $\mathcal{K}$. The call-by-value ω-rules are given in Table 8.

Definition 42. Given $M \in \Lambda\circ$. The call-by-value information content of M is given by the function ω_{value}, which given M returns the normal form of M with

Table 7. The kernelizing system $\mathcal{K}$

$$\begin{aligned}
\langle M \mid D\rangle N &\doteq \langle M\,N \mid D\rangle \\
M\langle N \mid D\rangle &\doteq \langle M\,N \mid D\rangle \\
\langle M \mid x = \langle N \mid D\rangle, D_1\rangle &\doteq \langle M \mid x = N, D, D_1\rangle \\
\langle\langle M \mid D\rangle \mid D'\rangle &\doteq \langle M \mid D, D'\rangle \\
\langle M \mid\rangle &\doteq M \\
C_{\text{safe}}[M\ N] &\doteq C_{\text{safe}}[\langle x \mid x = M\ N\rangle]\ x \text{ a new variable}
\end{aligned}$$

respect to the call-by-value ω-rules and system $\mathcal{K}$. We define $M \preceq_{\text{value}} N$ if $\omega_{\text{value}}(M) \leq_\Omega \omega_{\text{value}}(N)$.

We can then show that $\langle(\Lambda\circ, \xrightarrow[\lambda\circ_{\text{value}}]{}, \preceq_{\text{value}}), \omega_{\text{value}}, (\omega_{\text{value}}(\Lambda\circ), \leq_\Omega)\rangle$ is an ARS with ordered information content. We do not prove confluence up to $\preceq_{\text{value}}$ directly, but instead we introduce a kernelized reduction ($\xrightarrow[\lambda\mathcal{K}]{}$) and prove confluence up to $\preceq_{\text{value}}$ for that reduction relation. $\xrightarrow[\lambda\mathcal{K}]{}$ is defined for $\mathcal{K}$-normal forms as: $M \xrightarrow[\lambda\mathcal{K}]{} N$ if $M \xrightarrow[\lambda\circ_{\text{value}}]{} P$ and $\mathcal{K}(P) \equiv N$. Confluence up to of $\lambda\circ_{\text{value}}$ then follows from the facts: (i) $\xrightarrow[\lambda\mathcal{K}]{} \subseteq \xrightarrow[\lambda\circ_{\text{value}}]{}{}^{\!\!*}$ and (ii) if $M \xrightarrow[\lambda\circ_{\text{value}}]{}{}^{\!\!*} N$ then $\mathcal{K}(M) \xrightarrow[\lambda\mathcal{K}]{}{}^{\!\!*} \mathcal{K}(N)$. The infinite normal form induced by this notion of information, denoted by $\text{Inf}_{\text{value}}$, also leads to a model.

9.2 The Cyclic Call-by-Value Calculus, Moggi's Computational Lambda Calculus and Hasegawa's Calculus

We use the infinite call-by-value normal form to relate our calculus to the commutative version of the computational lambda calculus of Moggi (λ_c) [30]. However, since Moggi's calculus is acyclic, we first need to relate a cyclic term to the acyclic terms approximating it, as we did for the call-by-name calculus. However, this relation only works for a subset of the set of terms. For example, the answer of $\langle x \mid x = yx\rangle$ is the term itself. Instead, the answer of any of its approximations is Ω. If we consider only internal merge normal forms then the restriction is that if several declarations are mutually recursive then all those declarations must only involve values. For example, $\langle y \mid y = x, x = \lambda z.y\rangle$ and $\langle y \mid y = \lambda z.x, x = \lambda z.y\rangle$ are good terms, but $\langle x \mid x = \lambda y.z, z = x\ x\rangle$ is not. An arbitrary term is good if its internal merge normal form is good. We denote the set of good terms by $\Lambda\circ_{\text{value}}$.

To define call-by-value expansions, we introduce the notation $M \xrightarrow[\text{GK(as)}]{}{}^{\!\!*n} N$ which denotes n-steps of the Gross-Knuth strategy applied to acyclic value substitution redexes occurring in M. The notion of acyclic substitution redex is taken from [6]. An acyclic value substitution redex is any value substitution redex that is not of the form $\langle M \mid x = C[y], y = V, D\rangle$, where x and y are mutually recursive. In $\langle \underline{x} \mid x = \lambda z.\underline{y}, y = \lambda z.\underline{w}, w = \lambda z.y\rangle$, the underlined x and y are acyclic value substitution redexes, and the underlined w is not. Since a

Table 8. The call-by-value ω-rules

$$
\begin{array}{rll}
(\lambda x.M)N & \doteq \Omega & \\
\langle M \mid x = y, D\rangle & \doteq \langle M[x := y] \mid D[x := y]\rangle & x \not\equiv y,\ D \neq \{\} \\
\langle M \mid x = y\rangle & \doteq M[x := y] & x \not\equiv y \\
\langle M \mid x = x, D\rangle & \doteq \Omega & \\
\langle M \mid x_1 = x_2\ M_1, \cdots, x_n = x_1\ M_n, D\rangle & \doteq \Omega & \\
\langle M \mid x = \lambda y.N, D\rangle & \doteq \langle M \mid D\rangle[x := \lambda y.\Omega] & D \neq \{\} \\
\langle M \mid x = \lambda y.N\rangle & \doteq M[x := \lambda y.\Omega] & \\
\Omega M & \doteq \Omega & \\
M\Omega & \doteq \Omega & \\
\langle \Omega \mid D\rangle & \doteq \Omega & \\
\langle M \mid x = \Omega, D\rangle & \doteq \Omega &
\end{array}
$$

value substitution redex can be obstructed by an environment, we first compute the internal merge normal form of a term M denoted by $\mathrm{nf}_{\mathrm{im}}(M)$.

Definition 43. Given $M \in \Lambda\circ_{\mathrm{value}}$. The n^{th} call-by-value expansion of M, written as M^n_V, is the term $\mathrm{strip}_{\mathrm{V}}(M^V_n)$ such that $\mathrm{nf}_{\mathrm{im}}(M) \xrightarrow[\mathrm{GK(as)}]{}{}^{n} M^V_n$ and $\mathrm{strip}_{\mathrm{V}}(N)$ is the normal form of N with respect to the rules:

$$
\begin{array}{rll}
\langle M \mid x_1 = x_2, \cdots, x_n = x_1, D\rangle & \rightarrow \langle M \mid x_1 = \Omega, \cdots, x_n = \Omega, D\rangle & \\
\langle M \mid x = \lambda y.N, D\rangle & \rightarrow \langle M \mid x = \lambda y.\Omega, D\rangle & N \not\equiv \Omega
\end{array}
$$

Theorem 44. *Given* $M \in \Lambda\circ_{\mathrm{value}}$. $\mathrm{Inf}_{\mathrm{value}}(M) = \bigcup\{\mathrm{Inf}_{\mathrm{value}}(M^i_V) \mid i \geq 0\}$.

Let $[\![M]\!]$ denote a translation of term M into a single-equation term, *i.e.,* each $\langle N \mid D\rangle$ in $[\![M]\!]$ is of the form $\langle N \mid x_1 = N_1\rangle$. Let $\lambda_c\backslash\eta_V$ denote λ_c without the η_V-axiom (*i.e.,* $\lambda x.Vx = V$ if x does not occur free in V).

Theorem 45.

- *Given* $M, N \in \Lambda_c$. *If* $\lambda_c\backslash\eta_V \vdash M = N$ *then* $\lambda\circ_{\mathrm{value}} \vdash M = N$.
- *Given acyclic terms* $M, N \in \Lambda\circ_{\mathrm{value}}$. *If* $\lambda\circ_{\mathrm{value}} \vdash M = N$ *then* $\lambda_c\backslash\eta_V \vdash [\![M]\!] = [\![N]\!]$.

Recently, Hasegawa [18] proposed a cyclic extension of Moggi's calculus. There are three major differences between Hasegawa's calculus and our own: (i) Hasegawa uses simply typed terms and treats values differently depending on them being cyclic or acyclic, (ii) he does not have lifting of values out of a lambda as an axiom [3], and (iii) he restricts garbage collection to acyclic values only. There is one minor difference and that is that environments are always non-empty. Unfortunately, Hasegawa does not study the rewriting aspects of the calculus. But in a sense, his calculus is complete with respect to our calculus:

[3] For acyclic values the axiom is derivable from substitution and garbage collection. The case for cyclic values is not derivable.

Theorem 46. *Given $M \in \Lambda\circ$, such that there is no subterm of the form $\langle N \mid\rangle$. Then* $\mathrm{Inf}_{\mathrm{value}}(M) =\downarrow \{\omega_{\mathrm{value}}(N) \mid \mathrm{Hasegawa} \setminus \eta_0 \vdash M = N\}$.

Remark. The η-axioms present in Moggi's and Hasegawa's calculi are not sound in our model, *e.g.*, $\lambda x.y\ x$ and y, $\langle M \mid x = x\rangle$ and $\langle M \mid x = \lambda z.x\ z\rangle$ do not have the same infinite normal forms.

10 Conclusions

We have developed a precise connection between the class of well-formed cyclic lambda-graphs and the terms of the lambda calculus extended with `letrec`. On the set of cyclic terms we have developed three cyclic lambda calculi. These calculi correspond to the parameter-passing techniques of call-by-name, call-by-need and call-by-value. The ability to define mutually recursive objects makes these calculi more suitable then lambda calculus [13], λ_{need} [5, 4] and λ_{V} [34] to express the operational semantics, compilation and optimization of current functional languages. The sharing calculus is the kernel language of Haskell and the functional core of Id and Parallel Haskell. The call-by-value calculus is the functional kernel of languages such as SML. In [3] we have also shown how to extend our calculi with data-structures.

What distinguishes our calculi from current theories is that we do not impose any restrictions on where the rewriting takes place. This makes our theories useful for reasoning about not only run-time issues but also about compilation issues. The development of these calculi is non-trivial due to the loss of the confluence property. Our calculi satisfy an approximate notion of confluence, which guarantees uniqueness of infinite normal forms. For each calculus, the infinite normal form provides a term model which allows us to relate our calculi to existing ones.

Acknowledgements

The research of the first author has been supported by NSF grants CCR-9410237 and CCR-9624711. The research of the second author has been supported by NSF grant CCR-9624711 and by SIR-grants from NWO. The second author thanks the University of Oregon to make his visits possible.

We thank Femke van Raamsdonk, Vincent van Oostrom, Jan Willem Klop, Amr Sabry, Miley Semmelroth, Mariangiola Dezani-Ciancaglini and Arvind for stimulating discussions about a draft of this paper.

References

1. M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy. Explicit substitutions. *Journal of Functional Programming*, 4(1):375–416, 1991.
2. Z. M. Ariola. Relating graph and term rewriting via Böhm models. *Applicable Algebra in Engineering, Communication and Computing*, 7(5), 1996.

3. Z. M. Ariola and S. Blom. Lambda calculi plus letrec. Technical Report CIS-TR-97-05, Department of computer and information science, University of Oregon. ftp://ftp.cs.uoregon.edu/pub/ariola/cyclic-calculi.ps.
4. Z. M. Ariola and M. Felleisen. The call-by-need lambda calculus. *Journal of Functional Programming*, 7(3), 1997.
5. Z. M. Ariola, M. Felleisen, J. Maraist, M. Odersky, and P. Wadler. The call-by-need lambda calculus. In *Proc. ACM Conference on Principles of Programming Languages*, pages 233–246, 1995.
6. Z. M. Ariola and J. W. Klop. Lambda calculus with explicit recursion. Technical Report CIS-TR-96-04, Department of computer and information science, University of Oregon. To appear in Information and computation.
7. Z. M. Ariola and J. W. Klop. Cyclic lambda graph rewriting. In *Proc. Ninth Symposium on Logic in Computer Science (LICS'94), Paris, France*, pages 416–425, 1994.
8. Z. M. Ariola and J. W. Klop. Equational term graph rewriting. *Fundamentae Informaticae*, 26(3,4):207–240, 1996. Extended version: CWI Report CS-R9552.
9. Z. M. Ariola, J. W. Klop, J. R. Kennaway, F. J. de Vries, and M. R. Sleep. Syntactic definitions of undefined: On defining the undefined. In *Proc. TACS 94, Sendai, Japan*, 1994.
10. Arvind, L. Augusston, J. Hicks, R. S. Nikhil, S. Peyton-Jones, J. Stoy, and W. Williams. pH: A Parallel Haskell. Technical report, MIT Laboratory for Computer Science, September 1993.
11. Arvind, J-W. Maessen, R.S. Nikhil, and J. E. Stoy. λ_s: an implicitly parallel λ-calculus with letrec, synchronization and side-effects. Technical Report 393, MIT Laboratory for Computer Science, 1997.
12. A. Asperti and C. Laneve. Interaction systems I: The theory of optimal reductions. *Mathematical structures for computer science*, 4:457–504, 1994.
13. H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics.* North-Holland, Amsterdam, 1984.
14. H. P. Barendregt, M. C. J. D. van Eekelen, J. R. W. Glauert, J. R. Kennaway, M. J. Plasmeijer, and M. R. Sleep. Term graph rewriting. In J. W. de Bakker, A. J. Nijman, and P. C. Treleaven, editors, *Proc. Conference on Parallel Architecture and Languages Europe (PARLE '87), Eindhoven, The Netherlands, Springer-Verlag LNCS 259*, pages 141–158, 1987.
15. Z. Benaissa, P. Lescanne, and K.H. Rose. Modeling sharing and recursion for weak reduction strategies using explicit substitution. In *PLIP'96*, 1996.
16. S. Billot and B. Lang. The structure of shared forests in ambiguous parsing. In *Proceedings of the Twenty-Seventh Annual Meeting of the Association for Computational Linguistics*, 1989. Association for Computational Linguistics.
17. B. Harper. Introduction to Standard ML. Technical report, ECS-LFCS-86-14, Laboratory for the Foundation of Computer Science, Edinburgh University, 1986.
18. M. Hasegawa. Recursion from cyclic sharing: traced monoidal categories and models of cyclic lambda calculi. In *Proc. Conference on Typed Lambda Calculi and Applications*, April 1997.
19. P. Hudak, S. Peyton Jones, P. Wadler, B. Boutel, J. Fairbairn, J. Fasel, K. Hammond, J. Hughes, T. Johnsson, D. Kieburtz, R. Nikhil, W. Partain, and J. Peterson. Report on the programming language Haskell. *ACM SIGPLAN Notices*, 27(5):1–64, 1992.
20. G. Huet. Confluent reductions: Abstract properties and applications to term rewriting systems. *JACM*, 27(4), 1980.

21. J. R. Kennaway, J. W. Klop, M. R. Sleep, and F. J. de Vries. Infinitary lambda calculus. In *Proc. Rewriting Techniques and Applications, Kaiserslautern*, 1995.
22. P. Klint. A meta-environment for generating programming environments. In *Algebraic Methods II: Theory, Tools and Applications. Springer-Verlag LNCS 490*, pages 105–124, 1991.
23. P. Klint and E. Visser. Using filters for the disambiguation of context-free grammars. In G. Pighizzini and P. San Pietro, editors, *Proc. ASMICS Workshop on Parsing Theory*, pages 1–20, Milano, Italy, October 1994. Tech. Rep. 126–1994, Dipartimento di Scienze dell'Informazione, Università di Milano.
24. Y. Lafont. Interaction nets. In *Proc. ACM Conference on Principles of Programming Languages, San Francisco*, 1990.
25. J. Launchbury. A natural semantics for lazy evaluation. In *Proc. ACM Conference on Principles of Programming Languages*, pages 144–154, 1993.
26. J.-J. Lévy. An algebraic interpretation of the $\lambda\beta\kappa$-calculus and an application of a labelled λ-calculus. *Theoretical Computer Science*, 2(1):97–114, 1976.
27. J.-J. Lévy. *Réductions Correctes et Optimales dans le Lambda-Calcul.* PhD thesis, Universite Paris VII, October 1978.
28. I.C. Mackie. *The geometry of implementation.* PhD thesis, University of London, 1994.
29. J. Maraist, M. Odersky, D. Turner, and P. Wadler. Call-by-name, call-by-value, call-by-need, and the linear lambda calculus. In *Proc. of Mathematical Foundations of Programming Semantics (MFPS)*, 1995.
30. E. Moggi. Computational lambda calculus and monads. Technical Report ECS-LFCS-88-86, Edinburgh University, 1988.
31. J. Niehren. Functional computation as concurrent computation. In *Proc. ACM Conference on Principles of Programming Languages*, pages 333–343, 1996.
32. R. S. Nikhil. Id (version 90.1) reference manual. Technical Report 284-2, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, 1991.
33. S. L. Peyton Jones. *The implementation of Functional Programming Languages.* Prentice-Hall International, Englewood Cliffs, N.J., 1987.
34. G. D. Plotkin. Call-by-name, call-by-value and the lambda calculus. *Theoretical Computer Science*, 1:125–159, 1975.
35. K. H. Rose. Explicit cyclic substitutions. In M. Rusinowitch and J. L. Rémy, editors, *Proc. 3rd International Workshop on Conditional Term Rewriting Systems (CTRS-92), Pont-à-Mousson, France, Springer-Verlag LNCS 656*, pages 36–50, 1992.
36. D. Sangiorgi and R. Milner. Techniques of "weak bisimulation up to". Technical report, 1993.
37. P. Selinger. Order-incompleteness and finite lambda models. In *Proc. Symposium on Logic in Computer Science (LICS'96)*, 1996.
38. M. Tomita. *Efficient Parsing for Natural Languages. A Fast Algorithm for Practical Systems.* Kluwer Academic Publishers, 1985.
39. C. Wadsworth. *Semantics and Pragmatics of the Lambda-Calculus.* 1971. PhD thesis, University of Oxford.

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