HAL
open science

Priority-Independent Rewrite Systems for Pointer-based Data-Structures

Rachid Echahed, Nicolas Peltier

To cite this version:

Rachid Echahed, Nicolas Peltier. Priority-Independent Rewrite Systems for Pointer-based Data-Structures. 2010. hal-00940674

HAL Id: hal-00940674

https://hal.science/hal-00940674v1

Submitted on 2 Feb 2014

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Priority-Independent Rewrite Systems for Pointer-based Data-Structures

Rachid Echahed and Nicolas Peltier

January 2010

Abstract

We define a syntactic class of graphs and graph rewrite systems for which the normal forms are independent from the order in which the nodes are reduced. This result, that is not covered by existing approaches in graph rewriting, allows us to devise simple confluence criteria and efficient normalization algorithms. It is based on a static analysis of the rewrite system, including a thorough analysis of the shape of the graphs generated during the rewriting process. The considered graphs naturally encode pointer-based data structures that are commonly used in practical programming and the rewrite rules can simulate any elementary transformation on these data structures (edge redirection, node relabeling etc.).

Rewriting is a central paradigm in declarative programming. Most algorithms can be conveniently specified as sets of oriented equations, interpreted as rewrite rules. Efficient reduction strategies have been designed for computing the normal forms and narrowing algorithms can be used to solve goals. Unfortunately, some algorithms (e.g. the Schorr-Waite algorithm [15] or in-situ algorithms for sorting or reversing a list) are difficult (or even impossible) to specify by term rewrite rules (except by using inefficient encodings) because they strongly rely on the use of complex pointer-based data-structures and pointer redirections. The most natural way to specify these algorithms is to use rewrite rules operating on *graphs*, that are a very convenient and precise formalism for denoting the data-structures used in everyday programming. Numerous approaches exist in graph rewriting, with various kinds of graphs and rewrite systems. Most of them use algebraic formalisms based on category theory (using single or double push-outs). Many implementations have been developed and used for solving practical problems: GraphSynth, AGG, PROGRES,... We refer to [8, 10, 7] for overviews of this research domain and additional references.

A drawback of graph rewrite systems is that confluence is very hard to satisfy [13, 1]. Confluence is an essential property, especially when rewrite rules are used to define functions, because it ensures that the normal forms are unique and enables more efficient reduction algorithms. Unfortunately, graph rewrite systems are seldom confluent, because two distinct defined functions may operate on the same nodes or edges. Even a rewrite system as simple as $f(x) \to x, g(x) \to x$ turns out to be non confluent when applied on the cyclic term-graph $\alpha = f(g(\alpha))$ (even modulo isomorphism). This is even more obvious if the defined functions affect their arguments (e.g. by redirecting edges). Almost all non trivial graph rewrite systems have non joinable critical pairs. Assume for instance that one defines a function *rev* that *physically*[1] reverses a list and a function *length* returning the length of a list. Then the normal form of the graph (DAG) $f(\mathit{length}(\gamma)), \mathit{rev}(\gamma))$ where $\gamma = [1, 2]$ obviously depends on the order on which the arguments of f are evaluated. If $\mathit{length}(\gamma)$ is evaluated first we get $f(2, [2, 1])$. But if $\mathit{rev}(\gamma)$ is evaluated first we get $f(1, [2, 1])$ since after the evaluation of $\mathit{rev}([1, 2])$ the argument of *length* (i.e. the cell containing 1) becomes the *last* cell of the list.

In [5] we have proposed to overcome this problem by ordering the nodes of the graphs. In this approach, a *priority ordering* must be provided by the programmer to specify the order in which the nodes are to be reduced, thereby ensuring the uniqueness of the normal form (for orthogonal rewrite systems). The drawback is that a set of relations, called the *dependency schema*, needs to be computed during the reduction of the graphs, in order to handle the dependencies between the defined functions operating on the same edges and nodes. This entails some computational overcost which may be unacceptable for applications in which efficiency is critical and/or for which the considered graphs are very large. This paper proposes another solution to the confluence problem in graph rewriting, which completely avoids the use of any priority ordering

[1] i.e. by redirecting existing edges instead of created new cells.

between the nodes (thus removing any computational overcost) and relies instead on a *static* analysis of the rewrite system. More precisely, we devise an algorithm that, given a graph rewrite system $\mathcal{R}$ and a graph t, effectively detects that the normal forms of t w.r.t. $\mathcal{R}$ do not depend on the order of reduction of the nodes (not only the nodes occurring in t, but also those introduced during rewriting). The graphs and rules for which the order of reduction is relevant are called *clash-reducible*. The algorithm runs in polynomial time w.r.t. the size of $\mathcal{R}$ and t. Furthermore, the most important part of the computation may be performed once and for all for a given graph rewrite system (see Section 5).

This result has many important applications that are of great interest in the context of declarative programming. Firstly, confluence criteria are easy to define for non clash-reducible graphs: it suffices to check that two distinct rules cannot be applied on the *same* node[2]. Secondly, efficient evaluation strategies can be defined, without any computational overcost w.r.t. the usual approaches in term rewriting (the computation of the dependency schemata which was required in [6] is avoided). Furthermore, it makes possible to use the narrowing algorithm of [4] to solve goals and synthesize complex data-structures. This last point is interesting because this narrowing algorithm is *not* compatible with priorities[3].

Determining whether a graph rewrite system is clash-reducible or not is an undecidable problem. Thus the class of systems that are accepted by our algorithm, that are called *well-defined*, is strictly contained into the class of non clash-reducible systems. Roughly speaking, our algorithm checks that if two distinct defined functions f and g operate on the same node (or edge) γ in the reduced graph, then neither f nor g can physically affect γ and – more importantly – that this property is preserved during the rewriting process. This is done by automatically analyzing the shape of the graphs occurring in the derivation in order to find *decidable approximations* of the graph obtained after rewriting. It is worth to mention that well-defined graph rewrite systems *may have non joinable critical pairs* in the sense that there exist graphs on which two distinct rules apply, yielding non joinable results. Thus confluence *cannot be proven* by traditional approaches. Our automated analysis ensures that those graphs will never occur in the derivation (if the initial graph and the rewrite rules are well-defined).

As illustrated by the examples in Sections 1.3 and 4, the class of well-defined graph rewrite systems is comprehensive enough to express many interesting graph transformation algorithms. If a graph rewrite system is not well-defined, then it is usually easy to construct a well-defined system that computes the same normal forms (in the worst case this may be done by composing graph transformations in a purely sequential way). Well-defined graph rewrite systems may be a good candidate for defining efficient declarative programming languages operating on graphs and offering similar features than the languages based on term rewriting such as Haskell or Curry: efficient (or even optimal) lazy evaluation strategies, etc. The use of well-defined graph rewrite systems would significantly increase the expressiveness of declarative languages. As far as we are aware, no existing approach in graph rewriting offers comparable features (in particular, the confluence criteria are much less general).

The rest of the paper is structured as follows. In Section 1 we introduce the basic terminology on graphs and graph rewrite rules. In Section 2 we introduce a semantic criterion ensuring that the results of the derivations do not depend on the reduction ordering. We define simple confluence criteria for such graphs and efficient reduction strategies. In Section 3, we introduce a syntactic class of graphs called *well-defined*, that satisfy the condition above. Section 4 provides examples of well-defined graph rewrite systems. All these systems are confluent on well-defined graphs, although they have non joinable critical pairs (modulo isomorphism). Section 5 discusses the possible use of the results and concludes the paper.

1 Graph Rewriting

In this section, we introduce our framework for graph rewriting. The definitions are much simpler than the ones of [5, 6] because we restrict ourselves to injective mappings.

1.1 Basic Definitions

We assume that the following sets of symbols are given: a set of *sorts* $\mathcal{S}$ (denoted by $\mathtt{s}, \mathtt{nat}, \mathtt{bool}, \ldots$) a set of *nodes* $\mathcal{N}$ (denoted by Greek letters) and a set of *function symbols* Σ (denoted by $f, g, \ldots$). Σ is divided into two disjoint sets of symbols: a set of *defined symbols* $\mathcal{D}$ and a set of *constructors* $\mathcal{C}$. Every node α is mapped to a unique sort $\mathtt{sort}(\alpha)$ and every function symbol f is mapped to a unique profile of the form $\mathtt{s}_1 \times \ldots \times \mathtt{s}_n \to \mathtt{s}$. n is called the arity of f and denoted by $ar(f)$.

[2]This (obviously decidable) condition is *not* sufficient in general.

[3]Defining efficient narrowing algorithms handling node ordering is a very difficult problem.

Definition 1 *A* term-graph *(or simply a* graph*) t is defined by:*

- *A set of* nodes $\mathcal{N}(t) \subseteq \mathcal{N}$.
- *A node* $root(t) \subseteq \mathcal{N}(t)$ *called the* root *of* t.
- *A partial function (also denoted by t) mapping node $\alpha \in \mathcal{N}(t)$ to a term of the form $f(\beta_1, \ldots, \beta_n)$ s.t. f is of profile $\mathtt{s}_1 \times \ldots \times \mathtt{s}_n \to \mathtt{s}$, $\beta_1, \ldots, \beta_n \in \mathcal{N}(t) \cup \{\bot\}$, $\mathtt{sort}(\alpha) = \mathtt{s}$ and $\forall i \in [1..n], \beta_i \neq \bot \Rightarrow \mathtt{sort}(\beta_i) = \mathtt{s}_i$.*

The symbol $\bot$ is used to denote undefined arguments. The set of nodes α s.t. $t(\alpha)$ is undefined is the set of *variables*, denoted by $var(t)$. If $t(\alpha) = f(\beta_1, \ldots, \beta_n)$ then f is called the *label* of the node α and denoted by $l_t(\alpha)$.

A *position* is a sequence of natural numbers. "." denotes the concatenation operator and ϵ denotes the empty sequence. If p is a position then $p_t(\alpha)$ denotes the node reachable from α following the position p:

$$p_t(\alpha) \stackrel{\text{def}}{=} \begin{cases} \alpha & \text{if } p \text{ is empty} \\ q_t(\beta_i) & \text{if } p = i.q,\ t(\alpha) = f(\beta_1, \ldots, \beta_n),\ i \in [1..n] \text{ and } \beta_i \neq \bot \\ \text{undefined} & \text{otherwise} \end{cases}$$

In particular, if i is a natural number, then $i_t(\alpha)$ is the i-th argument of $t(\alpha)$. The set of positions p s.t. $p_t(\alpha)$ is defined is written $pos_t(\alpha)$. If i is a natural number and if $i_t(\alpha) = \beta$ then we say that t *contains an edge from α to β, labeled by i*. By definition, given a node α and a label i, there can be at most one edge starting from α and labeled by i and this edge points to $i_t(\alpha)$.

Two graphs t, s are *disjoint* iff they share no nodes: $\mathcal{N}(t) \cap \mathcal{N}(s) = \emptyset$. We write $t \subseteq s$ iff $\mathcal{N}(t) \subseteq \mathcal{N}(s)$ and $\forall i \in \mathbb{N} \cup \{l\}, i_t(\alpha) \neq \bot \Rightarrow i_s(\alpha) = i_t(\alpha)$.

t, s are *compatible* iff there exists a graph containing both t and s, i.e. if the following property holds: $\forall \alpha \in \mathcal{N}(t) \cap \mathcal{N}(s), \forall i \in \mathbb{N} \cup \{l\}$, $(i_t(\alpha) \neq \bot \wedge i_s(\alpha) \neq \bot) \Rightarrow i_t(\alpha) = i_s(\alpha)$.

A *path* in a graph t from a node α to β is a sequence of nodes $\gamma_1, \ldots, \gamma_n$ s.t. $\gamma_1 = \alpha, \gamma_n = \beta$ and for every $i \in [1..n-1]$, t contains an edge from γ_i to γ_{i+1}. If moreover every node in $\gamma_1, \ldots, \gamma_n$ is labeled by a constructor then $\gamma_1, \ldots, \gamma_n$ is a *constructor path*. We write $\alpha \to_t \beta$ (resp. $\alpha \stackrel{\mathcal{C}}{\to}_t \beta$) if there exists a path (resp. a constructor path) from α to β. Obviously, $\alpha \to_t \beta$ iff $\beta = t_p(\alpha)$ for some position p. $\to_t$ is reflexive but not $\stackrel{\mathcal{C}}{\to}_t$ ($\stackrel{\mathcal{C}}{\to}_t$ is reflexive only on nodes labeled by constructors).

α, β are *joinable* (written $\alpha \downarrow_t \beta$) if there exists a node γ s.t. $\alpha \to_t \gamma$ and $\beta \to_t \gamma$. If moreover $\mathtt{sort}(\gamma) = \mathtt{s}$, then we write $\alpha \downarrow_t^{\mathtt{s}} \beta$.

Denoting graphs by labeled terms

For the sake of conciseness and readability, we use a linear notation for denoting graphs. The term $\alpha{:}f(t_1, \ldots, t_n)$ will be used as an abbreviation for denoting a graph t of root α, containing $t_1, \ldots, t_n$ and s.t. $t(\alpha) = f(root(t_1), \ldots, root(t_n))$. We assume that $t_1, \ldots, t_n$ are compatible and that α is a non variable node in $t_1, \ldots, t_n$. The node α can be left unspecified, and in this case it is simply replaced by an arbitrary node not occurring elsewhere. A term $\alpha \in \mathcal{N}$ denotes a graph with a unique node α and no edge (i.e. a variable).

1.2 Some Basic Operations on Graphs

Union

If t, s are two graphs, we denote by $t; s$ the graph u defined as follows:

- $\mathcal{N}(u) \stackrel{\text{def}}{=} \mathcal{N}(t) \cup \mathcal{N}(s)$ and $root(u) \stackrel{\text{def}}{=} root(t)$.
- $\forall \alpha \in \mathcal{N}(u)$, $\forall i \in \mathbb{N} \cup \{l\}$:

$$i_t(\beta) \stackrel{\text{def}}{=} \begin{cases} i_s(\beta) & \text{if } i_s(\beta) \text{ is defined.} \\ i_t(\beta) & \text{if } i_s(\beta) \text{ is undefined and } i_t(\beta) \text{ is defined.} \\ \text{undefined} & \text{if both } i_s(\beta) \text{ and } i_t(\beta) \text{ are undefined} \end{cases}$$

Informally, $t; s$ is the union of t and s, where the labels/edges in s "overwrite" those in t. Notice that ";" is commutative iff t and s are compatible and have the same root.

Replacement

If t is a graph, we denote by $t|_\alpha$ the subgraph s of t of root α, formally defined as follows:

- $\mathcal{N}(s) \stackrel{\text{def}}{=} \{\beta \mid \alpha \to_t \beta\}$.
- $root(s) \stackrel{\text{def}}{=} \alpha$.

- $\forall \beta \in \mathcal{N}(s), s(\alpha) \stackrel{\text{def}}{=} t(\alpha)$.

This notation extends naturally to positions: $t|_p$ denotes the graph $t|_\alpha$ where $\alpha = p_t(root(t))$.

If t is a graph and α, β are two nodes s.t. $\texttt{sort}(\alpha) = \texttt{sort}(\beta)$, we denote by $t_{\beta/\alpha}$ the graph s obtained by redirecting every edge pointing to α to the node β, formally defined as follows:

- $\mathcal{N}(s) \stackrel{\text{def}}{=} (\mathcal{N}(t) \setminus \{\alpha\}) \cup \{\beta\}$, $root(s) \stackrel{\text{def}}{=} \begin{cases} root(t) & \text{if } \alpha \neq root(t) \\ \beta & \text{if } \alpha = root(t) \end{cases}$
- $\forall \gamma \in \mathcal{N}(s)$, $l_s(\gamma) \stackrel{\text{def}}{=} l_t(\gamma)$ and $\forall i \in \mathbb{N}$, $i_s(\gamma) \stackrel{\text{def}}{=} \begin{cases} \beta & \text{if } i_s(\gamma) = \alpha. \\ i_t(\gamma) & \text{otherwise} \end{cases}$

Then $t[s]_\alpha$ denotes the graph: $t[s]_\alpha \stackrel{\text{def}}{=} t_{root(s)/\alpha}; s$.

Renamings

A *renaming* σ is a partial *injective*[4] function from $\mathcal{N}$ to $\mathcal{N}$. As usual, σ is extended into an homomorphism on the terms built on $\Sigma \cup \mathcal{N} \cup \{\bot\}$. If σ is a renaming and t is a graph, $\sigma(t)$ denotes the graph s defined as follows:

- $\mathcal{N}(s) \stackrel{\text{def}}{=} \{\sigma(\alpha) \mid \alpha \in \mathcal{N}(t)\}$.
- $root(s) \stackrel{\text{def}}{=} \sigma(root(t))$.
- $s(\sigma(\alpha)) \stackrel{\text{def}}{=} \sigma(t(\alpha))$.

Notice that s is necessarily well-defined since σ is injective. We write $t \equiv s$ if $s = \sigma(t)$ for some renaming σ.

1.3 Graph Rewrite Rules

Definition 2 *(Rewrite Rule) A* graph rewrite rule *is an expression of the form $L \to R$ where L, R are graphs s.t.* $\texttt{sort}(root(L)) = \texttt{sort}(root(R))$. *A* graph rewrite system *(GRS for short) is a set of rewrite rules.*

Our definition is close to the ones in [2, 3]. However, L and R are not necessarily compatible: the nodes in L can be redefined (as in the rule: $f(\alpha{:}0) \to \alpha{:}s(0)$). Thanks to the use of injective mappings, this definition is simpler, more condensed and much more natural than the one used in [6]. It is also very close to term rewriting rules. In [6], node constraints were attached to the rule and the right-hand side R was defined as a sequence of elementary actions, operating on the left-hand side L. This is needed if non injective mappings are considered because in this case the image of the right-hand side is not necessarily a graph (the same node may have two incompatible definitions, for instance if the rule $f(\alpha{:}a, \beta{:}a) \to f(\alpha{:}a, \beta{:}b)$ is applied on $f(\alpha{:}a, \alpha)$).

$\mathcal{N}(\rho) \stackrel{\text{def}}{=} \mathcal{N}(L) \cup \mathcal{N}(R)$ denotes the set of nodes occurring in ρ. $\mathcal{C}N(\rho) \stackrel{\text{def}}{=} \mathcal{N}(R) \setminus \mathcal{N}(L)$ denotes the set of nodes occurring in R but not in L, i.e. the nodes that are *created* by the rewrite rule. Finally, $\mathcal{R}N(\rho)$ is the set of nodes $\alpha \in \mathcal{N}(L)$ s.t. either $\alpha = root(L)$ or $R(\alpha) \neq L(\alpha)$. The nodes in $\mathcal{R}N(\rho)$ are *affected* by ρ in the sense that either α is relabeled or an edge starting from α is redirected ($\mathcal{R}$ stands for "redirected").

A rule $\rho = L \to R$ is:

- **Connex** iff for every node $\alpha \in \mathcal{N}(L)$, $root(L) \to_L \alpha$. Every node occurring in the left-hand side must be reachable from the root.
- **Constructor-based** iff $l_L(\alpha) \in \mathcal{D} \Leftrightarrow \alpha = root(L)$. The root is the only node that is labeled by a non constructor symbol. This condition is usual in constructor-based rewrite systems.
- **Safe** iff:
  - For every node α, if $L(\alpha)$ is defined then $R(\alpha)$ is defined.
  - $root(L)$ does not occur in R.
  - If $l_R(\alpha) \in \mathcal{D}$ then α is a created node (only the created nodes may be labeled by a defined symbol).

These last three conditions are technically convenient and obviously non restrictive.

From now on, we assume that all the rules are connex, constructor-based and safe.

Definition 3 *Let $\rho : L \to R$ be a rule. A* ρ-matcher *for a graph t* at *a node $\alpha \in \mathcal{N}(t)$ is a renaming σ satisfying the following conditions.*

[4] In this paper we only consider injective mappings in contrast to what is done in [5, 6] in which explicit equational constraints were attached to the graphs. The use of injective mappings is very convenient from a theoretical point of view and is well-known to be non restrictive, although it should be avoided in practice because it may exponentially increase the size of the graphs.

1. *$\sigma(L) \subseteq t$ (i.e. $\sigma(L)$ must be a subgraph of t).*

2. *$\alpha = \sigma(root(L))$.*

3. *If β is a node in $\sigma(L)$ distinct from α, then $l_t(\beta) \notin \mathcal{D}$.*

ρ-matchers can be easily computed using standard matching algorithms for graphs. Conditions 1 and 2 are very natural: they ensure that the left-hand side of ρ matches the subgraph of t at root α. Condition 3 may seem rather restrictive but it is needed to delay the application of a rule until its left-hand side is fully known. We will come back to this problem later (see Example 1). Condition 3 does *not* prevent the use of lazy evaluation strategies. It merely ensures that the *names* of the nodes are known before applying the graph-rewrite rules, which is rather natural since the rule may depend on these names (this is not the case of term rewrite systems).

Let $\rho = L \to R$, we write $t \to_{\rho,\sigma} s$ if $s = t[\sigma(R)]_\alpha$, where σ is a ρ-matcher for t at α. We write $t \to_\rho s$ if $t \to_{\rho,\sigma} s$ for some renaming σ and $t \to_{\mathcal{R}} s$ iff there exists a rule $\rho \in \mathcal{R}$ s.t. $t \to_\rho s$. As usual, $\to^*$ denotes the reflexive and transitive closure of the relation $\to$. A *derivation from* a graph t_1 *in* a GRS $\mathcal{R}$ is a sequence of graphs $t_1, \ldots, t_n$ s.t. for every $i \in [1..n-1]$ $t_i \to_{\mathcal{R}} t_{i+1}$.

If s is obtained from a t by applying a rule at a node α, and if β is a node in s then β^- denotes the "ancestor" of the node β: $\beta^- \stackrel{\text{def}}{=} \beta$ if β is a node in t and $\beta^- \stackrel{\text{def}}{=} \alpha$ otherwise.

Examples of Graph Rewrite Systems

The following rules define a function physically appending two (distinct) lists. We use the constructor *cons* and *nil* to denote lists (we also use Prolog notations: $cons(1, cons(2, \ldots, cons(n, nil)\ldots))$ is denoted by $[1, 2, \ldots, n]$ and $cons(1, cons(2, \ldots, cons(n, \alpha)\ldots))$ by $[1, 2, \ldots, n|\alpha]$).

$$append(nil, \alpha) \to \alpha \quad \zeta{:}append(\alpha{:}cons(\beta, \gamma), \delta) \to \alpha{:}cons(\beta, append(\gamma, \delta))$$

The definition is almost identical to the usual one, except that we reuse the node α in the right-hand side of the second rule instead of creating a new node.

The following rules define an in-situ reverse function:

$$\begin{array}{ll} rev(\alpha) \to rev'(\alpha, nil) & rev'(nil, \alpha) \to \alpha \\ rev'(\alpha{:}cons(\beta, \gamma), \delta) \to rev'(\gamma, \alpha{:}cons(\beta, \delta)) & \end{array}$$

The following function replaces the leaves c, d, h in a binary tree by new constant symbols $\clubsuit, \diamondsuit, \heartsuit$ respectively. The trees are represented by $g(\alpha, \beta)$, where α, β are the two children. No stack is used and the algorithm works by redirecting the edges in the graph (so that no additional memory is needed). We use a constant symbol *dummy* and two constructors g', g'' to mark the nodes in the tree s.t. the first (resp. second) child has already been handled.

$$\begin{array}{lcll} inc(\alpha) & \to & inc'(\alpha, dummy) & \\ inc'(\alpha{:}dummy, \beta) & \to & \beta & \\ inc'(\alpha{:}x, \beta) & \to & inc'(\beta, \alpha{:}\star) & \text{where } x, \star \in \{(c, \clubsuit), (d, \diamondsuit), (h, \heartsuit)\} \\ inc'(\alpha{:}g(\beta, \gamma), \delta) & \to & inc'(\beta, \alpha{:}g'(\delta, \gamma)) & \\ inc'(\alpha{:}g'(\beta, \gamma), \delta) & \to & inc'(\gamma, \alpha{:}g''(\delta, \beta)) & \\ inc'(\alpha{:}g''(\beta, \gamma), \delta) & \to & inc'(\gamma, \alpha{:}g(\beta, \delta)) & \end{array}$$

Notice that if a straightforward algorithm is used then in the worst case $o(n)$ additional nodes would be created, where n is the size of the graph. Our algorithm uses only 1 additional node (this is obvious since the only rule that contains a created node distinct from the root of the right-hand side is the first one, which applies only once).

For instance we have the derivation:

$$
\begin{array}{rcl}
inc(g(g(c,d),h)) & \to & inc'(g(g(c,d),h),dummy) \\
 & \to & inc'(g(c,d),g'(dummy,h)) \\
 & \to & inc'(c,g'(g'(dummy,h),d)) \\
 & \to & inc'(g'(g'(dummy,h),d),\clubsuit) \\
 & \to & inc'(d,g''(\clubsuit,g'(dummy,h))) \\
 & \to & inc'(g''(\clubsuit,g'(dummy,h)),\diamondsuit) \\
 & \to & inc'(g'(dummy,h),g(\clubsuit,\diamondsuit)) \\
 & \to & inc'(h,g''(g(\clubsuit,\diamondsuit),dummy)) \\
 & \to & inc'(g''(g(aaaa,\diamondsuit),dummy),\heartsuit) \\
 & \to & inc'(dummy,g(g(\clubsuit,\diamondsuit),\heartsuit)) \\
 & \to & g(g(\clubsuit,\diamondsuit),\heartsuit)
\end{array}
$$

The following example shows the importance of Condition 3 in Definition 3:

Example 1 *Consider the rules* $e(\alpha,\beta) \to false, e(\alpha,\alpha) \to true, g(\alpha) \to \alpha$ *and the graph:* $t = e(\gamma{:}g(\lambda),\delta{:}g(\lambda))$.

If Condition 3 *is not taken into account then* t *is reduced to false by the first rule. However, by applying the third rule on the nodes* γ *and* δ *both the nodes* γ *and* δ *are redirected to* λ*, thus the second rule applies and yields true. Condition* 3 *avoids this behavior by preventing the application of the first rule until both* α *and* β *are labeled by constructor symbols. This condition was omitted in [6] because it was handled by the dependency schema (it is obvious that the use of node constraints does not solve this problem).*

2 Clash-Reducible Graphs

2.1 Definition

Our goal is to ensure that the normal forms of a graph do not depend on the order in which the nodes are reduced. Intuitively, the reduction ordering matters when a function affects a node γ that is used by another function, as for instance in the term $f(\beta{:}length(\gamma{:}[1,2]),\alpha{:}rev(\gamma))$. In this case the result of the function depends on the order in which the nodes α and β are reduced[5]. This kind of configuration (which we want to avoid) will be called a *clash*. In order to formally define this notion, we need to introduce some additional terminology. The *constructor shape* of a graph t is the graph obtained from t by removing every variable and every node labeled by a defined symbol, except the root of the graph.

Formally, it is the graph s defined as follows:

- $\mathcal{N}(s) \stackrel{\text{def}}{=} \{root(t)\} \cup \{\beta \in \mathcal{N}(t) \mid l_t(\beta) \in \mathcal{C}\}$.
- $root(s) \stackrel{\text{def}}{=} root(t)$.
- $\forall \alpha \in \mathcal{N}(s)$, $s(\alpha) \stackrel{\text{def}}{=} f(\gamma_1,\ldots,\gamma_n)$ iff $t(\alpha) = f(\lambda_1,\ldots,\lambda_n)$ and $\forall i \in [1..n]$, $\gamma_i \stackrel{\text{def}}{=} \lambda_i$ if $\lambda_i \in \mathcal{N}(s)$ and $\gamma_i \stackrel{\text{def}}{=} \bot$ otherwise.

Let $\rho = L \to R$ be a rule. A renaming σ is a *potential* ρ*-matcher* for a graph t at a node α iff $\sigma(root(L)) = \alpha$ and $\sigma(L)$ is compatible with the constructor shape of $t|_\alpha$. Intuitively, σ is a renaming that may be extended into a matcher after some rewriting steps.

We write $\alpha \sqsupset_t^\rho \beta$ iff β occurs both in $\sigma(L)$ and in the constructor shape of $t|_\alpha$ and $\alpha \sqsupset_t^{\mathcal{R}} \beta$ iff $\exists \rho \in \mathcal{R}, \alpha \sqsupset_t^\rho \beta$. A node γ is *demanded* for α w.r.t. ρ if there exists a potential ρ-matcher σ and a node β s.t. $\alpha \sqsupset_t^\rho \beta$ and $\gamma = i_{\sigma(L)}(\beta)$.

Let $\rho : f(cons(\alpha,\beta),\alpha) \to true$ and $t = \gamma{:}f(cons(g(\delta),\lambda{:}nil),h(\delta))$, where g,h denote defined functions. The constructor shape of t is $\gamma{:}f(cons(\bot,\lambda{:}nil),\bot)$. This graph is compatible with the left-hand side of ρ and we have $\gamma \sqsupset_t^\rho \gamma,\lambda$. The nodes of $g(\delta)$ and $h(\delta)$ are demanded (but δ is not).

Definition 4 *Let* $\mathcal{R}$ *be a GRS. A* clash *for a graph* t *is a pair of distinct nodes* α,β *satisfying the following conditions:*

1. *There exist a rule* $\rho = L \to R$ *in* $\mathcal{R}$ *and a* ρ*-matcher* σ *at* α.
2. *There exists a node* $\gamma \in \mathcal{R}N(\sigma(\rho))$ *s.t.:*

[5] After the execution of *rev*, γ becomes the *last* cell of the list.

(a) Either $\beta = \gamma$ and $root(t) \xrightarrow{\mathcal{C}}_t \beta$

(b) or $\beta \sqsupset_t^{\mathcal{R}} \gamma$.

Proposition 1 *Let $\rho = L \to R$ and $\pi = G \to D$ be two rules, let σ be a renaming, and let t, s be two graphs s.t. $t \to_{\rho,\sigma} s$. Let α be a node in t, distinct from $root(\sigma(L))$.*

If θ is a π-matcher for s at α, then one of the following conditions hold:

- *t contains a clash.*
- *$root(\sigma(L))$ is demanded for α.*
- *θ is a π-matcher for t and π at α.*

Proof *Assume that θ is not a π-matcher for t. This means that $\theta(L) \not\subseteq t$. Since $\theta(L) \subseteq s$, a node $\beta = p_t(\alpha)$ where p is a position in G must be affected by ρ. Assume that p is the minimal position having this property. By definition, t and s coincide on every node $q_t(\alpha)$ where q is a proper prefix of p. Since π is constructor-based, every node in the path from α to β, except α and β, is labeled by a constructor. Thus this path occurs in the constructor shape of $t|_\alpha$, and β is demanded for α. If $\beta = root(\sigma(L))$ then the proof is completed. Otherwise, β must occur in $\sigma(L)$ thus must be labeled by a constructor symbol and in this case we have $\alpha \sqsupset_t^{\pi} \beta$. By Condition 2b in Definition 4 t contains a clash.*

The following lemma states that two rules applicable on distinct nodes of a given graph containing no clash necessarily commute.

Lemma 1 *Let t be a graph containing no clash. Let $\rho = L \to R, \pi = G \to D$ be two rules. Let s, u be two graphs s.t. $t \to_{\rho,\sigma} s$, $t \to_{\pi,\theta} u$. Assume that s, u share no nodes except those occurring in t. Let $\alpha = \sigma(root(L)), \beta = \theta(root(G))$. If $\alpha \neq \beta$ then there exists a graph v s.t. $s \to_{\pi,\theta} v$, $u \to_{\rho,\sigma} v$.*

Proof *If $\mathcal{R}N(\sigma(R)) \cap \mathcal{N}(\theta(G))$ contains a node γ then it is easy to check that all the conditions in Definition 4 are satisfied (in particular Condition 2b) thus α, β is a clash in t. Therefore $\mathcal{R}N(\sigma(R)) \cap \mathcal{N}(\theta(G)) = \mathcal{N}(\sigma(L)) \cap \mathcal{R}N(\theta(D)) = \emptyset$ (by symmetry).*

We have $\theta(G) \subseteq t$. Since $\mathcal{R}N(\sigma(R)) \cap \mathcal{N}(\theta(G)) = \emptyset$ no node occurring in $\theta(G)$ may be directed by $\sigma(R)$ (neither globally nor locally, hence no edge in $\theta(G)$ can be redirected). Thus $\theta(G) \subseteq s$. Since the created nodes in σ, θ are distinct, this implies that θ is a π-matcher for s. By symmetry, σ is a ρ-matcher for u.

Let α', β' be the roots of $\sigma(R)$ and $\theta(D)$ respectively. We have seen that $\mathcal{R}N(\theta(D)) \cap \mathcal{N}(\sigma(L)) = \emptyset$. This implies in particular that $\alpha' \neq \beta$. Similarly, $\beta' \neq \alpha$. Since the non created nodes in $\theta(D)$ (except β) are labeled by constructors, α' cannot occur in $\theta(D)$ and similarly, β' cannot occur in $\sigma(R)$. This implies that $\alpha' \neq \beta'$. Moreover, the graphs $\sigma(R)$ and $\theta(D)$ must be compatible. Indeed, the created nodes must be mapped to distinct nodes (since s, u share no nodes other than those in s). Furthermore, since $\mathcal{R}N(\sigma(R)) \cap \mathcal{N}(\theta(G)) = \mathcal{N}(\sigma(L)) \cap \mathcal{R}N(\theta(D)) = \emptyset$, the nodes that are affected by $\sigma(R)$ (resp. $\theta(L)$) cannot occur in $\theta(G)$ (resp. $\sigma(L)$). Therefore the nodes that are defined both in $\sigma(R)$ and $\theta(D)$ must have the same definition as in t. Thus we have:

$$\begin{array}{rcl} t[\sigma(R)]_\alpha[\theta(D)]_\beta & = & (t_{\alpha'/\alpha}; \sigma(R))_{\beta'/\beta}; \theta(D) \\ & = & t_{\alpha'/\alpha\beta'/\beta}; \sigma(R); \theta(D) \\ & = & t_{\beta'/\beta\alpha'/\alpha}; \theta(D); \sigma(R) \\ & = & t[\theta(D)]_\beta[\sigma(R)]_\alpha. \end{array}$$

A graph t is *clash-reducible* (w.r.t. a GRS $\mathcal{R}$) iff there exists a graph s s.t. $t \to_{\mathcal{R}}^* s$ and s contains a clash. We now establish interesting properties of non clash-reducible graphs.

2.2 A Confluence Criterion

A GRS $\mathcal{R}$ is *orthogonal* iff it contains no pair of distinct rules $L \to R, G \to D \in \mathcal{R}$ (up to a renaming of the nodes) s.t. there exists a renaming σ s.t. $\sigma(root(L)) = \sigma(root(G))$ and $\sigma(L)$ and $\sigma(G)$ are compatible.

Theorem 1 *Let $\mathcal{R}$ be an orthogonal GRS. $\to_{\mathcal{R}}$ is confluent on non clash-reducible graphs.*

Proof *As well-known, it suffices to prove that the relation $\to_{\mathcal{R}}$, restricted to non clash-reducible graphs, is* strongly *confluent. Let t be a graph s.t. $t \to_{\mathcal{R}} s$ and $t \to_{\mathcal{R}} s'$. There exists two rules ρ and π, a ρ-matcher σ at a node α and a π-matcher θ at a node β s.t. $t \to_{\rho,\sigma} s$ and $t \to_{\pi,\theta} s'$. If $\alpha = \beta$ then we have $\rho = \pi$ (since $\mathcal{R}$ is orthogonal). Thus s, s' are identical up to a renaming of created nodes. If $\alpha \neq \beta$ the proof follows by Lemma 1.*

2.3 A Needed Reduction Strategy

We now define a demand-driven reduction strategy for non clash-reducible graphs. Let $\prec$ be an arbitrary chosen total ordering on positions, containing the prefix ordering. The set of *demanded* nodes in a graph t is the least set s.t.:

- If p is the minimal position (w.r.t. $\prec$) s.t. $p_t(root(t))$ is labeled by a defined symbol then $p_t(root(t))$ is demanded.
- If α is demanded and p is the minimal position s.t. $p_t(\alpha)$ is demanded for α then $p_t(\alpha)$ is demanded.

We write $t \rightarrow^n_{\mathcal{R}} s$ if s is obtained from t by applying a rule in $\mathcal{R}$ on a demanded node in t. Notice that this strategy is much simpler than the more general one defined in [6] (in the sense that the set of demanded nodes is much easier to compute). Theorem 2 shows that the above strategy is normalizing for every graph t that is not clash-reducible.

We need the following results:

Lemma 2 *Let $\mathcal{R}$ be a GRS and let $\rho = L \rightarrow R$. If $t \rightarrow_{\rho,\sigma} s$ and if $\alpha = \sigma(root(L))$ is not demanded, then any node that is demanded in s is also demanded in t.*

Proof *The proof is by induction on the set of demanded nodes. Let β be a demanded node in s.*

Assume that $root(s)$ is labeled by a constructor and that $\beta = p_s(root(s))$ where p is the minimal position in s s.t. $p_s(root(s))$ is labeled by a defined symbol. By definition there exists a node β' s.t. $root(s) \xrightarrow{\mathcal{C}}_s \beta'$ and $\beta = i_s(\beta')$, for some $i \in \mathbb{N}$.

If $\mathcal{R}N(\sigma(\rho))$ contains a node γ s.t. $root(t) \xrightarrow{\mathcal{C}}_t \gamma$ then according to Condition 2a in Definition 4, α, γ is a clash, which is impossible (notice that γ must be distinct from α since α is labeled by a defined symbol). Thus the nodes γ s.t. $root(t) \xrightarrow{\mathcal{C}}_t \gamma$ are not affected by ρ. Since $root(s) \xrightarrow{\mathcal{C}}_s \beta'$ this implies that $root(t) \xrightarrow{\mathcal{C}}_t \beta'$ and that $p_t(root(t)) = i_t(\beta')$. Moreover, if $i_t(\beta')$ is distinct from β then this means that $i_t(\beta')$ must have been redirected, thus this node must be α. Let q be the minimal position s.t. $q_t(root(t))$ is labeled by a defined symbol. Since α is not demanded, this node cannot be α hence it cannot be redirected by ρ. Furthermore, we have seen that the nodes occurring at a position lower than q cannot be affected by ρ. Thus $q_s(root(s)) = q_t(root(t))$, hence $q \succeq p$ (by definition p is the minimal position in s s.t. $p_s(root(s))$ is labeled by a defined symbol).

We have seen that if $p_t(root(t)) \neq \beta$ then we have $\alpha = p_t(root(t))$. Thus in any case $p_t(root(t))$ is labeled by a defined symbol, hence $p \succeq q$ and $p = q$. Since α is not demanded we cannot have $\alpha = p_t(root(t))$, therefore $p_t(root(t)) = \beta$, hence β is demanded in t.

Now assume that there exists a demanded node γ in s, a rule π in $\mathcal{R}$ and a position p s.t. $p_s(\gamma)$ is demanded for γ in s w.r.t. π and $\beta = p_s(\gamma)$. By the induction hypothesis we know that γ is demanded in t. If $\mathcal{R}N(\sigma(\rho))$ contains a node λ s.t. $\gamma \sqsupset^{\pi}_{\lambda}$ then according to Condition 2b, α, γ is a clash, which is impossible. Otherwise, we prove, by using exactly the same reasoning as above (since the node γ cannot be redirected nor created by ρ) than β is demanded for γ in t and that $\beta = p_t(\gamma)$. Thus β is demanded in t.

A *value* is a graph that contains no node reachable from the root and labeled by a defined symbol.

Proposition 2 *Let $\mathcal{R}$ be a GRS and let $\rho = L \rightarrow R$. Let s be a value. If $t \rightarrow_{\rho,\sigma} s$ and if $\sigma(root(L))$ is not demanded, then t is a value.*

Proof *Assume that t is not a value. Then t contains a demanded node β. We have $root(t) \xrightarrow{\mathcal{C}}_t \beta'$ and $i_t(\beta') = \beta$. By the same reasoning as in the proof of Lemma 2, we prove that ρ cannot affect any node along the path from $root(t)$ to β'. Thus we have $root(s) \xrightarrow{\mathcal{C}}_s \beta'$ and $i_t(\beta') = \beta$. Furthermore, the label of β must be a defined symbol and s cannot be a value.*

Theorem 2 *Let $\mathcal{R}$ be a GRS. Let t_1 be a graph that is not clash-reducible. If there exists a derivation $t_1 \rightarrow_{\mathcal{R}} \ldots \rightarrow_{\mathcal{R}} t_n$ s.t. t_n is a value, then there exists a derivation $s_1 \rightarrow^n_{\mathcal{R}} \ldots \rightarrow^n_{\mathcal{R}} s_m$ s.t. $t_1 = s_1$ and a renaming σ s.t. $s_m|_\epsilon = \sigma(t_n|_\epsilon)$. Moreover $m \leq n$ and every node that is reduced in $s_1 \rightarrow^n_{\mathcal{R}} \ldots \rightarrow^n_{\mathcal{R}} s_m$ is also reduced in $t_1 \rightarrow_{\mathcal{R}} \ldots \rightarrow_{\mathcal{R}} t_n$.*

Notice that by the previous definitions, $t|_\epsilon$ denotes the graph obtained from t by removing all nodes not reachable from the root.

Proof *We reason by induction on n. If $n = 1$ the proof is obvious. Otherwise, since t_1 is not clash-reducible, t_2 cannot be clash-reducible, thus by the induction hypothesis we may assume that $t_2 \to^n_{\mathcal{R}} \ldots \to^n_{\mathcal{R}} t_n$. There exist $\rho = L \to R$ and σ s.t. $t_1 \to_{\rho,\sigma} t_2$. Let $\alpha = \sigma(root(L))$. If α is demanded then $t_1 \to^n_{\mathcal{R}} t_2$ thus the proof is completed.*

Thus we assume that α is not demanded. If $n = 2$, then by definition t_2 must be a value and we get the desired result by Proposition 2. Assume that $n > 2$. Then there exist $\pi = G \to D$ and θ s.t. $t_2 \to_{\pi,\theta} t_3$. Let $\beta = \theta(root(G))$. We must have $\beta \neq \alpha$ (since α cannot occur in t_2). By Lemma 2, β is demanded in t_1. By Proposition 1, θ is a matcher for t_1 and π. By Lemma 1, we have $t_1 \to_{\pi,\theta} t'_1 \to_{\rho,\sigma} t_3$. Thus $t_1 \to^n_{\mathcal{R}} t'_1 \to t_3 \to^n_{\mathcal{R}} \ldots \to^n_{\mathcal{R}} t_k$. By reapplying the induction hypothesis on the derivation from t'_1 to t_k, we get the desired result.

We now define a class of GRS for which the previous strategy is optimal (in the sense that the selected nodes are reduced in every derivation).

If t is a graph and p is a position, we denote by $t|_{\prec p}$ the graph s obtained from t by removing every node α s.t. $\forall q \prec p, \alpha \neq q_t(root(t))$. A GRS $\mathcal{R}$ is *$\prec$-sequential* iff for every pair of rules $\rho = L \to R$ and $\pi = G \to D$ in $\mathcal{R}$, either $L \equiv G$ or there exists a position $p \in pos_L(root(L)) \cap pos_G(root(G))$ s.t. $L|_{\prec p} \equiv G|_{\prec p}$ and $L|_{\prec p.1} \not\equiv G|_{\prec p.1}$. Checking whether a given GRS is $\prec$-sequential or not is straightforward. All the GRS in this paper are $\prec$-sequential. Every inductively/strongly sequential GRS [11, 12, 9] and every elementary GRS [6] are $\prec$-sequential.

Proposition 3 *Let t be a graph and let $\mathcal{R}$ be a $\prec$-sequential GRS. There exists at most one node in t that is both demanded and reducible.*

Proof *We consider the definition of demanded nodes. This definition is inductive. We denote by N_0 the set of nodes that are generated by the base case and by N_i the nodes that are generated from the node $\bigcup_{j=0}^{i-1} N_j$ at step i. We show that for every $i \in \mathbb{N}$, N_i contains at most one element and that if the node in N_i is reducible then N_{i+1} is empty. These properties immediately entail the desired result.*

By definition N_0 contains exactly one element (since $\prec$ is total). Assume that N_i contains only one element α. Let β, β' be two nodes generated at step $i + 1$. By definition there exist two positions p, p' s.t. $\beta = p_t(\gamma)$ and $\beta' = {p'}_t(\gamma')$ are demanded for γ and γ' respectively. Furthermore, we must have $\gamma' = \gamma = \alpha$, otherwise the node β and/or β' would have been generated before step $i + 1$.

α must be labeled by a defined symbol and there exist two rules $\rho = L \to R, \rho' = L' \to R'$ s.t. p, p' are the minimal positions in $pos_L(root(L))$ and $pos_{L'}(root(L'))$ s.t. $p_t(\alpha), {p'}_t(\alpha)$ are labeled by defined symbols.

Since $\mathcal{R}$ is $\prec$-sequential, we have either $L \equiv L'$ or there exists a position q s.t. $L|_{\prec q} \equiv L'|_{\prec q}$ and $L|_{\prec q.1} \not\equiv L'|_{\prec q.1}$. If $L \equiv L'$ then $p = p'$ (since the set of demanded nodes depends only on t and on the left-hand side of the rule). Otherwise, by definition since $L|_{\prec q} \equiv L'|_{\prec q}$ then if $p \neq p'$ we must have $p, p' \succeq q$, furthermore, since $L|_{\prec q.1} \not\equiv L'|_{\prec q.1}$, we have $p = p' = q$. Thus $\beta = \beta'$.

Moreover, if α is reducible then by definition there exists a rule $\pi = G \to D$ that is applicable on t at α. But then either L and G are identical (up to a renaming) hence ρ is applicable (which is impossible since p is a position in $pos_L(root(L))$ and $p_t(\alpha)$ is labeled by a defined symbol) or $L|_{\prec r.1} \not\equiv L|_{\prec r.1}$ for some position $r \in pos_L(root(L)) \cap pos_G(root(G))$ which is also impossible: indeed, there exists a renaming θ s.t. $\theta(G) \subseteq t$ hence the graph t' obtained from t by removing every node distinct from α and not labeled by a defined symbol contains $\theta(G)$. Since $\sigma(L)$ and $\theta(G)$ are not compatible, t' and $\sigma(L)$ cannot be compatible, thus there is no potential ρ-matcher for t at α.

Lemma 3 *Let $\mathcal{R}$ be a $\prec$-sequential GRS. Let t be a non clash-reducible graph. If α is demanded in t and reducible, then for every derivation $t_1 \to_{\rho_1,\sigma_1} \ldots \to_{\rho_n,\sigma_n} t_n$ s.t. $\rho_1, \ldots, \rho_n \in \mathcal{R}$, $t_1 = t$ and t_n is a value, there exists $i \in [1..n]$ s.t. $\alpha = \sigma_i(root(L_i))$, where $\rho_i = L_i \to R_i$.*

Proof *By Theorem 2, we assume that $t_1 \to^n_{\rho_1,\sigma_1} \ldots \to^n_{\rho_n,\sigma_n} t_n$. By Proposition 3 t_1 contains only one reducible demanded node, thus $\alpha = \sigma_1(root(R_1))$.*

3 A Decidable Criterion Ensuring Clash-Irreducibility

The results in the previous section demonstrate that non clash-reducible graphs have very interesting computational properties. Therefore restricting oneself to non clash-reducible graphs is a very natural idea. Unfortunately, clash-reducibility is a *semantic* property: there is no algorithm to check whether a graph is clash-reducible or not (it is easy to see that this problem is undecidable). Notice that it is not sufficient to

check that a graph contains no clash (which is trivially decidable): we must ensure that no clash is generated during the rewriting process. Consequently, we need to define a decidable criterion, that is strong enough to ensure non clash-reducibility but that is weak enough to capture the systems arising naturally in practical programming. This is the goal of the present section.

Our approach can be summarized as follows. In Section 3.1, we introduce a notion of *potential clashes* that is weaker than the notion of clashes but that has the advantage of being invariant by rewriting: if a clash appears during the reduction of a graph t between two nodes α, β in t then there must be a potential clash between α and β (or their ancestors) already in t. This is unfortunately not sufficient to dismiss every clash-reducible graph, because the rewrite rules themselves may introduce clashes. Section 3.2 imposes additional conditions to forbid rewrite rules creating new clashes. To ensure this property, we need to impose additional conditions on the form of the graphs that may occur in the left-hand side of the rule. These conditions are called *shape relations*, because they describe the shape of the considered data-structures. We show how to determine automatically such shape relations and how to exploit them to define conditions that are comprehensive enough to capture many rewrite systems occurring in practice but in the same time sufficiently strong to prevent the rules to prevent clashes. This is done in Section 3.3, and Section 3.4 shows the soundness of the whole algorithm.

3.1 Potential Clashes

We need to detect "potential" clashes, i.e. clashes that do not occur in the initial graph, but that are introduced by rewriting. The first step is to control the side-effects performed by the defined functions.

3.1.1 Approximating the Effects of a Defined Function

We associate to each function symbol f of arity $[1..n]$ a *side-effect profile* which is composed by three sets $w_f, in_f, out_f \subseteq [1..n] \times \mathcal{S}$. These sets will be used to approximate the effects of a defined function f on the graphs on which it is applied. Intuitively, the intended interpretation of these sets is the following:

- $(i, \mathtt{s}) \in w_f$ iff f may affect a node α of sort $\mathtt{s}$ occurring in its i-th argument (by relabeling α or by redirecting some edges starting from α).
- $(i, \mathtt{s}) \in in_f$ iff f may access to a node α of sort $\mathtt{s}$ occuring in its i-th argument.
- $(i, \mathtt{s}) \in out_f$ iff the value returned by f may contain a node α of sort $\mathtt{s}$ initially occurring in its i-th argument.

Example 2 *Consider the function append defined in Section 1.3. Assume that cons and append have profile* $\mathtt{nat} \times \mathtt{list} \to \mathtt{list}$ *and* $\mathtt{list} \times \mathtt{list} \to \mathtt{list}$ *respectively. append affects it first argument but not the second, thus* $w_{append} = \{(1, \mathtt{list})\}$*. append accesses to the first element of the first list (of sort* $\mathtt{nat}$*), to the tail of the second list and to the second list (both of sort* $\mathtt{list}$*), thus* $in_{append} = \{(1, \mathtt{nat}), (1, \mathtt{list}), (2, \mathtt{list})\}$*. Finally,* $out_{append} = \{(1, \mathtt{nat}), (2, \mathtt{nat}), (1, \mathtt{list}), (2, \mathtt{list})\}$ *(both the first argument and the second are reachable from the result).*

Now, consider the function copy that returns a copy of a list, defined as follows: $copy(nil) \to nil, copy(cons(\alpha, \beta)) \to cons(\alpha, copy(\beta))$*. We have* $w_{copy} = \emptyset$ *(no side-effects),* $in_{copy} = \{(1, \mathtt{list}), (1, \mathtt{nat})\}$ *and* $out_f = \{(1, \mathtt{nat})\}$ *(the initial list is not reachable from the result returned by copy but its elements are).*

Let inc be a function (physically) incrementing every element occurring in a list (without affecting the list itself). Then we would have $(1, \mathtt{nat}) \in w_{inc}$*, but* $(1, \mathtt{list}) \notin w_{inc}$*.*

Finally, let length be a function computing the length of a list, defined as usual. Then we have $w_{length} = \emptyset$ *(length produces no side effect) and* $out_{length} = \emptyset$ *(the result is a created node, thus no node occurring in the list occurs in the result). Furthermore,* $in_{length} = \{(1, \mathtt{list})\}$*.*

Definition 6 will formalize this and will show how to compute these relations.

In particular, the set out_f allows one to compute from a graph t, an *approximation* of the graph s obtained after some reduction steps, and more precisely of the reachability relation $\xrightarrow{\mathcal{C}}_s$. This is the goal of the next section.

3.1.2 Approximating the Reachability Relation

Being able to determine in advance whether a node α will be reachable or not from a node β after some rewrite steps is essential for detecting side-effects. Consider for instance the graph $t = f(rev(\alpha), rev(id(\alpha)))$ where *rev* is a function physically reversing its argument and *id* is the identity function. Is t clash-reducible? This depends on the function *id*. The answer is clearly "no" if *id* returns a *copy* of its argument. In this case

t is not clash-reducible because the two occurrences of *rev* operate on (physically) distinct lists: there is no *physical* connection between the result of the function *id* and its argument, that belongs to a different data space. However if *id* is simply defined as $id(\alpha) \to \alpha$ then there exists a clash because *id* merely redirects the argument of the second occurrence of *rev* to the node α, thus the two occurrences of *rev* actually operate on the same node (after reducing *id*, the graph becomes $f(rev(\alpha), rev(\alpha))$ with an obvious clash). To approximate the reachability relation obtained after some rewrite steps we define a family of relations $\rhd_t^\Gamma$ (where $\Gamma \subseteq \mathcal{N}(t)$) on the nodes in t. $\alpha \rhd_t^\Gamma \beta$ indicates that a constructor path possibly exists from α to β after reducing the nodes in Γ. More formally:

Definition 5 *Let t be a graph and let $\Gamma \subseteq \mathcal{N}(t)$. We denote by $\rhd_t^\Gamma$ the smallest relation (implicitly parameterized by out_f) on $\mathcal{N}(t) \times \mathcal{N}(t)$ s.t.the following conditions hold:*

- *If $\alpha \in \Gamma$ or if $l_t(\alpha) \in \mathcal{C}$ then $\alpha \rhd_t^\Gamma \alpha$.*
- *$\beta = i_t(\alpha)$, $l_t(\alpha) \in \mathcal{C}$ and either $\beta \in \Gamma$ or $l_t(\beta) \in \mathcal{C}$ then $\alpha \rhd_t^\Gamma \beta$.*
- *If $\alpha \rhd_t^\Gamma \beta$ and $\beta \rhd_t^\Gamma \gamma$ then $\alpha \rhd_t^\Gamma \gamma$.*
- *If $i_t(\alpha) \rhd_t^\Gamma \beta$, $l_t(\alpha) = f \in \mathcal{D}\alpha$, $(i, \mathtt{sort}(\beta)) \in out_f$ and $\alpha \in \Gamma$ then $\alpha \rhd_t^\Gamma \beta$.*

Using an obvious fixpoint algorithm, $\rhd_t^\Gamma$ may be computed in time $|\mathcal{N}(t)|^5$ (if out_f and Γ are given). There are at most $|\mathcal{N}(t)|^2$ steps (the maximal cardinality of the computed relation) and at each step we need to check every triple of nodes in the graph. $\rhd_t^\Gamma$ may be seen as an approximation of the relation $\xrightarrow{\mathcal{C}}_s$, where s denotes a graph obtained from t by reducing all the nodes in Γ (and their descendants). The "ideal" reachability relation could be obtained, from a theoretical point of view, by reducing all the nodes in Γ, but this is of course infeasible in practice. The second item in the definition handles the edges in t. The third item states that the relation is transitive, and the last item covers the case in which the reduction of a node $\gamma \in \Gamma$ (labeled by a defined symbol f) creates a constructor path from α to β.

Example 3 *We use the same notations as in Example 2. Let $t = \alpha{:}append(\beta{:}cons(\gamma{:}0, \lambda{:}nil), \zeta{:}copy(\delta{:}cons(\epsilon{:}0, \lambda)))$. We have:*

$$\begin{array}{lcl} \rhd_t^\emptyset & = & \{(\beta,\beta),(\beta,\gamma),(\beta,\lambda),(\delta,\delta),(\delta,\epsilon),(\delta,\lambda),(\epsilon,\epsilon),(\gamma,\gamma),(\lambda,\lambda)\}. \\ \rhd_t^{\{\alpha\}} & = & \rhd_t^\emptyset \cup \{(\alpha,\alpha),(\alpha,\beta),(\alpha,\gamma),(\alpha,\lambda)\}. \\ \rhd_t^{\{\alpha,\zeta\}} & = & \rhd_t^{\{\alpha\}} \cup \{(\zeta,\zeta),(\zeta,\epsilon),(\alpha,\zeta),(\alpha,\epsilon)\}. \end{array}$$

Notice that we do not have $\alpha \rhd_t^{\{\alpha,\zeta\}} \delta$. This indicates that a function operating on t cannot access to δ (since δ is not reachable from α, after the evaluation of append and copy).

We do not have either $\beta \rhd^{\{\alpha,\zeta\}} \epsilon$, which may be seem surprising, since append creates a path from β to ϵ. However, we only consider non clash-reducible graphs, thus the only functions that access to β will actually operate on α. Since we have $\alpha \rhd_t^{\{\alpha,\zeta\}} \epsilon$, the path from α to ϵ does not need to be considered.

3.1.3 Automatically Computing w_f, in_f, out_f

We now provide an algorithm to compute automatically the side-effect profile from a given GRS. We write $\mathtt{s} \geq \mathtt{s}'$ iff there exists a constructor graph whose root is of sort $\mathtt{s}$ containing a node of sort $\mathtt{s}'$ ($\geq$ is obviously decidable). A node α *accesses to a node β in t* (resp. *affects a node β in t*) iff α is labeled by a defined symbol f and there exists i s.t. $(i, \mathtt{sort}(\beta)) \in in_f$ (resp. $(i, \mathtt{sort}(\beta)) \in w_f$) and $i_t(\alpha) \rhd_t^\mathcal{N} \beta$ where $\Gamma = \mathcal{N}(t) \setminus \{\alpha\}$. If f, g are two defined functions, we write $(f, i) > (g, j)$ iff there exist an f-rule $L \to R$, a node $\alpha \in \mathcal{N}(L|_i)$ s.t. $\mathtt{sort}(\alpha) \geq \mathtt{s}$ and a node β in R labeled by g s.t. $j_R(\beta) \rhd_R^\mathcal{N} \alpha$.

The following definition expresses the conditions that must be satisfied by w_f, in_f and out_f (following the informal definition in Section 3.1.1).

Definition 6 *A side-effect profile $(w_f, in_f, out_f)_{f \in \mathcal{D}}$ is* adequate *for a GRS $\mathcal{R}$ if the three following conditions hold:*

- *$(i, s) \in w_f$ if there exist an f-rule $L \to R \in \mathcal{R}$ and a node $\alpha \in \mathcal{N}(L|_i)$ s.t. $\alpha \in \mathcal{R}N(\rho)$ and $\mathtt{sort}(\alpha) = \mathtt{s}$. This condition corresponds to the case in which an f-rule affects a node α reachable from the i-th argument of f.*
- *$(i, s) \in in_f$ if there exists an f-rule $L \to R \in \mathcal{R}$ and a node $\alpha \in \mathcal{N}(L|_i)$ s.t. α is of sort s.*

- *If* $(f,i) >_{\mathtt{s}} (g,j)$ *and* $(j,\mathtt{s}) \in w_g$ *then* $(i,\mathtt{s}) \in w_f$.
- *If* $(f,i) >_{\mathtt{s}} (g,j)$ *and* $(j,\mathtt{s}) \in in_g$ *then* $(i,\mathtt{s}) \in in_f$.
- $(i,s) \in out_f$ *if there exist an* f*-rule* $L \to R \in \mathcal{R}$ *and a node* $\alpha \in \mathcal{N}(L|_i)$ *s.t.* $root(R) \rhd_R^{\mathcal{N}} \alpha$ *and* $\mathtt{sort}(\alpha) \geq s$.

Example 4 *Let* $f : \mathtt{nat} \times \mathtt{nat} \to \mathtt{nat}$ *and* $g : \mathtt{list} \to \mathtt{nat}$. *We consider the following GRS:*

$$\begin{array}{lcl} f(\alpha{:}0,\beta) & \to & \beta;\alpha{:}s(0) \\ g(cons(\alpha,\beta)) & \to & f(\alpha,\alpha) \end{array}$$

The node α *is affected by the first rule. Moreover, this node is reachable from the first argument of* f. *Thus, by the first item in Definition 6, we have* $(1,\mathtt{nat}) \in w_f$. *Similarly by the second item we have* $(1,\mathtt{nat}) \in in_f$ *and* $(2,\mathtt{nat}) \in in_f$. β *is reachable from the root of the right-hand side of the first rule, thus* $(2,\mathtt{nat}) \in out_f$.

Consider the second rule. By the first item we have $(1,\mathtt{nat}),(1,\mathtt{list}) \in in_g$. *It is easy to check that* $(g,1) >_{\mathtt{nat}} (f,1),(g,1)$ *(since* g *calls the function* f *on* α,α*). Thus by the third item in Definition 6, we deduce* $(1,\mathtt{nat}) \in w_g$ *and* $(1,\mathtt{nat}) \in w_g$. *Finally, by the last item we have* $(1,\mathtt{nat}) \in out_g$.

Notice that we have $(1,\mathtt{list}) \notin w_g \cup out_g$. *This means that for instance if we consider the graph* $\alpha{:}g(\beta{:}cons(0,nil))$, *we know that (i)* β *is not affected by* α *and (ii)* β *is not reachable from the image of* g.

It is easy to compute a *minimal* family of side-effect profiles for a given GRS, using a standard fixpoint algorithm. This algorithm runs in polynomial time w.r.t. the size of the GRS. There are at most $3 \times a \times |\mathcal{S}| \times |\mathcal{D}|$ steps, where a is the maximal arity of the function symbols. At each step one must compute the relation $\rhd_R^{\mathcal{N}}$ for each rule $\rho = L \to R$ and then check that the conditions of Definition 6 are satisfied. The maximal complexity is thus $3 \times n^5 \times a \times |\mathcal{S}| \times |\mathcal{D}| \times m$, where n denotes the maximal size of the rules in $\mathcal{R}$ and m the maximal number of rules corresponding to the same defined function. Notice that in practice, $|\mathcal{D}|, m$ and $|\mathcal{S}|$ may be high, but n, a are usually small.

In the rest of the section, we assume that (w_f, in_f, out_f) is the minimal side-effect profile that is adequate for the considered GRS. $\rhd_t^{\Gamma}$ denotes the corresponding approximation of the reachability relation.

3.1.4 Identifying Potential Clashes

Building on the previous results, we can weaken the definition of clashes:

Definition 7 *A* potential clash *for a graph* t *is a pair of distinct nodes* α,β *satisfying the following conditions:*

- α *is labeled by a defined symbol* f.
- *There exist a node* γ *and* $i \in [1..ar(f)]$ *s.t.* $(i,\mathtt{sort}(\gamma)) \in w_f$ *and* $i_t(\alpha) \rhd_t^{\Gamma} \gamma$ *where* $\Gamma \stackrel{def}{=} \mathcal{N}(t) \setminus \{\alpha,\beta\}$.
- *Either* $\beta = \gamma$ *and* $root(t) \xrightarrow{\mathcal{C}}_t \beta$ *or* β *is labeled by a defined symbol* g *and there exists* $j \in [1..ar(g)]$ *s.t.* $(j,\mathtt{sort}(\gamma)) \in in_g$ *and* $j_t(\alpha) \rhd_t^{\Gamma} \gamma$.

  A graph t *is* clash-free *if the graph* t *contains no potential clash.*

Intuitively, α affects the node γ and either there exists a constructor path from the root of t to γ or there exists a defined node β that accesses to γ.

For instance the graph $t = \beta{:}append(\alpha{:}rev(\gamma{:}[1,2]),\gamma)$ contains a potential clash between α and β, since $1_t(\alpha) \rhd_t^{\emptyset} \gamma$, $2_t(\beta) \rhd_t^{\emptyset} \gamma$, $(1,\mathtt{list}) \in w_{rev}$ and $(2,\mathtt{list}) \in in_{append}$. Notice that t contains no clash. The graph $s = [1|\beta{:}[2,3]];\alpha{:}rev(\beta)$ also contains a potential clash, since $1_t(\alpha) \rhd_t^{\emptyset} \beta$ and β is reachable from the root of t.

Proposition 4 *Every clash is a potential clash.*

Proof *Assume that* t *contains a clash. According to Definition 4, there exists a pair of nodes* α,β *satisfying the following conditions:*

- $\alpha \neq \beta$.
- *There exist a rule* $\rho = L \to R$ *in* $\mathcal{R}$ *and a* ρ*-matcher* σ *at* α.
- *There exist a node* $\gamma \in \mathcal{R}N(\sigma(\rho))$ *s.t. either* $\beta = \gamma$ *and* $root(t) \xrightarrow{\mathcal{C}}_t \beta$ *or* $\beta \sqsupset_t^R \gamma$.

Since $\gamma \in \mathcal{R}N(\sigma(\rho))$, γ occurs in $\sigma(L)$. Since ρ is connex, we have $i_{\sigma(L)}(\alpha) \xrightarrow{\mathcal{C}}_{\sigma(L)} \gamma$, for some $i \in [1..ar(f)]$ hence $i_t(\alpha) \xrightarrow{\mathcal{C}}_t \gamma$. We remark that according to Definition 5, the relation $\rhd_t^{\emptyset}$ contains $\xrightarrow{\mathcal{C}}_t$. Therefore we have $i_t(\alpha) \rhd_t^{\emptyset} \gamma$. Moreover, by the first item in Definition 6, we have $(i, \mathtt{sort}(\gamma)) \in w_f$.

If $\beta = \gamma$ and $root(t) \xrightarrow{\mathcal{C}}_t \beta$ the proof is completed. Otherwise, if $\beta \sqsupset_t^R \gamma$, then there exists j s.t. $j_s(\beta) \xrightarrow{\mathcal{C}}_s \gamma$, thus $j_s(\delta) \rhd_s^{\emptyset} \gamma$. Moreover, by the second item in Definition 6, we have $(j, \mathtt{sort}(\gamma)) \in in_g$.

Therefore, the conditions of Definition 14 are satisfied and s contains a potential clash.

3.2 Describing Data-Structures by Shape Relations

Since all clashes are also *potential* clashes, a clash-free graph cannot contain clashes. In order to get the desired result (namely ensuring clash-irreducibility), it remains to check that the class of clash-free graphs is closed under rewriting. Unfortunately this property does not hold in general and we need to infer additional information on the shape of the reduced graphs. Consider for instance the rule: $ar(\alpha, \beta) \to a(rev(\alpha), rev(\beta))$, where a denotes the usual function appending two lists (with no side effects) and *rev* denotes a function that physically reverses its argument. Can this rule introduce a clash in the reduced graph ? Clearly this is possible, if α, β share some nodes. For instance, the normal form of the graph $ar(cons(1, \alpha{:}[2, 3]), \alpha)$ depends on the order in which the two occurrences of *rev* are evaluated. The reader can easily check that depending on the strategy, we could obtain the lists $[3, 2, 1, 2]$ or $[2, 1, 3, 2, 1]$. However, if α and β denote distinct lists, belonging to different data spaces (i.e. sharing no nodes), then there is no clash. The assertion "α and β belong to different data spaces" can be seen as a "precondition" of the defined function *ar*. These preconditions should be compared to the connective $*$ in separation logic (see e.g. [14]).

We introduce some relations which allow one to restrict the form (or shape) of a graph. The idea is to specify, for every function symbol f, which are the features (or arguments) of f that can be joinable (from a programming point of view this means that they belong to the same data-space), and which are the ones that may loop to the node of f itself (this implies that there is a cycle in the graph, containing the node of f). We assume the following relations and sets are given, for every function symbol f of arity n in Σ:

- A symmetric and reflexive relation $\sim_f \subseteq [1..n] \times [1..n]$.
- A set $\circlearrowleft_f \subseteq [1..n]$.

These relations will be used to impose additional constraints on the considered graphs. $i \not\sim_f j$ indicates that the arguments i and j of the function f cannot share nodes and $i \notin \circlearrowleft_f$ states that the argument i of f does not contain the node of f itself (i.e. if $t = \alpha{:}f(\beta_1{:}t_1, \ldots, \beta_n{:}t_n)$ then there is no path from β_i to α, i.e. t_i is is a *proper* subgraph of t). This is formalized by Definition 8. We denote by $\mathcal{N}_{\mathcal{D}}(t)$ the set of nodes reachable from a defined node in t, i.e. $\mathcal{N}_{\mathcal{D}}(t) \stackrel{\text{def}}{=} \{\alpha \mid \exists \beta \in \mathcal{N}(t), \beta \to_t \alpha, l_t(\beta) \in \mathcal{D}\}$.

Definition 8 *A node α is* well-formed *in a graph t iff either α is a variable or α is a node labeled by f and the two following conditions hold:*

- *If there exists a node γ and $i, j \in [1..ar(f)]$ s.t. $i_t(\alpha) \xrightarrow{\mathcal{C}}_t \gamma$ and $j_t(\alpha) \xrightarrow{\mathcal{C}}_t \gamma$ then $i \sim_f j$.*
- *If there exists $i \in [1..ar(f)]$ s.t. $i_t(\alpha) \xrightarrow{\mathcal{C}}_t \alpha$ then $i \in \circlearrowleft_f$.*

A graph t is well-formed *iff every node in $\mathcal{N}_{\mathcal{D}}(t)$ is well-formed.*

Assume for instance that we have $\sim_f = \{(1, 2), (2, 1), (1, 1), (2, 2)\}$. Then $f(s(0), 0, s(s(0)))$ or $f(s(\alpha{:}0), s(\alpha), 0)$ are well-formed, but not $f(0, s(\alpha{:}0), \alpha)$. If $\circlearrowleft_f = \{2, 3\}$ then $\alpha{:}f(\alpha, 1, 2)$ is not well-formed, but $\alpha{:}f(\beta{:}1, \alpha, 2)$ is well-formed.

If the constructor symbol *cons* is intended to denote non looping lists (without sharing) then we should have $1 \not\sim_{cons} 2$ (the first element is non joinable with the tail of the list) and $1, 2 \notin \circlearrowleft_{cons}$ (the list cannot occur in the first element nor in the tail). If the elements can be (physically) repeated, as in the list $[\alpha, \alpha]$, we should have $1 \sim_{cons} 2$. On the other hand, if the list are looping then we should have $\circlearrowleft_{cons} = \{2\}$. If moreover, the list may occur in itself then we should have $\circlearrowleft_{cons} = \{1, 2\}$ (notice in this case *cons* must have the profile: $\mathtt{list} \times \mathtt{list} \to \mathtt{list}$).

Although we assume in this section that these relations are given, in practice they do not need to be provided by the programmer: as explained in Section 5 they can be automatically inferred from the given GRS (by interpreting Definitions 8 and 15 as fixpoint operators on shape relations). To ensure that well-formedness is preserved by rewriting, we use the approximation of the reachability relation to strengthen the condition on the graphs:

Definition 9 *A node α is* $\rhd$well-formed in a graph t *iff either α is a variable or α is a node labeled by f and the two following conditions hold:*

- *If there exists a node γ and $i, j \in [1..ar(f)]$ s.t. $i_t(\alpha) \rhd_t^{\mathcal{N}} \gamma$ and $j_t(\alpha) \rhd_t^{\mathcal{N}} \gamma$ then $i \sim_f j$.*
- *If there exists $i \in [1..ar(f)]$ s.t. $i_t(\alpha) \rhd_t^{\mathcal{N}} \alpha$ then $i \in \circlearrowleft_f$.*

A graph t is $\rhd$well-formed *iff every node in $\mathcal{N}(t)$ is $\rhd$-well-formed.*

If a graph t is $\rhd$well-formed, then not only it is known to be well-formed (since $\rhd_t^{\mathcal{N}}$ contains $\xrightarrow{\mathcal{C}}_t$) but also it is ensured that every graph s that can be derived from t is well-formed.

Proposition 5 *Let t, s be two graphs. Assume that $\rhd_t^{\mathcal{N}}$ contains $\rhd_s^{\mathcal{N}}$ and that $i_t(\alpha) = i_s(\alpha)$ for every node $\alpha \in \mathcal{N}(s)$ and for every $i \in \mathbb{N} \cup \{l\}$. If t is $\rhd$-well-formed then so is s.*

Proof *This follows straightforwardly from the definition (since the condition on a node α only depend on $i_t(\alpha)$ (where $i \in \mathbb{N} \cup \{l\}$) and on the relation $\rhd_t^{\mathcal{N}}$).*

We denote by $\mathcal{WG}$ (resp. $\mathcal{WG}_{\mathtt{s}}$) the set of $\rhd$well-formed graphs (resp. of $\rhd$well-formed graphs of sort $\mathtt{s}$). We write $\mathtt{s}' \succ \mathtt{s}$ if there exists $t \in \mathcal{WG}_{\mathtt{s}'}$ and a node α of sort $\mathtt{s}$ s.t. $root(t) \rightarrow_t \alpha$.

Proposition 6 *There exists an algorithm to check whether $\mathcal{WG}_{\mathtt{s}}$ is empty. Furthermore, $\succ$ is decidable.*

Proof *A graph is* minimal *iff for every pair of nodes α, β of sort $\mathtt{s}$, if $\alpha \rightarrow_t \beta$ then $\alpha = \beta$. This means that along every position in the graph, one can find at most one node of each sort. By the pigeonhole argument this implies that the length of the position is bounded, thus there are finitely many minimal graphs (up to a renaming). We now show that if $\mathcal{WG}_{\mathtt{s}} \neq \emptyset$ then $\mathcal{WG}_{\mathtt{s}}$ contains a minimal graph. This obviously ensures that the test "$\mathcal{WG}_{\mathtt{s}} = \emptyset$?" is decidable: it suffices to enumerate the set of minimal graphs and check whether it contains a $\rhd$-well-formed one. Assume that t is a non minimal $\rhd$-well-formed graph. W.l.o.g. we assume that the number of nodes in t is minimal. By definition, t contains two distinct nodes α, β of the same sort s.t. $\alpha \rightarrow_t \beta$. Let $s = t_{\beta/\alpha}$. By definition, the edges in s are either edges in t or edges from nodes γ s.t. $i_t(\gamma) = \alpha$ to β. For each of those nodes we have $\gamma \xrightarrow{\mathcal{C}}_t \alpha$, hence $\gamma \xrightarrow{\mathcal{C}}_t \beta$. Consequently, $\rightarrow_t$ contains $\rightarrow_s$. Furthermore, for every node $\lambda \in \mathcal{N}(s)$, we have $\lambda \in \mathcal{N}(t)$ and $l_t(\lambda) = l_s(\lambda)$. By Proposition 5, since t is $\rhd$-well-formed, s must be also $\rhd$-well-formed.*

By using the same principle, we can decide whether $\mathtt{s} \succ \mathtt{s}'$: it suffices to enumerate the set of minimal graphs in $\mathcal{WG}_{\mathtt{s}}$ and check for each of them whether there exists a node of sort $\mathtt{s}'$ reachable from the root.

Exploiting Shape Relations

If we assume that all the graphs occurring in a derivation are well-formed, we can infer some interesting information about the variables occurring in the left-hand side of the rules that are applied along the derivation. For instance, if we know that $1 \not\sim_f 2$ and if the left-hand side of a rule ρ contains the term $f(\alpha, \beta)$ then α and β cannot be joinable (in particular no path can exist from β to α nor from β to α). More precisely, if σ is a ρ-matcher for a well-formed term t then since $\sigma(f(\alpha, \beta)) \subseteq t$ we must have by Definition 8, $\sigma(\beta) \not\downarrow_t \sigma(\alpha)$. This information allows one to discard many potential clashes. Assume for instance that ρ is of the form $f(\alpha, \beta) \rightarrow g(h(\alpha), h'(\beta))$, where f, g, h, h' denote defined symbols. Assume that h physically affects its argument. α, β denotes variables that may be instantiated arbitrarily thus the nodes α and β can be joinable and there is a potential clash between $h(\alpha)$ and $h'(\beta)$ in the right-hand side. Assume for instance that h physically increments its arguments and that h' returns *true* iff its argument is 0. If the GRS is applied on the term $f([\beta{:}0], \beta)$ the result can be $g([s(0)], true)$ or $g([s(0)], false)$ depending on the strategy. Now, assume that we know that $1 \not\sim_f 2$. Then, since the term in the left-hand side is matched to a well-formed term, α, β cannot be joinable and the previous case can be discarded. The term $f(h(\alpha), h'(\beta))$ contains no clash.

Knowing that the reduced graph is well-formed allows us to restrict the paths that may occur "outside" the left-hand side of a rule. This is important in practice because those paths are not known at the time the static analysis is performed (only the left-hand sides of the rules are known). During the static analysis, we must take into account *all the possible paths*, except those that are explicitly forbidden by the shape relations. In order to capture the paths that may exist in the reduced node, but outside the left-hand side, we consider expressions of the form $\alpha \rightarrow \beta$ or $\alpha \downarrow^{\mathtt{s}} \beta$ and we introduce an algorithm to check whether these expressions are *compatible* with a given a graph, i.e. intuitively hold in at least one extension of t.

Definition 10 *We consider expressions of the form $\alpha \to \beta$ or $\alpha \downarrow^{\mathtt{s}} \beta$, where $\mathtt{s} \in \mathcal{S}$. An expression ϕ is* compatible *with a graph t if there exists a $\rhd$-well-formed graph s containing t s.t.:*

- *Either ϕ is of the form $\alpha \to \beta$, $\alpha \in var(t)$, $\beta \in \mathcal{N}(t)$ and $\alpha \rhd_s^{\mathcal{N}} \beta$.*
- *Or ϕ is of the form $\alpha \downarrow^{\mathtt{s}} \beta$, $\alpha, \beta \in var(t)$ and there exists a node γ of sort $\mathtt{s}$ s.t. $\alpha \rhd_s^{\mathcal{N}} \gamma$ and $\beta \rhd_s^{\mathcal{N}} \gamma$.*

We use the approximation of the reachability relation $\rhd_s^{\mathcal{N}}$ and not the reachability relation itself because this allows us to discard some paths: for instance if $t = f(\alpha, \beta{:}copy(\alpha))$ then α, β are not joinable.

The next lemma states that the previous conditions can be decided effectively.

Lemma 4 *There exists an algorithm deciding, for every expression ϕ and for every graph t, whether ϕ is compatible or not with t. This algorithm runs in polynomial time w.r.t. the size of t and ϕ.*

Proof *We only provide the proof for expressions of the form $\alpha \to \beta$ (the case $\phi = (\alpha \downarrow^{\mathtt{s}} \beta)$ is similar).*

$\mathtt{s}, \mathtt{s}'$ be the sorts of α, β respectively. If $\mathtt{s} \not\succ \mathtt{s}'$ then by definition there is no well-formed graph s containing a path from α to β. Thus $\alpha \to \beta$ cannot be compatible with t. Now assume that $\mathtt{s} \succ \mathtt{s}'$. In this case, there exists a graph w and two nodes α', β' of sort $\mathtt{s}$ and $\mathtt{s}'$ respectively and s.t. $\alpha' = root(w), \alpha' \to_w \beta'$. By using appropriate renaming we assume that $\alpha = \alpha', \beta = \beta'$ and that the nodes in w that are distinct from α, β does not occur in t. We also assume that:

- *For every subgraph $\gamma{:}f(w_1, \ldots, w_n)$ occurring in w, $w_1, \ldots, w_n$ are disjoint. Moreover if w_i contains γ then it must contain β. If this property does not hold then we simply replace one of the arguments w_i of f by a disjoint copy of w_i. In order to ensure that $\beta = \beta'$, we cannot rename the subgraph w_i that contains β, which explains the restriction in the second condition.*
- *$\beta \in var(w)$. This condition is obviously not restrictive (if needed it suffices to replace β by a new variable of type $\mathtt{s}'$).*
- *Let $\alpha_1, \ldots, \alpha_k$ be the path from α to β. For every $i \in [1..k-1]$ there exists j s.t. $j_w(\alpha_i) = \alpha_{i+1}$. We assume that $j \in \circlearrowleft_g$, if it is possible (i.e. if a graph satisfying this condition exists).*

In order to check that the conditions of Definition 10 hold we consider the graph $u \stackrel{def}{=} t; w$. If u is $\rhd$-well-formed, then $\alpha \to \beta$ is compatible with t, by definition. Now assume that that there exists a $\rhd$-well-formed graph s containing t and s.t. $\alpha \to_s \beta$. We show that u is $\rhd$-well-formed.

Let Λ be the set of nodes in w that are distinct from α, β. W.l.o.g. we assume that s does not contain the nodes in Λ. Let $\Gamma \subseteq \mathcal{N}$. $\rhd_s^{\Gamma}$ contains $\rhd_t^{\Gamma} \cup \{(\alpha, \beta)\}$. Let $\delta \in \Lambda$. The only node λ in t s.t. $\delta \to_w \lambda$ is the node β. Thus $\rhd_s^{\Gamma}$ contains the restriction of $\rhd_u^{\Gamma}$ to the nodes in t.

Assume that u is not $\rhd$-well-formed. Then there exists a node $\lambda \in \mathcal{N}(u)$ that is non $\rhd$-well-formed. Let $g = l_u(\lambda)$. By definition, one of the two following conditions holds:

1. *There exist $i, j \in [1..ar(g)]$ s.t. $i \not\sim_g j$ and $i_u(\lambda) \geq_u^{\mathcal{N}} \circ \leq_u^{\mathcal{N}} j_u(\lambda)$. By definition of w, λ cannot occur in w (since the arguments of the node in w are disjoint). Thus λ and $i_u(\lambda)$ both occur in t and we have $i_s(\lambda) \geq_s^{\mathcal{N}} \circ \leq_s^{\mathcal{N}} j_s(\lambda)$ which is impossible since s is $\rhd$-well-formed.*

2. *There exists $i \notin \circlearrowleft_g$ s.t. $i_u(\lambda) \rhd_u^{\Gamma} \lambda$. If λ does not occur in w the proof is similar to the previous one. Otherwise, by definition of w this means λ occurs along the path from α to β in w and that the path from α to β in u contains an edge labeled by i from a node labeled by a function h s.t. $i \notin \circlearrowleft_h$. Then this node cannot be $\rhd$-well-formed in s, which is impossible.*

The (approximated) reachability relation $\rhd_t^{\Gamma}$ is now extended to rewrite rules:

Definition 11 *For any rule $\rho = L \to R$, $\rhd_\rho^{\Gamma}$ denotes the smallest transitive relation containing $\rhd_R^{\Gamma}$ and every pair (α, β) s.t. $\alpha \to \beta$ is compatible with L.*

Intuitively, $\alpha \rhd_\rho^{\Gamma} \beta$ if there exists a $\rhd$-well-formed graph t on which ρ is applicable and such that $\alpha \rhd_s^{\Gamma} \beta$, where $t \to_\rho s$ (i.e. the graph obtained from t by applying ρ and reducing the nodes in Γ may contain a constructor path from α to β). The first point captures the paths that already exist in the left-hand side or that are created by the rule and the second one handles paths that exist in the reduced graph, but *outside* L. Let ρ be the rule $\alpha{:}f(\beta, \gamma) \to \beta$. Assume that $\sim_f = \{(1,1), (1,2), (2,1), (2,2)\}$ and that $\circlearrowleft_f = \emptyset$. We have $\alpha \rhd_\rho^{\beta} \rhd_\rho^{\gamma}$ and $\gamma \rhd_\rho^{\beta}$ (since both $\beta \to \gamma$ and $\gamma \to \beta$ are compatible with L) but $\beta \not\rhd_\rho^{\alpha}$ (since $\gamma, \beta \to \alpha$ are not compatible with L). $\rhd_\rho^{\Gamma}$ can be computed in polynomial time since this is the case for $\rhd_R^{\Gamma}$ and for the set of compatible inequations.

Definition 8 can be adapted to apply it on rewrite rules instead of graphs:

Definition 12 *A node α occurring in a rule $L \to R$ is* well-formed *iff either $\alpha \in var(R)$ or $l_R(\alpha) = f$ and the two following conditions hold:*

- *If there exist two nodes β, γ s.t. either $\beta = \gamma$ or $\beta \downarrow^{\mathtt{s}} \gamma$ is compatible with L (for some $\mathtt{s} \in \mathcal{S}$), and two indices $i, j \in [1..ar(f)]$ s.t. $i_R(\alpha) \rhd_\rho^{\mathcal{N}} \beta$ and $j_R(\alpha) \rhd_\rho^{\mathcal{N}} \gamma$ then $i \sim_f j$.*
- *If there exists an index $i \in [1..ar(f)]$ s.t. $i_R(\alpha) \rhd_\rho^{\mathcal{N}} \alpha$ then $i \in \circlearrowleft_f$.*

A rule ρ is well-formed *iff every node in $\mathcal{N}(R)$ is well-formed.*

3.3 Well-defined Graphs

Definition 13 *A graph is* well-defined *if it is both $\rhd$-well-formed and clash-free.*

This definition is then extended to GRS by replacing the reachability relation $\rhd_t^\Gamma$ by the (weaker) relation $\rhd_\rho^\Gamma$. As we have seen, this is necessary to take into account the paths that may exist in the reduced graph but outside the left-hand side.

Definition 14 *A* potential clash *for a rule $\rho = L \to R$ is a pair of distinct nodes α, β labeled by defined functions f, g respectively and satisfying the following conditions:*

- *There exist $i, j \in \mathbb{N}$ and $\gamma, \delta \in \mathcal{N}$ s.t. $i_t(\alpha) \rhd_\rho^\Gamma \gamma$, $j_t(\beta) \rhd_\rho^\Gamma \delta$, where $\Gamma \stackrel{def}{=} \mathcal{N}(t) \setminus \{\alpha, \beta\}$.*
- *Either $\gamma = \delta$ and $\mathtt{sort}(\gamma) = \mathtt{s}$, or $\gamma \downarrow^{\mathtt{s}} \delta$ is compatible with L.*
- *$(i, \mathtt{s}) \in w_f$ and $(j, \mathtt{s}) \in in_g$.*

*A graph t containing no potential clash for ρ is ρ-*clash-free*. Notice that every ρ-clash-free graph is also clash-free.*

We now provide the most important definition, namely the condition that must be fulfilled by the GRS. We assume that $\mathcal{D}$ contains a special symbol $\top$ of arity 1 (not occurring in the GRS) s.t. for every sort $\mathtt{s}$: $(1, \mathtt{s}) \in w_\top \cap in_\top$. A sort $\mathtt{s}$ is *restricted* if there exists a function symbol f of profile $\mathtt{s}_1 \times \ldots \times \mathtt{s}_n \to \mathtt{s}$ s.t. either $\circlearrowleft_f \neq [1..n]$ or $i \not\sim_f j$ for some $i \neq j$. A rule $\rho = \alpha{:}f(L_1, \ldots, L_n) \to R$ is *shape-preserving* if for every node β of sort $\mathtt{s}$ in $\mathcal{R}N(\rho)$ for every node γ of sort $\mathtt{s}'$ in $\alpha{:}f(L_1, \ldots, L_n)$ if there exists a restricted sort $\mathtt{s}''$ s.t. $\mathtt{s}' \succ \mathtt{s}'' \succ \mathtt{s}$ then $\gamma \to \beta$ is not compatible with L.

Definition 15 *A rule $\rho = L \to R$ is* clash-free *if the graph $\top(R)$ is ρ-clash-free. It is* well-defined *if it is shape-preserving, $\rhd$-well-formed and clash-free.*

A GRS is well-defined *if every rule in it is well-defined.*

$\top$ is a generic function symbol intended to represent all the defined symbols operating on the root of ρ. It is *not* sufficient to check that R is well-formed and clash-free because clashes may appear between a defined function occurring in R and a defined function (possibly performing side-effects) "outside" L. For instance, consider the rule $f(s(\alpha)) \to s(f(\alpha))$. The term $s(f(\alpha))$ is obviously clash-free (since it contains only one defined node). However, if f is applied on a cyclic term then a clash can appear, e.g. the term $g(f(\alpha{:}s(\alpha)))$ (that is clash-free) can be transformed into $g(\alpha{:}s(f(\alpha)))$ which contains a clash since g and f operate on the same node α. In order to check that a graph or GRS is well-formed or clash-free, one has to compute the relations $\rhd_t^\Gamma$ and $\rhd_\rho^\Gamma$. This can be done in polynomial time w.r.t. the size of t, ρ (if the relations $\sim_f$ and $\circlearrowleft_f$ are fixed). According to Definitions 14 and 9 we do not have to compute these relations for every possible set of nodes Γ (which would be very inefficient since there are exponentially many different sets), but only for sets of the form $\Gamma = \mathcal{N}$ or $\Gamma = \mathcal{N}(t) \setminus \{\alpha, \beta\}$, where α, β are nodes in t.

3.4 Invariance

In this section we show that the class of well-defined graphs is closed by rewriting (if the rules are well-defined), which ensures that well-defined graphs cannot be clash-reducible. This shows the soundness of the static analysis as defined by the previous definitions.

We first state an important property of the relation $\rhd_t^\Gamma$. In the whole section, $\mathcal{R}$ denotes a well-defined GRS and $(w_f, in_f, out_f)_{f \in \mathcal{D}}$ denotes a side-effect profile adequate for $\mathcal{R}$.

Lemma 5 *Let $\rho = L \to R$ be a rule in $\mathcal{R}$ and let t be a clash-free graph. Let σ be a ρ-matcher for t at a node α. Let s be a graph s.t. $t \to_{\rho,\sigma} s$ and let $\beta, \gamma \in \mathcal{N}(s)$. Let Γ be a set of nodes in s. Assume that there exists a node $\lambda \neq \alpha$ labeled by a defined symbol f s.t. $i_t(\lambda) \rhd_t^\emptyset \beta$, where $(i, \mathtt{sort}(\beta)) \in in_f$.*

If $\beta \rhd_s^\Gamma \gamma$ then $\beta \rhd_t^{\Gamma'} \gamma^-$, where $\Gamma' = \Gamma^- \cup \{\alpha\}$.

Proof *We reason by induction on the relations $\rhd_s^\Gamma$. Let $f = l_t(\alpha)$.*

- *If $\beta = \gamma$, then we must have $\beta \in \Gamma$, hence $\beta^- = \beta \in \Gamma'$ and the proof is immediate.*
- *Assume that $\sigma(R)$ contains a $\rhd_t^\Gamma$-path from β to γ. If $\sigma(L)$ already contains a path from β to γ then obviously this path is in t, thus $\beta \rhd_t^{\Gamma'} \gamma$ (the only non constructor node along this path is α). Otherwise, this means that $\sigma(R)$ affects a node β' such that $\beta \rhd_t^\emptyset \beta'$. Since $i_t(\lambda) \rhd_t^\emptyset \beta$, we have $i_t(\lambda) \rhd_t^\emptyset \beta'$. Since the graph t contains no potential clash, we must have $\beta' = \alpha$ and α is redirected to a node γ' s.t. $\gamma' \rhd_{\sigma(R)}^\Gamma \gamma$. If γ is a created node then the proof is completed. Otherwise, γ must occur in $\sigma(L)$ thus there exist i s.t. $i_{\sigma(L)}(\alpha) \rightarrow_t \gamma$ hence $i_t(\alpha) \rhd_t^\emptyset \beta$. By definition of out_f, $i \in out_f$, thus we have $\alpha \rhd_t^{\Gamma'} \gamma$, and the proof is completed.*
- *Assume that we have $\beta \rhd_s^\Gamma \delta$ and $\delta \rhd_s^\Gamma \gamma$. We assume that δ is not a created node, since this case is already covered by the previous item (the path from β to δ is inside $\sigma(R)$). Then by the induction hypothesis, we have $\beta \rhd_t^{\Gamma'} \delta$ and $\delta \rhd_t^{\Gamma'} \gamma$. Thus $\beta \rhd_s^\Gamma \gamma$, by transitivity.*

We have the following:

Proposition 7 *Let $\rho = L \rightarrow R$ be a rewrite rule and let t, s be two graphs s.t. $t \rightarrow_{\rho,id} s$. If $\alpha \rhd_s^\Gamma \beta$ where α is a node in R then:*

- *Either β is a node in R and $\alpha \rhd_R^\Gamma \beta$.*
- *Or $\beta \notin \mathcal{N}(R)$ and there exists a node $\gamma \in var(R)$ s.t. $\alpha \rhd_\rho^\Gamma \gamma$ and $\gamma \rhd_t^\Gamma \beta$.*

Proof *The proof is by induction of $\rhd_R^\Gamma$. Assume that $\beta = i_s(\alpha)$. If β is not in R then the edge from α to β necessarily occurs in t, furthermore α, β cannot be relabeled, thus $\alpha \rhd_t^\Gamma \beta$. Assume that $\alpha \rhd_s^\Gamma \lambda \rhd_s^\Gamma \beta$. Assume that λ occurs in R. By the induction hypothesis, we deduce that $\alpha \rhd_R^\Gamma \lambda$. Furthermore, if $\beta \in \mathcal{N}(R)$ then $\lambda \rhd_R^\Gamma \beta$, otherwise there exists $\gamma \in var(R)$ s.t. $\lambda \rhd_\rho^\Gamma \gamma$ and $\lambda \rhd_t^\Gamma \beta$. By transitivity of $\rhd_\rho^\Gamma$, we deduce that in the former case $\alpha \rhd_R^\Gamma \beta$ and in the latter $\alpha \rhd_\rho^\Gamma \gamma$. Thus the proof is completed. Now, assume that λ is not in R. In this case we may assume, w.l.o.g. that no node in the path from λ to β is in R, thus $\lambda \rhd_t^\Gamma \beta$. By the induction hypothesis, there exists a node $\gamma \in var(R)$ s.t. $\alpha \rhd_\rho^\Gamma \gamma$ and $\gamma \rhd_t^\Gamma \lambda$ and by transitivity $\gamma \rhd_t^\Gamma \beta$.*

The lemmata 6 and 7 below state that the classes of $\rhd$-well-formed and clash-free graphs are stable by rewriting.

Lemma 6 *Let $\rho = L \rightarrow R$ be a well-formed rewrite rule and let t, s be two graphs s.t. $t \rightarrow_\rho s$. If t is well-defined then s is $\rhd$-well-formed.*

Proof *We only consider the case where $t \rightarrow_{\rho,id} s$ (it suffices to rename the nodes in ρ). Assume that s is not $\rhd$-well-formed. There exists a node α in $\mathcal{N}_\mathcal{D}(s)$ labeled by function symbol f s.t. one of the following conditions hold:*

1. *Either $i \not\sim_f j$ and $i_s(\alpha) \geq_s^\mathcal{N} \circ \leq_s^\mathcal{N} j_s(\alpha)$.*
2. *Or $i \notin \circlearrowleft_f$ and $i_s(\alpha) \rhd_s^\mathcal{N} \alpha$.*

Since $\alpha \in \mathcal{N}_\mathcal{D}(s)$, there exists a node λ, labeled by a defined symbol g, s.t. $\lambda \gg_s^i \alpha$, where $(i, \mathtt{sort}(\alpha)) \in in_g$. Assume that λ does not occur in t, i.e. that it is a created node. This implies that there exists a variable λ' in t s.t. $\lambda \gg_R^i \lambda'$, $\lambda' \xrightarrow{\mathcal{C}}_t \alpha$. Hence $\mathtt{sort}(\lambda') \succ \mathtt{sort}(\alpha)$. If a node α' occurring in a path from α is affected by ρ, then α' must occur in $\mathcal{R}N(\rho)$ and we must have $\alpha \rhd_t^\mathcal{N} \alpha'$, thus $\mathtt{sort}(\alpha) \succ \mathtt{sort}(\alpha')$. But since ρ is shape-preserving, this implies that either $\mathtt{sort}(\alpha)$ is unrestricted, which contradicts the assumptions 1 and 2 above, or that $\lambda' \rightarrow \alpha'$ is not compatible with R. But this last condition cannot hold since by definition $\lambda' \rhd_t^\mathcal{N} \alpha \rhd_t^\mathcal{N} \alpha'$ and t is a $\rhd$-well-formed graph containing R.

Now, assume that λ occurs in t. If ρ redirects an edge starting from a node in this path, then t contains a clash between α and β. Thus we have $\lambda \gg_t^i \alpha$. Let h be the label of the root L. Assume that α occurs in t. Note that we have $\alpha \neq \beta$ since β does not occur in s. ρ cannot redirect $c_s(\alpha)$ for any feature c since otherwise there would be a potential clash between λ and β. We distinguish the two cases above.

1. *$i \not\sim_f j$ and $i_s(\alpha) \geq_s^\mathcal{N} \circ \leq_s^\mathcal{N} j_s(\alpha)$. There exists a node γ s.t. $i_s(\alpha) \rhd_s^\mathcal{N} \gamma$ and $j_s(\alpha) \rhd_s^\mathcal{N} \gamma$.*

   By Lemma 5 we have $i_s(\alpha)^- \rhd_t^\mathcal{N} \gamma^-$ and $j_s(\alpha)^- \rhd_t^\mathcal{N} \gamma^-$, i.e. (since $i_t(\alpha)$ and $j_t(\alpha)$ cannot be redirected): $i_t(\alpha) \rhd_t^\mathcal{N} \gamma^-$ and $j_t(\alpha) \rhd_t^\mathcal{N} \gamma^-$. Thus t is not $\rhd$-well-formed, which is impossible.

2. *In the second case, we have similarly* $a_t(\alpha) \rhd_s^{\mathcal{N}} \alpha^- = \alpha$, *hence* t *is not* $\rhd$*-well-formed.*

Consequently, α *cannot occur in* t*, hence is created by* ρ*. Then* α *occurs in* R*, as the nodes* $j_s(\alpha)$*, for every* $j \in [1..ar(f)]$*. Once again, we consider each case separately.*

1. *There exists a node* γ *s.t.* $i_s(\alpha) \rhd_s^{\mathcal{N}} \gamma$ *and* $j_s(\alpha) \rhd_s^{\mathcal{N}} \gamma$*. By Proposition 7 we have* $i_s(\alpha) \rhd_\rho^\Gamma \gamma'$ *and* $j_s(\alpha) \rhd_\rho^\Gamma \gamma''$ *where* $\gamma', \gamma'' \rhd_t^{\mathcal{N}} \gamma$*. But then* $\gamma' \downarrow^{\mathtt{sort}(\gamma)} \gamma''$ *is compatible with* L*, thus the rule is not well-formed, a contradiction.*

2. *If* $i_s(\alpha) \rhd_s^{\mathcal{N}} \alpha$*, then by Proposition 7, we have* $i_s(\alpha) \rhd_\rho^{\mathcal{N}} \alpha$*, hence* ρ *is not well-formed, which is impossible.*

Lemma 7 *Let* $\rho = L \to R$ *be a clash-free rewrite rule and let* t, s *be two graphs s.t.* $t \to_\rho s$*. If* t *is well-defined then* s *is clash-free.*

Proof *As usual, we assume that* $t \to_{\rho,id} s$*. We denote by* λ *the root of the left-end side of* ρ*. Assume that* s *contains a potential clash. There exist two nodes* $\alpha \neq \beta$*, labeled by defined symbols* f *and* g *respectively, two natural numbers* i, j *and a node* γ *of sort* $\mathtt{s}$ *s.t.* $i_s(\alpha) \rhd_s^\Gamma \gamma$*,* $j_s(\beta) \rhd_s^\Gamma \gamma$*, where* $\Gamma = \mathcal{N}(s) \setminus \{\alpha, \beta\}$*,* $(i, \mathtt{s}) \in w_f, (j, \mathtt{s}) \in w_g$*.*

Assume that both α *and* β *occurs in* t*. Then since the nodes labeled by defined symbols cannot be redirected (and since the arguments of these nodes cannot be redirected, otherwise the graph* t *would contain a potential clash), we have* $l_s(\alpha) = l_t(\alpha) = f$*,* $l_s(\beta) = l_t(\beta) = g$*,* $i_s(\alpha) = i_t(\alpha)$ *and* $j_s(\beta) = j_t(\beta)$*. By Lemma 5 we deduce* $i_t(\alpha) \rhd_t^{\Gamma^- \cup \{\lambda\}} \gamma^-$*,* $j_t(\beta) \rhd_t^{\Gamma^- \cup \{\lambda\}} \gamma^-$*. Thus* t *contains a clash, which is impossible.*

Assume that α *is in* t *but not* β*. We have* $i_t(\alpha) \rhd_t^{\Gamma^- \cup \{\lambda\}} \gamma^-$*. Let* γ' *be the last node in the path from* $j_s(\beta)$ *to* γ *s.t.* γ'^- *occurs in* L *(this node necessarily exists, because* $b_s(\beta)^-$ *must be in* $\sigma(L)$ *since* β *is created by* ρ*). There exists* k *s.t.* $k_t(\lambda) \to_L \gamma'^-$ *(since* ρ *is connex). Moreover, if* $(j, \mathtt{s}) \in in_f$ *then we must have* $(k, \mathtt{sort}(\gamma'^-)) \in in_{l_t(\lambda)}$*. Consequently, there must be a clash in* t *between* λ *and* α*. The proof is similar if* β *is in* t *but not* α*.*

Thus both α *and* β *are created. By Proposition 7, there exist two nodes* $\alpha', \beta' \in var(R)$ *s.t.* $\alpha' \rhd_t^\Gamma \gamma, \beta' \rhd_t^\Gamma \gamma$*,* $\alpha \rhd_\rho^\Gamma \alpha', \beta \rhd_\rho^\Gamma \beta'$*. Then* $\alpha' \downarrow^{\mathtt{sort}(\gamma)} \beta'$ *is compatible with* L *and* ρ *contains a potential clash which is impossible.*

The next theorem states the main result of our paper.

Theorem 3 *Let* $\mathcal{R}$ *be a well-defined GRS. There is no clash-reducible well-defined graph.*

Proof *This is an easy consequence of Theorem 1, Lemmata 6 and 7 and Proposition 4.*

4 Examples

We provide examples of well-defined and non well-defined GRS. It is worth mentioning that every standard term rewrite system is well-defined, as well as any GRS that does not perform any side-effect. Obviously this is also the case if the "defined part" of every graph in the right-hand side has a tree structure (with non joinable leaves).

Example 5 *The GRS defining the in-situ reverse in Section 1.3 is well-defined (for any family of relation* $\sim_f, \circlearrowleft_f$*). Notice that* this system has a non joinable critical pair, *e.g.* $rev'(\alpha{:}cons(\beta, \gamma), \lambda); rev'(\alpha, \lambda')$*, where* λ, λ' *denote distinct lists (obviously this is also the case if other graph rewriting formalisms are used, e.g. algebraic approaches). This shows the interest of our analysis, which guarantees that no such graph will be generated during the rewriting process (if the initial graph is well-defined).*

The graph $rev(\alpha{:}[1, 2]); rev(\alpha)$ *is not well-defined (since the two occurrences of rev affect the same graph* α*).* $rev(rev(\alpha))$ *is well-defined (since the path from the first occurrence of rev to* α *necessarily contains the second occurrence of rev, there is no potential clash).*

Let eq be a function comparing two lists and let copy be a function returning a copy of its argument, defined as in Example 2.

$$\begin{array}{ll} eq(nil, nil) \to true & eq(cons(\alpha, \beta), cons(\alpha, \gamma)) \to eq(\beta, \gamma) \\ eq(cons(\alpha, \bot), cons(\delta, \bot)) \to false & palindrom(\alpha) \to eq(\alpha, rev(copy(\alpha)) \end{array}$$

The above GRS is well-defined. We assume that the constructors cons, nil have the profiles $\mathtt{nat} \times \mathtt{list} \to \mathtt{list}$ *and* $\mathtt{list}$*, where* $\mathtt{nat}$ *denotes (for instance) natural numbers. If instead, cons has profile* $\mathtt{list} \times \mathtt{list} \to$

`list`, *then the above GRS is not well-defined. Indeed, in the last rule the arguments of rev and eq occur in the same data space since copy(α) and α share the* elements *of the list. If these elements are of type* `nat` *then they cannot be affected by rev (which only affects nodes of sort* `list`*) thus there is no potential clash, but if the elements are also of sort* `list` *then a clash is possible.*

Example 6 *The GRS defining the function inc in Section 1.3 is well-defined for every shape relation (this is obvious since the right-hand sides of the rules contain only one defined symbol, occurring at root position). However the GRS defining the function append (more precisely the third rule) is not well-defined in general. Consider for instance the graph length($append(\alpha{:}[0,1|alpha], nil)$). After one reduction step we get: $length(\alpha{:}[0|append([1|\alpha], nil)))$. The term obviously contains a potential clash since append may affect α. If we assume that $2 \notin \circlearrowleft_{cons}$ (non looping rules are considered) and that $1 \not\sim_{append} 2$ (this is necessary to ensure that the rules are shape-preserving) then the GRS becomes well-defined.*

Example 7 *The following rules specify an in-situ insertion sort.*

$$\begin{array}{l} sort(nil) \to nil \quad sort(\lambda{:}cons(\alpha,\beta)) \to insert(sort(\beta), \lambda{:}cons(\alpha, nil)) \\ insert(nil, \lambda{:}cons(\alpha, nil)) \to \lambda \\ insert(\delta{:}cons(\beta,\gamma), \lambda{:}cons(\alpha, nil)) \to \\ \quad if\ \alpha \leq \beta\ then\ \lambda{:}cons(\alpha,\delta)\ else\ \delta{:}cons(\beta, insert(\gamma,\lambda)) \end{array}$$

The rules for if ... then ... else ... and $\leq$ are defined as usual. One can check that the above rules are well-defined if we have $1, 2 \notin \circlearrowleft_{cons}$ (i.e. there is no cyclic list).

Example 8 *The following function (physically) transforms a cyclic list into a standard one:*

$$\begin{array}{ll} c(\alpha) \to c'(\alpha,\alpha) & c'(\alpha{:}cons(\beta,\gamma),\delta) \to \alpha; c'(\gamma,\delta) \\ c'(\alpha{:}cons(\beta,\gamma),\gamma) \to \alpha{:}cons(\beta, nil) & \end{array}$$

This GRS is not well-defined. For instance in the rule to the right, α may be reachable from δ, which entails that a potential clash exists between c' (in the right-hand side) and a defined function operating on α. Fortunately, one can easily specify the same function in such a way that the GRS becomes well-defined, it suffices to return the result only when the end of the list is reached, so that no other function can be applied on the list before the evaluation of c is completed:

$$\begin{array}{ll} c(\alpha) \to c'(\alpha,\alpha) & c'(\alpha{:}cons(\beta,\gamma),\delta) \to c'(\gamma,\delta) \\ c'(\alpha{:}cons(\beta,\gamma),\gamma) \to \gamma; \alpha{:}cons(\beta, nil) & \end{array}$$

5 Discussion

We have identified a syntactic class of graphs and graph rewrite systems, called well-defined, for which the normal forms do not depend on the reduction ordering among the nodes. This result (that is not covered by traditional approaches in graph rewriting) allowed us to devise very general confluence criteria and efficient evaluation strategies for the considered class of graphs. The previous property is ensured by automatically analyzing the shape of the reduced graphs as well as the side-effects performed by the defined symbols. Examples showed that well-defined systems are expressive enough to denote many algorithms useful in practical programming.

The class is parameterized by the shape relations $\sim_f$ and $\circlearrowleft_f$. The above results can be used in at least three ways. First, $\sim_f$ and $\circlearrowleft_f$ can be provided by the programmer. Then it is easy to check automatically that the graphs and rules are well-defined (in polynomial time w.r.t. the size of the graphs and GRS). Alternatively, one can check for every set of rules $\mathcal{R}$ and for every graph t whether there exist shape relations s.t. t and $\mathcal{R}$ are well-defined. To this purpose, it suffices to use a fixpoint algorithm, starting with the minimal relations $i \sim_f j \Leftrightarrow i = j$ and $\circlearrowleft_f = \emptyset$ (for every function symbol f) and weakening iteratively these relations in such a way that the conditions of Definition 15 are satisfied (until a fixpoint is reached). According to Definition 8, one can always ensure that the conditions hold by adding elements in $\sim_f$ and $\circlearrowleft_f$. This has the advantage that the process is fully automatic. However, the complexity of the computation is much higher (it increases by a factor $n \times m^2$ where m is the number of features and n the number of function symbols). This may be unacceptable in practice, in particular if one has to evaluate many distinct terms t using the same GRS, since the shape relations must be recomputed for each term. Therefore, a good compromise could be to compute automatically, for a given GRS, a *maximal* pair of relations $(\sim_f, \circlearrowleft_f)$ such that $\mathcal{R}$ is well-defined. Then it is easy to check whether the terms are well-defined w.r.t. these relations. This is easy to automatize,

and the advantage is that the computation is performed once and for all for a given GRS (it does not need to be computed again each time a graph is evaluated). A drawback is that there exist in general many maximal relations.

The previous results suggest that well-defined graph rewrite systems may provide a convenient high-level framework for handling complex pointer-based data-structures and algorithms operating on them. An important aspect is how to handle "rejected" GRS, i.e. GRS that are not well-defined. In this case one should provide the programmer a clear and intuitive explanation of why the GRS is to be rejected (e.g. an example showing the ambiguity of the rewrite relation). The programmer could react either by correcting the GRS or by providing additional information (e.g. refined shape relations) ensuring that the GRS becomes well-defined. This point obviously deserves further investigations. Algorithms "compiling" non well-defined GRS into (equivalent) well-defined ones could also be defined.

References

[1] Z. M. Ariola, J. W. Klop, and D. Plump. Bisimilarity in term graph rewriting. *Inf. Comput.*, 156(1-2):2–24, 2000.

[2] H. Barendregt, M. van Eekelen, J. Glauert, R. Kenneway, M. J. Plasmeijer, and M. Sleep. Term Graph Rewriting. In *PARLE'87*, pages 141–158. Springer, LNCS 259, 1987.

[3] R. Echahed and J.-C. Janodet. Admissible graph rewriting and narrowing. In *Proceedings of 15th International Conference and Symposium on Logic Programming*, pages 325–340, Manchester, 1998. MIT Press.

[4] R. Echahed and N. Peltier. Narrowing data-structures with pointers. In *Proceedings of ICGT (International Conference of Graph Transformation)*. Springer LNCS 4178, September 2006.

[5] R. Echahed and N. Peltier. Non Strict Confluent Rewrite Systems for Data-Structures with Pointers. In *Proceedings of RTA (Rewriting Techniques and Applications)*. Springer LNCS 4533, June 2007.

[6] R. Echahed and N. Peltier. A needed rewriting strategy for data-structures with pointers. In *Rewriting Techniques and Applications, 19th International Conference, RTA 2008*, volume 5117 of *Lecture Notes in Computer Science*, pages 63–78. Springer, 2008.

[7] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of Algebraic Graph Transformation (Monographs in Theoretical Computer Science. An EATCS Series)*. Springer, March 2006.

[8] H. Ehrig, H.-J. Kreowski, U. Montanari, and G. Rozenberg, editors. *Handbook of graph grammars and computing by graph transformation*. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1999.

[9] M. Hanus, S. Lucas, and A. Middeldorp. Strongly sequential and inductively sequential term rewriting systems. *Information Processing Letters*, 67(1):1–8, 1998.

[10] R. Heckel. Graph transformation in a nutshell. *Electronic Notes in Theoretical Computer Science*, 148(1):187–198, February 2006.

[11] G. Huet and J.-J. Levy. Computations in orthogonal rewriting systems. In J.-L. Lassez and G. Plotkin, editors, *Computational Logic: Essays in Honor of Alan Robinson*, pages 395–443. MIT Press, Cambridge, MA, 1991.

[12] J. W. Klop and A. Middeldorp. Sequentiality in orthogonal term rewriting systems. *J. Symb. Comput.*, 12(2):161–195, 1991.

[13] D. Plump. Confluence of graph transformation revisited. In *Processes, Terms and Cycles*, volume 3838 of *LNCS*, pages 280–308. Springer, 2005.

[14] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. *Logic in Computer Science, Symposium on*, 0:55, 2002.

[15] H. Schorr and W. M. Waite. An Efficient Machine Independent Procedure for Garbage Collection in Various List Structures. *Communication of the ACM*, 10:501–506, August 1967.