A rewriting calculus for cyclic higher-order term graphs

C. Bertolissi [a], P. Baldan [b], H. Cirstea [a], C. Kirchner [a]

[a] *LORIA INRIA INPL NANCY II*
54506 Vandoeuvre-lès-Nancy BP 239 Cedex France

[b] *Dipartimento di Informatica, Università Ca' Foscari di Venezia, Italy*

Abstract

Introduced at the end of the nineties, the Rewriting Calculus (ρ-calculus, for short) is a simple calculus that fully integrates term-rewriting and λ-calculus. The rewrite rules, acting as elaborated abstractions, their application and the obtained structured results are first class objects of the calculus. The evaluation mechanism, generalizing beta-reduction, strongly relies on term matching in various theories.

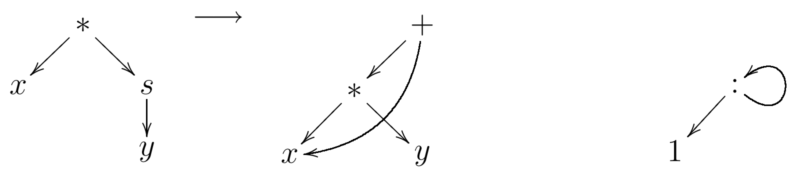
In this paper we propose an extension of the ρ-calculus, handling graph like structures rather than simple terms. The transformations are performed by explicit application of rewrite rules as first class entities. The possibility of expressing sharing and cycles allows one to represent and compute over regular infinite entities.

The calculus over terms is naturally generalized by using unification constraints in addition to the standard ρ-calculus matching constraints. This therefore provides us with the basics for a natural extension of an explicit substitution calculus to term graphs. Several examples illustrating the introduced concepts are given.

Introduction

Main interests for term rewriting steams from functional and rewrite based languages as well as from theorem proving. In particular, we can describe the behaviour of a functional or rewrite based program by analyzing some properties of the associated term rewrite system. In this framework, terms are often seen as trees but in order to improve the efficiency of the implementation of such languages, it is of fundamental interest to think and implement terms as graphs [BvEG+87]. In this case, the possibility of sharing subterms saves space (by avoiding the duplication of the subterm by means of multiple pointers to the same subterm) and saves time (a redex appearing in that subterm will be reduced at most once and equality tests can be done in constant time when the sharing is maximal). We can take as example the definition of multiplication in a rewrite system $\mathcal{R} = \{x * 0 \to 0, \quad x * s(y) \to (x * y) + x\}$. If

This is a preliminary version. The final version will be published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs



(a) Multiplication using a graph rewrite rule (b) Infinite list of ones

Fig. 1. Examples of term graphs

we represent it using graphs, we will write the second rule by duplicating the reference to the x instead of duplicating the x itself (see Figure 1).

Graph rewriting is a useful technique for the optimization of functional and declarative languages implementation [PJ87]. Moreover, the possibility to define cycles leads to an increased expressive power that allows one to represent easily regular infinite data structures. For example, the circular list $ones = 1 : ones$, where ":" denotes the concatenation operator, can be represented by the cyclic graph of Figure 1. Cyclic term graph rewriting has been widely studied, both from an operational [BvEG$^+$87,HKP91,AK96] and from a categorical/logical point of view [CG99] (see [SPvE93] for a survey on term graph rewriting).

In this context, an abstract model generalizing λ-calculus and adding cycles and sharing features has been proposed by Z. M. Ariola and J. W. Klop [AK97]. Their approach consists of an equational framework that models λ-calculus extended with explicit recursion. A λ-graph is treated as a system of recursion equations involving λ-terms and rewriting is described as a sequence of equational transformations. This work allows for the combination of graph structures with the higher-order capabilities of λ-calculus. A last important ingredient is still missing: pattern matching. This possibility to discriminate using pattern matching could be encoded, in particular in λ-calculus, but it is much more attractive to directly discriminate and to use indeed rewriting. Programs become quite compact and the encoding of data type structures is no longer necessary.

The rewriting calculus (ρ-calculus, for short) has been introduced in the late nineties as a natural generalization of term rewriting and of the λ-calculus [CK01]. It has been shown to be a very expressive framework e.g. to express object calculi [CKL01] and it can be equipped with powerful type systems [BCKL03]. One essential component of the ρ-calculus are the matching constraints that are generated by the generalization of the β-reduction called ρ-reduction. By making this matching step explicit and the matching constraints first class objects of the calculus, we can allow for an explicit handling of constraints instead of substitutions [CFK04].

The first contribution of this paper consists of a new system, called the ρ_{g}-calculus, that generalizes cyclic λ-calculus as the standard ρ-calculus generalizes the classical λ-calculus. The ρ_{g}-calculus deals with cyclic terms with bound variables and can express vertical sharing as well as horizontal sharing by means of a list of recursion equations. In the ρ_{g}-calculus computations related to the matching are made explicit and performed at the object-level.

We then show that the ρ_{g}-calculus can simulate the ordinary ρ-calculus. For doing this, we prove that the matching in the ρ_{g}-calculus behaves well *w.r.t.* the matching algorithm of the ρ-calculus and that for any ρ-reduction there exists a corresponding reduction in the ρ_{g}-calculus. We also show that ρ_{g}-calculus is a natural extension of the cyclic λ-calculus by proving that cyclic λ-terms can be translated into the ρ_{g}-calculus and that cyclic λ-reductions can be simulated in our system. We therefore get a common generalization of the cyclic λ-calculus and the ρ-calculus, providing a framework where matching, graph like structures and higher-order capabilities are primitive.

The paper is organized as follows. In the first section we briefly review the two systems which inspired our new calculus: the standard ρ-calculus [CK01] and the cyclic λ-calculus [AK97]. Section 2 and Section 3 describe respectively the syntax and the small-step semantics of the ρ_{g}-calculus giving some examples of terms and term reductions in the system. In Section 4 we show that the ρ_{g}-calculus is a generalization of the ρ-calculus and we show how cyclic λ-reductions can be simulated in ρ_{g}-calculus. We conclude in Section 5 by presenting some perspectives of future work.

1 Rewriting calculus and cyclic λ-calculus

We briefly present here the two formalisms that inspired the calculus introduced in this paper.

1.1 The rewriting calculus

The ρ-calculus was introduced to make all the basic ingredients of rewriting explicit objects, in particular the notions of rule abstraction ($\twoheadrightarrow$), rule application and set of results (;). In the ρ-calculus, the usual λ-abstraction $\lambda x.t$ is replaced by a rule abstraction $T_1 \twoheadrightarrow T_2$, where T_1 and T_2 are two arbitrary terms, and the free variables of T_1 are bound in T_2.

The set of ρ-terms is defined as follows:

$$\mathcal{T} ::= \mathcal{X} \mid \mathcal{K} \mid \mathcal{T} \twoheadrightarrow \mathcal{T} \mid [\mathcal{T} \ll \mathcal{T}]\mathcal{T} \mid \mathcal{T}\ \mathcal{T} \mid \mathcal{T} ; \mathcal{T}$$

The symbols $T, U, L, R, \ldots$ range over the set $\mathcal{T}$ of terms, the symbols $x, y, z, \ldots$ range over the set $\mathcal{X}$ of variables, the symbols $a, b, c, \ldots, f, g, h$ range over a set $\mathcal{K}$ of constants.

The small-step reduction semantics is defined by the evaluation rules presented in Figure 2. The application of a rewrite rule (abstraction) to a term is always evaluated to the application of the corresponding constraint to the

$$
\begin{array}{llll}
(\rho) & (\mathcal{T}_1 \to \mathcal{T}_2)\mathcal{T}_3 & \mapsto_\rho & [\mathcal{T}_1 \ll \mathcal{T}_3]\mathcal{T}_2 \\
(\sigma) & [\mathcal{T}_1 \ll \mathcal{T}_3]\mathcal{T}_2 & \mapsto_\sigma & \sigma_{(\mathcal{T}_1 \ll \mathcal{T}_3)}(\mathcal{T}_2) \\
(\delta) & (\mathcal{T}_1; \mathcal{T}_2)\,\mathcal{T}_3 & \mapsto_\delta & \mathcal{T}_1\,\mathcal{T}_3; \mathcal{T}_2\,\mathcal{T}_3
\end{array}
$$

Fig. 2. Small-step semantics of ρ-calculus

right-hand side of the rewrite rule. Such a construction is called a *delayed matching constraint*. The body of the constrained term will be evaluated or delayed according to the result of the corresponding matching problem. If a solution exists, the delayed matching constraint self-evaluates to $\sigma(\mathcal{T}_2)$, where σ is the solution of the matching between $\mathcal{T}_1$ and $\mathcal{T}_3$. The matching power of the ρ-calculus can be regulated using arbitrary theories. Here we consider the ρ-calculus with the empty theory (*i.e.* syntactic matching) that is decidable and has a unique solution.

Starting from these top-level rules we define, as usually, the context closure denoted $\mapsto_{\rho\sigma\delta}$. The many-step evaluation $\mapsto\!\!\!\!\to_{\rho\sigma\delta}$ is defined as the reflexive-transitive closure of $\mapsto_{\rho\sigma\delta}$.

1.2 The cyclic λ-calculus

The cyclic λ-calculus introduced by Ariola and Klop consists in an equational framework for term graph rewriting with cycles. The axiom system is an extension of λ-calculus with the `letrec` construct and λ-graphs are represented as a system of (possibly nested) recursion equations on standard λ-terms. If the system is used without restrictions on the rules, the confluence is lost. The authors restore it by controlling the operations on the recursion equations. The resulting calculus, called $\lambda\phi$ [AK97], is powerful enough to incorporate the classical λ-calculus [Bar84] and also the $\lambda\mu$-calculus [Par92] and the $\lambda\sigma$-calculus with names [ACCL91] extended with horizontal and vertical sharing respectively. The syntax of $\lambda\phi$ is the following:

$$
t ::= x \mid f(t_1, \dots t_n) \mid t_0\ t_1 \mid \lambda x.t \mid \langle t_0 \mid\ \ x_1 = t_1, \dots, x_n = t_n \rangle
$$

The set of $\lambda\phi$-terms is composed of the ordinary λ-terms (*i.e.* variables, functions of fixed arity, applications, abstractions) plus new terms built using the `letrec` construct: $\langle t_0 \mid\ \ x_1 = t_1, \dots, x_n = t_n \rangle$, where we suppose the recursion variables x_i, $i = 1, \dots, n$, all distinct. We denote by E an unordered sequence of equations $x_1 = t_1, \dots, x_n = t_n$ and by ϵ the empty sequence. Terms are denoted by the symbols $t, s, \dots$, variables are denoted by the symbols $x, y, z, \dots$ and constants by the symbols $a, b, c, \dots, f, g, h$. A context $\mathsf{Ctx}\{\_\}$ is a term with a single hole $\square$ in the place of a subterm. Filling the context $\mathsf{Ctx}\{\square\}$ with a term t yields the term $\mathsf{Ctx}\{t\}$. Variables are bound either by the lambda abstraction, or by a recursion equation. We denote by $\leq$ the least pre-order on recursion variables such that $x \geq y$ if $x = \mathsf{Ctx}\{y\}$, for some

$$
\begin{array}{llll}
(\beta) & (\lambda x.t_1)\ t_2 & \to_\beta & \langle t_1 \mid x = t_2\rangle \\
(\textit{external sub}) & \langle \mathsf{Ctx}\{y\} \mid y = t, E\rangle & \to_{es} & \langle \mathsf{Ctx}\{t\} \mid y = t, E\rangle \\
(\textit{acyclic sub}) & \langle t_1 \mid y = \mathsf{Ctx}\{x\}, x = t_2, E\rangle & \to_{ac} & \langle t_1 \mid y = \mathsf{Ctx}\{t_2\}, x = t_2, E\rangle \\
 & & & \textit{if } y > x \\
(\textit{black hole}) & \langle \mathsf{Ctx}\{x\} \mid x =_\circ x, E\rangle & \to_\bullet & \langle \mathsf{Ctx}\{\bullet\} \mid x =_\circ x, E\rangle \\
 & \langle t \mid y = \mathsf{Ctx}\{x\}, x =_\circ x, E\rangle & \to_\bullet & \langle t \mid y = \mathsf{Ctx}\{\bullet\}, x =_\circ x, E\rangle \\
 & & & \textit{if } y > x \\
(\textit{garbage collect}) & \langle t \mid E, E'\rangle & \to_{gc} & \langle t \mid E\rangle \\
 & & & \textit{if } E' \neq \epsilon \textit{ and } E \perp (E', t) \\
 & \langle t \mid \epsilon\rangle & \to_{gc} & t
\end{array}
$$

Fig. 3. Evaluation rules of the $\lambda\phi_0$-calculus

context $\mathsf{Ctx}\{\_\}$. We write $x > y$ if $x \geq y$ and $x \not\equiv y$, where $\equiv$ is the equivalence induced by the pre-order, *i.e.* $x \equiv y$ if $x \geq y \geq x$ (variables x and y occur in a cycle). We write $E \perp (E', t)$, E is orthogonal to a sequence of equations E' and a term t, if the recursion bound variables of E do not intersect with the set of free variables of E' and t. The notation $x =_\circ x$ is an abbreviation for the sequence of recursion equations $x = x_1, \ldots, x_n = x$.

The reduction rules of the basic $\lambda\phi_0$-calculus are given in Figure 3. Some extensions of this basic set of rules can be considered [AK97] by adding either box distribution rules ($\lambda\phi_1$) or box merging and elimination rules ($\lambda\phi_2$). In the following we will concentrate our attention on the basic system of Figure 3. In the β-rule, the variable x bound by λ becomes bound by a recursion equation after the reduction. The two substitution rules are used to make a copy of a graph associated to a recursion variable. The restriction on the order of recursion variables is introduced to ensure confluence in the case of cyclic configurations of lambda redexes. The proviso $y > x$ in the rules *acyclic sub* and *black hole* is necessary in order to prove the confluence of the system. The condition $E' \neq \epsilon$ in the rule *garbage collect* rule avoids trivial non-terminating reductions.

We denote by $\mapsto_{\lambda\phi}$ the rewrite relation induced by the set of rules of Figure 3 and by $\mapsto\mathrel{\mkern-14mu}\rightarrow_{\lambda\phi}$ its reflexive and transitive closure.

2 The syntax of ρ_{g}-calculus

The syntax of ρ_{g}-calculus presented in Figure 4 extends the syntax of the standard ρ-calculus and of the ρ_{x} [CFK04], the ρ-calculus with explicit matching and substitution application. The term $G_1 \twoheadrightarrow G_2$ represents a rewrite rule (*i.e.* an abstraction), where the term G_1 is called the pattern. There are two different application operators: the functional application operator is de-

Terms

$$\begin{array}{lll}
\mathcal{G} ::= & \mathcal{X} & \text{(Variables)} \\
 & \mid \mathcal{K} & \text{(Constants)} \\
 & \mid \mathcal{G} \rightarrowtriangle \mathcal{G} & \text{(Abstraction)} \\
 & \mid \mathcal{G}\ \mathcal{G} & \text{(Functional application)} \\
 & \mid \mathcal{G};\mathcal{G} & \text{(Structure)} \\
 & \mid \mathcal{G}\ [\mathcal{C}] & \text{(Constraint application)}
\end{array}$$

Constraints

$$\begin{array}{lll}
\mathcal{C} ::= & \epsilon & \text{(Empty constraint)} \\
 & \mid \mathcal{X} = \mathcal{G} & \text{(Recursion equation)} \\
 & \mid \mathcal{G} \ll \mathcal{G} & \text{(Match equation)} \\
 & \mid \mathcal{C}, \mathcal{C} & \text{(Conjunction of constraints)}
\end{array}$$

Fig. 4. Syntax of the ρ_{g}-calculus

noted simply by concatenation (and by @ in graphical presentations), while the constraint application operator is denoted by the " [ ]" operator. Terms can be grouped together into *structures* built using the operator ";" and depending on the theory behind this operator we can obtain, for example, a multi-set (for an associative-commutative operator) or a set (for an associative-commutative-idempotent operator). This operator is useful for representing the (non-deterministic) application of a set of rewrite rules and consequently, the non-deterministic results. Starting from this point of view, term rewrite systems (and underlying strategies) can be encoded in the ρ-calculus [CLW03] and we conjecture that this encoding can be extended to term graph rewrite systems in ρ_{g}-calculus.

As the ρ_{x}, the ρ_{g}-calculus deals explicitly with matching constraints of the form $\mathcal{G} \ll \mathcal{G}$ but introduces also a new kind of constraint, the recursion equations. A recursion equation is a constraint of the form $x = \mathcal{G}$ and can be seen as a delayed substitution, or as an environment associated to a term. In the ρ_{g}-calculus constraints are conjunctions (built using the operator ",") of match equations and recursion equations. The operator "," is supposed to be associative, commutative and idempotent, with ϵ as neutral element. The empty constraint is denoted by ϵ.

We assume that the application operator associates to the left, while the other operators associate to the right. To simplify the syntax operators have different priorities. Here are the operators ordered from higher to lower priority: " ", "$\rightarrowtriangle$", ";", " [ ]" , "$\ll$", "$=$" and ",".

The symbols $G, H, \ldots$ range over the set $\mathcal{G}$ of terms, $x, y, z, \ldots$ range over the set $\mathcal{X}$ of variables ($\mathcal{X} \subseteq \mathcal{G}$), $a, b, c, \ldots, f, g, h$ range over a set $\mathcal{K}$ of constants ($\mathcal{K} \subseteq \mathcal{G}$). The symbols $E, F, \ldots$ range over the set $\mathcal{C}$ of constraints. We call *algebraic* the terms of the form $(((f\ G_1)\ G_2) \ldots)\ G_n$ with $f \in \mathcal{K}$ and we usually denote them by $f(G_1, G_2, \ldots, G_n)$.

We denote by $\bullet$ (black hole) a constant, already introduced by Ariola and Klop [AK96] using the equational approach and also by Corradini [Cor93] using the categorical approach, to give a name to "undefined" terms that correspond to the expression $x\ [x = x]$ (self-loop). The notation $x =_\circ x$ is again an abbreviation for the sequence $x = x_1, \ldots, x_n = x$.

We use the symbol $\mathsf{Ctx}\{\Box\}$ for a context with exactly one hole $\Box$. We say that a ρ_{g}-term is *acyclic* if it contains no recursive sequences of constraints of the form $\mathsf{Ctx}_0\{x_0\} \lll \mathsf{Ctx}_1\{x_1\}, \mathsf{Ctx}_2\{x_1\} \lll \mathsf{Ctx}_3\{x_2\}, \ldots, \mathsf{Ctx}_m\{x_n\} \lll \mathsf{Ctx}_{m+1}\{x_0\}$, with $n, m \in \mathbb{N}$ and $\lll \in \{=, \ll\}$. This kind of sequence is called a *cycle*.

The notions of free and bound variables of ρ_{g}-terms take into account the three binders of the calculus: the abstraction, the recursion and the match. In particular, to ease the definition, we also introduce the domain of a constraint $\mathcal{C}$, denoted $\mathcal{DV}(\mathcal{C})$, as the set of variables (potentially) defined by the recursion and matching equations it contains. The set $\mathcal{DV}(\mathcal{C})$ includes, for any recursion equation $x = H$ in $\mathcal{C}$, the variable x and for any match $G \ll G'$ in $\mathcal{C}$, the set of free variables of G.

Definition 2.1 [Free, bound, and defined variables] Given a ρ_{g}-term G, its free variables, denoted $\mathcal{FV}(G)$, and its bound variables, denoted $\mathcal{BV}ar(G)$, are recursively defined below:

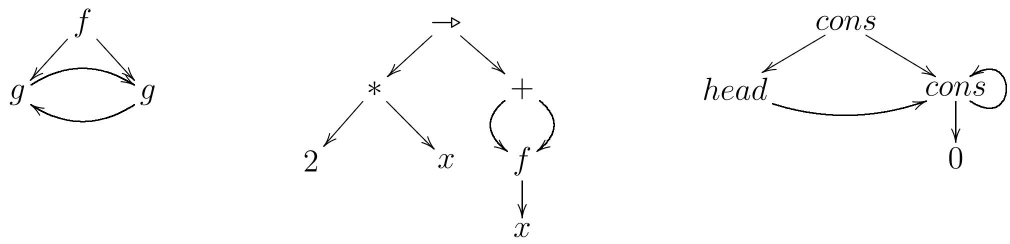
G	$\mathcal{BV}(G)$	$\mathcal{FV}(G)$
x	$\emptyset$	$\{x\}$
k	$\emptyset$	$\emptyset$
$G_1\ G_2$	$\mathcal{BV}(G_1) \cup \mathcal{BV}(G_2)$	$\mathcal{FV}(G_1) \cup \mathcal{FV}(G_2)$
$G_1; G_2$	$\mathcal{BV}(G_1) \cup \mathcal{BV}(G_2)$	$\mathcal{FV}(G_1) \cup \mathcal{FV}(G_2)$
$G_1 \twoheadrightarrow G_2$	$\mathcal{FV}(G_1) \cup \mathcal{BV}(G_1) \cup \mathcal{BV}(G_2)$	$\mathcal{FV}(G_2) \setminus \mathcal{FV}(G_1)$
$G_0\ [\mathcal{C}]$	$\mathcal{BV}(G_0) \cup \mathcal{BV}(\mathcal{C})$	$(\mathcal{FV}(G_0) \cup \mathcal{FV}(\mathcal{C})) \setminus \mathcal{DV}(\mathcal{C})$

For a given constraint C, the free variables, denoted $\mathcal{FV}(C)$, the bound variables, denoted $\mathcal{BV}ar(C)$, and the defined variables, denoted $\mathcal{DV}(C)$, are defined as follows:

C	$\mathcal{BV}(C)$	$\mathcal{FV}(C)$	$\mathcal{DV}(C)$
ϵ	$\emptyset$	$\emptyset$	$\emptyset$
$x = G_0$	$x \cup \mathcal{BV}(G_0)$	$\mathcal{FV}(G_0)$	$\{x\}$
$G_1 \ll G_2$	$\mathcal{FV}(G_1) \cup \mathcal{BV}(G_1) \cup \mathcal{BV}(G_2)$	$\mathcal{FV}(G_2)$	$\mathcal{FV}(G_1)$
C_1, C_2	$\mathcal{BV}(C_1) \cup \mathcal{BV}(C_2)$	$\mathcal{FV}(C_1) \cup \mathcal{FV}(C_2)$	$\mathcal{DV}(C_1) \cup \mathcal{DV}(C_2)$

The notion of α-conversion used in the λ-calculus can be naturally extended to deal with the terms of the ρ_{g}-calculus.

As in the cyclic λ-calculus we define an order on recursion variables, i.e., variables bound by the recursion and match equations: we denote by $\leq$ the



(i) (ii) (iii)

Fig. 5. Some ρ_{g}-terms

least pre-order on recursion variables such that $x \geq y$ if $x = \mathsf{Ctx}\{y\}$, for some context $\mathsf{Ctx}\{\_\}$. The equivalence induced by the pre-order is denoted $\equiv$ and we say that x and y are cyclically equivalent ($x \equiv y$) if $x \geq y \geq x$ (they lye on a common cycle). We write $x > y$ if $x \geq y$ and $x \not\equiv y$. As we will see later on, this order gives us the possibility to allow substitution only upwards.

In order to support the intuition, in what follows we sometimes give a graphical representation of ρ_{g}-terms not including matching constraints. This correspondence is used only informally in the paper, but it could be made precise, e.g., along the lines of the work in [Blo01] for cyclic term graphs with binders. Roughly, any term without constraints is represented as an acyclic graph in the obvious way, a constraint $G\ [x_1 = G_1, \ldots, x_n = G_n]$ is read as a letrec construct $\texttt{letrec}\ x_1 = G_1, \ldots, x_n = G_n\ \texttt{in}\ G$ and represented through a cyclic structure. Here the correspondence between a variable in the right-hand side of a rule and its binding occurrence in the pattern is represented by keeping the variable names (instead of using backpointers). This correspondence does not extend straightforwardly to general ρ_{g}-terms, possibly including matching constraints, for which a suitable graphical representation is still under investigation.

Example 2.2 [Some ρ_{g}-terms] For a graphical representation of the terms see Figure 5.

(i) The ρ_{g}-term $f(x, y)\ [x = g(y), y = g(x)]$ is an example of twisted sharing that can be expressed using the `letrec` construct. We have that $x \geq y$ and $y \geq x$, so $x \equiv y$.

(ii) The sharing in the right-hand side of rule $(2 * x) \twoheadrightarrow ((y + y)\ [y = f(x)])$ avoids the copying of the object instantiating $f(x)$, when the rule is applied to a ρ_{g}-term.

(iii) The ρ_{g}-term $cons(head(x), x)\ [x = cons(0, x)]$ represents an infinite list of zeros. Notice that we do not need duplication for the argument of *head* since there is no need to create a new node *cons* which will have the same successors as the existing one.

As usually, we work modulo *α-conversion* (such that different bound variables have different names) and we use Barendregt's *"hygiene-convention"*, *i.e.* free and bound variables have different names [Bar84]. We point out that the set of bound variables in the subterm G of a constraint application $G\ [E]$ is the domain of E plus the bound variables of G. For example, the term $x\ [x \ll a, x \ll b]$ is equivalent modulo α-conversion to the term $y\ [y \ll a, y \ll b]$. Note also that the visibility of a recursion variable is limited to the ρ_{g}-term appearing in the list of constraints where the recursion variable is defined and the ρ_{g}-term to which this list is applied. For example, in the term $f(x, y)\ [x = g(y)\ [y = a]]$ the variable y defined in the recursion equation bounds its occurrence in $g(y)$ but not in $f(x, y)$. In fact, the term does not satisfy the naming conditions since y occurs both free and bound.

This naming conventions allows us to disregard some terms (see the examples below) and thus to apply replacements (like for the evaluation rules in Figure 6) quite straightforwardly, since no variable capture is possible.

Besides the naming conventions, some structural properties are required for a ρ_{g}-term to be well-formed.

Definition 2.3 [Well-formed terms] A ρ_{g}-term is *well-formed* if

- each variable occur at most once as left hand side of a recursion equation;
- the left hand side of abstractions and match equations are linear and acyclic, and all their subterms not containing constraints are algebraic.

For instance, the ρ_{g}-term $(f(y)\ [y = g(y)] \twoheadrightarrow a)$ is not well-formed since the abstraction has a cyclic left-hand side. All the ρ_{g}-terms considered in the sequel will be implicitly well-formed, unless stated otherwise.

Example 2.4 [Free and bound variables should not have the same name]
The reduction of the ρ_{g}-term $z\ [z = x \twoheadrightarrow y, y = x + x]$ (by instantiating the variable y) can lead to a variable capture. However this term does not respect our naming conventions: the variable capture is no longer possible if we consider the legal ρ_{g}-term $z\ [z = x_1 \twoheadrightarrow y, y = x + x]$ obtained after α-conversion. In order to have the occurrences of the variable x appearing in the second constraint bounded by the arrow, we should use a nested constraint as in the ρ_{g}-term $z\ [z = x \twoheadrightarrow (y\ [y = x + x])]$.

Example 2.5 [Different bound variables should have different names]
Intuitively, by the notions of free and bound variable, in a term there cannot be any sharing between the left hand side of rewrite rules and the rest of a ρ_{g}-term. In other words, the left hand side of a rewrite rule is self-contained. Sharing inside the left hand side is allowed. No restrictions are imposed on the right hand side. For example, in the ρ_{g}-term $f(y, y \twoheadrightarrow g(y))\ [y = x]$ the first occurrence of y is bound by the recursion variable, while the scope of the y in the abstraction $\twoheadrightarrow$ is limited to the right-hand side of the abstraction itself. The ρ_{g}-term should be in fact written (by α-conversion) as $f(y, z \twoheadrightarrow g(z))\ [y = x]$.

3 The small-step semantics of ρ_{g}-calculus

In the classical ρ-calculus, when reducing the application of a constraint to a term, *i.e.*, a delayed matching constraint, the corresponding matching problem is solved and resulted substitutions are applied at the meta-level of the calculus. In the $\rho_{\mathtt{x}}$, this reduction is decomposed into two steps, one computing the substitution and the other one describing the application of this substitution. Matching computations leading from constraints to substitutions and the application of the substitution are clearly separated and made explicit. In the ρ_{g}-calculus, the computation of the substitution from the matching constraint is performed explicitly and, if the computation is successful, the result is a recursion equation added to the list of constraints of the term. The resulting substitution is not applied immediately to the term but kept in the environment for a possible delayed application.

The evaluation rules of the ρ_{g}-calculus presented in Figure 6 can be split into three categories:

- Rules describing the application of abstractions and structures on ρ-terms.
- Rules that describe the solving of match equations.
- Rules handling the replacements and the garbage collection.

The first two rules ρ and δ come from the ρ-calculus. The rule δ deals with the distributivity of the application on the structures built with the ";" operator while the rule ρ triggers the application of a rewrite rule to a ρ_{g}-term by applying the appropriate constraint to the right hand side of the rule. For each of these rules an additional one taking into account a possible context is added. Without these rules the application of abstraction ρ_{g}-terms like $x\ [x = f(y) \twoheadrightarrow x\ f(y)]$ (that can encode a recursive application as in Example 3.4) cannot be reduced. Alternatively, appropriate distributivity rules could be introduced but this approach is not considered in this paper.

The Matching rules and in particular the rule *decompose* are strongly related to the theory modulo which we want to compute the solution of the matching. In this first version of the ρ_{g}-calculus, we have chosen to present the ρ_{g}-calculus with an empty theory, but extensions to more complicated theories are possible. Due to the restrictions imposed on the left-hand sides of rewrite rules, we only need to decompose algebraic terms.

The goal of this set of rules is to produce a constraint of the form $x_1 = G_1, \ldots, x_n = G_n$ starting from a matching equation. This is possible when the left and right hand sides of the matching equation are algebraic but some replacements might be needed (as defined by the Graph rules) as soon as the terms contain some sharing.

A matching equation containing constraints is reduced (by the *propagate* rule) to a constraint containing the same matching equation without the constraints that are propagated to the top level. Since the left hand sides of matching equations are acyclic, there is no need for an evaluation rule prop-

BASIC RULES:

$$\begin{array}{lll}
(\rho) & (G_1 \twoheadrightarrow G_2)\ G_3 & \to_\rho\ G_2\ [G_1 \ll G_3] \\
 & (G_1 \twoheadrightarrow G_2)\ [E]\ G_3 & \to_\rho\ G_2\ [G_1 \ll G_3, E] \\
(\delta) & (G_1; G_2)\ G_3 & \to_\delta\ G_1\ G_3; G_2\ G_3 \\
 & (G_1; G_2)\ [E]\ G_3 & \to_\delta\ (G_1\ G_3; G_2\ G_3)\ [E]
\end{array}$$

MATCHING RULES:

$$\begin{array}{lll}
(\textit{propagate}) & G_1 \ll (G_2\ [E_2]) & \to_p\ G_1 \ll G_2, E_2 \\
(\textit{decompose}) & K(G_1, \ldots, G_n) \ll K(G'_1, \ldots, G'_n) & \to_{dk}\ G_1 \ll G'_1, \ldots, G_n \ll G'_n \\
(\textit{eliminate}) & K \ll K, E & \to_e\ E \\
(\textit{solved}) & x \ll G, E & \to_s\ x = G, E \quad \textit{if}\ x \notin \mathcal{DV}(E)
\end{array}$$

GRAPH RULES:

$$\begin{array}{lll}
(\textit{external sub}) & \mathsf{Ctx}\{y\}\ [y = G, E] & \to_{es}\ \mathsf{Ctx}\{G\}\ [y = G, E] \\
(\textit{acyclic sub}) & G\ [G_0 \lll \mathsf{Ctx}\{y\}, y = G_1, E] & \to_{ac}\ G\ [G_0 \lll \mathsf{Ctx}\{G_1\}, y = G_1, E] \\
 & & \quad \textit{if}\ x > y,\ \forall x \in \mathcal{FV}(G_0) \\
 & & \quad \textit{where} \lll \in \{=, \ll\} \\
(\textit{garbage}) & G\ [E, x = G'] & \to_{gc}\ G\ [E] \\
 & & \quad \textit{if}\ x \notin \mathcal{FV}(E) \cup \mathcal{FV}(G) \\
 & G\ [\epsilon] & \to_{gc}\ G \\
(\textit{black hole}) & \mathsf{Ctx}\{x\}\ [x =_\circ x, E] & \to_{bh}\ \mathsf{Ctx}\{\bullet\}\ [x =_\circ x, E] \\
 & G\ [y = \mathsf{Ctx}\{x\}, x =_\circ x, E] & \to_{bh}\ G\ [y = \mathsf{Ctx}\{\bullet\}, x =_\circ x, E] \\
 & & \quad \textit{if}\ y > x
\end{array}$$

Fig. 6. Evaluation rules

agating the constraints from the left hand side of the matching equation; the possible constraints on this side of the matching equation can be pushed down in the term using the substitution and garbage collection rules. The algebraic terms are decomposed and the trivial equations are eliminated. A match constraint $x \ll G_1$ is transformed into a recursion equation $x = G_1$ if there exist no other constraints of the form $x = G_2$ or $x \ll G_2$ in the list of constraints. For example, the constraint $x \ll a, x \ll b$ cannot be reduced showing that the original (non-linear) matching problem has no solution.

The GRAPH RULES are inherited from the cyclic λ-calculus of Ariola and Klop. The first two rules make a copy of a ρ_{g}-term associated to a recursion variable into a term that is inside the scope of the corresponding constraint. This is important when a redex should be made explicit (*e.g.* $x\ a\ [x = a \twoheadrightarrow b]$) or when a matching equation should be solved (*e.g.* $(a \ll x)\ [x = a]$). As we have already mentioned, the order on the variables of a ρ_{g}-term allows one to make the copies only upwards. Without this condition confluence is surely broken: the ρ_g-term $z_1\ [z_1 = x \twoheadrightarrow z_2\ s(x), z_2 = y \twoheadrightarrow z_1\ s(y)]$ reduces either to $z_1\ [z_1 = x \twoheadrightarrow z_1\ s(s(x))]$ or to $z_1\ [z_1 = x \twoheadrightarrow z_2\ s(x), z_2 = y \twoheadrightarrow z_2\ s(s(y))]$ (see [AK97] for the complete counterexample). As mentioned in the conclusions, we conjecture that, as it happens for the cyclic λ-calculus, with some restrictions on the shape of the rewrite rules, this is one of the key ingredients for confluence also for the ρ_g-calculus.

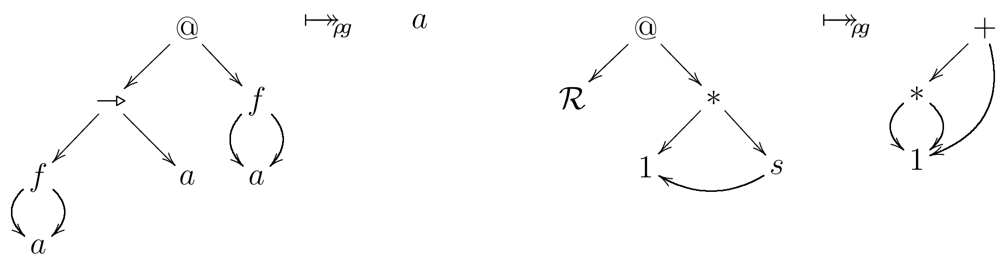
The *garbage* rules get rid of recursion equations that represent non connected parts of the ρ_g-term. Matching constraints are never eliminated, keeping thus the trace of matching failures during a non successful reduction. The *black hole* rules replace the undefined ρ_g-terms with the constant $\bullet$.

As usually, we define the one step relations $\mapsto_{\mathcal{M}}$ and $\mapsto_{\rho g}$ and the many steps relations $\mapsto\!\!\!\!\rightarrow_{\mathcal{M}}$ and $\mapsto\!\!\!\!\rightarrow_{\rho g}$ *w.r.t.* the subset of Matching rules and the whole set of rules of Figure 6 respectively.

Example 3.1 [A simple reduction with sharing]
For a graphical representation see Figure 7(a).

$$
\begin{array}{ll}
 & (f(x,x)\ [x=a] \twoheadrightarrow a)\ (f(y,y)\ [y=a]) \\
\mapsto_{p} & a\ [f(x,x)\ [x=a] \ll f(y,y)\ [y=a]] \\
\mapsto\!\!\!\!\rightarrow_{es} & a\ [f(a,a)\ [x=a] \ll f(y,y)\ [y=a]] \\
= & a\ [f(a,a)\ [x=a,\epsilon] \ll f(y,y)\ [y=a]] \\
\mapsto_{gc} & a\ [f(a,a)\ [\epsilon] \ll f(y,y)\ [y=a]] \\
\mapsto_{gc} & a\ [f(a,a) \ll f(y,y)\ [y=a]] \\
\mapsto_{p} & a\ [f(a,a) \ll f(y,y), y=a] \\
\mapsto_{dk} & a\ [a \ll y, y=a] \quad (\textit{by idempotency}) \\
\mapsto_{ac} & a\ [a \ll a, y=a] \\
\mapsto_{e} & a\ [y=a] \\
= & a\ [y=a,\epsilon] \\
\mapsto_{gc} & a\ [\epsilon] \\
\mapsto_{gc} & a
\end{array}
$$

Example 3.2 [Multiplication] If we use an infix notation for the constant "$*$" the following ρ_g-term corresponds to the application of the rewrite rule $\mathcal{R} = (x * s(y) \twoheadrightarrow ((x_1 * y + x_1)\ [x_1 = x])$ (see Figure 1) to the term $1 * s(1)$ where the term 1 is shared. The result is shown graphically in Figure 7(b).



(a) (b)

Fig. 7. Examples of reductions

$$
\begin{array}{ll}
 & (x * s(y) \multimap (x_1 * y + x_1\ [x_1 = x]))\ (z * s(z)\ [z = 1]) \\
\mapsto_\rho & x_1 * y + x_1\ [x_1 = x]\ [x * s(y) \ll (z * s(z)\ [z = 1])] \\
\mapsto_p & x_1 * y + x_1\ [x_1 = x]\ [x * s(y) \ll z * s(z), z = 1] \\
\mapsto\!\!\!\rightarrow_{dk} & x_1 * y + x_1\ [x_1 = x]\ [x \ll z, y \ll z, z = 1] \\
\mapsto\!\!\!\rightarrow_{s} & x_1 * y + x_1\ [x_1 = x]\ [x = z, y = z, z = 1] \\
\mapsto\!\!\!\rightarrow_{es} & (z * z + z)\ [x_1 = z]\ [x = z, y = z, z = 1] \\
\mapsto\!\!\!\rightarrow_{gc} & (z * z + z)\ [x_1 = z]\ [z = 1] \\
\mapsto\!\!\!\rightarrow_{gc} & (z * z + z)\ [z = 1]
\end{array}
$$

Example 3.3 [Non-linearity] The matching involving non-linear patters can lead to a normal form that is either a constraint consisting only of recursion equations (that represent a successful matching) or a constraint that contain some matching equations (representing a matching failure).

$$
\begin{array}{llcll}
 & f(x, x)\ [x = y] \ll f(a, a) & \qquad & & f(x, x)\ [x = y] \ll f(a, b) \\
\mapsto\!\!\!\rightarrow_{es} & f(y, y)\ [x = y] \ll f(a, a) & & \mapsto\!\!\!\rightarrow_{es} & f(y, y)\ [x = y] \ll f(a, b) \\
\mapsto\!\!\!\rightarrow_{gc} & f(y, y) \ll f(a, a) & & \mapsto\!\!\!\rightarrow_{gc} & f(y, y) \ll f(a, b) \\
\mapsto_{dk} & y \ll a\ \ (\textit{by idempotency}) & & \mapsto_{dk} & y \ll a, y \ll b \\
\mapsto_s & y = a & & &
\end{array}
$$

Example 3.4 Consider the term rewrite rule $R_1 = Y\ x \to x\ (Y\ x)$ which expresses the behaviour of the fixed point combinator Y of the λ-calculus. Given the a term t, we have the infinite rewrite sequence

$$Y\ t\ \to_{R_1}\ t\ (Y\ t)\ \to_{R_1}\ t\ (t\ (Y\ t))\ \to_{R_1}\ \ldots$$

which, in a sense which can be formalized, converges to the infinite term $t\ (t\ (t\ (\ldots)))$.

We can represent this reduction in the ρ_{g}-calculus using the ρ_{g}-term

$$(x_0\ [x_0 = Y\ x \twoheadrightarrow x\ (x_0\ (Y\ x))])\ (Y\ G)$$

where Y behaves like a constant.

If we denote $R = Y\ x \twoheadrightarrow x\ (x_0\ (Y\ x))$, the reduction is performed as follows:

$$\begin{array}{ll}
 & (x_0\ [x_0 = R])\ (Y\ G) \\
\mapsto_{es} & ((Y\ x \twoheadrightarrow x\ (x_0\ (Y\ x)))\ [x_0 = R])\ (Y\ G) \\
\mapsto_{\rho} & x\ (x_0\ (Y\ x))\ [Y\ x \ll Y\ G, x_0 = R] \\
\mapsto_{dk} & x\ (x_0\ (Y\ x))\ [x \ll G, x_0 = R] \\
\mapsto_{s} & x\ (x_0\ (Y\ x))\ [x = G, x_0 = R] \\
\mapsto\!\!\!\!\to_{es} & G\ (x_0\ (Y\ G))\ [x = G, x_0 = R] \\
\mapsto_{gc} & G\ (x_0\ (Y\ G))\ [x_0 = R] \\
\mapsto\!\!\!\!\to_{\rho g} & G(G \ldots (x_0\ (Y\ G)))\ [x_0 = R] \\
\mapsto\!\!\!\!\to_{\rho g} & \ldots
\end{array}$$

Continuing the reduction, this will "converge" to the term of Figure 8(a).

We can have a more efficient implementation of the same term reduction using a method introduced by Turner [Tur79] that models the rule R_1 by means of the cyclic ρ_{g}-term depicted in Figure 8(b). This gives in the ρ_{g}-calculus the ρ_{g}-term

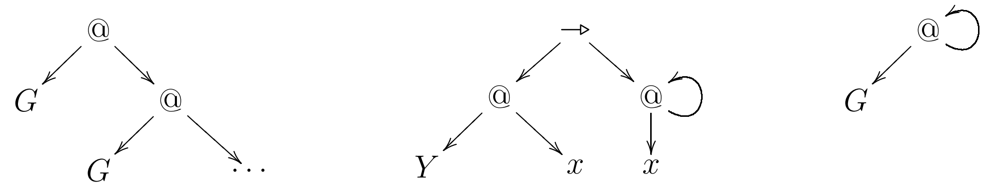
$$(Y\ x \twoheadrightarrow (z\ [z = x\ z]))\ (Y\ G)$$

The reduction in this case is the following:

$$\begin{array}{ll}
 & (Y\ x \twoheadrightarrow (z\ [z = x\ z]))\ (Y\ G) \\
\mapsto_{\rho} & z\ [z = x\ z]\ [Y\ x \ll Y\ G] \\
\mapsto_{dk} & z\ [z = x\ z]\ [x \ll G] \\
\mapsto_{s} & z\ [z = x\ z]\ [x = G] \\
\mapsto_{es} & z\ [z = G\ z]\ [x = G] \\
\mapsto\!\!\!\!\to_{gc} & z\ [z = G\ z]
\end{array}$$

The resulting ρ_{g}-term is depicted in Figure 8(c). If we "unravel", in the intuitive sense, this cyclic ρ_{g}-term we obtain the infinite term shown in Figure 8(a).

This reduction captures the fact that a finite sequence of rewritings on cyclic ρ_{g}-terms can correspond to an infinite term reduction sequence. In particular, the "unravelling" of a term can be done in a finite number of steps, while in term rewriting an infinite sequence of steps is needed.



(a) (b) (c)

Fig. 8. Example of reductions

4 ρ_{g}-calculus versus ρ-calculus and cyclic λ-calculus

The set of terms of the ρ-calculus is a strict subset of the set of terms of the ρ_{g}-calculus (modulo some syntactic conventions). The main difference for ρ-terms is the restriction of the list of constraints to a single constraint necessarily of the form $\_ \ll \_$ (delayed matching constraint).

Before proving that the ρ-calculus is simulated in the ρ_{g}-calculus we need to show that the Matching rules of the ρ_{g}-calculus are well-behaving with respect to the ρ-calculus matching algorithm restricted to patterns [CKL02].

Lemma 4.1 *Given an algebraic ρ-term T with $\mathcal{FV}(T) = \{x_1, \ldots, x_n\}$, a matching problem $T \ll U$ and its solution $\sigma = \{x_1/U_1, \ldots, x_n/U_n\}$ such that $\sigma(T) = U$. Then we have $T \ll U \mapsto\!\!\!\rightarrow_{\mathcal{M}} x_1 \ll U_1, \ldots, x_n \ll U_n$. In particular, if T is linear then $T \ll U \mapsto\!\!\!\rightarrow_{\mathcal{M}} x_1 = U_1, \ldots, x_n = U_n$.*

Proof. By structural induction on the term T.

- Basic case: The term T is a variable or a constant. The case where $T = x$ is trivial. If $T = a$ then $\sigma = \{\}$ and $U = a$. In the ρ_{g}-calculus we have $a \ll a \mapsto_e \epsilon$ and the property obviously holds.
- Induction case: $T = f(T_1, \ldots, T_m)$ with $m > 0$.
  Since a substitution σ exists and the matching is syntactic, we have $U = f(V_1, \ldots, V_m)$ and $\sigma(f(T_1, \ldots, T_m)) = f(\sigma(T_1), \ldots, \sigma(T_m))$ with $\sigma(T_i) = V_i$, $i = 1 \ldots m$. By induction hypothesis, $T_j \ll V_j \mapsto\!\!\!\rightarrow_{\mathcal{M}} x_j \ll U_j, \ldots, x_l = U_l$ with $x_j, \ldots, x_l \in \mathcal{FV}(T)$ and thus, we have $f(T_1, \ldots, T_m) \ll f(V_1, \ldots, V_m) \mapsto_{dk} T_1 \ll V_1, \ldots, T_m \ll V_m \mapsto\!\!\!\rightarrow_{\mathcal{M}} x_1 \ll U_1, \ldots, x_n \ll U_n$

The assertion about linear terms follows immediately. □

We can show now that a reduction in the ρ-calculus with linear patterns can be simulated in the ρ_{g}-calculus.

Lemma 4.2 *Given an algebraic ρ-term T and a ρ-term T'. If there exists a reduction $T \mapsto\!\!\!\rightarrow_{\rho\sigma\delta} T'$ in the ρ-calculus then there exists a corresponding one $T \mapsto\!\!\!\rightarrow_{\rho g} T'$ in the ρ_g-calculus.*

Proof. We show that for each reduction step in the ρ-calculus we have a corresponding sequence of reduction steps in the ρ_{g}-calculus.

- If $T \mapsto_{\rho} T'$ or $T \mapsto_{\delta} T'$ in the ρ-calculus, then we trivially have the same reduction in the ρ_{g}-calculus using the corresponding rules.

- If $T = [T_1 \ll T_3]T_2 \mapsto_{\sigma} \sigma(T_2) = T'$ where T_1 is a linear pattern and the substitution $\sigma = \{U_1/x_1, \ldots, U_m/x_m\}$ is solution of the matching then, in the ρ_{g}-calculus the corresponding reduction is the following:

$$
\begin{aligned}
& T = T_2\ [T_1 \ll T_3] \\
& \mapsto\!\!\!\rightarrow_{\mathcal{M}} T_2\ [x_1 = U_1, \ldots, x_m = U_m] \quad \textit{by Lemma 4.1} \\
& \mapsto\!\!\!\rightarrow_{es} \{U_1/x_1, \ldots, U_m/x_m\}T_2\ [x_1 = U_1, \ldots, x_m = U_m] \\
& \mapsto\!\!\!\rightarrow_{gc} \{U_1/x_1, \ldots, U_m/x_m\}T_2\ [\epsilon] \\
& \mapsto_{gc} \{U_1/x_1, \ldots, U_m/x_m\}T_2 = T'
\end{aligned}
$$

where we denote by $\{U_1/x_1, \ldots, U_n/x_n\}T_2$ the term T_2 in which every occurrence of the variable x_i is replaced by the term U_i, for all $i = 1 \ldots m$.

□

In the case of matching failures, the two calculi handle errors in a slightly different way, even if in both cases matching clashes are not reduced and kept as constraint application failures. In particular we can have a deeper decomposition of a matching problem in the ρ_{g}-calculus than in the ρ-calculus and thus a ρ-term in normal form can be further reduced in the ρ_{g}-calculus.

Example 4.3 [Matching failure in ρ-calculus and ρ_{g}-calculus] In both calculi, non successful reduction lead to a non solvable match equation in the list of constraints of the term.

$$
\begin{array}{ll}
(f(a) \twoheadrightarrow b)\ f(c) & \qquad (f(a) \twoheadrightarrow b)\ f(c) \\
\mapsto_{\rho\sigma\delta} [f(a) \ll f(c)]b & \qquad \mapsto_{\rho}\ b\ [f(a) \ll f(c)] \\
 & \qquad \mapsto_{dk}\ b\ [a \ll c]
\end{array}
$$

Notice that in the ρ-calculus, since the matching algorithm cannot compute a substitution solution of the match equation $f(a) \ll f(c)$, the (σ) rule cannot be applied and thus the reduction is stuck. On the other hand, in the ρ_{g}-calculus the Matching rules can partially decompose the match equation until the clash $a \ll c$ is reached.

The terms of $\lambda\phi_0$ can be easily translated into terms of the ρ_{g}-calculus. The main difference of $\lambda\phi_0$ *w.r.t.* ρ_{g}-calculus is the restriction of the list of constraints to a list of recursion equations. Delayed matching constraint are not needed since in the λ-calculus the matching is always trivially satisfied.

Definition 4.4 [Translation] The translation of a $\lambda\phi_0$-term t into a ρ_g-term, denoted $\overline{t}$, is inductively defined as follows:

$$
\begin{array}{llll}
\overline{x} & \triangleq x & \overline{f^n(t_1,\ldots,t_n)} & \triangleq f(\overline{t_1},\ldots,\overline{t_n}) \\
\overline{\lambda x.t} & \triangleq x \twoheadrightarrow \overline{t} & \overline{\langle t_0|\ x_1 = t_1,\ldots,x_n = t_n\rangle} & \triangleq \overline{t_0}\ [x_1 = \overline{t_1},\ldots,x_n = \overline{t_n}] \\
\overline{t_0\ t_1} & \triangleq \overline{t_0}\ \overline{t_1} & &
\end{array}
$$

We can see the evaluation rules of the ρ_{g}-calculus as the generalization of those of the $\lambda\phi_0$-calculus. The *β-rule* can be simulated using the Basic rules of the ρ_{g}-calculus. The rest of the rules can be simulated using the corresponding ones in the subset Graph rules of the ρ_{g}-calculus.

We show next that a reduction in the $\lambda\phi_0$-calculus can be simulated in the ρ_{g}-calculus.

Lemma 4.5 *Given two $\lambda\phi_0$-terms t_1 and t_2. If $t_1 \mapsto_{\lambda\phi} t_2$ in the cyclic λ-calculus, then there exists a reduction $\overline{t_1} \mapsto\!\!\!\!\to_{\rho g} \overline{t_2}$ in the ρ_g-calculus.*

Proof. We proceed by analyzing each reduction axiom of $\lambda\phi_0$.

- The β-rule:

$$t_1 = (\lambda x.s_1)\ s_2 \to_\beta \langle s_1|\ x = s_2\rangle = t_2$$

In the ρ_{g}-calculus we have:

$$\overline{t_1}\ =\ (x \twoheadrightarrow \overline{s_1})\ \overline{s_2}\ \mapsto_\rho\ \overline{s_1}\ [x \ll \overline{s_2}]\ \mapsto_s\ \overline{s_1}\ [x = \overline{s_2}]\ =\ \overline{t_2}$$

- The *external sub* rule: trivial.
- The *acyclic sub* rule: trivial ($\lll$ stands always for $=$ in this case).
- The *black hole* rule: trivial.
- The garbage collect rule: The proviso $E \perp (E', t)$ is equivalent to the one expressed using the definition of free variables in the ρ_{g}-calculus. The condition $E' \neq \epsilon$ is implicit in the ρ_{g}-calculus since we eliminate one recursion equation at time. For this reason, a single step of the *garbage collect* rule in $\lambda\phi_0$ can correspond to several steps of the corresponding *garbage* rule in the ρ_{g}-calculus: if $\langle t|E, E'\rangle \to_{gc} \langle t|E\rangle$ then $\overline{t}\ [E, E'] \mapsto\!\!\!\!\to_{gc} \overline{t}\ [E]$.

□

5 Conclusions and future work

In this paper we have proposed ρ_{g}-calculus, an extension of the ρ-calculus able to deal with graph like structures, where sharing of subterms and cycles (which can be used to represent regular infinite data structures) can be expressed. The ρ_{g}-calculus has been shown to be a generalization of the cyclic λ-calculus as well as of the standard ρ-calculus.

The work is still in a preliminary stage and there are several interesting directions of future research.

Taking inspiration from analogous work on cyclic λ-calculus [AK97] and on the ρ-calculus [BCKL03], it would be interesting to understand under which restrictions the ρ_{g}-calculus can be made confluent. We conjecture that, if we consider a syntactic matching, it suffices to restrict to rewrite rules and matching problems where the left hand side respect the so-called "Rigid Pattern Condition" [vO90] adapted to our syntax. This condition corresponds in fact to the restrictions we have already imposed for the patterns in Section 2.

At the same time, an appealing problem is the generalization of ρ_{g}-calculus to deal with different, non syntactic, matching theories. For example, in the case of a matching involving cyclic graphs, the reduction of a matching constraint can be stuck even if a solution of the matching problem actually exists. For instance, the term $g(x, x) \ll (g(f(z), f(f(y)))\ [y = f(y), z = f(z)]$ can be reduced to $[x \ll f(z), x \ll f(f(y)), y = f(y), z = f(z)]$ but it is stuck at this point. In order to recover from this failure, we should be able to compare the right hand sides of the two match equations and decide if their "unravelling" is the same. In other words, we should be able to deal with general cyclic matching. One should notice that this is not straightforward, since, in ρ_{g}-calculus matching is internalized rather than being carried out at metalevel.

Moreover, in this paper we have only informally scratched the problem of defining the (cyclic term) graph associated to a term of the ρ_{g}-calculus. While for the fragment of the ρ_{g}-calculus without matching constraints some clear suggestions could come from existing work on cyclic term graphs with binders in [Blo01,Kah98], the generalization to the full calculus will require further investigations.

After making this correspondence formal, a quite interesting question arise asking whether we can encode term graph rewriting into the ρ_{g}-calculus in the same way as term rewriting systems (and their underlying strategies) can be encoded in the ρ-calculus. Furthermore, a term of the ρ_{g}-calculus, possibly with sharing and cycles, can be seen as a "compact" representation of a possibly infinite ρ-calculus term, obtained by "unravelling" the original term. On the one hand, it would be interesting to define an infinitary version of the ρ-calculus, taking inspiration, e.g., from the work on the infinitary λ-calculus [KKSd97] and on infinitary rewriting. On the other hand, to enforce the view of the ρ_{g}-calculus as efficient implementations of terms and rewriting in the infinitary ρ-calculus one should have an adequacy result in the style of [KKSd94,CD97].

Acknoledgments. We are grateful to Andrea Corradini and Fabio Gadducci for fruitful discussions on earlier versions of this paper.

References

[ACCL91] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Levy. Explicit substitutions. *Journal of Functional Programming*, 4(1):375–416, 1991.

[AK96] Z. M. Ariola and J. W. Klop. Equational term graph rewriting. *Foundamenta Informaticae*, 26(3-4):207–240, 1996.

[AK97] Z. M. Ariola and J. W. Klop. Lambda calculus with explicit recursion. *Journal of Information and Computation*, 139 (2):154–233, 1997.

[Bar84] H. P. Barendregt. *The Lambda-Calculus, its syntax and semantics.* Studies in Logic and the Foundation of Mathematics. Elsevier Science Publishers B. V. (North-Holland), Amsterdam, 1984. Second edition.

[BCKL03] G. Barthe, H. Cirstea, C. Kirchner, and L. Liquori. Pure Patterns Type Systems. In *Proceedings of POPL'03: Principles of Programming Languages, New Orleans, USA*, volume 38, pages 250–261. ACM, 2003.

[Blo01] S. Blom. *Term Graph Rewriting - Syntax and Semantics.* PhD thesis, Vrije Universiteit Amsterdam, 2001.

[BvEG+87] 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 *Proceedings of PARLE'87, Parallel Architectures and Languages Europe*, volume 259 of *Lecture Notes in Computer Science*, pages 141–158, Eindhoven, 1987. Springer-Verlag.

[CD97] A. Corradini and F. Drewes. (Cyclic) term graph rewriting is adequate for rational parallel term rewriting. Technical Report TR-97-14, Dipartimento di Informatica, Pisa, 1997.

[CFK04] H. Cirstea, G. Faure, and C. Kirchner. A rho-calculus of explicit constraint application. In *Proceedings of WRLA'04, the 5th Workshop on Rewriting Logic and Applications, Barcelona, Spain.* Electronic Notes in Theoretical Computer Science, 2004.

[CG99] A. Corradini and F. Gadducci. Rewriting on cyclic structures: Equivalence of operational and categorical descriptions. *Theoretical Informatics and Applications*, 33:467–493, 1999.

[CK01] H. Cirstea and C. Kirchner. The rewriting calculus — Part I *and* II. *Logic Journal of the Interest Group in Pure and Applied Logics*, 9(3):427–498, May 2001.

[CKL01] H. Cirstea, C. Kirchner, and L. Liquori. Matching Power. In A. Middeldorp, editor, *Proceedings of RTA'01, Rewriting Techniques and Applications*, volume 2051 of *Lecture Notes in Computer Science*, pages 77–92, Utrecht, The Netherlands, May 2001. Springer-Verlag.

[CKL02] H. Cirstea, C. Kirchner, and L. Liquori. Rewriting calculus with(out) types. In F. Gadducci and U. Montanari, editors, *Proceedings of*

WRLA'02, the 4th Workshop on Rewriting Logic and Applications, volume 71 of *Electronic Notes in Theoretical Computer Science*, Pisa (Italy), September 2002.

[CLW03] H. Cirstea, L. Liquori, and B. Wack. Rewriting calculus with fixpoints: Untyped and first-order systems. In *Types for Proofs and Programs (TYPES)*, volume 3085 of *Lecture Notes in Computer Science*, pages 147–171, Torino (Italy), May 2003.

[Cor93] A. Corradini. Term rewriting in CT_Σ. In M. C. Gaudel and J. P. Jouannaud, editors, *Proceedings of TAPSOFT'93, Theory and Practice of Software Development / 4th International Joint Conference CAAP/FASE*, pages 468–484. Springer, Berlin, Heidelberg, 1993.

[HKP91] A. Habel, H. J. Kreowski, and D. Plump. Jungle evaluation. *Fundamamenta Informaticae*, 15(1):37–60, 1991.

[Kah98] W. Kahl. Relational treatment of term graphs with bound variables. *Logic Journal of the Interest Group in Pure and Applied Logics*, 6(2):259–303, 1998.

[KKSd94] J. R. Kennaway, J. W. Klop, M. R. Sleep, and F. J. de Vries. On the adequacy of graph rewriting for simulating term rewriting. *ACM Transactions on Programming Languages and Systems*, 16(3):493–523, May 1994.

[KKSd97] J. R. Kennaway, J. W. Klop, M. R. Sleep, and F. J. de Vries. Infinitary lambda calculus. *Theoretical Computer Science*, 175(1):93–125, 1997.

[Par92] M. Parigot. $\lambda\mu$-calculus: An algorithmic interpretation of classical natural deduction. In A. Voronkov, editor, *Proceedings of LPAR'92, Logic Programming and Automated Reasoning, St Petersburg, Russia, July 1992*, volume 624 of *Lecture Notes in Artificial Intelligence*, pages 190–201. Springer-Verlag, Berlin, 1992.

[PJ87] S. L. Peyton-Jones. *The implementation of functional programming languages*. Prentice-Hall International, 1987.

[SPvE93] M. R. Sleep, M. J. Plasmeijer, and M. C. J. D. van Eekelen, editors. *Term graph rewriting: theory and practice*. Wiley, London, 1993.

[Tur79] D. A. Turner. A new implementation technique for applicative languages. *Software Practice and Experience*, 9:31–49, 1979.

[vO90] V. van Oostrom. Lambda Calculus with Patterns. Technical Report IR-228, Faculteit der Wiskunde en Informatica, Vrije Universiteit Amsterdam, 1990.