Math. Struct. in Comp. Science (2007), *vol.* 17, *pp.* 363–406.
doi:10.1017/S0960129507006093 Printed in the United Kingdom

A rewriting calculus for cyclic higher-order term graphs

PAOLO BALDAN[†], CLARA BERTOLISSI[‡‖],
HORATIU CIRSTEA[§‖] and CLAUDE KIRCHNER[¶‖]

[†]*Dipartimento di Matematica, Pura e Applicata, Università di Padova, Italy*
Email: baldan@math.unipd.it
[‡]*Université Henri Poincaré*
Email: Clara.Bertolissi@loria.fr
[§]*Université Nancy 2*
Email: Horatiu.Cirstea@loria.fr
[¶]*INRIA*
Email: Claude.Kirchner@loria.fr
[‖]*LORIA, BP 239, 54506 Vandoeuvre-lès-Nancy Cedex France*

Received 15 November 2005; revised 10 September 2006

The Rewriting Calculus (ρ-calculus, for short) was introduced at the end of the 1990s and fully integrates term-rewriting and λ-calculus. The rewrite rules, acting as elaborated abstractions, their application and the structured results obtained are first class objects of the calculus. The evaluation mechanism, which is a generalisation of beta-reduction, relies strongly on term matching in various theories.

In this paper we propose an extension of the ρ-calculus, called ρ_g-calculus, that handles structures with cycles and sharing rather than simple terms. This is obtained by using recursion constraints in addition to the standard ρ-calculus matching constraints, which leads to a term-graph representation in an equational style. Like in the ρ-calculus, 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.

We show that the ρ_g-calculus, under suitable linearity conditions, is confluent. The proof of this result is quite elaborate, due to the non-termination of the system and the fact that ρ_g-calculus-terms are considered modulo an equational theory. We also show that the ρ_g-calculus is expressive enough to simulate first-order (equational) left-linear term-graph rewriting and λ-calculus with explicit recursion (modelled using a `letrec`-like construct).

1. Introduction

The main interest in term rewriting stems from functional and rewrite-based languages and from theorem proving. In particular, we can describe the behaviour of a functional or rewrite-based program by analysing some properties of the associated term rewriting 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 represent and implement terms as graphs (Barendregt *et al.* 1987). In this case, the possibility of sharing sub-terms allows us to save space (by using multiple pointers to the

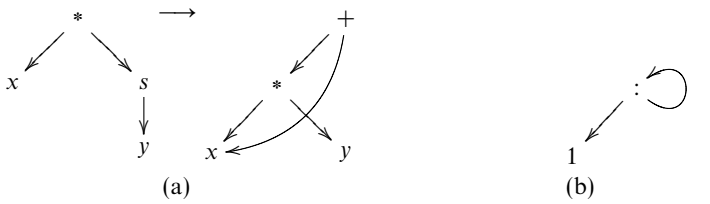


Figure 1. (a) Rule for multiplication with sharing (b) Cyclic representation of an infinite list of ones.

same sub-term instead of duplicating the sub-term) and to save time (for example, when the sharing is maximal, a redex appearing in a shared sub-term will be reduced at most once and equality tests can be done in constant time). Consider, as an example, the definition of multiplication given by the rewrite system $\mathscr{R} = \{x * 0 \to 0, \quad x * s(y) \to (x * y) + x\}$. If we represent it using graphs, we can write the second rule by duplicating the reference to x instead of duplicating x itself (see Figure 1(a)).

Graph rewriting is a useful technique for the optimisation of functional and declarative languages implementation (Peyton-Jones 1987). Moreover, the possibility of defining cycles leads to an increased expressive power that makes it easy to represent regular infinite data structures. For example, if ':' denotes the concatenation operator, an infinite list of ones can be modelled as a cyclic list $ones = 1 : ones$, represented by the cyclic graph of Figure 1(b). Cyclic term graph rewriting has been widely studied, both from an operational (Barendregt *et al.* 1987; Ariola and Klop 1996) and from a categorical/logical point of view (Corradini and Gadducci 1999) – see also Sleep *et al.* (1993) and Plump (1999) for a survey on term graph rewriting.

In this context, an abstract model generalising the λ-calculus and adding cycles and sharing features has been proposed in Ariola and Klop (1997). Their approach consists of an equational framework that models the 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 graphical structures with the higher-order capabilities of the λ-calculus. A final important ingredient is still missing: pattern matching. The possibility of discriminating using pattern matching could be encoded, in particular, in the λ-calculus, but it is much more attractive to have an explicit matching construct and, indeed, to use rewriting. Programs become quite compact and the encoding of data type structures is no longer necessary.

The rewriting calculus (ρ-calculus, for short) was introduced in the late nineties as a natural generalisation of term rewriting and of the λ-calculus (Cirstea and Kirchner 2001). It has been shown to be a very expressive framework, for example, for expressing object calculi (Cirstea *et al.* 2001), and it has been equipped with powerful type systems (Barthe *et al.* 2003; Wack 2005). The notion of ρ-reduction of the ρ-calculus generalises β-reduction by considering matching on patterns, which can be more elaborated than simple variables. First, the matching constraint is built explicitly and then the substitution possibly obtained by solving such a constraint is applied to the body of the abstraction. By making this

https://doi.org/10.1017/S0960129507006093 Published online by Cambridge University Press

matching step explicit and the matching constraints first-class objects of the calculus, we can allow for an explicit, fine-grained handling of constraints instead of generating and applying substitutions at once (Cirstea *et al.* 2005).

The main contribution of this paper consists of a new system, called the ρ_g-calculus, that generalises the cyclic λ-calculus in the same way as the standard ρ-calculus generalises the λ-calculus.

In the ρ_g-calculus any term is associated with a list of constraints consisting of recursion equations, which are used to express sharing and cycles, and matching constraints, arising from the fact that computations related to matching are made explicit and performed at the object-level. The matching algorithm described by the evaluation rules of the ρ_g-calculus corresponds to syntactic matching. The order and multiplicity of constraints in a list is inessential, and the addition of an empty list of constraints is irrelevant in a ρ_g-term. Hence, formally, the conjunction operator, which is used to build lists of constraints, is assumed to be associative, commutative and idempotent, with the empty list of constraints as neutral element. As a consequence, reductions take place over equivalence classes of terms rather than over single terms, and this fact must be considered when reasoning on the rewrite relation induced by the evaluation rules of the calculus.

The calculus is shown to be confluent, under some linearity and acyclicity restrictions on patterns. The proof method generalises the proof of confluence of the cyclic λ-calculus (Ariola and Klop 1997) to the setting of rewriting, modulo an equational theory (Jouannaud and Kirchner 1986; Ohlebusch 1998), and, moreover, it adapts the proof to deal with terms containing patterns and match equations. More precisely, the concept of 'development' and the property of the 'finiteness of developments', as defined in the theory of the λ-calculus (Barendregt 1984), play a central role in the proof.

The ρ_g-calculus is shown to be an expressive formalism that generalises both the plain ρ-calculus and the λ-calculus extended with explicit recursion, providing a homogeneous framework for pattern matching and higher-order graphical structures. Moreover, we show that (equational) left-linear term graph rewriting can be naturally encoded in the ρ_g-calculus. More specifically, we prove that matching in the ρ_g-calculus is well behaved with respect to the notion of homomorphism on term graphs and that any reduction step in a term graph rewrite system can be simulated in the ρ_g-calculus.

The paper is organised as follows. In the next section we review the two systems that inspired our new calculus, the standard ρ-calculus (Cirstea and Kirchner 2001) and the cyclic λ-calculus (Ariola and Klop 1997), and we briefly describe first-order term graph rewrite systems following an equational approach (Ariola and Klop 1996). In Section 3, we present the ρ_g-calculus with its syntax and its small-step semantics, giving some examples of ρ_g-graphs and reductions in the system. In Section 4, after recalling some notions of rewriting in an equational setting, we prove the confluence of the calculus. First, a general outline of the proof is given, and then we provide the full details. In Section 5, we show that the ρ_g-calculus is a generalisation of the ρ-calculus and of the cyclic λ-calculus. We show also that first-order term graph rewriting reductions can be simulated in the ρ_g-calculus. We conclude in Section 6 by presenting some perspectives of future work.

https://doi.org/10.1017/S0960129507006093 Published online by Cambridge University Press

2. General setup

2.1. *The rewriting calculus*

The ρ-calculus was introduced as a calculus in which all the basic ingredients of rewriting are made explicit, in particular, the notions of rule abstraction (represented by the operator '$\twoheadrightarrow$'), rule application (represented by term juxtaposition) and collection of results (represented by the operator '$\wr$'). Depending on the theory behind the operator '$\wr$', the results can be grouped together, for example, in lists (when '$\wr$' is associative) or in multi-sets (when '$\wr$' is associative and commutative) or in sets (when '$\wr$' is associative, commutative and idempotent). This operator is useful for representing the (non-deterministic) application of a set of rewrite rules and, consequently, the set of possible results.

With respect to the lambda-calculus, the usual λ-abstraction $\lambda x.t$ is extended to a rule abstraction $P \twoheadrightarrow T$, where, in the most general case, P is a general ρ-term that may, for example, contain rules. Some restrictions are usually imposed on the shape of P to get desirable properties for the calculus.

The set of ρ-terms is defined as follows:

$$\mathscr{T}, \mathscr{P} ::= \mathscr{X} \mid \mathscr{K} \mid \mathscr{P} \twoheadrightarrow \mathscr{T} \mid \mathscr{T}[\mathscr{P} \ll \mathscr{T}] \mid \mathscr{T}\ \mathscr{T} \mid \mathscr{T} \wr \mathscr{T}$$

The symbols $T, U, L, R, \ldots$ range over the set $\mathscr{T}$ of terms, the symbols $x, y, z, \ldots$ range over the set $\mathscr{X}$ of variables, and the symbols a, b, c, d, e, f, g, h range over a set $\mathscr{K}$ of constants. We assume that the (hidden) application operator $(\_\ \_)$ associates to the left, while the other operators associate to the right. The priority of the application operator is higher than that of $\_[\_ \ll \_]$, which is higher than that of $\_ \twoheadrightarrow \_$ which is, in turn, of higher priority than $\_ \wr \_$. Terms of the form $(T_0\ T_1 \cdots T_n)$ will often be denoted $T_0(T_1, \ldots, T_n)$. Given a term $T = f_1(T_1, \ldots, T_{i_1}, \ldots)$, a *position* ω in T is either the empty sequence, denoted by ϵ, corresponding to the head position of t, or a sequence $f_1 i_1 f_2 i_2 \ldots f_n i_n$, such that $f_2 i_2 \ldots f_n i_n$ is a position in T_{i_1}. A sub-term of T at position ω in T is denoted $T_{|\omega}$. We will use the notation $T_{[U]_\omega}$ to specify that T has a sub-term U at position ω, and the notation $T_{[U]_\omega}$ to denote the term obtained from T by replacing the sub-term $T_{|\omega}$ by U.

We define next the set of free variables of a ρ-term, which generalises the notion of free variables in the λ-calculus.

Definition 1 (Free, bound and active variables). Given a ρ-term T, its sets of *free* variables $\mathscr{FV}(T)$ and *bound* variables $\mathscr{BV}(T)$ are defined by:

T	$\mathscr{BV}(T)$	$\mathscr{FV}(T)$
x	$\varnothing$	$\{x\}$
k	$\varnothing$	$\varnothing$
$T_1\ T_2$	$\mathscr{BV}(T_1) \cup \mathscr{BV}(T_2)$	$\mathscr{FV}(T_1) \cup \mathscr{FV}(T_2)$
$T_1 \wr T_2$	$\mathscr{BV}(T_1) \cup \mathscr{BV}(T_2)$	$\mathscr{FV}(T_1) \cup \mathscr{FV}(T_2)$
$P \twoheadrightarrow T_1$	$\mathscr{FV}(P) \cup \mathscr{BV}(P) \cup \mathscr{BV}(T_1)$	$\mathscr{FV}(T_1) \setminus \mathscr{FV}(P)$
$T_1[P \ll T_2]$	$\mathscr{FV}(P) \cup \mathscr{BV}(P) \cup \mathscr{BV}(T_1) \cup \mathscr{BV}(T_2)$	$(\mathscr{FV}(T_1) \setminus \mathscr{FV}(P)) \cup \mathscr{FV}(T_2)$

https://doi.org/10.1017/S0960129507006093 Published online by Cambridge University Press

$$\begin{array}{lllll} (\rho) & (P \twoheadrightarrow T)U & \mapsto_{\rho} & T[P \ll U] \\ (\sigma) & T[P \ll U] & \mapsto_{\sigma} & \sigma_{P \ll U}(T) \\ (\delta) & (T_1 \wr T_2)\ T_3 & \mapsto_{\delta} & T_1\ T_3 \wr T_2\ T_3 \end{array}$$

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

A term is said to be *closed* if it has no free variables. A variable is *active* in a term T when it appears free in the left-hand side of an application occurring in T.

In an abstraction $P \twoheadrightarrow T$, the free variables of P bind the corresponding variables in T, while in $T_2[P \ll T_1]$, the free variables of P are bound in T_2 (but not in T_1).

As is common in calculi involving binders, we work modulo the α-*convention* (Church 1941), that is, two terms that differ only in the names of their bound variables are considered α-equivalent and, slightly abusing the notation, single terms will be used to denote the corresponding equivalence classes. Additionally, representatives in α-equivalence classes will be chosen according to the *hygiene-convention* of Barendregt (1984), that is, forcing free and bound variables to have different names. The application of a substitution σ to a term T, denoted by $\sigma(T)$ or $T\sigma$, is defined, as usual, to avoid variable captures.

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 evaluates via the rule (ρ) to the application of the corresponding constraint to the right-hand side of the rewrite rule. Such a construction is called a *delayed matching constraint*. By rule (σ), if the matching problem between P and U admits a solution σ, the delayed matching constraint evaluates to $\sigma(T)$. Finally, rule (δ) distributes the application of structures.

Note that the matching power of the general ρ-calculus can be regulated by using arbitrary theories, possibly leading to multiple solutions for a matching problem. In this case, if $\sigma_1, \ldots, \sigma_n$ are the substitutions arising as solutions of a matching problem $P \ll U$, the application of the constraint $T[P \ll U]$ would evaluate to a structure $\sigma_1(T) \wr \ldots \wr \sigma_n(T)$. The simplified version of rule (σ) above is motivated by the fact that here we consider the ρ-calculus with the empty theory, that is, with syntactic matching, which is decidable and has a unique solution.

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

Example 2. Let $head(cons(x, y)) \twoheadrightarrow x$ be the ρ-abstraction returning the head of a list. If we apply it to $head(cons(a, b))$, we obtain the following reduction:

$$\begin{array}{lll} (head(cons(x, y)) \twoheadrightarrow x)\ head(cons(a, b)) & \mapsto_{\rho} & x[head(cons(x, y)) \ll head(cons(a, b))] \\ & \mapsto_{\sigma} & x\{x/a, y/b\} \\ & = & a. \end{array}$$

2.2. *The cyclic lambda calculus*

The cyclic λ-calculus introduced in Ariola and Klop (1997) generalises the ordinary λ-calculus by allowing us to represent sharing and cycles in the λ-calculus terms. This is

https://doi.org/10.1017/S0960129507006093 Published online by Cambridge University Press

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

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

obtained by adding to the λ-calculus a `letrec`-like construct in such a way that the new terms, called λ-graphs, are essentially systems of (possibly nested) recursion equations on standard λ-terms. If the system is used without restrictions on the rules, confluence is lost. The authors restore it by controlling the operations on the recursion equations. The resulting calculus, called $\lambda\phi$, is powerful enough to incorporate the λ-calculus (Barendregt 1984), the $\lambda\mu$-calculus (Parigot 1992) and the $\lambda\sigma$-calculus with names (Abadi *et al.* 1991) extended with horizontal and vertical sharing, respectively. The syntax of $\lambda\phi$ is

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

The set of $\lambda\phi$-terms consists of the ordinary λ-terms (that is, variables, functions of fixed arity, applications, abstractions) and new terms built using a `letrec` construct $\langle t_0 \mid \ x_1 = t_1, \ldots, x_n = t_n\rangle$, where all the recursion variables x_i are assumed distinct, for $i = 1, \ldots, n$. Variables are bound either by lambda abstractions or by recursion equations. Let E denote any unordered sequence of equations $x_1 = t_1, \ldots, x_n = t_n$ and let ϵ be the empty sequence. The notation $x =_\circ x$ is an abbreviation for a sequence of recursion equations $x = x_1, \ldots, x_n = x$. Terms are denoted by the symbols $t, s, \ldots$, variables are denoted by the symbols $x, y, z, \ldots$ and constants by the symbols a, b, c, d, e, f, g, h.

A context $\mathsf{Ctx}\{\Box\}$ is a term with a single hole $\Box$ in place of a sub-term. Filling the context $\mathsf{Ctx}\{\Box\}$ with a term t yields the term $\mathsf{Ctx}\{t\}$. We use $<$ to denote the least transitive relation on recursion variables such that $x > y$ if $x = \mathsf{Ctx}\{y\}$ for some context $\mathsf{Ctx}\{\Box\}$. We write $x \equiv y$ if $x > y$ and $y > x$ (intuitively, if variables x and y occur in a cycle). We write $E \perp (E', t)$ and say that E is *orthogonal* to a sequence of equations E' and a term t if the recursion variables of E do not intersect the free variables of E' and t. The reduction rules of the basic $\lambda\phi$-calculus, referred to as $\lambda\phi_0$-calculus, are given in Figure 3. Some extensions of this basic set of rules are considered in Ariola and Klop (1997) – these add either distribution rules ($\lambda\phi_1$) or merging and elimination rules ($\lambda\phi_2$) for the $\langle \_ \mid \_ \rangle$ construct. In the following we will concentrate on the basic system in 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 λ-graph associated to a recursion variable. The restriction on the order of recursion variables is

https://doi.org/10.1017/S0960129507006093 Published online by Cambridge University Press

introduced to ensure confluence in the case of cyclic configurations of lambda redexes (see Section 4 of Ariola and Klop (1997) for a counterexample). The condition $E' \neq \epsilon$ in the *garbage collect* rule avoids trivial non-terminating reductions.

We use $\mapsto_{\lambda\phi}$ to denote the rewrite relation induced by the set of rules of Figure 3 and $\mapsto\!\!\!\!\rightarrow_{\lambda\phi}$ for its reflexive and transitive closure.

Example 3. Consider the λ-graph $t = \langle y \mid y = plus(z\ 0, z\ 1), z = \lambda x.s(x)\rangle$ where 0 and 1 are constants and s is meant to represent the successor function. We have the following reduction, where at each step the considered redex is underlined.

$$\begin{aligned}
\langle y \mid y = plus(\underline{z}\ 0, z\ 1), \underline{z = \lambda x.s(x)}\rangle &\mapsto_{ac} \langle y \mid y = plus(\underline{\lambda x.s(x)\ 0}, z\ 1), z = \lambda x.s(x)\rangle \\
&\mapsto_{\beta} \langle y \mid y = plus(\langle \underline{s(x)} \mid \underline{x = 0}\rangle, z\ 1), z = \lambda x.s(x)\rangle \\
&\mapsto_{es} \langle y \mid y = plus(\langle s(0) \mid \underline{x = 0}\rangle, z\ 1), z = \lambda x.s(x)\rangle \\
&\mapsto\!\!\!\!\rightarrow_{gc} \langle y \mid y = plus(s(0), \underline{z}\ 1), \underline{z = \lambda x.s(x)}\rangle \\
&\mapsto_{ac} \langle y \mid y = plus(s(0), \underline{\lambda x.s(x)\ 1}), z = \lambda x.s(x)\rangle \\
&\mapsto\!\!\!\!\rightarrow_{\lambda\phi} \langle y \mid y = plus(s(0), s(1)), \underline{z = \lambda x.s(x)}\rangle \\
&\mapsto_{gc} \langle y \mid y = plus(s(0), s(1))\rangle\,.
\end{aligned}$$

2.3. *Term graph rewriting*

Several presentations have been proposed for term graph rewriting (TGR) (see Sleep *et al.* (1993) for a survey). Here we consider an equational presentation in the style of Ariola and Klop (1996). Given a set of variables $\mathscr{X}$ and a first-order signature $\mathscr{F}$ with symbols of fixed arity, a term graph over $\mathscr{X}$ and $\mathscr{F}$ is a system of equations of the form $G = \{x_1 \mid x_1 = t_1, \ldots, x_n = t_n\}$ where $t_1, \ldots, t_n$ are terms over $\mathscr{X}$ and $\mathscr{F}$ and the recursion variables x_i are pairwise distinct, for $i = 1, \ldots, n$. The variable x_1 on the left represents the root of the term graph. We call the list of equations the *body* of the term graph and we denote it by E_G, or simply E when G is clear from the context. The empty list is denoted by ϵ. The variables $x_1, \ldots, x_n$ are bound in the term graph by the associated recursion equation. The other variables occurring in the term graph G are called *free*, and the set of free variables is denoted by $\mathscr{FV}(G)$. A term graph without free variables is called *closed*. We use $\mathrm{Var}(G)$ to denote the collection of variables appearing in G. Two α-equivalent term graphs, that is, two term graphs that differ only in the name of bound variables, are considered equal. Cycles may appear in the system and degenerated cycles, that is, equations of the form $x = x$, are replaced by $x = \bullet$ (black hole). A term graph is said to be in *flat form* if all its recursion equations are of the form $x = f(x_1, \ldots, x_n)$, where the variables $x, x_1, \ldots, x_n$ are not necessarily distinct from each other. In the following we will consider only term graphs in flat form and without useless equations (garbage), which we remove systematically during rewriting. A term graph in flat form can be easily

https://doi.org/10.1017/S0960129507006093 Published online by Cambridge University Press

interpreted and depicted as a graph taking the set of variables as nodes. We will use the graphical interpretation as an aid for intuition in the examples.

Rewriting is done by means of term graph rewriting rules.

Definition 4 (Term graph rewrite rule). A term graph rewrite rule is a pair of term graphs (L, R) such that L and R have the same root, L is not a single variable and $\mathscr{FV}(R) \subseteq \mathscr{FV}(L)$. We say that a rewrite rule is *left-linear* if L is acyclic and each variable appears at most once in the right-hand side of the recursion equations of L.

In the rest of the paper all term graph rewrite rules will be assumed to be left-linear, unless explicitly stated otherwise.

Definition 5 (Term graph rewrite system). A *term graph rewrite system* (TGR) consists of a pair $TGR = (\Sigma, \mathscr{R})$ where Σ is a signature and $\mathscr{R}$ is a set of rewrite rules over this signature.

A rewrite rule can be applied to a term graph if there exists a match of its left-hand side into the graph. The notion of match is formalised as a possibly non-injective homomorphism from the left-hand side of the rule into the term graph. Thus a rule can match term graphs containing more sharing than its left-hand side. Also notice that, since we consider term graphs in flat form, a homomorphism will simply be a (possibly non-injective) variable renaming.

Definition 6 (Substitution, matching and redex).

— A substitution $\sigma = \{x_1/y_1, \ldots, x_n/y_n\}$ is a map from variables to variables. Its application to a term graph G, denoted $\sigma(G)$, is defined inductively as follows:

$$\sigma(\{z_1 \mid z_1 = t_1, \ldots, z_n = t_n\}) \triangleq \{\sigma(z_1) \mid \sigma(z_1) = \sigma(t_1), \ldots, \sigma(z_n) = \sigma(t_n)\}$$

$$\sigma(z_i) \triangleq \begin{cases} y_i & \text{if } z_i = x_i \in \{x_1, \ldots, x_n\} \\ z_i & \text{otherwise} \end{cases} \qquad \sigma(f(t_1, \ldots, t_n)) \triangleq f(\sigma(t_1), \ldots, \sigma(t_n)).$$

— A homomorphism (matching) from a term graph L to a term graph G is a substitution σ such that $\sigma(L) \subseteq G$, where the inclusion means that all recursion equations of $\sigma(L)$ are in G, that is, if $\sigma(L) = \{x_1 \mid E\}$, then $G = \{x_1' \mid E, E'\}$.

— A redex in a term graph G is a pair $((L, R), \sigma)$ where (L, R) is a rule and σ is a homomorphism from the left-hand side L of the rule to G. If x is the root of L, we call $\sigma(x)$ the head of the redex.

We introduce next the notions of path and position, which we will use later to define a rewrite step.

Definition 7 (Path and position). A *path* in a closed term graph G is a sequence of function symbols interleaved by integers $p = f_1 i_1 f_2 \ldots i_{n-1} f_n$ such that f_{j+1} is the i_j-th argument of f_j, for all $j = 1, \ldots, n$. The sequence of integers $i_1, \ldots, i_{n-1}$ is called the *position* of the node labelled f_n and still denoted by the letter p.

https://doi.org/10.1017/S0960129507006093 Published online by Cambridge University Press

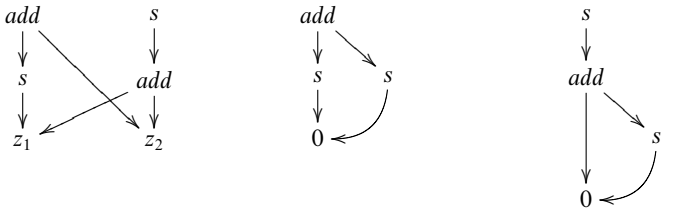


Figure 4. Examples of term graphs.

In the same way as we did for terms, we introduce the notation $G_{|p}$ for the subgraph of G at the position p in G, while $G_{\lceil G' \rceil_p}$ specifies that G contains a term graph G' at the position p. In the same situation, if z is the root of G' and $z = t$ is the corresponding equation, we will also write $G_{\lceil z=t \rceil_p}$. We use the notation $G_{[G']_p}$ to denote the term graph G where the subgraph $G_{|p}$, or, more precisely, the equation defining the root of $G_{|p}$, has been replaced by G'. Given, for instance, the two term graphs $G_{\lceil z=t \rceil_p}$ and $G' = z\ [E_{G'}]$, the term graph $G_{[G']_p}$ is obtained from G by replacing the equation $z = t$ by $E_{G'}$, and then possibly performing garbage collection. Intuitively, the term graph G' is attached to the node z in G.

The notions of path and position are used to define a rewrite step.

Definition 8 (Rewrite step). Let $((L, R), \sigma)$ be a redex occurring in G at the position p. A rewrite step that reduces the redex above consists of removing the equation defining the root of the redex and replacing it with the body of $\sigma(R)$, with a fresh choice of bound variables. Using a context notation, we have $G_{\lceil \sigma(x)=t \rceil_p} \to G_{[\sigma(R)]_p}$.

We next give an example of rewriting. Note that only the root equation of the match gets rewritten, and it is replaced by several equations. Renaming is needed to avoid collisions with other variables already in the term graph.

Example 9 (Rewriting). Let

$$G_1 = \{x_1 \mid x_1 = add(x_2, x_3), x_2 = s(x_4), x_3 = s(x_4), x_4 = 0\}$$

be a closed term graph in flat form, and

$$(L, R) = (\{y_1 \mid y_1 = add(y_2, z_2), y_2 = s(z_1)\}, \{y_1 \mid y_1 = s(y_2), y_2 = add(z_1, z_2)\})$$

be a rewrite rule (see Figure 4). Note that in the rule the bound variables are y_1 and y_2, while the free variables are z_1 and z_2. A matching of L in G_1 is given by the substitution $\sigma = \{y_1/x_1, y_2/x_2, z_1/x_4, z_2/x_3\}$. The rewrite step is performed at the root of G_1. We have

$$\begin{aligned} G_1 &= \{x_1 \mid \underline{x_1 = add(x_2, x_3)}, x_2 = s(x_4), x_3 = s(x_4), x_4 = 0\} \\ &\to \{x_1 \mid \underline{x_1 = s(x_2'), x_2' = add(x_4, x_3)}, x_2 = s(x_4), x_3 = s(x_4), x_4 = 0\} = G_2 \end{aligned}$$

where the underlined equation in G_1 is replaced by the underlined equations in G_2. The resulting term graph G_2 is depicted in Figure 4.

https://doi.org/10.1017/S0960129507006093 Published online by Cambridge University Press

Terms			Constraints		
$\mathscr{G},\mathscr{P}$ ::=	$\mathscr{X}$	Variables	$\mathscr{C}$ ::=	ϵ	Empty constraint
|	$\mathscr{K}$	Constants	|	$\mathscr{X} = \mathscr{G}$	Recursion equation
|	$\mathscr{P} \twoheadrightarrow \mathscr{G}$	Abstraction	|	$\mathscr{P} \ll \mathscr{G}$	Match equation
|	$\mathscr{G}\ \mathscr{G}$	Functional application	|	$\mathscr{C},\mathscr{C}$	Conjunction of constraints
|	$\mathscr{G} \wr \mathscr{G}$	Structure			
|	$\mathscr{G}\ [\mathscr{C}]$	Constraint application			

Figure 5. Syntax of the ρ_g-calculus.

3. The graph rewriting calculus

3.1. *The syntax of the ρ_g-calculus*

The syntax of the ρ_g-calculus presented in Figure 5 extends the syntax of the standard ρ-calculus and of the ρ_x-calculus (Cirstea *et al.* 2005), that is, the ρ-calculus with explicit matching and substitution application. As in the plain ρ-calculus, λ-abstraction is generalised by a rule abstraction $P \twoheadrightarrow G$, where P, referred to as a *pattern*, is taken from a suitable subclass of terms. There are two different application operators: the functional application operator, denoted simply by concatenation, and the constraint application operator, denoted by '_ [_]'. Terms can be grouped together into *structures* built using the operator '_ ≀ _'.

As with the ρ_x-calculus, the ρ_g-calculus deals explicitly with matching constraints of the form $P \ll G$, but it also introduces a new kind of constraint, the recursion equations. A recursion equation is a constraint of the form $X = G$ 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 empty constraint is denoted by ϵ. The operator '_, _' is assumed to be associative, commutative and idempotent, with ϵ as neutral element. Hence, as we will see in the next section, the evaluation rules of the calculus are applied modulo this theory.

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, which are given, in order from highest to lowest priority, by the following list: '_ _'(application), '_ ↠ _', '_ ≀ _', '_ [_]' , '_ ≪ _', '_ = _' and '_, _'. The symbols $G, H, \ldots$ range over the set $\mathscr{G}$ of ρ_g-graphs; $x, y, z, \ldots$ range over the set $\mathscr{X}$ of variables; and a, b, c, d, e, f, g, h range over a set $\mathscr{K}$ of constants. The symbols $E, F, \ldots$ range over the set $\mathscr{C}$ of constraints.

We say a ρ_g-graph is *well-formed* if each variable occurs at most once on the left-hand side of a recursion equation. All the ρ_g-graphs considered in the rest of the paper will be implicitly assumed to be well-formed.

We use the symbol $\mathsf{Ctx}\{\Box\}$ for a context with exactly one hole $\Box$, and $\mathsf{Ctx}\{G\}$ for the ρ_g-graph obtained by filling the hole of such a context with G, which can be defined formally in the obvious way.

https://doi.org/10.1017/S0960129507006093 Published online by Cambridge University Press

Definition 10 (Order and cycle). We use $<$ to denote the least transitive relation on recursion variables such that $x > y$ if $\mathsf{Ctx}_1\{x\} \lll \mathsf{Ctx}_2\{y\}$ for some contexts $\mathsf{Ctx}_i\{\Box\}$, $i = 1, 2$, where the symbol $\lll$ can be the recursion operator $=$ or the match operator $\ll$. We say that x and y are *cyclically equivalent*, written $x \equiv y$, if $x > y > x$.

A ρ_g-graph G is said to be *acyclic* if the relation $>$ is a strict partial order[†] (and thus $\equiv$ is the empty relation). It is said to be *cyclic* otherwise, that is, if there exist two variables x and y in G such that $x \equiv y$.

Observe that any cyclic ρ_g-graph will contain a *cycle*, that is, a sequence 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 $n, m \in \mathbb{N}$, where $x_0 \equiv x_1 \equiv \ldots \equiv x_n$.

We use $\bullet$ (black hole) to denote a constant that represents 'undefined' ρ_g-graphs corresponding to the expression $x\ [x = x]$ (self-loop). This notation was introduced by Ariola and Klop (Ariola and Klop 1996), using the equational approach, and by Corradini (Corradini 1993), using the categorical approach. The notation $x =_\circ x$ is again an abbreviation for the sequence $x = x_1, \ldots, x_n = x$.

Definition 11 (Algebraic ρ_g-graphs and patterns). We say that the ρ_g-graphs defined by the following grammar are *algebraic*:

$$\mathscr{A} \ ::= \ \mathscr{X} \ \mid \ \mathscr{K} \ \mid \ (((\mathscr{K}\ \mathscr{A})\ \mathscr{A})\ldots)\ \mathscr{A} \ \mid \ \mathscr{A}\ [\mathscr{X} = \mathscr{A}, \ldots, \mathscr{X} = \mathscr{A}]\,.$$

An algebraic term of the form $(((f\ \ G_1)\ \ G_2)\ldots)\ \ G_n$ will usually be written as $f(G_1, G_2, \ldots, G_n)$.

A *pattern* is an algebraic acyclic ρ_g-graph.

For the purposes of this paper we will assume that all terms appearing as left-hand sides of abstractions and constraints (set $\mathscr{P}$ in the syntax) are patterns. For instance, the ρ_g-graph $(f(y)\ [y = g(y)] \twoheadrightarrow a)$ is not allowed in our calculus since the abstraction has a cyclic left-hand side.

The notions of free and bound variables of ρ_g-graphs take into account the three binders of the calculus: abstraction, recursion and match. Intuitively, variables that occur free in the left hand-side of any of these operators bind the occurrences of the same variable in the right-hand side of the operator.

Given a constraint $\mathscr{C}$, we will also refer to the set $\mathscr{DV}(\mathscr{C})$ of variables 'defined' in $\mathscr{C}$. This set includes, for any recursion equation $x = G$ in $\mathscr{C}$, the variable x and for any matching equation $P \ll G$ in $\mathscr{C}$, the set of free variables of P.

[†] Recall that a *strict partial order* is a transitive and irreflexive relation.

https://doi.org/10.1017/S0960129507006093 Published online by Cambridge University Press

Definition 12 (Free, bound and defined variables). Given a ρ_g-graph G, its free variables, denoted $\mathscr{FV}(G)$, and its bound variables, denoted $\mathscr{BV}(G)$, are recursively defined below:

G	$\mathscr{BV}(G)$	$\mathscr{FV}(G)$
x	$\varnothing$	$\{x\}$
k	$\varnothing$	$\varnothing$
$G_1\ G_2$	$\mathscr{BV}(G_1) \cup \mathscr{BV}(G_2)$	$\mathscr{FV}(G_1) \cup \mathscr{FV}(G_2)$
$G_1 \wr G_2$	$\mathscr{BV}(G_1) \cup \mathscr{BV}(G_2)$	$\mathscr{FV}(G_1) \cup \mathscr{FV}(G_2)$
$G_1 \multimap G_2$	$\mathscr{FV}(G_1) \cup \mathscr{BV}(G_1) \cup \mathscr{BV}(G_2)$	$\mathscr{FV}(G_2) \setminus \mathscr{FV}(G_1)$
$G\ [\mathscr{C}]$	$\mathscr{BV}(G) \cup \mathscr{BV}(\mathscr{C})$	$(\mathscr{FV}(G) \cup \mathscr{FV}(\mathscr{C})) \setminus \mathscr{DV}(\mathscr{C})$

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

C	$\mathscr{BV}(C)$	$\mathscr{FV}(C)$	$\mathscr{DV}(C)$
ϵ	$\varnothing$	$\varnothing$	$\varnothing$
$x = G$	$\{x\} \cup \mathscr{BV}(G)$	$\mathscr{FV}(G)$	$\{x\}$
$G_1 \ll G_2$	$\mathscr{FV}(G_1) \cup \mathscr{BV}(G_1) \cup \mathscr{BV}(G_2)$	$\mathscr{FV}(G_2)$	$\mathscr{FV}(G_1)$
C_1, C_2	$\mathscr{BV}(C_1) \cup \mathscr{BV}(C_2)$	$\mathscr{FV}(C_1) \cup \mathscr{FV}(C_2)$	$\mathscr{DV}(C_1) \cup \mathscr{DV}(C_2)$

This definition generalises Definition 1. It is worth noting that the set of bound variables in $G\ [E]$ includes the domain of E and the bound variables of G. For example, the set of bound variables in the term $(f(y) \multimap y)\ (g(x,z)\ [x \ll f(a)])$ is $\{x, y\}$. Note also that the visibility of a recursion variable is limited to the ρ_g-graphs appearing in the list of constraints in which the recursion variable is defined and the ρ_g-graph 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 $y = a$ binds its occurrence in $g(y)$ but not in $f(x,y)$. To avoid confusion and guarantee that free and bound variables always have different names in a ρ_g-graph, we work modulo *α-conversion* and use Barendregt's *'hygiene-convention'*. Using α-conversion, the previous term becomes $f(x,y)\ [x = g(z)\ [z = a]]$, where it is clear that the variable y is free. The operation of α-conversion is also used for defining a capture-free substitution operation over ρ_g-graphs.

We will now give some examples describing the visibility of bound variables and the need for renaming variables in order to help the understanding of the notion of free (and bound) variables, which are more elaborate than usual due to the presence of different binders in the calculus and to the fact that the sets of variables of the different constraints in a list are not necessarily disjoint.

Example 13 (Free and bound variables should not have the same name). Given the ρ_g-graph $z\ [z = x \multimap y, y = x + x]$, one might naively think to replace the variable y by $x + x$ in the right-hand side of the abstraction, which would lead to a variable capture.

https://doi.org/10.1017/S0960129507006093 Published online by Cambridge University Press

This could happen because the previous term does not respect our naming conventions: the variable capture is no longer possible if we consider the ρ_g-graph $z\ [z = x_1 \twoheadrightarrow y, y = x + x]$ obtained after α-conversion. In order to have the occurrences of the variable x appearing in the second constraint bounded by the abstraction, we should use a nested constraint as in the ρ_g-graph $z\ [z = x \twoheadrightarrow (y\ [y = x + x])]$.

Example 14 (Different bound variables should have different names). According to the notions of free and bound variable, there cannot be any sharing in a term between the left-hand side of the rewrite rules and the rest of a ρ_g-graph. In other words, the left-hand side of a rewrite rule is self-contained. Sharing inside the left-hand side is allowed, and no restrictions are imposed on the right-hand side.

For example, in $f(y, y \twoheadrightarrow g(y))\ [y = x]$, 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-graph should in fact be written (by α-conversion) as $f(y, z \twoheadrightarrow g(z))\ [y = x]$.

Notice also that it is not possible to express sharing between the left and right hand sides of an abstraction. For example, in the term $(x \twoheadrightarrow x)\ [x = a]$, the variable x in the left-hand side of the abstraction is bound by the x in the right-hand side, and thus the term can be α-converted to $(z \twoheadrightarrow z)\ [x = a]$, which respects the naming convention.

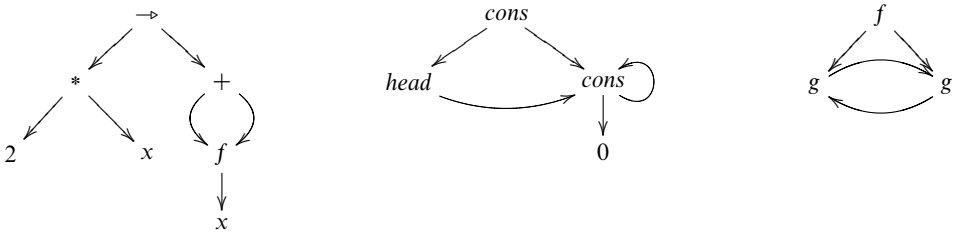
These naming conventions allow us to consider suitable representatives in α-equivalence classes of ρ_g-graphs, over which replacements (like those required by the evaluation rules in Figure 7) can be performed quite straightforwardly, disregarding the problem of variable captures.

In order to support the intuition, in the rest of the paper we will sometimes provide a graphical representation of ρ_g-graphs that does not include matching constraints. Roughly speaking, any term without constraints can be represented as a tree in the obvious way, while a ρ_g-graph $G\ [x_1 = G_1, \ldots, x_n = G_n]$ can be read as a letrec construct $\texttt{letrec}\ x_1 = G_1, \ldots, x_n = G_n\ \texttt{in}\ G$ and represented as a structure with sharing and cycles. 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). The correspondence between a term and its graphical representation can be extended to general ρ_g-graphs, possibly including matching constraints, as described in Bertolissi (2005). In this paper we will only use this correspondence informally and for simple ρ_g-graphs not containing match equations.

Example 15 (Some ρ_g-graphs). The graphical representation of the following ρ_g-graphs is given in Figure 6:

1 In the rule $(2 * x) \twoheadrightarrow ((y + y)\ [y = f(x)])$ the sharing in the right-hand side avoids the copying of the object instantiating $f(x)$ when the rule is applied to a ρ_g-graph.
2 The ρ_g-graph $cons(head(x), x)\ [x = cons(0, x)]$ represents an infinite list of zeros. Notice that the recursion variable x binds the occurrence of x in the right-hand side $cons(0, x)$ of the constraint and those in the term $cons(head(x), x)$ to which the constraint is applied.

https://doi.org/10.1017/S0960129507006093 Published online by Cambridge University Press



(1) (2) (3)

Figure 6. Some ρ_g-graphs.

3 The ρ_g-graph $f(x, y)$ $[x = g(y), y = g(x)]$ is an example of 'twisted sharing', which can be expressed using mutually recursive constraints (to be read as a `letrec` construct). Here the preorder over variables is $x \geqslant y$ and $y \geqslant x$, and thus $x \equiv y$.

3.2. *The small-step semantics of ρ_g-calculus*

In the classical ρ-calculus, when reducing the application of a constraint to a term, that is, a delayed matching constraint, the corresponding matching problem is solved and the resulting substitutions are applied at the meta-level of the calculus. In the ρ_X-calculus, this reduction is decomposed into two phases, one computing substitutions and the other describing the application of these substitutions. Matching computations leading from constraints to substitutions and the application of the substitutions are clearly separated and made explicit. In the ρ_g-calculus, the computation of the substitutions solving a matching constraint is performed explicitly and, if the computation is successful, the result is a recursion equation added to the list of the term's constraints. This means that the substitution is not applied immediately to the term but kept in the environment for a possible delayed application.

The complete set of evaluation rules of the ρ_g-calculus is presented in Figure 7. As in the plain ρ-calculus, in the ρ_g-calculus the application of a rewrite rule to a term is represented as the application of an abstraction. A redex can be 'activated' using the ρ rule in the BASIC RULES, which creates the corresponding matching constraint. The computation of the substitution that solves the matching is then performed explicitly by the MATCHING RULES and, if the computation is successful, the result is a recursion equation added to the list of the term's constraints. This means that the substitution is not applied immediately to the term but is kept in the environment for a delayed application or for deletion if useless, as expressed by the GRAPH RULES.

In more detail, the first two rules ρ and δ are typical of the ρ-calculus. The ρ rule triggers the application of a rewrite rule to a ρ_g-graph by applying the appropriate constraint to the right-hand side of the rule. The δ rule distributes the application over the structures built with the '$\wr$' operator. For each of these rules, an additional rule dealing with the presence of constraints is considered. Without these rules, the application of abstraction ρ_g-graphs like R $[x = R]$ $f(a)$, where $R = f(y) \twoheadrightarrow x\ f(y)$ (which can encode a recursive application as in Example 22), could not be reduced. An alternative solution would be

https://doi.org/10.1017/S0960129507006093 Published online by Cambridge University Press

BASIC RULES:

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

MATCHING RULES:

$$
\begin{array}{lll}
(\textit{propagate})\ \ P \ll (G\ [E]) & \to_p & P \ll G, E \quad \textit{if } P \notin \mathscr{X} \\
(\textit{decompose})\ \ K(G_1,\ldots,G_n) \ll K(G'_1,\ldots,G'_n) & \to_{dk} & G_1 \ll G'_1, \ldots, G_n \ll G'_n \\
 & & \textit{with } n \geqslant 0 \\
(\textit{solved})\ \ x \ll G, E & \to_s & x = G, E \quad \textit{if } x \notin \mathscr{DV}(E)
\end{array}
$$

GRAPH RULES:

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

Figure 7. Small-step semantics of the ρ_g-calculus.

to introduce appropriate distributivity rules, but this approach is not considered in this paper.

The MATCHING RULES and, in particular, the *decompose* rule are strongly related to the theory modulo which we want to compute the solutions of the matching. In this paper we consider the syntactic matching, which is known to be decidable for finite as well as regular trees (Colmerauer 1984; Courcelle 1980), but extensions to more elaborate theories are possible, for example, to take care of associativity or commutativity. Due to the assumptions on the left-hand sides of abstractions and constraints, 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 match equation. Some replacements might be needed (as defined by the GRAPH RULES) when the terms contain some sharing. The *propagate* rule flattens a list of constraints, thus propagating such constraints to a higher level. Note that, since left-hand sides of match equations are acyclic, there is no need for an evaluation rule propagating the constraints from the left-hand side of a match equation: the substitution and garbage collection rules can be used to obtain the same result. The algebraic terms are decomposed and the trivial constraints $K \ll K$ are eliminated. The *solved* rule transforms a matching constraint $x \ll G$ into a recursion equation $x = G$. The proviso requiring that x is not defined elsewhere in the constraint is necessary in the case of matching problems involving non-linear constraints. For example, the constraint $x \ll a, x \ll b$ should not be reduced, showing that the original (non-linear) matching problem has no solution.

https://doi.org/10.1017/S0960129507006093 Published online by Cambridge University Press

The GRAPH RULES are inherited from the cyclic λ-calculus (Ariola and Klop 1997). The *sub*(stitution) rules copy a ρ_g-graph associated with a recursion variable into a term inside the scope of the corresponding constraint. This is important for making a redex explicit (for example, in $x\ a\ [x = a \twoheadrightarrow b]$) or for solving a match equation (for example, in $a\ [a \ll x, x = a]$). As already mentioned, the *acyclic sub* rule only allows one to make the copies upwards with respect to the order defined on the variables of ρ_g-graphs. In the cyclic λ-calculus this is needed for the confluence of the system (see Ariola and Klop (1997) for a counterexample) and it will also be essential when proving the confluence of the ρ_g-calculus. Without this condition, confluence is broken as one can see for the ρ_g-graph $z_1\ [z_1 = x \twoheadrightarrow z_2\ s(x), z_2 = y \twoheadrightarrow z_1\ s(y)]$, which reduces to either $z_1\ [z_1 = x \twoheadrightarrow z_1\ s(s(x))]$ or $z_1\ [z_1 = x \twoheadrightarrow z_2\ s(x), z_2 = y \twoheadrightarrow z_2\ s(s(y))]$. The *garbage* rule gets rid of recursion equations whose left-hand side variables do not appear in the scope of the equation itself (intuitively, they represent non-connected parts of the ρ_g-graph). Matching constraints are not eliminated, thereby keeping track of matching failures during an unsuccessful reduction. The *black hole* rules replace the undefined ρ_g-graphs, which correspond to self-loop graphs, with the constant $\bullet$.

As we have already mentioned, for the purposes of this paper we will consider some restrictions on the terms of the ρ_g-calculus and the subsequent matching. These restrictions are summarised in the definition below.

Definition 16 (Algebraic ρ_g-calculus). The *algebraic* ρ_g*-calculus* is the ρ_g-calculus with syntactic matching, where the terms in the left-hand side of abstractions and all constraints are patterns.

From now on, the qualification 'algebraic' will often be omitted, and we will simply write ρ_g-calculus for algebraic ρ_g-calculus.

Definition 17 (Rewrite relations). We use $\mapsto_{\mathcal{M}}$ and $\mapsto_{\rho g}$ to denote the one-step relations induced by the subset of MATCHING RULES and the whole set of rules given in Figure 7, respectively. Analogously, we use $\mapsto\!\!\!\!\rightarrow_{\mathcal{M}}$ and $\mapsto\!\!\!\!\rightarrow_{\rho g}$ to denote the corresponding multistep relations.

Note that all the evaluation steps are performed modulo the underlying theory associated with the conjunction operator '$\_,\_$'.

With a view to a future efficient implementation of the calculus, it would be interesting to study suitable strategies that delay the application of the *external sub* and *acyclic sub* substitution rules in order to keep hold of the sharing information as long as possible. Basically, the idea consists of applying the substitution rules only if needed for generating new redexes for the BASIC RULES and to possibly unfreeze match equations where otherwise the computation of the matching is stuck. In addition, substitution rules can be used to 'remove' trivial recursion equations of the kind $x = y$.

Definition 18. The evaluation strategy *SharingStrat* is defined as follows. All the evaluation rules apart from *external sub* and *acyclic sub* are applicable without any restriction. The *external sub* and *acyclic sub* rules are applicable only if their application causes:

https://doi.org/10.1017/S0960129507006093 Published online by Cambridge University Press

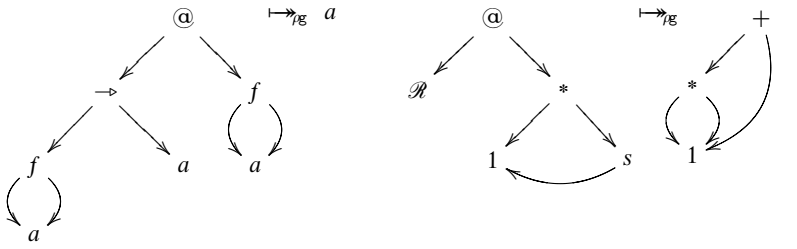


Figure 8. Examples of reductions.

— the instantiation of an active variable by an abstraction or a structure; or
— the instantiation of a variable in a stuck match equation; or
— the instantiation of a variable by a variable.

This strategy is followed in the following examples. Note that the theory underlying the constraint conjunction operator ',' is used during the reduction.

Example 19 (A simple reduction with sharing). A graphical representation is given in Figure 8(a).

$$
\begin{aligned}
(f(x,x)\ [x=a] \twoheadrightarrow a)\ (f(y,y)\ [y=a]) \ &\mapsto_{\rho}\ a\ [f(x,x)\ [x=a] \ll f(y,y)\ [y=a]] \\
&\mapsto\!\!\!\!\twoheadrightarrow_{es}\ a\ [f(a,a)\ [x=a] \ll f(y,y)\ [y=a]] \\
&=\ a\ [f(a,a)\ [x=a,\epsilon] \ll f(y,y)\ [y=a]] \\
&\qquad\qquad (\textit{by the neutral element axiom}) \\
&\mapsto_{gc}\ a\ [f(a,a)\ [\epsilon] \ll f(y,y)\ [y=a]] \\
&\mapsto_{gc}\ a\ [f(a,a) \ll f(y,y)\ [y=a]] \\
&\mapsto_{p}\ a\ [f(a,a) \ll f(y,y), y=a] \\
&\mapsto_{dk}\ a\ [a \ll y, a \ll y, y=a] \\
&=\ a\ [a \ll y, y=a] \qquad (\textit{by idempotency}) \\
&\mapsto_{ac}\ a\ [a \ll a, y=a] \\
&\mapsto_{dk}\ a\ [y=a] \\
&=\ a\ [y=a,\epsilon] \qquad (\textit{by the neutral element axiom}) \\
&\mapsto_{gc}\ a\ [\epsilon] \\
&\mapsto_{gc}\ a
\end{aligned}
$$

Example 20 (Multiplication). We will use an infix notation for the constant '$*$'. The following ρ_g-term corresponds to the application of the rewrite rule $\mathscr{R} = x * s(y) \twoheadrightarrow (x * y + x)$ to the term $1 * s(1)$ where the constant 1 is shared. The result is shown graphically in Figure 8(b).

$$
\begin{aligned}
(x * s(y) \twoheadrightarrow (x * y + x))\ (z * s(z)\ [z=1]) \ &\mapsto_{\rho}\ x * y + x\ [x * s(y) \ll (z * s(z)\ [z=1])] \\
&\mapsto_{p}\ x * y + x\ [x * s(y) \ll z * s(z), z=1]
\end{aligned}
$$

https://doi.org/10.1017/S0960129507006093 Published online by Cambridge University Press

$$\begin{aligned}
&\mapsto\!\!\!\!\rightarrow_{dk} \; x * y + x \; [x \ll z, y \ll z, z = 1] \\
&\mapsto\!\!\!\!\rightarrow_{s} \; x * y + x \; [x = z, y = z, z = 1] \\
&\mapsto\!\!\!\!\rightarrow_{es} \; (z * z + z) \; [x = z, y = z, z = 1] \\
&\mapsto\!\!\!\!\rightarrow_{gc} \; (z * z + z) \; [z = 1]
\end{aligned}$$

Note that the term $(z * z + z)\ [z = 1]$ is in normal form with respect to the strategy SharingStrat, but can be reduced to $(1 * 1 + 1)$ if no evaluation strategy is used.

Example 21 (Non-linearity). The matching involving non-linear patterns can lead to a normal form that is either a constraint consisting only of recursion equations, representing a successful matching:

$$\begin{aligned}
f(y, y) \ll f(a, a) &\mapsto_{dk} \; y \ll a \quad (\textit{by idempotency}) \\
&\mapsto_{s} \; y = a
\end{aligned}$$

or a constraint that contains some match equations, representing a matching failure:

$$f(y, y) \ll f(a, b) \mapsto_{dk} \; y \ll a, y \ll b$$

Example 22 (Fixed point combinator). Consider the term rewrite rule $R_Y = Y\ x \to x\ (Y\ x)$, which expresses the behaviour of the fixed point combinator Y of the λ-calculus. Given a term t, we have the infinite rewrite sequence

$$Y\ t \;\to_{R_Y}\; t\ (Y\ t) \;\to_{R_Y}\; t\ (t\ (Y\ t)) \;\to_{R_Y}\; \ldots$$

which, in the sense formalised in Kennaway *et al.* (1995) and Corradini (1993), converges to the infinite term $t\ (t\ (t\ (\ldots)))$.

We can represent the Y-combinator in the ρ_{g}-calculus as the following term:

$$Y \triangleq x_0\ [x_0 = x \twoheadrightarrow x\ (x_0\ x)].$$

If we define $R = x \twoheadrightarrow x\ (x_0\ x)$, we have the following reduction:

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

Continuing the reduction, this will 'converge' to the term of Figure 9(a).

We can have a more efficient implementation of the same term reduction using a method introduced by Turner (Turner 1979), which models the rule R_Y by means of the cyclic term depicted in Figure 9(b). In the ρ_{g}-calculus this gives the ρ_{g}-graph

$$Y_T \triangleq x \twoheadrightarrow (z\ [z = x\ z]).$$

https://doi.org/10.1017/S0960129507006093 Published online by Cambridge University Press

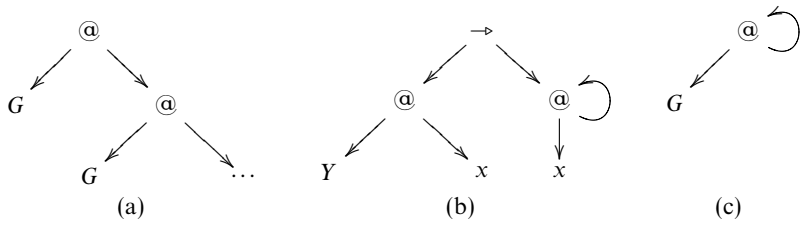


Figure 9. Example of reductions.

The reduction in this case is

$$\begin{aligned} Y_T\ G &\mapsto_{\rho}\ z\ [z = x\ z]\ [x \ll G] \\ &\mapsto_{s}\ z\ [z = x\ z]\ [x = G] \\ &\mapsto_{es}\ z\ [z = G\ z]\ [x = G] \\ &\mapsto\!\!\!\!\!\rightarrow_{gc}\ z\ [z = G\ z]. \end{aligned}$$

The resulting ρ_{g}-graph is depicted in Figure 9(c). If we 'unravel' (in the intuitive sense) this cyclic ρ_{g}-graph, we obtain the infinite term shown in Figure 9(a).

This means that a finite sequence of rewritings on cyclic ρ_{g}-graphs can correspond to an infinite reduction sequence on the corresponding acyclic term.

4. Confluence of the ρ_{g}-calculus

As we mentioned earlier, rewriting in the ρ_{g}-calculus is performed over equivalence classes of ρ_{g}-graphs modulo the theory associated with the constraint conjunction operator. This fact must be taken into account when analysing the confluence of the rewriting relation. In the next section we formally introduce the notion of rewriting modulo an equational theory and some related properties. Then we prove the confluence of the ρ_{g}-calculus: we begin by sketching an outline of our proof technique and then give the details.

4.1. *Equational rewriting*

Given a set of equations E over a set of terms $\mathscr{T}$, we use $\sim_E^1$ to denote the one-step equality, that is, for any context $\mathsf{Ctx}\{\Box\}$ and any substitution σ, if $T_1 = T_2$ is an equation in E, then $\mathsf{Ctx}\{\sigma(T_1)\} \sim_E^1 \mathsf{Ctx}\{\sigma(T_2)\}$. Let $\sim_E$ be the reflexive, symmetric and transitive closure of $\sim_E^1$ over the set $\mathscr{T}$; two terms T_1 and T_2 are said to be equivalent modulo E if $T_1 \sim_E T_2$.

We next define a notion of rewriting where the rewrite rules act over equivalence classes of terms modulo $\sim_E$ (Lankford and Ballantyne 1977; Huet 1980). This approach is rather general, but might be very inefficient since in order to reduce a given term all the terms in the same equivalence class must be taken into consideration, and such a class can be quite large or even infinite. A possible refinement of this reduction relation is studied in Peterson and Stickel (1981) and Jouannaud and Kirchner (1986), and called rewriting modulo E. Using this notion of reduction, the rules apply to terms rather than to equivalence classes, but matching modulo E is performed at each step of the reduction.

https://doi.org/10.1017/S0960129507006093 Published online by Cambridge University Press

Strong normalisation	SN	*no infinite* $\mapsto_S$ *reductions*
Diamond property modulo E	$D_{\sim}$	$\leftarrow_S \cdot \mapsto_S \subseteq \mapsto_S \cdot \sim_E \cdot \leftarrow_S$
Local confluence modulo E	$LCON_{\sim}$	$\leftarrow_S \cdot \mapsto_S \subseteq \mapsto_S^* \cdot \sim_E \cdot \leftarrow_S^*$
Confluence modulo E	$CON_{\sim}$	$\leftarrow_S^* \cdot \mapsto_S^* \subseteq \mapsto_S^* \cdot \sim_E \cdot \leftarrow_S^*$
Church–Rosser modulo E	$CR_{\sim}$	$\leftrightarrow_{S \cup E}^* \subseteq \mapsto_S^* \cdot \sim_E \cdot \leftarrow_S^*$
Commutation modulo E	$COM_{\sim}$	$\leftarrow_{S_1}^* \cdot \mapsto_{S_2}^* \subseteq \mapsto_{S_2}^* \cdot \sim_E \cdot \leftarrow_{S_1}^*$
Strong commutation modulo E	$SCOM_{\sim}$	$\leftarrow_{S_1} \cdot \mapsto_{S_2} \subseteq \mapsto_{S_2}^{0/1} \cdot \sim_E \cdot \leftarrow_{S_1}^*$
Compatibility with E	$CPB_{\sim}$	$\leftarrow_S \cdot \sim_E \subseteq \sim_E \cdot \leftarrow_S$
Coherence with E	$CH_{\sim}$	$\leftarrow_S^* \cdot \sim_E \subseteq \mapsto_S^* \cdot \sim_E \cdot \leftarrow_S^*$

Figure 10. Properties of rewriting modulo E.

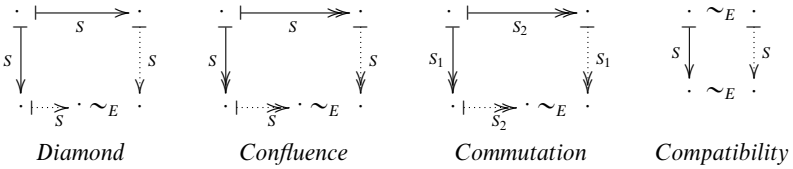


Figure 11. Properties of rewriting modulo E, graphically.

Definition 23 (E-class rewriting and rewriting modulo E). Let $\mathscr{R}$ be a set of rewrite rules and E be a set of equations over a set of terms $\mathscr{T}$. Let $L \to R \in \mathscr{R}$ be a rewrite rule and σ be a substitution. Then:

1 A term T_1 E-class rewrites to a term T_2, denoted $T_1 \mapsto_{\mathscr{R}/E} T_2$ iff $T_1 \sim_E \mathsf{Ctx}\{\sigma(L)\}$ and $T_2 \sim_E \mathsf{Ctx}\{\sigma(R)\}$.
2 A term T_1 rewrites modulo E to a term T_2, denoted $T_1 \mapsto_{\mathscr{R},E} T_2$ iff $T_1 = \mathsf{Ctx}\{T\}$ with $T \sim_E \sigma(L)$ and $T_2 = \mathsf{Ctx}\{\sigma(R)\}$.

Given a rewriting relation $\mapsto_S$, we use $\mapsto_S^*$ to denote its reflexive and transitive closure, and $\leftrightarrow_S^*$ for its symmetric, reflexive and transitive closure. A zero- or one-step reduction is denoted by $\mapsto_S^{0/1}$. Figure 10 collects the definitions of several classical properties of term rewrite systems, with some generalised to rewriting modulo a set of axioms. We will write $PROP(S)$ if a property $PROP$ holds for a relation $\mapsto_S$. Some of these properties are represented graphically in Figure 11.

It is easy to see that $CPB_{\sim_E^1}$ and $CPB_{\sim_E}$ coincide and that $CPB_{\sim}$ implies $CH_{\sim}$. Several other relationships between the above properties are stated and proved in Ohlebusch (1998), but here we will just recall two propositions concerning confluence. Generally speaking, compared to standard rewriting, in order to have confluence for rewriting modulo a set of equations E, an additional compatibility property of the rewrite system with respect to the congruence relation generated by E comes into play.

https://doi.org/10.1017/S0960129507006093 Published online by Cambridge University Press

Proposition 24 (Ohlebusch 1998).

1 A terminating relation S that is locally confluent modulo E and compatible with E is confluent modulo E.
2 The union of two relations S_1 and S_2 commuting modulo E that are both confluent modulo E and compatible with E is confluent modulo E.

In the following we will also need the following two properties that ensure the commutation of two sets of rewrite rules.

Proposition 25.

1 If S_1 and S_2 verify the property $\leftarrowtail_{S_1} \cdot \twoheadrightarrow_{S_2} \subseteq \twoheadrightarrow_{S_2} \cdot \sim_E \cdot \twoheadleftarrow_{S_1}$ (denoted $PR_{\sim}(S_1, S_2)$) and are compatible with E, then S_1 and S_2 commute modulo E.
2 Two strongly commuting relation S_1 and S_2 compatible with E commute modulo E.

Proof. Point (1) is proved by induction on the number of steps of S_1. Point (2) follows by using (1) and an induction on the number of steps of S_2. □

4.2. *General outline of the confluence result*

The confluence for higher-order systems dealing with non-linear matching is a difficult issue since, as shown by Klop in the setting of the λ-calculus (Klop 1980), we usually obtain non-joinable critical pairs. Klop's counterexample can be encoded in the ρ-calculus (Wack 2005), showing that the non-linear ρ-calculus is not confluent if no evaluation strategy is imposed on the reductions. The counterexample is still valid when generalising the ρ-calculus to the ρ_g-calculus, so in the following we consider a version of the ρ_g-calculus with some linearity assumptions.

Definition 26 (Linear ρ_g-calculus). The class of (algebraic) *linear patterns* is defined by

$$\mathscr{L} \ ::= \ \mathscr{X} \ \mid \ \mathscr{K} \ \mid \ (((\mathscr{K} \ \mathscr{L}_0) \ \mathscr{L}_1)\ldots) \ \mathscr{L}_n \ \mid \ \mathscr{L}_0 \ [\mathscr{X}_1 = \mathscr{L}_1, \ldots, \mathscr{X}_n = \mathscr{L}_n]$$

where we assume that $FV(\mathscr{L}_i) \cap FV(\mathscr{L}_j) = \varnothing$ for $i \neq j$. A constraint $[L_1 \lll G_1, \ldots, L_n \lll G_n]$, where $\lll \in \{=, \ll\}$, is *linear* if all patterns $L_1, \ldots, L_n$ are linear and $FV(L_i) \cap FV(L_j) = \varnothing$ for $i \neq j$. The *linear ρ_g-calculus* is the ρ_g-calculus in which all the patterns in the left-hand side of abstractions and all constraints are linear.

It is easy to see that the class of ρ_g-graphs in the linear ρ_g-calculus is closed under reduction.

In the general ρ_g-calculus, the operator '$\_, \_$' is supposed to be associative, commutative and idempotent, with the empty list of constraints ϵ as neutral element. However, in the linear ρ_g-calculus, idempotency is not needed since constraints of the form $x \ll G, x \ll G$ are not allowed (and cannot arise from reductions). Therefore, in the linear ρ_g-calculus, rewriting can be thought of as acting over equivalence classes of ρ_g-graphs with respect to the congruence relation, which is denoted by $\sim_{AC1}$ or simply $AC1$, generated by the associativity, commutativity and neutral element axioms for the '$\_, \_$' operator.

Following the notation in Definition 23, the rewriting relation induced over $AC1$-equivalence classes is denoted by $\mapsto_{\rho_g/AC1}$. Concretely, in most of the proofs we will use

https://doi.org/10.1017/S0960129507006093 Published online by Cambridge University Press

the notion of rewriting modulo $AC1$ (Peterson and Stickel 1981), denoted by $\mapsto_{\rho_g,AC1}$. On the one hand, as already mentioned, this notion of rewriting is more convenient from a computational point of view than $AC1$-class rewriting. On the other hand, as we will see in Section 4.3, under suitable assumptions satisfied by our calculus, the confluence of the relation $\rho_g, AC1$ implies the confluence of the $\rho_g/AC1$ relation.

According to the definition of $\mapsto_{\rho_g,AC1}$, matching modulo $AC1$ is performed at each evaluation step. Note that matching modulo $AC1$ may lead to infinitely many solutions, but the complete set of solutions is finitary and has a canonical representative in which terms are normalised with respect to the neutral element (Kirchner 1990).

The confluence proof is quite elaborate, so we will decompose it into a number of lemmas to achieve the final result. Its complexity is mainly due to the non-termination of the system and to the fact that equivalence modulo $AC1$ on ρ_{g}-graphs has to be considered throughout the proof.

We start by proving a fundamental compatibility lemma showing that the ρ_{g}-calculus rewrite relation is particularly well behaved with respect to the congruence relation $AC1$. This lemma ensures that if there exists a rewrite step from a ρ_{g}-graph G, then the 'same' step can be performed starting from any term $AC1$-equivalent to G.

Lemma 27 (Compatibility of ρ_g). Compatibility with $AC1$ holds for any rule r of the ρ_{g}-calculus:

$$\leftarrow_{r,AC1} \cdot \sim_{AC1} \subseteq \sim_{AC1} \cdot \leftarrow_{r,AC1}$$

Proof. We use case analysis on the rules of the ρ_{g}-calculus. Consider, for instance, the diagram for the *acyclic sub* rule with a commutation step:

$$\begin{array}{ccc} G\ [G_0 \lll \mathsf{Ctx}\{y\}, y = G_1, F] & \sim^1_{AC1} & G\ [y = G_1, G_0 \lll \mathsf{Ctx}\{y\}, F] \\ \Big\downarrow{\scriptstyle ac,AC1} & & \Big\downarrow{\scriptstyle ac,AC1} \\ G\ [G_0 \lll \mathsf{Ctx}\{G_1\}, y = G_1, F] & \sim_{AC1} & G\ [y = G_1, G_0 \lll \mathsf{Ctx}\{G_1\}, F] \end{array}$$

A different order of the constraints in the list does not prevent the application of the *acyclic sub* rule, thanks to the fact that matching is performed modulo $AC1$. Moreover, the extension variable E in the definition of the *acyclic sub* rule ensures the applicability of the rule to ρ_{g}-graphs having an arbitrary number of constraints in the list. In particular, the extension variable E can be instantiated by ϵ if the term to be reduced is simply $G\ [G_0 \lll \mathsf{Ctx}\{y\}, y = G_1]$. In this case the application of the rule is possible since there exists a match of the *acyclic sub* rule in the term $[G_0 \lll \mathsf{Ctx}\{y\}, y = G_1, \epsilon]$, which is equivalent to the given term using the neutral element axiom.

So it is easy to close the diagram. The same reasoning can be applied for the other rules of the ρ_{g}-calculus. The extension to several steps of $\sim_{AC1}$ holds trivially. □

Note that since compatibility holds for any rule of the ρ_{g}-calculus, it also holds for any subset of rules, including the entire set of rules of the ρ_{g}-calculus.

In order to prove the confluence of $\mapsto\!\!\!\!\rightarrow_{\rho_g,AC1}$, we use a technique inspired by the method adopted for proving confluence of the cyclic λ-calculus (Ariola and Klop 1997). The larger

https://doi.org/10.1017/S0960129507006093 Published online by Cambridge University Press

number of evaluation rules of the ρ_g-calculus and the explicit treatment of the congruence relation on ρ_g-graphs make the proof for the ρ_g-calculus much more elaborate.

The main idea of the proof is to split the rules into two subsets and show the confluence of each of the subsets separately. We then prove the confluence of their union using a commutation lemma for the two sets of rules.

So we begin by dividing the ρ_g-calculus rules into the following two subsets:

— Σ-*rules*, including the substitution rules *external sub* and *acyclic sub*, plus the δ rule;
— τ-*rules*, including all the remaining rules (that is, ρ, *propagate*, *decompose*, *solved*, *garbage*, *black hole*).

The Σ-rules include the substitution rules, which represent the 'non-terminating part' of the ρ_g-calculus. The δ rule is also included in the Σ-rules, though it could be safely added to the τ-rules keeping this set of rules terminating. This choice is motivated by the fact that, because of its non-linearity, adding the δ rule to the τ-rules would have caused problems in the proof of the final commutation lemma (Lemma 45).

The confluence proof of $\twoheadrightarrow_{\rho_g,AC1}$ is in three parts:

— *Confluence modulo AC1 of the relation induced by the τ-rules.*
This is done by using the fact that a relation that is strongly normalising and locally confluent modulo *AC1* is confluent modulo *AC1* if the compatibility property holds (see Proposition 24). To prove the strong normalisation, we use a polynomial interpretation of the ρ_g-calculus. Local confluence modulo *AC1* is easy to prove by analysis of the critical pairs.
— *Confluence modulo AC1 of the relation induced by the Σ-rules.*
This is the most complex part of the proof. The idea is to exploit the *complete development method* of the λ-calculus by defining a terminating version of the relation induced by the Σ rules (the development) and using its properties to deduce the confluence of the original rewrite relation.
— *Confluence modulo AC1 of the relation induced by the union of the two sets.*
General confluence holds since we can prove the commutation modulo *AC1* of the two relations.

From the confluence of the relation $\twoheadrightarrow_{\rho_g,AC1}$ we can deduce the confluence of the relation $\twoheadrightarrow_{\rho_g/AC1}$ acting on *AC1*-equivalence classes of ρ_g-graphs. This is a consequence of the fact that the compatibility with *AC1* property holds for the rules of the ρ_g-calculus.

In the following, to lighten the notation we will simply write *AC1* or $\sim$ for $\sim_{AC1}$ and $\twoheadrightarrow_R$ for $\twoheadrightarrow_{R,AC1}$, where R may be any subset of rules of the ρ_g-calculus.

The outline of the proof is depicted in Figure 12, where all the lemmas are mentioned, except for the compatibility lemma, which is left implicit, since it is used for almost all the intermediate results.

4.3. *The complete confluence proof*

Confluence modulo AC_ϵ for the τ-rules

The confluence modulo *AC1* for the relation $\twoheadrightarrow_\tau$ induced by the τ-rules is proved by showing that this relation is strongly normalising and locally confluent modulo *AC1*. In

https://doi.org/10.1017/S0960129507006093 Published online by Cambridge University Press

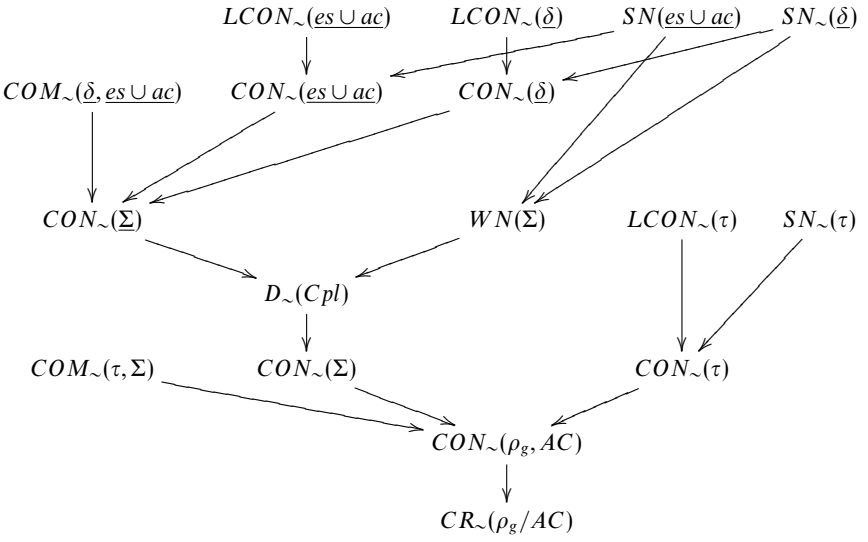


Figure 12. Confluence proof scheme.

the first part of this section we prove strong normalisation for $\mapsto_{\tau}$ by using a reduction order based on a polynomial interpretation of the ρ_g-graphs. In the second part of the section, the local confluence modulo $AC1$ is proved for the relation $\mapsto_{\tau}$ by case analysis of the critical pairs. On the basis of these results, we can then conclude that the relation $\mapsto_{\tau}$ is confluent.

We start by showing that $\mapsto_{\tau}$ is strongly normalising. In order to do this, we define a polynomial interpretation of the ρ_g-calculus syntax.

Definition 28 (Polynomial interpretation). We consider the following polynomial interpretation of ρ_g-graphs (assuming the standard order on natural numbers):

$$
\begin{aligned}
Size(\epsilon) &= 0 \\
Size(\bullet) &= 1 \\
Size(x) &= Size(f) = 2 \text{ for all } x \in \mathcal{X} \text{ and } f \in \mathcal{K} \setminus \{\bullet\} \\
Size(G_1 \wr G_2) &= Size(G_1) + Size(G_2) + 1 \\
Size(G_1\ G_2) &= Size(G_1) + Size(G_2) + 1 \\
Size(G_1 \ll G_2) &= Size(G_1) + Size(G_2) + 1 \\
Size(G_1 \multimap G_2) &= Size(G_1) + Size(G_2) + 2 \\
Size(G\ [E]) &= Size(G) + Size(E) + 1 \\
Size(E, E') &= Size(E) + Size(E') \\
Size(x = G) &= Size(x) + Size(G).
\end{aligned}
$$

https://doi.org/10.1017/S0960129507006093 Published online by Cambridge University Press

Note that the polynomial interpretation is compatible with respect to neutrality of ϵ for the constraint conjunction operator. In fact $E, \epsilon = E$ and $Size(E, \epsilon) = Size(E) + Size(\epsilon) = Size(E) + 0 = Size(E)$. Similarly, it is compatible with respect to the associativity and commutativity of the conjunction and with respect to α-conversion. Moreover, function $Size(\cdot)$ can be easily seen to be monotonic and closed under contexts.

Lemma 29 (Context closure). Let G_1 and G_2 be two ρ_g-graphs. If $Size(G_1) > Size(G_2)$, then $Size(\mathsf{Ctx}\{G_1\}) > Size(\mathsf{Ctx}\{G_2\})$, for all contexts $\mathsf{Ctx}\{\Box\}$.

Proof. Since addition is increasing on naturals, the lemma is clearly satisfied. □

We next show that for all the rules in τ, the polynomial interpretation of the left-hand side is larger than that of the right-hand side for any substitution of the (meta-)variables of the rule by positive naturals. As a consequence, we get the termination of the $\mapsto\!\!\!\!\!\rightarrow_\tau$ relation.

Lemma 30 (SN(τ)). The relation $\mapsto\!\!\!\!\!\rightarrow_\tau$ is strongly normalising.

Proof. Clearly, $Size(\cdot)$ associates a natural number to any constraint and ρ_g-graph (more precisely, $Size(\mathscr{C}) \geqslant 0$ for any constraint $\mathscr{C}$ and $Size(G) \geqslant 1$ for any ρ_g-graph G). Now, it is not difficult to check that for any rule $L \to R$ in τ, we have $Size(L) > Size(R)$ for all interpretations of the meta-variables of L and R over natural numbers. Hence, by Lemma 29, for all ρ_g-graphs G_1 and G_2 such that $G_1 \mapsto_\tau G_2$, we have $Size(G_1) > Size(G_2)$. Therefore the relation $\mapsto\!\!\!\!\!\rightarrow_\tau$ is strongly normalising. □

We next prove the local confluence modulo *AC1* of the relation $\mapsto\!\!\!\!\!\rightarrow_\tau$. This is done by inspection of the critical pairs generated by the τ-rules.

Lemma 31 (LCON$_\sim(\tau)$). The relation $\mapsto\!\!\!\!\!\rightarrow_\tau$ is locally confluent modulo *AC1*.

Proof. The proof is done by inspecting the possible critical pairs. The *decompose* rule and the *garbage* rule only generate trivial critical pairs with the other τ-rules. The ρ rule generates a joinable critical pair with the *black hole* rule as shown in the next diagram:

$$\begin{array}{ccc}
(P \twoheadrightarrow \mathsf{Ctx}\{x\})\ [x =_\circ x, E]\ G_3 & \xrightarrow[bh]{} & (P \twoheadrightarrow \mathsf{Ctx}\{\bullet\})\ [x =_\circ x, E]\ G_3 \\
\Big\downarrow{\scriptstyle \rho} & & \Big\Downarrow{\scriptstyle \rho} \\
\mathsf{Ctx}\{x\}\ [P \ll G_3, x =_\circ x, E] & \overset{}{\dashrightarrow}_{bh} & \mathsf{Ctx}\{\bullet\}\ [P \ll G_3, x =_\circ x, E]
\end{array}$$

The *propagate* rule generates a joinable critical pair with the *black hole* rule:

$$\begin{array}{ccc}
P \ll \mathsf{Ctx}\{x\}\ [x =_\circ x, E] & \xrightarrow[bh]{} & P \ll \mathsf{Ctx}\{\bullet\}\ [x =_\circ x, E] \\
\Big\downarrow{\scriptstyle p} & & \Big\Downarrow{\scriptstyle p} \\
P \ll \mathsf{Ctx}\{x\}, x =_\circ x, E & \overset{}{\dashrightarrow}_{bh} & P \ll \mathsf{Ctx}\{\bullet\}, x =_\circ x, E
\end{array}$$

https://doi.org/10.1017/S0960129507006093 Published online by Cambridge University Press

Finally, the *solved* rule generates a joinable critical pair with the *black hole* rule:

$$\begin{array}{ccc} y \ll \mathsf{Ctx}\{x\}, x =_\circ x, E & \xrightarrow[bh]{} & y \ll \mathsf{Ctx}\{\bullet\}, x =_\circ x, E \\ \Big\downarrow{\scriptstyle s} & & \Big\downarrow{\scriptstyle s} \\ y = \mathsf{Ctx}\{x\}, x =_\circ x, E & \xrightarrow[bh]{} & y = \mathsf{Ctx}\{\bullet\}, x =_\circ x, E \end{array}$$

□

As in standard term rewriting, we can use local confluence and strong normalisation to prove the confluence of a relation. By the fact that we consider (local) confluence modulo a set of equations, an additional compatibility property is needed to conclude the desired result, which corresponds to Newman's Lemma for rewriting modulo an equational theory.

Proposition 32 (CON$_\sim(\tau)$). The relation $\mapsto\!\!\!\!\twoheadrightarrow_\tau$ is confluent modulo *AC1*.

Proof. The proof is by Proposition 24, using Lemmas 27, 30 and 31. □

Confluence modulo AC1 for the Σ*-rules*

In this section we present the more elaborate part of the proof, namely, confluence modulo *AC1* for the relation $\mapsto\!\!\!\!\twoheadrightarrow_\Sigma$ induced by the Σ-rules. The difficulties arise from the fact that the rewrite relation $\mapsto\!\!\!\!\twoheadrightarrow_\Sigma$ is not strongly normalising. In particular, notice that neither of the rewrite relations induced by the substitution rules are terminating in the presence of cycles:

$$x\ [x = f(y), y = g(y)] \mapsto_{ac} x\ [x = f(g(y)), y = g(y)] \mapsto_{ac} \dots$$
$$y\ [y = g(y)] \mapsto_{es} g(y)\ [y = g(y)] \mapsto_{es} \dots$$

Consequently, the techniques used in the previous section for the $\mapsto\!\!\!\!\twoheadrightarrow_\tau$ relation do not apply in this case. Taking inspiration from the confluence proof of the cyclic λ-calculus in Ariola and Klop (1997), we use the so-called *complete development method* of the λ-calculus, adapting it to the relation $\mapsto\!\!\!\!\twoheadrightarrow_\Sigma$. The idea of this proof technique consists of first defining a new rewrite relation *Cpl* with the same transitive closure as the $\mapsto\!\!\!\!\twoheadrightarrow_\Sigma$ relation and then proving the diamond property modulo *AC1* for this relation. We can then conclude that the original $\mapsto\!\!\!\!\twoheadrightarrow_\Sigma$ relation is confluent.

Intuitively, a single step of *Cpl* rewriting on a term G consists of the complete development of a set of redexes initially fixed and marked in G. Concretely, an underlining operator is used to mark some redexes, and then the reductions on underlined redexes are performed using the following underlined versions of the Σ-rules:

$$\begin{array}{lll} (\underline{\textit{external sub}}) & \mathsf{Ctx}\{\underline{y}\}\ [y = G, E] & \to_{\underline{es}}\ \mathsf{Ctx}\{G\}\ [y = G, E] \\ (\underline{\textit{acyclic sub}}) & G\ [G_0 \lll \mathsf{Ctx}\{\underline{y}\}, y = G_1, E] & \to_{\underline{ac}}\ G\ [G_0 \lll \mathsf{Ctx}\{G_1\}, y = G_1, E] \\ & & \textit{if } \forall x \in \mathscr{FV}(G_0), x > \underline{y} \\ (\underline{\delta}) & (G_1 \underline{\wr} G_2)\ G_3 & \to_{\underline{\delta}}\ G_1\ G_3 \wr G_2\ G_3 \\ & (G_1 \underline{\wr} G_2)\ [E]\ G_3 & \to_{\underline{\delta}}\ (G_1\ G_3 \wr G_2\ G_3)\ [E] \end{array}$$

https://doi.org/10.1017/S0960129507006093 Published online by Cambridge University Press

We call the new rewrite relation $\mapsto\!\!\!\!\rightarrow_{\underline{\Sigma}}$ and the associated calculus $\underline{\Sigma}$-calculus. Terms belonging to the $\underline{\Sigma}$-calculus are ρ_g-graphs in which some recursion variables belonging to a $\underline{\Sigma}$-redex are underlined.

Example 33 (Terms of the $\underline{\Sigma}$-calculus). The following list shows some legal and illegal terms:

— $\underline{x}\ [x = f(x)]$ is a legal term.
— $x\ [x = f(\underline{x})]$ is not a legal term, since $x \equiv \underline{x}$ and thus the proviso for the application of the *acyclic sub* rule is not verified.
— Similarly, $f(x)\ [x \ll f(\underline{y}), y = g(\underline{z}, \underline{z}), z = \underline{y}]$ is not a legal term, since $x > \underline{y}$ but $\underline{z} \equiv \underline{y}$.
— $f(x)\ [x \ll f(\underline{y}), y = g(\underline{z}, \underline{z}), z = y]$ is a legal term, since here $x > \underline{y}$ and $y > \underline{z}$.

The *Cpl* rewrite relation is then defined as follows.

Definition 34 (*Cpl* relation). Given the ρ_g-graphs G_1 and G_2 in the Σ-calculus, we have that $G_1 \mapsto\!\!\!\!\rightarrow_{Cpl} G_2$ if there exists an underlining G_1' of G_1 such that $G_1' \mapsto\!\!\!\!\rightarrow_{\underline{\Sigma}} G_2$ and G_2 is in normal form with respect to the relation $\mapsto\!\!\!\!\rightarrow_{\underline{\Sigma}}$.

Example 35 (Reductions in the $\underline{\Sigma}$-calculus).

— The term $x\ [x = f(\underline{y}), y = g(y)]$ reaches the $\underline{\Sigma}$ normal form $x\ [x = f(g(y)), y = g(y)]$ in one ($\underline{ac}$)-step.
— We have the reduction

$$\begin{aligned} G_1 &= x\ [f(x,y) \ll f(\underline{z},\underline{z}), z = g(\underline{w}), w = a] \\ &\mapsto_{\underline{\Sigma}} x\ [f(x,y) \ll f(\underline{z},\underline{z}), z = g(a), w = a] \\ &\mapsto_{\underline{\Sigma}} x\ [f(x,y) \ll f(g(a),g(a)), z = g(a), w = a] \\ &= G_2 \end{aligned}$$

and thus $G_1 \mapsto_{Cpl} G_2$.

To ensure that for every possible underlining of redexes in a ρ_g-graph G_1 we have a corresponding *Cpl* reduction, we need to prove that for every underlined term there exists a normal form with respect to the $\mapsto\!\!\!\!\rightarrow_{\underline{\Sigma}}$ reduction, that is, we must prove that $\mapsto\!\!\!\!\rightarrow_{\underline{\Sigma}}$ is weakly normalising. To this end, we first prove that the relations induced by the δ rule and the underlined substitution rules are separately strongly normalising.

Lemma 36. $SN(\underline{\delta})$ and $SN(\{\underline{es}, \underline{ac}\})$ hold.

Proof. The strong normalisation of the relation induced by the $\underline{\delta}$ rule can be proved using the multiset path ordering induced by the following precedence on the operators of the ρ_g-calculus:

$$\_\ \_ \succ \_\langle\_ \succ \_[\_] \succ \_\ll\_ \succ \_,\_ \succ \_=\_$$

To prove the termination of the relation induced by $\{\underline{es}, \underline{ac}\}$, we exploit a technique inspired by Ariola and Klop (1997). We define the weight associated with a term of the $\underline{\Sigma}$-calculus as the multiset of all its underlined recursion variables, ordered by standard multiset ordering induced by the ordering $>$ among recursion variables (see Definition 10).

https://doi.org/10.1017/S0960129507006093 Published online by Cambridge University Press

Then we show that the weight decreases during the reduction. We analyse the two different cases:

— If we substitute an underlined recursion variable by a term containing no underlined variables, the weight trivially decreases. For example, $\underline{x}$ $[x = f(x)]$ has weight $\{\underline{x}\}$, while its reduct $f(x)$ $[x = f(x)]$ has weight $\varnothing$.

— If we substitute an underlined recursion variable $\underline{x}$ by a term containing one or more recursion variables $\underline{y_1}, \ldots, \underline{y_n}$, then we have $\underline{x} > \underline{y_i}$ for all $i = 1, \ldots, n$, otherwise the term would not be a legal $\underline{\Sigma}$-calculus term. It follows that the multiset of the reduced term is smaller than the one associated with the initial term. Consider, for example, the ρ_{g}-graph $G = x$ $[x = \mathsf{C}_0\{\underline{y_1}\}, y_1 = \mathsf{C}_1\{\underline{y_2}\}, y_2 = G']$ and the reduction

$$G \mapsto_{\underline{ac}} x\ [x = \mathsf{Ctx}_0\{\mathsf{C}_1\{\underline{y_2}\}\}, y_1 = \mathsf{C}_1\{\underline{y_2}\}, y_2 = G'].$$

The multiset associated with G is $\{\underline{y_1}, \underline{y_2}\}$. By the definition of the order on recursion variables, we have $x > y_1$ and $y_1 > y_2$, so the multiset $\{\underline{y_2}, \underline{y_2}\}$ associated with the ρ_{g}-graph obtained after the reduction is smaller. Notice that $y_1 \neq y_2$, otherwise the proviso of the *acyclic sub* rule would not be satisfied and G would not be a legal term. For the same reason, no $\underline{y_1}$ is allowed on the right-hand side of the recursion equation for y_2. □

Proposition 37 (WN($\underline{\Sigma}$)). The relation $\mapsto_{\underline{\Sigma}}$ is weakly normalising.

Proof. Given any $\underline{\Sigma}$-term, a normal form can be reached by using the rewriting strategy where $\underline{\delta}$ has greater priority than $\{\underline{es}, \underline{ac}\}$. By Lemma 36 we know that the relations induced by $\underline{\delta}$ and $\{\underline{es}, \underline{ac}\}$ are strongly normalising. Observe that the $\{\underline{es}, \underline{ac}\}$ induced relation does not generate $\underline{\delta}$ redexes. Hence we can normalise a term G first with respect to the $\underline{\delta}$ induced relation and then with respect to the $\{\underline{es}, \underline{ac}\}$ induced relation obtaining thus a finite reduction of G. □

The next goal is to prove the diamond property for the *Cpl* relation. In order to do this, the confluence modulo *AC1* of the $\mapsto_{\underline{\Sigma}}$ relation is needed. Since we know that the relations induced by $\underline{\delta}$ and $\{\underline{es}, \underline{ac}\}$ are both strongly normalising, we prove their local confluence modulo *AC1* by analysis of the critical pairs and then conclude that they are confluent modulo *AC1*. The confluence modulo *AC1* of the $\mapsto_{\underline{\Sigma}}$ relation will then follow using a commutation lemma.

Lemma 38. $LCON_{\sim}(\underline{\delta})$ and $LCON_{\sim}(\{\underline{es}, \underline{ac}\})$ hold.

Proof. We proceed by analysis of the critical pairs. The critical pairs of the $\underline{\delta}$ rule with itself are trivial. Among the critical pairs of the *external sub* and *acyclic sub* rules, we will just show the diagrams for two interesting cases. We consider the case in which $\lll$ is equal to $=$. The case in which $\lll$ is $\ll$ can be treated in the same way. To make the notation simpler, from now on we will just write $\mathsf{C}_i\{G\}$ for a context $\mathsf{Ctx}_i\{G\}$ in the diagrams representing the critical pairs.

— Consider the critical pair generated from a term having a list of constraints containing two non-disjoint *ac* redexes. Notice that the recursion variable $\underline{z}$ can be duplicated by the first *ac*-step.

https://doi.org/10.1017/S0960129507006093 Published online by Cambridge University Press

$$\begin{array}{ccc} G_0\ [y = \mathsf{C}_1\{\underline{x}\}, x = \mathsf{C}_2\{\underline{z}\}, z = G_1] & \xrightarrow[\underline{ac}]{} & G_0\ [y = \mathsf{C}_1\{\underline{x}\}, x = \mathsf{C}_2\{G_1\}, z = G_1] \\ \Big\downarrow{\scriptstyle \underline{ac}} & & \Big\downarrow{\scriptstyle \underline{ac}} \\ G_0\ [y = \mathsf{C}_1\{\mathsf{C}_2\{\underline{z}\}\}, x = \mathsf{C}_2\{\underline{z}\}, z = G_1] & \xrightarrow[\underline{ac}]{} & G_0\ [y = \mathsf{C}_1\{\mathsf{C}_2\{G_1\}\}, x = \mathsf{C}_2\{G_1\}, z = G_1] \end{array}$$

— Consider the critical pair in which the term duplicated by an $\underline{es}$ step contains an $\underline{ac}$ redex. Notice that we need both the substitution rules, that is, $\underline{ac \cup es}$, to close the diagram.

$$\begin{array}{ccc} \mathsf{C}_0\{\underline{y}\}\ [y = \mathsf{C}_1\{\underline{x}\}, x = \mathsf{C}_2\{x\}] & \xrightarrow[\underline{es}]{} & \mathsf{C}_0\{\mathsf{C}_1\{\underline{x}\}\}\ [y = \mathsf{C}_1\{\underline{x}\}, x = \mathsf{C}_2\{x\}] \\ \Big\downarrow{\scriptstyle \underline{ac}} & & \Big\downarrow{\scriptstyle \underline{ac \cup es}} \\ \mathsf{C}_0\{\underline{y}\}\ [y = \mathsf{C}_1\{\mathsf{C}_2\{x\}\}, x = \mathsf{C}_2\{x\}] & \xrightarrow[\underline{ac}]{} & \mathsf{C}_0\{\mathsf{C}_1\{x\}\}\ [y = \mathsf{C}_1\{\mathsf{C}_2\{x\}\}, x = \mathsf{C}_2\{x\}] \end{array}$$

□

At this point, it is worth noticing that the local compatibility with *AC1* holds for the underlined version of the rules. This property, together with the local confluence modulo *AC1* and the strong normalisation for the relations induced by the rules $\underline{\delta}$ and $\{\underline{es}, \underline{ac}\}$, is sufficient to prove their confluence.

Lemma 39. $CPB_{\sim}(\underline{\delta})$ and $CPB_{\sim}(\{\underline{es}, \underline{ac}\})$ hold.

Proof. By Lemma 27 we know that this property holds for the original version of the rules without underlining. Since equivalence steps in the *AC1* theory have no effect with respect to the underlining, we can conclude that the lemma is also true for the underlined rules. □

Lemma 40. $CON_{\sim}(\underline{\delta})$ and $CON_{\sim}(\{\underline{es}, \underline{ac}\})$ hold.

Proof. The proof is by Proposition 24 using Lemmas 36, 38 and 39. □

Having proved the confluence modulo *AC1* of the relations induced by the two subsets of rules independently, following Proposition 24, we need a commutation lemma in order to show the confluence of the relation induced by the union of the two subsets.

Lemma 41. $COM_{\sim}(\underline{\delta}, \{\underline{es}, \underline{ac}\})$ holds.

Proof. General commutation is not easy to prove, thus we prove a simpler property that implies commutation. By Lemma 39, we know that the compatibility property holds for our relations. Unfortunately, the two relations are not strongly commuting, since each of them can duplicate redexes of the other. Nevertheless, the relations do not interfere with each other, in the sense that, for example, a $\underline{\delta}$ redex will still be present (and possibly duplicated) after one or several steps of $\{\underline{es}, \underline{ac}\}$. Therefore, we will use Proposition 25, and we need simply to verify the property

$$\leftarrow\!\!\!\mapsto_{\{\underline{es},\underline{ac}\}} \cdot \mapsto\!\!\!\!\!\rightarrow_{\underline{\delta}} \ \subseteq\ \mapsto\!\!\!\!\!\rightarrow_{\underline{\delta}} \cdot \sim_E \cdot \leftarrow\!\!\!\!\!\mapsto_{\{\underline{es},\underline{ac}\}} \cdot$$

https://doi.org/10.1017/S0960129507006093 Published online by Cambridge University Press

We proceed by analysis of the critical pairs. We will just consider explicitly the critical pairs between the $\underline{\delta}$ rule and the $\underline{es}$ rule, since the treatment for the critical pairs between the $\underline{\delta}$ rule and the $\underline{ac}$ rule is similar. In particular, in the next diagram we consider a critical pair in which the $\underline{\delta}$ and the $\underline{es}$ redexes are not disjoint. We recall that, by Lemma 36, there exists no infinite $\underline{\delta}$ or $\{\underline{es}, \underline{ac}\}$ reduction. For simplicity, the diagram shows a single $\underline{\delta}$ step; a longer derivation would just bring a further duplication of the $\{\underline{es}, \underline{ac}\}$ redex, but the diagram could be closed in a similar way.

$$\begin{array}{ccc}
\mathsf{C}_0\{(G_1 \underline{\wr} G_2)\ \mathsf{C}_1\{\underline{x}\}\}\ [x = G, E] & \xrightarrow[\underline{\delta}]{} & \mathsf{C}_0\{(G_1\ \mathsf{C}_1\{\underline{x}\}) \wr (G_2\ \mathsf{C}_1\{\underline{x}\})\ [x = G, E] \\
\Big\downarrow{\scriptstyle \{\underline{es},\underline{ac}\}} & & \Big\downarrow{\scriptstyle \{\underline{es},\underline{ac}\}} \\
\mathsf{C}_0\{(G_1 \underline{\wr} G_2)\ \mathsf{C}_1\{G\}\}\ [x = G, E] & \xrightarrow[\underline{\delta}]{} & \mathsf{C}_0\{(G_1\ \mathsf{C}_1\{G\}) \wr (G_2\ \mathsf{C}_1\{G\})\ [x = G, E]
\end{array}$$

□

Taking advantage of the previous three lemmas, it is now possible to show the confluence modulo $AC1$ of the $\underline{\Sigma}$ relation.

Proposition 42 ($\mathbf{CON}_{\sim}(\underline{\Sigma})$). The relation $\underline{\Sigma}$ is confluent modulo $AC1$.

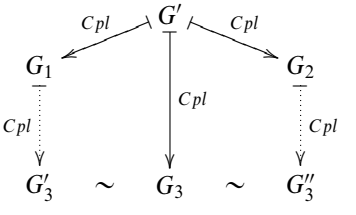
Proof. The proof is by Proposition 24, using Lemmas 39, 40 and 41. □

Using the weak termination of the relation $\underline{\Sigma}$ and its confluence modulo $AC1$, we can finally prove that the diamond property modulo $AC1$ holds for the Cpl relation.

Lemma 43 ($\mathbf{D}_{\sim}(Cpl)$). The rewrite relation Cpl enjoys the diamond property modulo $AC1$.

Proof. Given a term G', let $S = S_1 \cup S_2$ be a set of underlined redexes in G' such that we have $G' \mapsto\!\!\!\!\!\rightarrow_{Cpl} G_3$ reducing all the underlined redexes in S. Let G_1 and G_2 be the two partial developments relative to S_1 and S_2, respectively, that is, $G' \mapsto\!\!\!\!\!\rightarrow_{Cpl} G_1$ reducing only the redexes in S_1 and $G' \mapsto\!\!\!\!\!\rightarrow_{Cpl} G_2$ reducing only the redexes in S_2. In both cases, since $WN(\underline{\Sigma})$ holds by Proposition 37, we can continue reducing the remaining underlined redexes, obtaining $G_1 \mapsto\!\!\!\!\!\rightarrow_{Cpl} G_3'$ and $G_2 \mapsto\!\!\!\!\!\rightarrow_{Cpl} G_3''$.

Since all the steps in the Cpl reduction are $\underline{\Sigma}$ steps, using the fact that G_3, G_3' and G_3'' are completely reduced with respect to $\underline{\Sigma}$ and that $CON_{\sim}(\underline{\Sigma})$ holds by Proposition 42, we have the equivalence of G_3, G_3' and G_3'':



□

https://doi.org/10.1017/S0960129507006093 Published online by Cambridge University Press

The confluence of the $\mapsto\!\!\!\!\rightarrow_{\Sigma}$ relation follows easily from the observation that this relation and the *Cpl* relation have the same transitive closure.

Proposition 44 (CON$_{\sim}(\Sigma)$). The relation $\mapsto\!\!\!\!\rightarrow_{\Sigma}$ is confluent modulo *AC1*.

Proof. The result follows by Lemma 43, since if *Cpl* satisfies the diamond property modulo *AC1*, so does its transitive closure, and it is not difficult to show that the transitive closure of the relation *Cpl* is the same as that of the relation $\mapsto\!\!\!\!\rightarrow_{\Sigma}$. This follows from the fact that $\mapsto_{\Sigma} \subseteq \mapsto\!\!\!\!\rightarrow_{Cpl} \subseteq \mapsto\!\!\!\!\rightarrow_{\Sigma}$. The first inclusion can be proved by underlining the redex reduced by the Σ step. The second inclusion follows trivially from the definition of the *Cpl* relation. □

General confluence

So far we have shown confluence of the relations induced by two subsets of rules τ and Σ separately. In the last part of the proof we consider the union of these two subsets of rules. General confluence holds since we can prove the commutation modulo *AC1* between the relation $\mapsto\!\!\!\!\rightarrow_{\tau}$ and the relation $\mapsto\!\!\!\!\rightarrow_{\Sigma}$.

Lemma 45 (COM$_{\sim}(\tau, \Sigma)$). The relations $\mapsto\!\!\!\!\rightarrow_{\tau}$ and Σ commute modulo *AC1*.

Proof. Since the relation $\mapsto\!\!\!\!\rightarrow_{\Sigma}$ does not terminate, it is easier to show strong commutation between the two relations instead of general commutation:

$$\begin{array}{ccc} G & \overset{\tau}{\longmapsto} & G_1 \\ {\scriptstyle\Sigma}\downarrow & & {\scriptstyle\Sigma}\,\downarrow{\scriptstyle 0/1} \\ G_2 & \overset{\tau}{\mapsto\!\!\!\!\rightarrow} & G_1' \sim G_2' \end{array}$$

The result then follows by Proposition 25, using the compatibility with *AC1* for the relations $\mapsto\!\!\!\!\rightarrow_{\tau}$ and $\mapsto\!\!\!\!\rightarrow_{\Sigma}$, which follows from Lemma 27. The possibility of closing the diagram by using at most one step for the Σ-rules is ensured by the fact that none of the τ-rules is duplicating.

If the applied Σ-rule is the δ rule, it is easy to close the diagram since the τ-rules do not interfere with δ redexes (the generated critical pairs are trivial). Only the *garbage* rule can alter a δ redex by eliminating it, and in this case the diagram is closed with zero δ steps.

If the applied Σ-rule is a substitution rule, the interesting critical pairs are:

— The τ-rule applied to G is the *propagate* rule. The only interesting case is the following where the two Σ-rules applied are different.

$$\begin{array}{ccc} P \ll (\mathsf{Ctx}\{y\}\ [y = H, E]) & \overset{p}{\longmapsto} & P \ll \mathsf{Ctx}\{y\}, y = H, E \\ {\scriptstyle es}\downarrow & & {\scriptstyle ac}\,\downarrow \\ P \ll (\mathsf{Ctx}\{H\}\ [y = H, E]) & \overset{p}{\mapsto\!\!\!\!\rightarrow} & P \ll \mathsf{Ctx}\{H\}, y = H, E \end{array}$$

https://doi.org/10.1017/S0960129507006093 Published online by Cambridge University Press

— The τ-rule applied to G is the *decompose* rule. In this case the term G is of the form $H\ [f(H_1,\ldots,H_n) \ll f(\mathsf{Ctx}\{y\},\ldots,H'_n), y = H', E]$. The *decompose* rule transforms the match equation in a set of simpler constraints $H_1 \ll \mathsf{Ctx}\{y\},\ldots, H_n \ll H'_n$ in the same list. Since the *acyclic sub* rule is applied using matching modulo $AC1$, the substitution generated from $y = H'$ can be equivalently performed either before or after the decomposition.
— The τ-rule applied to G is the *solved* rule. In this case, there are no differences between G_1 and G from the point of view of the application of a substitution rule.
— The τ-rule applied to G is the *garbage* rule. The peculiarity here is that we can have zero steps of the Σ rules for closing the diagram when the substitution redex is part of the sub-term that is eliminated by garbage collection.
— The τ-rule applied to G is the *black hole* rule. We may have an overlap of the *external sub* rule and the *black hole* rule if the term duplicated by the substitution is a variable:

$$\begin{array}{ccc} \mathsf{Ctx}\{y\}\ [y = y, E] & \xrightarrow[bh]{} & \mathsf{Ctx}\{\bullet\}\ [y = y, E] \\ \Big\downarrow{\scriptstyle es} & & \Big\downarrow{\scriptstyle es\ 0} \\ \mathsf{Ctx}\{y\}\ [y = y, E] & \overset{bh}{\twoheadrightarrow} & \mathsf{Ctx}\{\bullet\}\ [y = y, E] \end{array}$$

If the cycle has length greater than one, that is, it is expressed by more than one recursion equation, the matching modulo $AC1$ allows us to apply the *black hole* rule even when the recursion equations are not in the correct order in the list, and this can happen as a consequence of the application of the *external sub* rule:

$$\begin{array}{ccc} \mathsf{Ctx}\{y\}\ [y = x, x = y, E] & \xrightarrow[bh]{} & \mathsf{Ctx}\{\bullet\}\ [y = x, x = y, E] \\ \Big\downarrow{\scriptstyle es} & & \Big\downarrow{\scriptstyle es\ 0} \\ \mathsf{Ctx}\{x\}\ [y = x, x = y, E] & \overset{bh}{\twoheadrightarrow} & \mathsf{Ctx}\{\bullet\}\ [y = x, x = y, E] \end{array}$$

We have similar cases for the *acyclic sub* rule. □

The confluence modulo $AC1$ of the sets of rules τ and Σ, the commutation modulo $AC1$ of the two sets, together with their compatibility property with $AC1$ ensure the confluence of their union.

Theorem 46 (CON$_\sim(\rho_g, AC1)$). In the linear ρ_g-calculus, the rewrite relation $\mapsto\!\!\!\!\!\rightarrow_{\rho_g, AC1}$ is confluent modulo $AC1$.

Proof. The result follows from Proposition 24 using Propositions 32 and 44, and Lemmas 27 and 45. □

As mentioned in the first section, our intention is to produce a more general result for rewriting on $AC1$-equivalence classes of ρ_g-graphs. Thanks to the property of compatibility of ρ_g with $AC1$, it is easy to derive the Church–Rosser property for $AC1$-equivalence classes for the ρ_g-calculus rewrite relation from the latter theorem.

https://doi.org/10.1017/S0960129507006093 Published online by Cambridge University Press

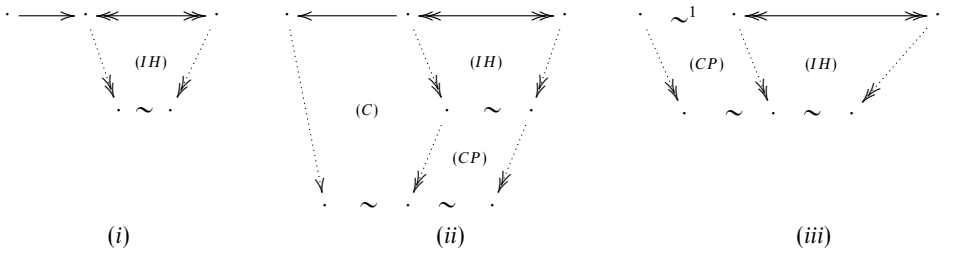


Figure 13. Church–Rosser property for $\rho_g/AC1$.

Theorem 47 ($CR_{\sim}(\rho_g/AC1)$). The linear ρ_{g}-calculus is Church–Rosser modulo $AC1$.

Proof. We use induction on the length of the reduction.

For the first step, we decompose the reduction $\twoheadleftarrow\!\!\!\!\twoheadrightarrow^{n}_{\rho_g/AC1}$ into $\twoheadleftarrow\!\!\!\!\twoheadrightarrow^{1}_{\rho_g/AC1}$ $\twoheadleftarrow\!\!\!\!\twoheadrightarrow^{n-1}_{\rho_g/AC1}$. We then have three possibilities for the first step. For each case we show the Church–Rosser diagram in Figure 13, using confluence modulo AC_ϵ, which was shown in Theorem 46 (denoted C), the compatibility property shown in Lemma 27 (denoted CP) and the induction hypothesis (denoted IH). □

5. Expressiveness of the ρ_{g}-calculus

5.1. ρ_{g}-*calculus versus* ρ-*calculus*

The set of terms of the ρ-calculus (with syntactic matching) is a strict subset of the set of ρ_{g}-graphs 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 behaved with respect to the ρ-calculus matching algorithm restricted to patterns (Cirstea *et al.* 2002).

Lemma 48. Let T be an algebraic ρ-term with $\mathscr{FV}(T) = \{x_1, \ldots, x_n\}$ and let $T \ll U$ be a matching problem with solution $\sigma = \{x_1/U_1, \ldots, x_n/U_n\}$, that is, $\sigma(T) = U$. Then we have $T \ll U \mapsto\!\!\!\!\twoheadrightarrow_{\mathscr{M}} x_1 = U_1, \ldots, x_n = U_n$, where $\mapsto\!\!\!\!\twoheadrightarrow_{\mathscr{M}}$ is the rewrite relation induced by the matching rules only, as defined in Definition 17.

Proof. We show by structural induction on the term T that there exists a reduction of the form $T \ll U \mapsto\!\!\!\!\twoheadrightarrow_{\mathscr{M}} x_1 \ll U_1, \ldots, x_n \ll U_n$, where the x_i's are all distinct – the thesis then follows.

— *Base 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.

https://doi.org/10.1017/S0960129507006093 Published online by Cambridge University Press

— *Induction step*: $T = f(T_1, \ldots, T_m)$ 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 $\sigma(T_i) = V_i$, for $i = 1, \ldots, m$. By the induction hypothesis, for any i, if $\mathscr{FV}(T_i) = \{x^i_1, \ldots, x^i_{k_i}\} \subseteq \mathscr{FV}(T)$, we have the reduction $T_i \ll V_i \mapsto\!\!\!\!\rightarrow_{\mathscr{M}} x^i_1 \ll \sigma(x^i_1), \ldots, x^i_{k_i} \ll \sigma(x^i_{k_i})$. Joining the various reductions, we have

$$\begin{aligned} f(T_1, \ldots, T_m) \ll f(V_1, \ldots, V_m) &\mapsto_{dk} T_1 \ll V_1, \ldots, T_m \ll V_m \\ &\mapsto\!\!\!\!\rightarrow_{\mathscr{M}} x_1 \ll \sigma(x_1), \ldots, x_n \ll \sigma(x_n). \end{aligned}$$

To understand the last step, note that in the list

$$x^1_1 \ll \sigma(x^1_1), \ldots, x^1_{k_1} \ll \sigma(x^1_{k_1}), \ldots, x^m_1 \ll \sigma(x^m_1), \ldots, x^m_{k_m} \ll \sigma(x^m_{k_m}),$$

constraints with the same left-hand side variable have identical right-hand sides. Hence, by idempotency, such a list coincides with $x_1 \ll \sigma(x_1), \ldots, x_n \ll \sigma(x_n)$. □

We can show now that a reduction in the ρ-calculus can be simulated in the ρ_g-calculus.

Lemma 49. Let T and T' be ρ-terms. If there exists a reduction $T \mapsto\!\!\!\!\rightarrow_{\rho\delta} T'$ in the ρ-calculus, 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, 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 ρ-calculus pattern and the substitution $\sigma = \{U_1/x_1, \ldots, U_m/x_m\}$ is the solution of the matching, then, in the ρ_g-calculus the corresponding reduction is

$$\begin{aligned} T &= T_2\ [T_1 \ll T_3] \\ &\mapsto\!\!\!\!\rightarrow_{\mathscr{M}} T_2\ [x_1 = U_1, \ldots, x_m = U_m] \quad (\textit{by Lemma 48}) \\ &\mapsto\!\!\!\!\rightarrow_{es} T'\ [x_1 = U_1, \ldots, x_m = U_m] \\ &\mapsto\!\!\!\!\rightarrow_{gc} T'\ [\epsilon] \\ &\mapsto_{gc} T' \end{aligned}$$

□

Theorem 50 (Completeness). Let T and T' be ρ-terms. If there exists a reduction $T \mapsto\!\!\!\!\rightarrow_{\rho\delta} T'$ in the ρ-calculus, then $T \mapsto\!\!\!\!\rightarrow_{\rho g} T'$ in the ρ_g-calculus.

Proof. The result follows from Lemma 49. □

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 are kept as constraint application failures. In particular, in the ρ_g-calculus we can have a deeper decomposition of a matching problem than in the ρ-calculus. Thus it can happen that a ρ-term in normal form can be further reduced in the ρ_g-calculus.

https://doi.org/10.1017/S0960129507006093 Published online by Cambridge University Press

Example 51 (Matching failure in ρ-calculus and ρ_g-calculus). In both calculi, non-successful reductions lead to a non-solvable match equation in the list of constraints of the term.

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

Notice that in the ρ-calculus, since the matching algorithm cannot compute a substitution solving 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.

5.2. *ρ_g-calculus versus cyclic λ-calculus*

The terms of $\lambda\phi_0$ can be easily translated into terms of the ρ_g-calculus. The main difference between $\lambda\phi_0$ and the ρ_g-calculus is the restriction of the list of constraints to a list of recursion equations. Delayed matching constraints are not needed since the matching is always trivially satisfied in the λ-calculus.

Definition 52 (Translation). The translation of a $\lambda\phi_0$-term t into a ρ_g-term, denoted $\bar{t}$, is defined inductively as follows:

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

We can see the evaluation rules of the ρ_g-calculus as the generalisation 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 will now show that a reduction in the $\lambda\phi_0$-calculus can be simulated in the ρ_g-calculus.

Lemma 53. Let t_1 and t_2 be two $\lambda\phi_0$-terms. If $t_1 \mapsto_{\lambda\phi} t_2$ in the cyclic λ-calculus, then there exists a reduction $\overline{t_1} \mapsto\!\!\!\!\twoheadrightarrow_{\rho g} \overline{t_2}$ in the ρ_g-calculus.

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

— 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:

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

— *external sub* rule: This case is trivial.
— *acyclic sub* rule: This case is trivial ($\lll$ stands always for $=$ in this case).
— *black hole* rule: This case is trivial.
— *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 a time. For this

https://doi.org/10.1017/S0960129507006093 Published online by Cambridge University Press

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 $\bar{t}\ [E, E'] \twoheadrightarrow_{gc} \bar{t}\ [E]$. □

Theorem 54 (Completeness). Let t_1 and t_2 be two $\lambda\phi$-terms. Given a reduction $t_1 \twoheadrightarrow_{\lambda\phi} t_2$ in the cyclic λ-calculus, then there exists a corresponding reduction $\overline{t_1} \twoheadrightarrow_{\rho g} \overline{t_2}$ in the ρ_g-calculus.

Proof. The result follows from Lemma 53. □

5.3. *Simulation of term graph rewriting into the ρ_g-calculus*

The possibility of representing structures with cycles and sharing naturally leads us to ask whether first-order term graph rewriting can be simulated in this context. In this section we provide a positive answer. For our purposes, we choose the equational description of term graph rewriting defined in Section 2.3. Recall that a term graph rewrite system $TGR = (\Sigma, \mathscr{R})$ is composed of a signature Σ over which the considered term graphs are built and a set of term graph rewrite rules $\mathscr{R}$. Both the term graphs over Σ and the set of rules are translated at the object level of the ρ_g-calculus, that is, into ρ_g-graphs.

Definition 55. For the various components of a TGR the corresponding element in the ρ_g-calculus is defined by:

— *(Terms)* Using the equational framework, the set of term graphs of a TGR is a strict subset of the set of terms of the ρ_g-calculus, modulo some obvious syntactic conventions. In particular, by abuse of notation, in the following we will sometimes confuse the two notations $\{x \mid E\}$ and $x\ [E]$.

— *(Rewrite rules)* A rewrite rule $(L, R) \in \mathscr{R}$ is translated into the corresponding ρ_g-graph $L \twoheadrightarrow R$.

— *(Substitution)* A substitution $\sigma = \{x_1/G_1, \ldots, x_n/G_n\}$ corresponds in the ρ_g-calculus to a list of constraints $E = (x_1 = G_1, \ldots, x_n = G_n)$, and its application to a term graph L corresponds to the addition of the list of constraints to the ρ_g-term L, that is, to the ρ_g-graph $L\ [E]$.

As seen in Section 2.3, it is convenient to work with a restricted class of term graphs in flat form and without useless equations. In general, the structure of a ρ_g-graph can be more complex than the structure of a flat term graph, since it can have nested lists of constraints and garbage. To recover the similarity, we define the canonical form of a ρ_g-graph G containing no abstractions and no match equations. Below, we will refer to the notion of ρ_g-graph in *flat form*, which, as for term graphs, is defined as a ρ_g-graph where all recursion equations are of the form $x = f(x_1, \ldots, x_n)$.

Definition 56 (Canonical form). Let G be a ρ_g-graph containing no abstractions and no match equations. We say that G is in *canonical form* if it is in flat form and contains neither garbage equations nor trivial equations of the form $x = y$.

Any ρ_g-graph G without abstractions and match equations can be transformed into a corresponding ρ_g-graph in canonical form, which will be denoted by $\overline{G}$, as follows. We

https://doi.org/10.1017/S0960129507006093 Published online by Cambridge University Press

first perform the flattening and merging of the lists of equations of G and introduce new recursion equations with fresh variables for every sub-term of G. In this way we obtain a ρ_g-graph in flat form. For instance, the ρ_g-term $G = x\ [x = f(g(y))\ [y = z,\ z = a]]$ would become $x\ [x = f(w),\ w = g(y), y = z,\ z = a]$. The canonical form can then be obtained from the flat form by removing the useless equations using the two substitution rules and the garbage collection rule of the ρ_g-calculus. In the example above we would get $\overline{G} = x\ [x = f(w),\ w = g(z),\ \ z = a]$. It is easy to see that the canonical form of a ρ_g-graph is unique, up to α-conversion and the $AC1$ axioms for the constraint conjunction operator, and a ρ_g-graph with no abstractions and no match equations in canonical form can be seen as a term graph in flat form.

Before proving the correspondence between rewriting in a TGR and in the ρ_g-calculus, we need a lemma showing that matching in the ρ_g-calculus is well behaved with respect to the notion of term graph homomorphism.

Lemma 57 (Matching). Let G be a closed term graph and let (L, R) be a rewrite rule with $\mathscr{V}ar(L) = \{x_1, \ldots, x_m\}$. Assume that there is a homomorphism from L to G, given by the variable renaming $\sigma = \{x_1/x'_1, \ldots, x_m/x'_m\}$.

Let $E = (x_n = x'_n, \ldots, x_m = x'_m, E_G)$ with $\{x_n, \ldots, x_m\} = \mathscr{FV}(L)$. Then, in the ρ_g-calculus we have the reduction $L \ll G \mapsto\!\!\!\!\rightarrow_{\rho g} E$ with $\tau(\overline{L\ [E]}) = G$, where τ is a variable renaming.

Proof. We consider functions of arity less than or equal to two. Note that this is not really a restriction since n-ary functions are encoded in the ρ_g-calculus as a sequence of nested binary applications.

Given the matching problem $L \ll G$, where $L = x_1\ [E_L]$ and $G = x'_1\ [E_G]$, in the ρ_g-calculus we have the reduction

$$L \ll G\ =\ x_1\ [E_L] \ll x'_1\ [E_G]\ \mapsto_p\ x_1\ [E_L] \ll x'_1, E_G \mapsto\!\!\!\!\rightarrow_{es,gc}\ T_L \ll x'_1, E_G$$

where T_L is a term without constraints, that is, a tree, which can be reached since L is acyclic by hypothesis.

We proceed by induction on the length of the list of recursion equations E_L of the term graph L, or, equivalently, on the height of T_L, seen as a tree:

— *Base case.* T_L is a variable x_1. We obtain the reduction $x_1 \ll x'_1, E_G \mapsto_s x_1 = x'_1, E_G$. We can verify immediately that $\overline{L\ [E]} = \overline{x_1\ [x_1 = x'_1, E_G]}$ is equal to G using the variable renaming $\tau = \{x_1/x'_1\}$.

— *Induction step.* Let G be of the form $G = x'_1\ [x'_1 = f(x'_2, x'_3), x'_2 = T_2, x'_3 = T_3, E']$. Continuing the reduction of the match equation $L \ll G$, we obtain

$$\begin{aligned} T_L \ll x'_1, E_G\ &=\ T_L \ll x'_1, x'_1 = f(x'_2, x'_3), \ldots \\ &\mapsto_{ac}\ T_L \ll f(x'_2, x'_3), E_G \end{aligned}$$

Since, by hypothesis, there is a homomorphism σ between L and G, we have $T_L = f(T'_2, T'_3)$, and thus

$$T_L \ll f(x'_2, x'_3), E_G \mapsto_{dk}\ T'_2 \ll x'_2, T'_3 \ll x'_3, E_G.$$

Using the induction hypothesis and the fact that L is acyclic, we obtain the reductions $T'_2 \ll x'_2, E_G \mapsto\!\!\!\!\rightarrow_{\rho g} E_2$ and $T'_3 \ll x'_3, E_G \mapsto\!\!\!\!\rightarrow_{\rho g} E_3$, and the variable renamings τ_2 and

https://doi.org/10.1017/S0960129507006093 Published online by Cambridge University Press

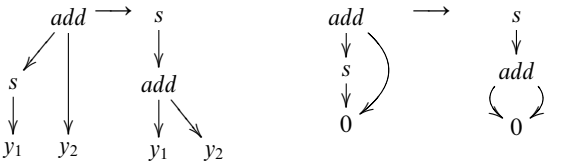


Figure 14. Example of rewriting in a TGR.

τ_3 such that $\tau_2(\overline{T_2' \ [E_2]}) = T_2$ and $\tau_3(\overline{T_3' \ [E_3]}) = T_3$. Therefore, since $\mathscr{FV}(L) = \mathscr{FV}(T_2') \cup \mathscr{FV}(T_3')$, it is easy to verify that $L \ll G \mapsto\!\!\!\!\!\rightarrow_{fg} E$, with $E = E_2, E_3, E_G$, and that the variable renaming $\tau = \tau_2\tau_3\{x_1/x_1'\}$ is such that $\tau(\overline{L \ [E]}) = G$. □

The previous lemma guarantees the fact that if there exists a homomorphism (represented as a variable renaming) of a term graph L into G, in the ρ_g-calculus, we obtain the variable renaming (in the form of a list of recursion equations) as the result of the evaluation of the matching problem $L \ll G$. In other words, this means that if a rewrite rule can be applied to a term graph, the application is still possible after the translation of the rule into a ρ_g-abstraction and of the term graph into a ρ_g-graph.

Example 58 (Matching). Consider the term graphs

$$L = \{x_1 \mid x_1 = add(x_2, y_2), x_2 = s(y_1)\}$$

and

$$G = \{z_0 \mid z_0 = add(z_1, z_2),\ z_1 = s(z_2), z_2 = 0\}$$

(see Figure 14) and the homomorphism

$$\sigma = \{x_1/z_0,\ x_2/z_1,\ y_1/z_2,\ y_2/z_2\}$$

from L to G. We will show how σ can be obtained in the ρ_g-calculus starting from the matching problem $L \ll G$.

$$\begin{aligned}
L \ll G \ &\mapsto_{p} && L \ll z_0, E_G \\
&\mapsto\!\!\!\!\!\rightarrow_{es,gc} && add(s(y_2), y_1) \ll z_0, E_G \\
&= && add(s(y_2), y_1) \ll z_0, z_0 = add(z_1, z_2), z_1 = s(z_2), z_2 = 0 \\
&\mapsto_{ac} && add(s(y_2), y_1) \ll add(z_1, z_2), z_0 = add(z_1, z_2), z_1 = s(z_2), z_2 = 0 \\
&\mapsto_{dk} && s(y_2) \ll z_1, y_1 \ll z_2, z_0 = add(z_1, z_2), z_1 = s(z_2), z_2 = 0 \\
&\mapsto\!\!\!\!\!\rightarrow_{ac,dk} && y_2 \ll z_2, y_1 \ll z_2, z_0 = add(z_1, z_2), z_1 = s(z_2), z_2 = 0 \\
&\mapsto\!\!\!\!\!\rightarrow_{s} && y_2 = z_2, y_1 = z_2, E_G
\end{aligned}$$

We can then verify that $\overline{L \ [y_2 = z_2, y_1 = z_2, E_G]}$ is equal to G up to variable renaming. In fact, the transformation into the canonical form leads to the graph

$$x_1 \ [x_1 = add(x_2, z_2), x_2 = s(z_2), z_2 = 0]$$

and it is easy to see that the variable renaming $\tau = \{x_1/z_0, x_2/z_1\}$ makes this graph equal to G.

https://doi.org/10.1017/S0960129507006093 Published online by Cambridge University Press

We will now analyse the relationship between derivations of a term graph rewrite system $TGR = (\Sigma, \mathscr{R})$ and reductions in the ρ_g-calculus. Given a term graph G in the TGR and a derivation with respect to the set of rules $\mathscr{R}$, we will show how to build a ρ_g-graph that reduces in the ρ_g-calculus to a term corresponding to the ending term graph of the original TGR reduction.

Since in the ρ_g-calculus the rule application is at the object level, we need to use a *position trace* ρ_g-term H encoding the position of the redex in the given term graph G.

Lemma 59 (Simulation). Let G be a ground term graph, (L, R) be a rewrite rule rooted at z, and σ be a homomorphism from L to G such that $G_{\lceil\sigma(z)=t\rceil_p} \to G_{\lceil\sigma(R)\rceil_p}$ with $\sigma(z)$ not in a cycle, that is, there does not exist $y \in G$ such that $\sigma(z) \equiv y$.

Define the ρ_g-term $H = G_{\lceil\sigma(z)=x\rceil_p} \twoheadrightarrow G_{\lceil\sigma(z)=(L \twoheadrightarrow R)\ x\rceil_p}$. Then there exists in the ρ_g-calculus a reduction $(H\ G) \mapsto\!\!\!\!\!\rightarrow_{\rho g} G'$ and a variable renaming τ such that $\tau(\overline{G'})$ is equal to $G_{\lceil\sigma(R)\rceil_p}$.

Proof. First, observe that, by definition, $G_{\lceil\sigma(z)=x\rceil_p}$ matches G. If the corresponding homomorphism is $\sigma' = \{x/x'\}$, by Lemma 57, after the resolution of the matching we obtain in the ρ_g-calculus a list of recursion equations $x = x', E_G$. We will simply use E'_G to denote such a list.

We obtain the following reduction in the ρ_g-calculus:

$$
\begin{aligned}
G_{\lceil\sigma(z)=x\rceil_p} \twoheadrightarrow G_{\lceil\sigma(z)=(L \twoheadrightarrow R)\ x\rceil_p}\ G \quad &\mapsto_{\rho} \quad G_{\lceil\sigma(z)=(L \twoheadrightarrow R)\ x\rceil_p}\ [G_{\lceil\sigma(z)=x\rceil_p} \ll G] \\
&\mapsto\!\!\!\!\!\rightarrow_{\rho g} \quad G_{\lceil\sigma(z)=(L \twoheadrightarrow R)\ x\rceil_p}\ [E'_G] \quad \textit{by Lemma 57} \\
&\mapsto_{es} \quad G_{\lceil\sigma(z)=(L \twoheadrightarrow R)\ x'\rceil_p}\ [E'_G] \\
&\mapsto_{\rho} \quad G_{\lceil\sigma(z)=R\ [L \ll x']\rceil_p}\ [E'_G] \\
&\mapsto\!\!\!\!\!\rightarrow_{\rho g} \quad G_{\lceil\sigma(z)=R\ [y_1=y'_1,\ldots,y_n=y'_n]\rceil_p}\ [E'_G] \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{by Lemma 57} \\
&\mapsto\!\!\!\!\!\rightarrow_{es,gc} \quad G_{\lceil R'\rceil_p}\ [E'_G] \ = \ G'_1
\end{aligned}
$$

where $\{y_1, \ldots, y_n\} = \mathscr{FV}(R)$ and R' is the term obtained from R by renaming y_i to y'_i, for $i = 1 \ldots n$. Using Lemma 57, it is not difficult to deduce that R' is equal to $\sigma(R)$ modulo α-conversion. We conclude by noticing that (the flat form of) $G_{\lceil\sigma(R)\rceil_p}\ [E'_G]$ is equal up to variable renaming to $G_{\lceil\sigma(R)\rceil_p}$. □

Notice that in the ρ_g-graph H we could have separated the rule from the information about its application position by choosing $H = y \twoheadrightarrow (\mathcal{P}_p(G)_{\lceil x\rceil_p} \twoheadrightarrow \mathcal{P}_p(G)_{[y\ x]_p})$ and then considering $H\ (L \twoheadrightarrow R)\ G$ as the starting term of the reduction.

Remark 60. Observe that in Lemma 59 it is essential that the head of the considered redex in G is not inside a cycle. As a counterexample, consider the ρ_g-term $R = f(x) \twoheadrightarrow g(x)$ corresponding to the term graph rewrite rule $(\{x_1 \mid x_1 = f(x)\}, \{x_1 \mid x_1 = g(x)\})$.

https://doi.org/10.1017/S0960129507006093 Published online by Cambridge University Press

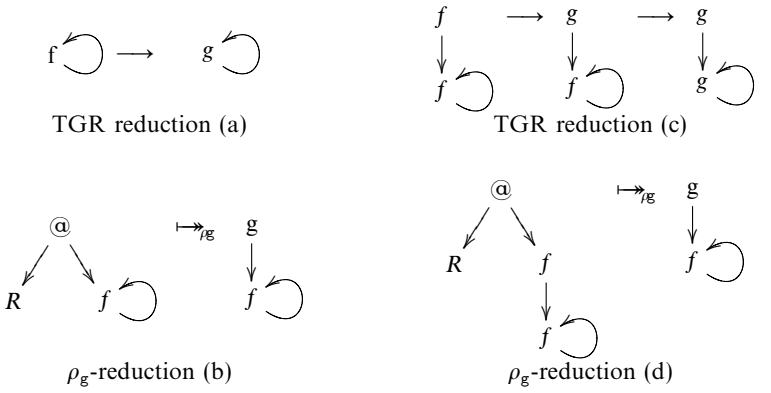


Figure 15. Counterexample for cyclic terms.

In a TGR, applying the given rule once to the term graph G, we get the reduction $G = \{y \mid y = f(y)\} \to \{y \mid g(y)\} = G_2$, where all the symbols 'f' are reduced simultaneously (see Figure 15 (a)).

We now consider the corresponding cyclic ρ_g-term $G = y\ [y = f(y)]$. We have the following ρ_g-calculus reduction (see Figure 15 (b)):

$$\begin{aligned} f(x) \twoheadrightarrow g(x)\ y\ [y = f(y)] &\mapsto\!\!\!\!\rightarrow_{\rho g}\ g(x)\ [f(x) \ll y, y = f(y)] \\ &\mapsto\!\!\!\!\rightarrow_{\rho g}\ g(x)\ [x = y, y = f(y)] \\ &\mapsto\!\!\!\!\rightarrow_{\rho g}\ g(y)\ [y = f(y)] = G_1 \end{aligned}$$

In this case only the top 'f' symbol is rewritten to 'g', while the rest of the term remains unchanged. This example reveals a clear difference in the behaviour of the two formalisms. What happens in the ρ_g-calculus is essential if we are to have a calculus that is confluent while still being coherent with the view of term graphs as objects defined up to bisimulation.

In fact, consider the ρ_g-term $G' = f(y)\ [y = f(y)]$, which is bisimilar to G, and which can be obtained from G with a step of external substitution. Then the ρ_g-term $(f(x) \twoheadrightarrow g(x))\ G'$ again reduces to G_1 (see Figure 15 (d)) and thus the rule $f(x) \twoheadrightarrow g(x)$ is no longer available. Instead, in the TGR, starting from G', we could obtain a term graph bisimilar to G_2 by applying the given rewrite rule *twice* (see Figure 15 (c)). This is not possible in the ρ_g-calculus, where the application of a rule consumes the rule itself.

Note that if we started in the proof of the previous lemma with a ρ_g-graph equal up to variable renaming to G, say G', we could have constructed an analogous reduction in the ρ_g-calculus. Indeed, in this case, by using the same reasoning as above, we obtain $G'_1 = \mathcal{P}_p(G')_{[R']_p}\ [E_G]$ as the final term, and this has a flat form equal up to variable renaming to $G'_{[\sigma(R)]_p}$, which is in turn equal up to variable renaming to $G_{[\sigma(R)]_p}$, so the lemma still holds.

Corollary 61. Let G be a term graph and let (L, R) be a rewrite rule such that $G_{\lceil\sigma(z)=t\rceil_p} \to G_{\lceil\sigma(R)\rceil_p} = G_1$, with $\sigma(z)$ not in a cycle. Then we can construct ρ_g-graph H such that

https://doi.org/10.1017/S0960129507006093 Published online by Cambridge University Press

for any term G' (whose flat form is) equal up to variable renaming to G there exists a reduction $(H\ G') \mapsto\!\!\!\!\rightarrow_{\rho g} G'_1$ such that (the flat form of) G'_1 is equal up to variable renaming to G_1.

The final ρ_g-graph we obtain is not exactly the same as the term graph resulting from the ρ_g-reduction in the TGR, and this is due to some unsharing steps that may occur in the reduction. In general, the two graphs are equal up to variable renaming, meaning that the ρ_g-graph G'_1 is possibly more 'unravelled' than the term graph G_1.

Example 62 (Addition). Let (L, R), where $L = x_1\{x_1 = add(x_2, y_1),\ x_2 = s(y_2)\}$ and $R = x_1\{x_1 = s(x_2),\ x_2 = add(y_1, y_2)\}$, be a term graph rewrite rule describing the addition of natural numbers. We then apply this rule to the term graph $G = z\{z = s(z_0), z_0 = add(z_1, z_2), z_1 = s(z_2), z_2 = 0\}$ using the variable renaming $\sigma = \{x_1/z_0, x_2/z_1, y_1/z_2, y_2/z_2\}$. This gives the term graph $G' = z\{z = s(z_0), z_0 = s(z'_1), z'_1 = add(z_2, z_2), z_2 = 0\}$. See Figure 14 for a graphical representation.

The corresponding reduction in the ρ_g-calculus is as follows. First, since the rule is not applied at the head position of G, we need to define the ρ_g-graph $H = s(x) \multimap s((L \multimap R)\ x)$ that pushes down the rewrite rule to the right application position, that is, under the symbol s. Then, applying the ρ_g-graph H to G, we obtain the following reduction:

$$
\begin{aligned}
s(x) \multimap s((L \multimap R)\ x)\ \ G & \\
&\mapsto_{\rho}\ \ s((L \multimap R)\ x)\ [s(x) \ll G] \\
&\mapsto_{p}\ \ s((L \multimap R)\ x)\ [s(x) \ll z, E_G] \\
&\mapsto\!\!\!\!\rightarrow_{\rho g}\ \ s((L \multimap R)\ x)\ [x = z_0, E_G] \\
&\mapsto_{es,gc}\ \ s((L \multimap R)\ z_0)\ [E_G] \\
&\mapsto_{\rho}\ \ s(R\ [L \ll z_0])\ [E_G] \\
&\mapsto\!\!\!\!\rightarrow_{\rho g}\ \ s(R\ [y_1 = z_2, y_2 = z_2])\ [E_G] \\
&=\ \ s(x_1\ [x_1 = s(x_2), x_2 = add(y_1, y_2)]\ [y_1 = z_2, y_2 = z_2])\ [E_G] \\
&\mapsto\!\!\!\!\rightarrow_{\rho g}\ \ s(x_1\ [x_1 = s(x_2), x_2 = add(z_1, z_2)])\ [E_G] = G''
\end{aligned}
$$

The canonical form of G'' is then obtained by removing the useless recursion equations in E_G and merging the lists of constraints. Then we get the graph $\overline{G''} = x\ [x = s(x_1), x_1 = s(x_2), x_2 = add(z_1, z_2), z_0 = 0]$, which is equal up to variable renaming to the term graph G'.

Theorem 63 (Completeness). Given an n step reduction $G \mapsto\!\!\!\!\rightarrow^n G_n$ in a TGR such that the heads of the n redexes are not in a cycle, there exist n ρ_g-graphs $H_1, \ldots, H_n$ such that $(H_n\ \ldots\ (H_1\ G)) \mapsto\!\!\!\!\rightarrow_{\rho g} G'_n$, and there exists a variable renaming τ such that $\tau(\overline{G'_n}) = G_n$.

Proof. The proof is by induction on the length of the reduction, using Lemma 59 and Corollary 61. □

https://doi.org/10.1017/S0960129507006093 Published online by Cambridge University Press

6. Conclusions and future work

We have introduced the ρ_g-calculus, which is an extension of the ρ-calculus able to deal with graph like structures, where sharing of sub-terms and cycles (which can be used to represent regular infinite data structures) can be expressed.

The ρ_g-calculus is shown to be confluent under suitable linearity assumptions over the considered patterns. The confluence result is obtained by adapting some techniques for confluence of term rewrite systems to the case of terms with constraints. An additional complication is caused by the fact that ρ_g-graphs are considered modulo an equational theory, and thus the rewriting relation formally acts on equivalence classes of terms. Since the ρ_g-calculus rewrite relation is not terminating, the 'finite development method' of the λ-calculus, together with several properties of 'rewriting modulo a set of equations', is needed to obtain the final result, thus making the complete proof quite elaborate.

The ρ_g-calculus has been shown to be an expressive calculus, being able to simulate standard ρ-calculus as well as cyclic λ-calculus and term graph rewriting (TGR). The main difference between the ρ_g-calculus and TGR lies in the fact that rewrite rules and their control (application position) are defined at the object-level of the ρ_g-calculus, while in the TGR the reduction strategy is left implicit, a discrepancy that also shows up in the technical comparison. The possibility of controlling the application of rewrite rules is particularly useful when the rewrite system is not terminating. It would certainly be interesting to define in the ρ_g-calculus iteration strategies and strategies for the generic traversal of ρ_g-graphs in order to simulate TGR rewritings guided by a given reduction strategy. Similar work has already been done for the representation of first-order term rewriting reductions in a typed version of the ρ-calculus (Cirstea *et al.* 2003). Intuitively, the ρ-term encoding of a first-order rewrite system is a ρ-structure consisting of the corresponding term rewrite rules wrapped in an iterator that allows for the repetitive application of the rules. We conjecture that this approach can be adapted and generalised in order to handle term-graphs and to simulate term-graphs reductions. As with term rewriting systems, this will require us to limit ourselves to suitable classes of TGRs (for example, confluent and terminating).

At the same time, an appealing problem is the generalisation of ρ_g-calculus to deal with different, possibly non-syntactic, matching theories. General cyclic matching, namely matching involving cyclic left-hand sides, could be useful, for example, for the modelling of reactions on cyclic molecules or transformations on distribution nets. One should notice that this extension is not straightforward, since in the ρ_g-calculus matching is internalised rather than being carried out at the metalevel.

Furthermore, a term of the ρ_g-calculus, which may have 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, for example, from the work on the infinitary λ-calculus (Kennaway *et al.* 1997) and on infinitary rewriting (Kennaway *et al.* 1995; Corradini 1993). On the other hand, to enforce the view of the ρ_g-calculus as an efficient implementation of terms and rewriting in the infinitary ρ-calculus, one should have an adequacy result in the style of Kennaway *et al.* (1994) and Corradini

https://doi.org/10.1017/S0960129507006093 Published online by Cambridge University Press

and Drewes (1997). In particular, for the ρ_g-calculus restricted to acyclic terms, adequacy could be investigated with respect to the standard ρ-calculus. Such a result could then formally justify the use of the acyclic ρ_g-calculus as an 'implementation' of the ρ-calculus. From this point of view, it would be worthwhile continuing the study of suitable strategies that will allow us to keep the sharing of terms as long as possible.

References

Abadi, M., Cardelli, L., Curien, P.-L. and Lévy, J.-J. (1991) Explicit Substitutions. *Journal of Functional Programming* **4** (1) 375–416.

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

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

Barendregt, H. (1984) *The Lambda-Calculus, its syntax and semantics*, Second edition, Studies in Logic and the Foundation of Mathematics, Elsevier Science Publishers.

Barendregt, H. P., van Eekelen, M. C. J. D., Glauert, J. R. W., Kennaway, J. R., Plasmeijer, M. J. and Sleep, M. R. (1987) Term Graph Rewriting. In: Proceedings of PARLE'87, Parallel Architectures and Languages Europe. *Springer-Verlag Lecture Notes in Computer Science* **259** 141–158.

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

Bertolissi, C. (2005) *The graph rewriting calculus: properties and expressive capabilities*, Thèse de Doctorat d'Université, Institut National Polytechnique de Lorraine, Nancy, France.

Church, A. (1941) A Formulation of the Simple Theory of Types. *Journal of Symbolic Logic* **5** 56–68.

Cirstea, H., Faure, G. and Kirchner, C. (2005) A rho-calculus of explicit constraint application. Proceedings of WRLA (Workshop on Rewriting Logic and Applications, Barcelona, Spain. March 2004). *Electronic Notes in Theoretical Computer Science* **117** 51–67.

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

Cirstea, H., Kirchner, C. and Liquori, L. (2001) Matching Power. In: Middeldorp, A. (ed.) Proceedings of RTA'01, Rewriting Techniques and Applications, Utrecht, The Netherlands. *Springer-Verlag Lecture Notes in Computer Science* **2051** 77–92.

Cirstea, H., Kirchner, C. and Liquori, L. (2002) Rewriting Calculus with(out) Types. *Electronic Notes in Theoretical Computer Science* **71** 3–19.

Cirstea, H., Liquori, L. and Wack, B. (2003) Rewriting Calculus with Fixpoints: Untyped and First-order Systems. In: Berardi, S., Coppo, M. and Damian, F. (eds.) Types for Proofs and Programs (TYPES). *Springer-Verlag Lecture Notes in Computer Science* **3085** 147–171.

Colmerauer, A. (1984) Equations and Inequations on Finite and Infinite Trees. In: *Proceedings of FGCS'84* 85–99.

Corradini, A. (1993) Term Rewriting in CT_Σ. In: Gaudel, M.-C. and Jouannaud, J.-P. (eds.) *Proceedings of TAPSOFT'93, Theory and Practice of Software Development—4th International Joint Conference CAAP/FASE*, Springer-Verlag 468–484.

Corradini, A. and Drewes, F. (1997) (Cyclic) Term Graph Rewriting is Adequate for Rational Parallel Term Rewriting. Technical Report TR-97-14, Dipartimento di Informatica, Pisa.

Corradini, A. and Gadducci, F. (1999) Rewriting on Cyclic Structures: Equivalence of Operational and Categorical Descriptions. *Theoretical Informatics and Applications* **33** 467–493.

Courcelle, B. (1980) Infinite Trees in Normal Form and Recursive Equations having a unique solution. *Math. Syst. Theory*.

Huet, G. (1980) Confluent Reductions: Abstract Properties and Applications to term Rewriting Systems. *Journal of the ACM* **27** (4) 797–821. (Preliminary version in 18th Symposium on Foundations of Computer Science, IEEE, 1977.)

Jouannaud, J.-P. and Kirchner, H. (1986) Completion of a set of rules modulo a set of Equations. *SIAM Journal of Computing* **15** (4) 1155–1194.

Kennaway, J. R., Klop, J. W., Sleep, M. R. and de Vries, F.-J. (1994) On the Adequacy of Graph Rewriting for Simulating Term Rewriting. *ACM Transactions on Programming Languages and Systems* **16** (3) 493–523.

Kennaway, J. R., Klop, J. W., Sleep, M. R. and de Vries, F.-J. (1997) Infinitary Lambda Calculus. *Theoretical Computer Science* **175** (1) 93–125.

Kennaway, J. R., Klop, J. W., Sleep, M. R. and de Vries, F.-J. (1995) Transfinite reductions in orthogonal term rewriting systems. *Information and Computation* **119** (1) 18–38.

Kirchner, C. (ed.) (1990) *Unification*, Academic Press.

Klop, J. W. (1980) *Combinatory Reduction Systems*, Ph.D. thesis, CWI.

Lankford, D. S. and Ballantyne, A. (1977) Decision procedures for simple equational theories with permutative axioms: complete sets of permutative reductions. Technical report, Univ. of Texas at Austin, Dept. of Mathematics and Computer Science.

Ohlebusch, E. (1998) Church–Rosser Theorems for Abstract Reduction Modulo an Equivalence Relation. In: Nipkow, T. (ed.) Proceedings of the 9th International Conference on Rewriting Techniques and Applications (RTA-98). *Springer-Verlag Lecture Notes in Computer Science* **1379** 17–31.

Parigot, M. (1992) $\lambda\mu$-Calculus: An Algorithmic Interpretation of Classical Natural Deduction. In: Voronkov, A. (ed.) Proceedings of LPAR'92, Logic Programming and Automated Reasoning, St Petersburg, Russia, July 1992. *Springer-Verlag Lecture Notes in Artificial Intelligence* **624** 190–201.

Peterson, G. and Stickel, M. E. (1981) Complete Sets of Reductions for Some Equational Theories. *Journal of the ACM* **28** 233–264.

Peyton-Jones, S. (1987) *The implementation of functional programming languages*, Prentice Hall.

Plump, D. (1999) Term graph rewriting. *Handbook of Graph Grammars and Computing by Graph Transformation* **2** 3–61.

Sleep, M. R., Plasmeijer, M. J. and van Eekelen, M. C. J. D. (eds.) (1993) *Term graph rewriting: theory and practice*, Wiley.

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

Wack, B. (2005) *Typage et déduction dans le calcul de réécriture*, Thèse de doctorat, Université Henri Poincaré – Nancy I.