Drags: A Simple Algebraic Framework for Graph Rewriting☆

Nachum Dershowitz

School of Computer Science, Tel Aviv University, Tel Aviv, Israel

Jean-Pierre Jouannaud

LIX & DEDUCTEAM[1], École Polytechnique, Palaiseau, France

Abstract

We are interested in a natural generalization of term-rewriting techniques to what we call *drags*, viz. finite, directed, ordered, rooted multigraphs, each vertex of which is labeled by a function symbol. To this end, we develop a rich algebra of drags that generalizes the familiar term algebra and its associated rewriting capabilities. Viewing graphs as terms provides an initial building block for rewriting with such graphs, one that should impact the many areas where computations take place on graphs.

This paper is dedicated to the memory of Maurice Nivat, a colleague and friend of both authors. During his career, Maurice impacted many research areas, including formal language theory, combinatorics, semantics of programming languages, concurrency, and discrete geometry. His interest in formal languages lead him to study confluence properties of rewriting words, and his interest in concurrency lead him to study languages of paths. The present work follows along that route.

1. Introduction

Rewriting with graphs has a long history in computer science, graphs being used to represent data structures, but also program structures and even concurrent and distributed computational models. They therefore play a key rôle in program evaluation, transformation, and optimization, and more generally program analysis; see, for example, [10]. Since graphs originate from topological investigations, it is no wonder seeing them also play an important rôle in modern algebraic topology, more specifically in the theory of operads.

Our initial interest in graph rewriting originated some needs of the study of operads, which has strong relationships with various areas of computer science, including concurrency theory and type theory. In the introduction to their book on the algebra of operads [18], Bremner and Dotsenko write:

> Elements of operads are conventionally represented by linear combinations of trees, "tree polynomials". Generalizations to algebraic structures where monomials are graphs that possibly have loops and are possibly disconnected, e.g. properads, PROPs, wheeled operads, etc., are still unknown, and it is not quite clear if it is at all possible to extend Gröbner-flavored methods to those structures.

This problem motivated our initial interest in first-order terms with both sharing and back-arrows. These graphs, qua expressions, can be composed to form monomials. In turn, these monomials can be added to form polynomials, addition being associative and commutative. These finite graphs and their polynomials are seen as expressions of an algebraic structure that is commonly referred to as an algebraic operad.

[1] `http://deducteam.gforge.inria.fr`

☆This paper is an extended version of [17].

Email addresses: `nachum.dershowitz@cs.tau.ac.il` (Nachum Dershowitz), `jeanpierre.jouannaud@gmail.com` (Jean-Pierre Jouannaud)

As rewriting graphs is very similar to rewriting algebraic terms, the same questions recur: What rewriting relation do we need? Is there an efficient pattern-matching algorithm? How can one determine if a particular rewriting system is confluent and/or terminating? And – last but not least – how should we represent graphs?

The above questions have, for these reasons, been addressed by the rewriting community since the mid-seventies in research initiated by Hartmut Ehrig and his collaborators, either in the context of general graphs [20], or for specific classes of graphs, namely those that do not admit cycles [47, 45]. In particular, termination and confluence techniques have been elaborated for various generalization of trees, such as rational trees, directed acyclic graphs, jungles, term-graphs, lambda-terms, and lambda-graphs, as well as for graphs in general. See [11, 12, 50] for detailed accounts of these techniques,and [29] for a survey of implementations of various forms of graph rewriting and of available analysis tools.

However, we quickly felt that the existing categorical framework for rewriting arbitrary graphs was rather heavy for many practical uses on the one hand, while, on the other hand, its instances disallowing cycles in graphs, namely dags (directed acyclic graphs), jungles and term graphs, were too specific for some applications, in particular for program transformations. At that point, our interest shifted towards the design of a convenient mathematical framework that would generalize term graphs by allowing for cycles generated by the possibility of having backarrows. This is not say that our interest in operads has vanished, but that it has become a secondary target until after a convenient framework emerges.

This paper describes the design of a general class of graphs – actually, multigraphs – that has good structural properties with respect to rewriting. A specificity used for that purpose is the presence of arbitrarily many roots, as well as of variables labeling some leaves – called "sprouts", which together define the graph "interface". Connecting the sprouts of one graph with the roots of another allows one to compose graphs in a particularly natural way, one that is reminiscent of several other compositional frameworks found in process algebra and homotopy type theory based on monoidal categories [8]. Graphs in this class come equipped with a simple algebraic structure that allows one to view them as terms with sharing and cycles, and, therefore, to mimic many techniques that have been developed for trees over the years. In particular, rewrite rules become pairs of graphs with matching numbers of roots and with no variables on the right-hand side that do not also appear on the left-hand side, while rewriting amounts to composing both sides of the rule with the same graph taken as the rewriting context. Although rewriting is general enough so that it can apply inside, outside, or across cycles, our approach ensures that rewriting cannot create dangling pointers, as can be the case with the traditional approaches to graph rewriting.

It should be noted that related ideas show up in recent works on symmetric monoidal categories and string diagrams and the representation of the latter as graphs with interfaces [5].

This work also opens the door for a notion of rewrite orderings for graphs and for syntactic termination proof methods that generalize the usual syntactic techniques used in term rewriting. Such generalizations were not previously possible in the presence of arbitrary cycles. Drag rewrite orderings are currently under exploration; see [15] and [16]. In the latter reference, a total well-founded order is developed for drags that fulfills the needs for developing Gröbner-based techniques for algebraic operads. For applications to program transformations, however, syntactic orders are often insufficient, so they need to be combined with interpretation based techniques. This question has not been considered yet, although our techniques have, in principal, the potential to allow for such a combination. Confluence of drag rewrite rules also has not been considered to date. Hopefully, the experience gained with the study of confluence for the categorical approaches to graph rewriting can ease the way [37, 19, 22, 5].

Despite being purely theoretical, the framework developed here should lead to easy implementations: the view of a drag as a term with sharing and cycles is extremely easy to implement, and suggests thereby a possibility to extend many existing term-graph rewriting implementations, to actually do graph rewriting.

The particular class of graphs we are interested in is introduced in Section 2. The algebra of drags is described in Section 3, followed by the view of drags as dags in Section 4. Drag rewriting is introduced in Section 5 and explored further in Section 6. Variations of the drag model are discussed in Section 7. A detailed example is presented in Section 8. Section 9 presents our language of drag expressions, whose semantics is studied in Section 10. Normal form expressions are studied in Section 11. Comparison with other frameworks, namely, the DPO approach, the term-graph frameworks, and monoidal categories, comes next in Section 12. This is followed by a brief discussion in conclusion.

2. Drags

The class of graphs with which we deal here is that consisting of finite directed graphs with labeled vertices, allowing multiple edges between vertices and an arbitrary number of roots. In the present work, we assume that the outgoing neighbors (vertices at the other end of outgoing edges) are ordered (from left to right, say, in the figures) and that their number is fixed, depending solely on the label of the vertex. Some vertices with no outgoing edges are designated *sprouts*. We shall call these finite ***d***irected ***r***ooted l***a***beled ***g***raphs, ***drag***s.

We presuppose a set of function symbols Σ, whose elements $f \in \Sigma$ will be used as labels for vertices, and which are each equipped with a fixed arity. (We have no associative-commutative symbols here.) In addition, we have a set of nullary variable symbols Ξ, disjoint from Σ, which will be used to label sprouts.

The successors of a vertex labeled f in a drag are implicitly interpreted as the arguments of the function symbol f. The graph describes therefore the inverse of the computational flow. (Some might prefer the converse choice; this is a matter of taste and convention.) This class of graphs appears to be a very general model for describing computations in the absence of binding constructs.

In contrast with the usual categorical approaches to graph rewriting considered in Section 12, our multigraph model allows for a very standard conception of rewriting. The novelty, however, lies in that we allow for arbitrarily many roots and arbitrarily complex cycles: there is no restriction on the structure of the drags we deal with. This is because we view drags as networks of processing units (the vertices) that accept a given number of data as inputs and deliver some datum as output that can be sent over an arbitrary number of one-way channels. Channels can of course be duplicated, allowing therefore for arbitrary sharing, and there is an order among the input channels of a processing unit so as to appropriately discriminate among the inputs. Finally, the generality of the model requires that these one-way channels connect the processing units in an arbitrary way. All this makes our drag model very general.

To ameliorate notational burden, we will use vertical bars $|\cdot|$ to denote various quantities, such as length of lists, size of sets or of expressions, and even the arity of function symbols. We use $\varnothing$ for an empty list, set, or multiset, $\cup$ for set and multiset union, as well as for list concatenation, and $\setminus$ for set or multiset difference. We mix these, too, and denote by $K \setminus V$ the sublist of a list K obtained by filtering out those elements belonging to a set V. We will also identify a singleton list, set, or multiset with its contents to avoid unnecessary clutter. A list of elements $e_1, \ldots, e_n$ will be written as $[e_1 \mathinner{..} e_n]$. In particular, we write $[1 \mathinner{..} n]$ for the natural numbers from 1 to n.

Definition 1 (Drags)**.** *A* drag *is a tuple $\langle V, R, L, X, S \rangle$, where*

1. *V is a finite set of* vertices*;*
2. *$R : [1 \mathinner{..} |R|] \to V$ is a finite list of vertices, called* roots*, so $R(n)$ refers to the nth root in the list;*
3. *$S \subseteq V$ is a set of* sprouts*, leaving $V \setminus S$ to be the* internal *vertices;*
4. *$L : V \to \Sigma \cup \Xi$ is the* labeling *function, mapping internal vertices $V \setminus S$ to labels from the vocabulary Σ and sprouts S to labels from the vocabulary Ξ;*
5. *$X : V \to V^*$ is the* successor *function, mapping each vertex $v \in V$ to a list of vertices in V whose length equals the arity of its label (that is, $|X(v)| = |L(v)|$).*

The pair (R, S) made of the list of roots and set of sprouts of the drag D will be called its interface.

By convention, R will denote both the function itself and its result, that is the list $[R(1) \mathinner{..} R(n)]$, where $n = |R|$.

A drag is closed *if it has no sprouts S. The last component $S = \varnothing$ will often be omitted from the tuple for closed drags. Non-closed drags act as* patterns.

If $b \in X(a)$, then (a, b) is a directed edge *with* source *a and* target *b. We also write aXb. The reflexive-transitive closure X^* of the relation X is called* accessibility. *A vertex v is said to be* accessible *from vertex u, and likewise that u* accesses *v, if uX^*v. Vertex v is* accessible *(without qualification) if it is accessible from some root. A root r is* maximal *if any root that can access it is accessible from it. We denote by $R^\bullet$ the set of maximal roots.*

A drag is clean *if all its vertices are accessible, and* linear *if no two sprouts have the same label.*

The labeling function extends to lists, sets, and multisets of vertices in the expected manner.

It will sometimes be convenient to consider roots as specific incoming edges and to identify a sprout with the variable symbol that is its label. We use these facilities unannounced.

The component R of a drag is a list of roots possibly with repetitions, whereas S is a set of sprouts. Having a list of roots, rather than a set, is one of the keys to a nice algebra of drags, and consequently for a general notion of rewriting.

Sprouts may be roots, as we shall see in examples. This ability plays a key rôle in our model, which would not be as expressive otherwise. It is also needed for the algebraic structure of drags developed in Section 3. Finally, nonlinear drags play an essential rôle in shared rewriting.

Given a drag $D = \langle V, R, L, X, S \rangle$, we make use of the following notations: $\mathcal{V}er(D) = V$ for its set of vertices; $\mathcal{I}nt(D)$ for its set of internal vertices; $X_D = X$ for its successor function; $\mathcal{R}(D) = R$ for its roots (list or set, depending on context); $\mathcal{S}(D) = S$ for its set of sprouts; $\mathcal{V}ar(D) = L(S)$ for the set of variables labeling its sprouts; $\mathcal{M}var(D)$ for the multiset of those variables; $D^{\overline{r}}_{\overline{s}}$ to indicate its list of roots and set of sprouts; $\mathcal{A}cc(D) = X^*(R)$ for its set of accessible vertices; and $D^\sharp$ for the clean drag obtained by removing inaccessible vertices and their related edges, which fits with the underlying intended behavioral semantics of drags. Note that cleanness, as can be expected, relates to garbage collection of unreachable vertices.

Ariola and Klop [1] introduced a colorful terminology that we shall adopt when convenient: *back arrows* refer to edges pointing back up the path from the root, *vertical sharing* refers to edges ending up in some vertex further away from the root, and *horizontal sharing* is used for edges going sideways between paths.

Terms as ordered trees, sequences of terms, and terms with shared subterms are all particular kinds of drags:

Definition 2. *A drag is referred to*

- *as a* jungle *if its successor function is acyclic and all its roots are maximal;*
- *as a* dag *if it is a jungle with only one root (that is, whose list of roots is a singleton);*
- *as a* forest *if it is a jungle whose list of roots has no repetition and vertices other than the root have a single incoming edge;*
- *and as a* tree *if it is a dag whose vertices other than the root have a single incoming edge, or, equivalently, if it's a forest whose list of roots is a singleton.*

Jungles were introduced in [24], where they were used to model programs. The definition given there is different as it uses hypergraphs, but it is equivalent to the present one.

A central concept used throughout will be that of subdrags:

Definition 3 (Subdrag)**.** *Given a drag $D = \langle V, R, L, X, S \rangle$ and a subset $W \subseteq V$ of its vertices, its* subdrag $D_{|W}$, *generated by W, is the clean drag $\langle V', R', L', X', S' \rangle$, where*

1. *its vertices $V' \subseteq V$ are the least superset of W that is closed under successors X;*
2. *its roots R' are $(R \cap V') \cup (X(V \setminus V') \cap V')$; and*
3. *L', X', S' are the restrictions of L, X, S to that V'.*

The roots of the subdrag include as new roots those vertices in V' that had an incoming edge in D that is no longer in the subdrag $D_{|W}$, with the corresponding multiplicity. The order of those additional roots in R' does not really matter for now.

In particular, restricting a drag D to its maximal roots, yields the clean drag $D^\sharp$. We will say of a drag, whose list of roots is R, that it is *generated* by $R^\bullet$, its maximal roots. On the other hand, restricting any drag to an empty set of vertices yields the empty drag $\varnothing \stackrel{\text{def}}{=} 1^{\varnothing}_{\varnothing} = \langle \varnothing, \varnothing, \varnothing, \varnothing, \varnothing \rangle$.

Drags are graphs. An isomorphism between two drags is a one-to-one mapping between their respective sets of (accessible) vertices that identifies their respective labels (up to renaming of the sprouts' labels done here by the very same mapping) and lists of roots, and commutes with their respective successor functions.

Definition 4 (Drag Isomorphism)**.** *Given two drags $D = \langle V, R, L, X, S \rangle$ and $D' = \langle V', R', L', X', S' \rangle$ whose sprouts are identified with their labels, an* isomorphism *from D to D' is a one-to-one mapping $o : \mathcal{A}cc(V) \to \mathcal{A}cc(V')$ such that*

1. *mapping o restricts to bijections between the lists of roots and the sets of accessible sprouts;*

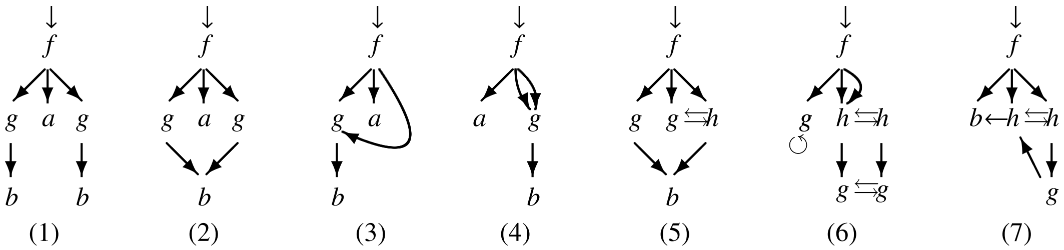

Figure 1: Seven pairwise non-isomorphic drags whose vertices are indicated by the labels.

2. *mapping o respects labels and successors of internal vertices: $\forall v \in \mathcal{A}cc(V) \setminus S$. $L(v) = L'(o(v))$ and $X(v) = X'(o(v))$.*

We write $D =_o D'$.

It is sometimes useful to consider multisets instead of lists of roots, resulting in a coarser equivalence on drags. Accordingly, we write $D \simeq_o D'$ and say that D, D' are quasi-isomorphic *if o restricts to a bijection between the multisets (instead of lists) of roots, that is, if their lists of roots coincide modulo permutation and renaming by o. Isomorphism is sometimes dubbed* strict *to distinguish it from quasi-isomorphism.*

The mapping o may be omitted, in particular when it is the identity.

Identifying sprouts with the variables that label them saved an additional one-to-one mapping between variables that would otherwise have been necessary.

Note that inaccessible vertices are ignored in the definition of drag isomorphism (this may be unusual, but is actually very natural for rooted drags), making any drag D isomorphic to its clean version $D^\sharp$:

Lemma 5. *For any drag D, $D = D^\sharp$.*

Example 6. *Consider the seven single-rooted drags depicted in Figure 1, with the convention that the order of arguments of function symbols begins with the leftmost outgoing edge and continues counterclockwise. An incoming down-arrow indicates a root.*

Let $\Sigma = \{a, b, f, g, h\}$ with $|f| = 3$, $|h| = 2$, $|g| = 1$, $|a| = |b| = 0$. We first consider the term $f(g(b), a, g(b))$ in which b or $g(b)$ can be shared. Among all possible cases, three are represented (1–3). The fifth (5) has both horizontal and vertical sharing. The sixth (6) has three cycles: an independent leftmost loop and a horizontal cycle sharing another horizontal cycle below. The last (7) has a single big cycle, which is both horizontal and vertical including an inner cycle.

3. Drag algebra

The key to working with drags is that they can be equipped with associative composition laws that preserve isomorphisms.

First, one can sum two drags D, D' having pairwise disjoint sets of vertices by putting them side by side, making a new drag $D \oplus D'$, or, following our wording, $D\ D'$. This addition is clearly associative, has a neutral element (the empty drag), but is non-commutative. However, it is *quasi-commutative* in the sense that $D\ D'$ and $D'\ D$ are quasi-isomorphic. This operation is sometimes called *horizontal composition* in the literature.

Next, we can also equip drags with a parameterized binary composition operator that connects sprouts of each of two drags with roots of the other according to a device we call a *switchboard*. This will be a primal notion, one that renders our drag model compositional. This second manner of composition can be dubbed *vertical*.

3.1. Switchboards

As usual, we denote by $\mathcal{D}om(\xi)$ and $\mathcal{I}m(\xi)$ the *domain (of definition)* and *image* of a (partial) function ξ, respectively, using $\xi_{A\to B}$ for its restriction going from subset A of its domain to subset B of its image, and omitting $\to B$ when irrelevant. The domain $\mathcal{D}om(\xi)$ is a set of sprouts, which we often identify with their variable labels. The image $\mathcal{I}m(\xi)$ is a set of natural numbers standing for positions in a list. Therefore, $\mathcal{I}m(\xi)$ may differ from $\xi(R)$, which is a list of natural numbers, possibly with repetitions.

Definition 7 (Switchboard). *Let $D = \langle V, R, L, X, S \rangle$ and $D' = \langle V', R', L', X', S' \rangle$ be drags. A* switchboard *ξ for D, D' is a pair $\langle \xi_D : S \rightarrow [1 .. |R'|]; \xi_{D'} : S' \rightarrow [1 .. |R|] \rangle$ of partial injective functions such that*

1. *$s \in \mathcal{D}om(\xi_D)$ and $L(s) = L(t)$ imply $t \in \mathcal{D}om(\xi_D)$ and $\xi_D(s) = \xi_D(t)$ for all sprouts $s, t \in S$;*
2. *$s \in \mathcal{D}om(\xi_{D'})$ and $L'(s) = L'(t)$ imply $t \in \mathcal{D}om(\xi_{D'})$ and $\xi_{D'}(s) = \xi_{D'}(t)$ for all sprouts $s, t \in S$;*
3. *ξ is* well-behaved*, that is, it does not induce any cycle among sprouts, using ξ, R, R' relationally:*

$$\nexists n > 0, s_1, \ldots, s_{n+1} \in S, t_1, \ldots, t_n \in S', s_1 = s_{n+1}. \forall i \in [1 .. n].\ s_i\ \xi_D R' X'^*\ t_i\ \xi_{D'} R X^*\ s_{i+1}$$

We also say that $\langle D', \xi \rangle$ is an extension *of D, and that it is* clean *when D is.*

Example 8. *Let $f(x)$ and $g(y)$ be two drags both of whose lists of roots have length 2, made, for each drag, of the internal node first (the root of the tree identified with its label f or g), and then the sprout (identified with its variable label x or y). Note that these drags are not trees, because each one has two roots. The mapping $\langle x \mapsto 2; y \mapsto 2 \rangle$ is then ill-behaved, hence is not a switchboard.*

A variable in a drag should be understood as a potential connection to some root of another drag. Therefore, two sprouts labeled by the same variable should be connected to the same vertex, as required by conditions (1,2). These conditions are of course automatically satisfied by linear drags, and by nonlinear drags whose switchboard, called *linear*, is defined for sprouts whose variables are all different. In the case where several sprouts $S_1, \ldots, S_p$ belonging to $\mathcal{D}om(\xi)$ are labeled by the same variable x, injectivity of ξ imposes that the root $R(\xi(S_1))$ be repeated p times in the list R. As a consequence, $\xi_D(\mathcal{D}om(\xi_D))$ must be a set, making the set difference $[1 .. |R'|] \setminus \xi_D(\mathcal{D}om(\xi_D))$ well defined.

In practice, we shall often enforce disjointness of the sprout labels of D and D' so as to write a linear switchboard as a list of assignments from variables to root numbers without ambiguity. But this is by no means necessary nor always possible. Both notations $\langle \overline{x = m}; \overline{y = n} \rangle$ when necessary, and $\{\overline{z = p}\}$ when possible, will therefore coexist.

We now move to the composition operation on drags induced by a switchboard. The essence of this operation is that the (disjoint) union of the two drags is formed, but with sprouts in the domain of the switchboards merged with the roots to which the switchboard images refer. This operation removes one sprout and one root at the same time from the sets of sprouts and lists of roots of the union. The list of roots in the union is obtained, therefore, by adding the lists of roots of the two drags and then subtracting those removed by the switchboard, that is, those in its domain.

Merging sprouts with their images requires one to worry about the case where multiple sprouts are merged successively, when the switchboards map sprout to rooted-sprout to rooted-sprout, until, eventually, an internal vertex of one of the two drags must be reached because a switchboard is well-behaved. Computing that vertex is the rôle of the coming recursive definition:

Definition 9 (Target). *Let $D = \langle V, R, L, X, S \rangle$ and $D' = \langle V', R', L', X', S' \rangle$ be drags such that $V \cap V' = \emptyset$, and ξ be a switchboard for D, D'. The* target *$\xi^*(s)$ is a mapping from sprouts in $S \cup S'$ to vertices in $V \cup V'$ defined as follows:*
Let $v = R'(n)$ if $s \in S$, and $v = R(n)$ if $s \in S'$, where $n = \xi(s)$.

1. *If $v \in (V \cup V') \setminus (S \cup S')$, then $\xi^*(s) = v$.*
2. *If $v \in (S \cup S') \setminus \mathcal{D}om(\xi)$, then $\xi^*(s) = v$.*
3. *If $v \in \mathcal{D}om(\xi)$, then $\xi^*(s) = \xi^*(v)$.*

The target mapping $\xi^*(\_)$ is extended to all vertices of D and D' by letting $\xi^*(v) = v$ when $v \in (V \setminus S) \cup (V' \setminus S')$.

The target of a sprout is defined by induction on the acyclic relation on $X \cup X'$, defined as $\{(x \in S, y \in S') : x \xi_D R' y$ or $y \xi_{D'} R x\}$, identifying sprouts with their variable label. Were the switchboard not assumed well-behaved, then this relation would have cycles and the target function would not be defined. Note also that the computation of the target needs to know the interfaces of D, D' only.

Example 10. *Consider the last of the three examples in Figure 2, in which a drag D, whose list of roots is $R = [f\,x\,h]$ (identifying vertices with their label) is composed with a second drag whose list of roots is $R' = [g\,y\,y]$, via the switchboard $\{x \mapsto 3, y \mapsto 2\}$. We calculate the target of the two sprouts: $x\,\xi\,3\,R'\,y\,\xi\,2\,R\,h$; hence $\xi^*(x) = \xi^*(y) = h$.*

3.2. Drag composition

We are finally ready for defining the composition of two drags. Its set of vertices will be the union of two components: the internal vertices of the drags, and their sprouts which are not in the domain of the switchboard. The labeling will be given by restricting the original one to the set of vertices that is obtained.

Definition 11 (Composition)**.** *Let $D = \langle V, R, L, X, S\rangle$ and $D' = \langle V', R', L', X', S'\rangle$ be drags such that $V \cap V' = \varnothing$, and let ξ be a switchboard for D, D'. Their* composition *is the drag $D \otimes_\xi D' = \langle V'', R'', L'', X'', S''\rangle$, with interface (R'', S'') denoted $(R, S) \otimes_\xi (R', S')$, where*

1. $V'' = (V \cup V') \setminus \mathcal{D}om(\xi)$;
2. $S'' = (S \cup S') \setminus \mathcal{D}om(\xi)$;
3. $R'' = \xi^*(R([1 .. |R|] \setminus \xi_{D'}(\mathcal{D}om(\xi_{D'}))) \cup \xi^*(R'([1 .. |R'|] \setminus \xi_D(\mathcal{D}om(\xi_{D'})))$;
4. $L''(v) = \begin{cases} L(v) & \text{if } v \in V \cap V'' \\ L'(v) & \text{if } v \in V' \cap V'' \end{cases}$
5. $X''(v) = \begin{cases} \xi^*(X(v)) & \text{if } v \in V \setminus S \\ \xi^*(X'(v)) & \text{if } v \in V' \setminus S \end{cases}$

First, note that items (1) and (2) express the simple fact that the sprouts that are in the domain of the switchboard have disappeared. The definition of the roots given at item (3) is slightly more delicate. We explain the first half of the expression, that describes the roots originating from D. The explanation for the second half follows by symmetry. The roots of D that belong to the image of a sprout of D' are not roots of the composition: this is why we need to calculate $[1 .. |R|] \setminus \xi_{D'}(\mathcal{D}om(\xi_{D'}))$. Note that we do not use here the image of $\xi_{D'}$ which is a set, but $\xi_{D'}(\mathcal{D}om(\xi_{D'}))$, which is a list possibly with repetitions. The obtained list of root positions is then transformed into root names by applying R. Furthermore, some roots in that list may be sprouts in the domain of ξ_D, which should be replaced by their associated target, hence explaining the role of the outer ξ^* (which is the identity for all internal vertices and those sprouts that do not belong to $\mathcal{D}om(\xi_D)$.)

It follows that, if ξ_D is surjective and $\xi_{D'}$ total, a kind of switchboard that will play a key rôle for rewriting, then *all* roots and sprouts of D' disappear in the composed drag. (If $\xi_{D'}$ is surjective and ξ_D total, those from D disappear.) Otherwise, the symmetry of the definition is broken by choosing the roots originating from D to come first (union of lists being concatenation).

Finally, it is easy to see that $D \otimes_\xi D'$ is a well-defined drag, that is, it satisfies the arity constraint required at each vertex. Furthermore, since all calculations done in the definition of $D \otimes_\xi D'$ require the sole knowledge of their interfaces, using the same notation for composing the interfaces is natural.

Example 12. *We show in Figure 2 three examples of compositions, the first two with similar drags. The first composition is a substitution of terms. The second uses a bi-directional switchboard, which induces a cycle. In the second example, the remaining root is the first (red) root of the first drag which has two roots, the first red, the other black. The third example shows how sprouts that are also roots connect to roots in the composition (colors black and blue indicate roots' origin, while red indicates a root that disappears in the composition). Since x points at y and y at the second root of the first drag, a cycle is created on the vertex of the resulting drag which is labeled by h. Further, the third root of the first drag has become the second root of the result, while the first (resp., second) root of the second drag has become the third (resp., fourth) root of the result. This agrees of course with the definition, as shown by the following calculations (started in Example 10): $\xi^*([1, 2, 3] \setminus [2]) = \xi^*([1, 3]) = [f, h]$; and $\xi^*([1, 2, 3] \setminus [2]) = \xi^*([1, 2]) = [g, h]$, hence the list of roots of the resulting drag is $[f, h, g, h]$.*

The definition of composition does not assume that the input drags are clean. Composing a drag D with a drag D' consisting to a single non-root sprout labeled x, has an observable effect on D. Let us assume that D is the tree a, where a is a constant. Being a tree, this drag has a single root, a itself. Consider now the composition $D \otimes_{\{x \mapsto 1\}} D'$. The result is the drag (no longer a tree) that has a single vertex labeled a, but no root, hence is equivalent to the empty drag.

It is easy to characterize the set of variables of a drag obtained by composition of two arbitrary drags:

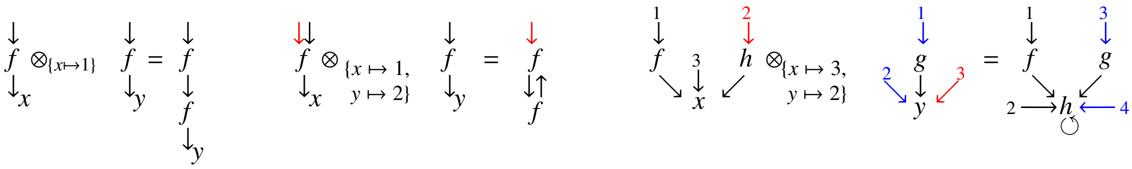

Figure 2: Three different effects of composition: substitution, formation of a cycle, and transfer of roots.

Lemma 13. *Let A, B be two drags whose sets of variables are disjoint, and ξ a switchboard for (A, B). Then, we have the following equalities between sets of variables:*

$$\mathcal{Var}(A \otimes_\xi B) = (\mathcal{Var}(A) \cup \mathcal{Var}(B)) \setminus \mathcal{Dom}(\xi)$$

Proof. This follows from the definition, replacing the sprouts by their variable names. □

Our definition of switchboard being (almost) symmetric, composition is commutative provided all roots of the resulting drag originate from the same side. Otherwise, composition of two drags D, D' in the reverse order yields the drag $D' \otimes_\xi D$ that is equal to $D \otimes_\xi D'$ up to cyclic permutation of their roots; we say that they are *quasi-commutative*.

Lemma 14 (Commutativity)**.** *Composition of drags is quasi-commutative, and becomes commutative for switchboards whose context or substitution is surjective.*

We now move to the next property, the existence of identity elements.

A linear drag all of whose vertices are its sprouts, whose set of edges is empty, and whose list of roots is a list made of its sprouts is called an *identity*. We denote it 1^Y_X, where X is its set of sprouts and Y is a list of elements of X. An identity drag is therefore clean if all its sprouts occur in its list of roots. We use $\varnothing$ for the clean drag $1^\varnothing_\varnothing$, called the empty drag.

Given a drag D, an *identity extension* of D is the pair $\langle 1^X_X, \iota\rangle$ made of an identity drag 1^X_X, such that $X \subseteq \mathcal{Var}(D)$, and an *identity* switchboard ι, such that $\mathcal{Dom}(\iota_D) = X$, ι_D is the identity (abusing our notations) and $\mathcal{Dom}(\iota_{1^X_X}) = \varnothing$.

Lemma 15 (Neutrality)**.** *Let D be a drag, $X \subseteq \mathcal{Var}(D)$, and $\langle 1^X_X, \iota\rangle$ an identity extension for D. Then $D \otimes_\iota 1^X_X = 1^X_X \otimes_\iota D = D$.*

Proof. Since ι_D is surjective on X, then $D \otimes_\iota 1^X_X = 1^X_X \otimes_\iota D$ by Lemma 14. Further, since ι_D is the identity, the obtained drag is D in which the sprouts have been changed to those of 1^X_X, yielding an isomorphic drag. □

In particular, this property holds when $X = \varnothing$, in which case the obtained drag is exactly D.

We now turn to associativity, for which we need a preliminary definition:

Definition 16 (Compatibility)**.** *Two switchboards ξ and ζ for the respective pairs of drags A, B and B, C are* compatible *if $\mathcal{Dom}(\xi_B) \cap \mathcal{Dom}(\zeta_B) = \varnothing$ and $\mathcal{Im}(\xi_A) \cap \mathcal{Im}(\zeta_C) = \varnothing$.*

Lemma 17 (Associativity)**.** *Let A, B, C be three drags, and ξ, ζ be compatible switchboards for A, B and B, C, respectively. Then, $(A \otimes_\xi B) \otimes_\zeta C = A \otimes_\xi (B \otimes_\zeta C)$.*

Proof. Compatibility ensures that ξ and ζ are switchboards for $A, B \otimes_\zeta C$ and $A \otimes_\xi B, C$, by eliminating the possibility that a sprout or root is used twice. Both sides of the associativity equation are thus defined. The equality itself results from easy checking. □

The assumption that ξ and ζ are compatible switchboards for A, B and B, C is important here. Associativity does *not* apply to any expression of the form $(A \otimes_\xi B) \otimes_\zeta C$, in which $\otimes_\zeta$ has two arguments $A \otimes_\xi B$ and C, instead of B and C as assumed in the statement. Note that ζ can be also denoted by ξ thanks to compatibility.

Horizontal and vertical compositions have many important properties, but one may wonder how they cooperate, in particular whether $\oplus$ distributes over $\otimes$. This is not the case in general, because of sharing, as shown by the following counter-example: consider the expression $s = (f(x) \oplus g(y)) \otimes_{x \mapsto 1, y \mapsto 2} a^2$, where a^2 denotes the term a with two roots. Then, s is a term with two roots, f and g, which share the same subterm a. Distributing $\oplus$ over $\otimes$ yields (intuitively) the expression $t = (f(x) \otimes_{x \mapsto 1} a) \oplus (g(y) \otimes_{y \mapsto 1} a) = f(a) \oplus g(a)$, in which a is no longer shared.

However, as for associativity of composition, they do cooperate in the following important case:

Lemma 18 (Middle four interchange)**.** *Let, A, B, C, D be arbitrary disjoint drags and ξ a switchboard for $(A \oplus C, B \oplus D)$ which decomposes into a pair of switchboards for (A, B) and (C, D) separately. Then, $(A \otimes_\xi B) \oplus (C \otimes_\xi D) = (A \oplus C) \otimes_\xi (B \oplus D)$.*

Proof. Under our assumption, both sides of the equation makes sense. The fact they yield the same drag (up to isomorphism) is then straightforward. $\square$

4. Drag structure

Next, we investigate ways of decomposing a drag, so as to exhibit its dag structure. This important question should have many applications that remain to be explored; we allude to some of them at the end of this paragraph.

We first give a general construction that splits a drag into the subdrag generated by some of its vertices, and the rest of the drag, its *antecedent*. This requires defining a specific kind of extensions that generalizes the one we have defined for identity drags:

Definition 19 (One-Way Switchboard)**.** *A switchboard ξ for D, D' is* one-way *if one of ξ_D and $\xi_{D'}$ has an empty domain. The pair $\langle D', \xi_{D'} \rangle$ is a* context extension *of D if $\mathcal{D}om(\xi_D) = \varnothing$ and a* substitution extension *of D if $\mathcal{D}om(\xi_{D'}) = \varnothing$.*

One-way switchboards correspond to the tree and dag cases, with all connections going from one drag to the other. One-way switchboards are automatically well-behaved. Note that context extensions and substitution extensions switch with each other by symmetry of the definition of a switchboard.

Lemma 20. *Given a drag D and a subset $W \subseteq V$ of its vertices, there is a drag A, called the* antecedent*, and a one-way linear switchboard ζ such that $D_{|W}$ is generated by $\xi(\mathcal{D}om(\xi))$ and $D \simeq A \otimes_\zeta D_{|W}$.*

Furthermore, A is clean if D is, and $D = A \otimes_\zeta D_{|W}$ iff the list $\mathcal{R}(D)$ is obtained by concatenating the list $\mathcal{R}(A)$ with the sublist of $\mathcal{R}(D_{|W})$ containing only those that are also roots of D.

Proof. For internal vertices, A has those vertices of D that do not belong to $D_{|W}$. Its sprouts are the sprouts of D that are not in $D_{|W}$, plus new sprouts that correspond to the new roots of $D_{|W}$, that is, to the vertices of $D_{|W}$ whose antecedents in D are not vertices of $D_{|W}$. These sprouts can be labeled by different variables so as to make A linear on them, hence ensuring unicity of the antecedent up to isomorphism. (We could also ensure unicity by labeling any two of these sprouts by the same variable iff they generate equal drags – or, perhaps, isomorphic drags – in D.) The roots of A are those of D that are not vertices of $D_{|W}$, plus the new sprouts that become roots with the appropriate order of multiplicity (the corresponding number of outgoing edges to $D_{|W}$ in D). The switchboard ξ mapping these sprouts to the corresponding roots of $D_{|W}$ is linear as a consequence of the labeling, one-way since subdrags are closed under successor, and total on the new roots which are maximal, hence generate $D_{|W}$. Note that the rooted-sprouts of A disappear in the composition. The fact that A is clean when W is clean follows from the construction. Likewise, there is no need of permuting roots of $A \otimes_\zeta D_{|W}$ to obtain D if all roots of A come first in D. $\square$

The fact that the switchboard ξ is one-way expresses the property that decomposing a drag into a subdrag and its antecedent does not break any of its cycles – by the definition of subdrag. Note also that ξ induces an order on the roots of the subdrag that are not roots of the whole drag.

A tree headed by the symbol f of arity n has one root labeled f, or equivalently a head corresponding to the expression $f(x_1, \ldots, x_n)$, plus several subtrees, seen here as a single drag with many roots. This unique decomposition is a fundamental property of trees that expresses the fact that the set of trees equipped with "head-operations" has an initial algebra structure.

Likewise, a drag has a head, which is intended to be the largest cycle containing the maximal roots of the drag – also the smallest nontrivial antecedent of the drag – and one tail, which is a list of several connected components. The head cycle may consist of a lone (root) vertex, in particular in the case of trees. As for trees, the head will have a list of new sprouts that are in one-to-one correspondence with the roots of the subdrag. A drag can therefore be seen as a bipartite graph made of its head, its tail, and a one-way switchboard specifying that correspondence.

Definition 21 (Tail)**.** *The* tail *∇D of a drag $D = \langle V, R, L, X, S \rangle$ is the subdrag generated by the set of vertices $V \setminus \{v \in V : vX^*\mathcal{R}^\bullet(D)\}$.*

We now move to heads, which belong to a very specific class of drags that we define first:

Definition 22. *A drag is* recycling *if every internal vertex can access some maximal root.*

The idea behind this notion of a recycling drag is that it is always possible, from any internal vertex, to go back to a maximal root (unless we are already there). Therefore, a recycling drag does not necessarily have a single cycle going through its vertices. In particular, a lone internal vertex that is a root is a recycling drag. Recycling drags constitute an essential ingredient here, one which helps us view a tree as a special case of a drag. For example, the ground (variable-free) term a and the terms $f(x)$, $g(x, f(\mathit{self}))$ – where *self* designates a back-arrow to the root of the expression – and $g(f(\mathit{self}), f(\mathit{self}))$ are all recycling drags, whereas the terms $f(a)$, $f(f(x))$ and $g(a, f(\mathit{self}))$ are not.

As an application of Lemma 20, we get:

Lemma 23 (Head)**.** *Given a drag D, there exists a recycling drag $\widehat{D}$, called the* head *of D, and a one-way linear switchboard ξ, such that $D \simeq \widehat{D} \otimes_\xi \nabla D$ (with equality iff all nonmaximal roots come after the maximal ones in $\mathcal{R}(D)$).*

The head of a drag is therefore the antecedent of its tail. By Lemma 20, it contains all maximal roots of D. Note further that its sprouts that were not already sprouts of D are linear and that ξ is total on those sprouts. More precisely, ξ is a bijection between those sprouts and the roots of the tail that were not already roots of the drag D. In the sequel, we assume that the sprouts of the head are ordered with respect to a depth first search initialized with its list of roots. This order on the sprouts of the head induces via the switchboard ξ, an order on the roots of the associated tail that were not roots of the original drag. The tail is therefore now canonically determined.

We could have defined the head and tail of D as a pair of drags whose head contains all maximal roots, is recycling, and its composition with the tail by a one-way linear switchboard is the drag D itself. This characterization of the head of a drag is a property that we should keep in mind.

We can now show the structure theorem of drags:

Theorem 24 (Structure)**.** *Isomorphic drags have isomorphic heads and tails.*

Proof. By Lemma 5, we restrict our attention to clean drags. Let D, D' be two clean drags such that $D =_o D'$. By definition of drag isomorphism, $o(\mathcal{R}(D)) = \mathcal{R}(D')$.

First, we claim that a root $r \in \mathcal{R}(D)$ is maximal in D iff the root $o(r)$ is maximal in D'. Let $o(r)$ be a maximal root in D', and let us assume that there exists $r' \in \mathcal{R}(D)$ such that there is a path in D from r' to r. By definition of drag isomorphism, paths are of course transported by o; hence there exists a path in D' from $o(r')$ to $o(r)$. By maximality of $o(r)$, there must exist a path in D' from $o(r)$ to $o(r')$. Transporting this path back to D yields a path from r to r', showing that r is maximal. The inverse direction is by symmetry. There are now two cases in the proof:

1. Assume first that the list $\mathcal{R}(D)$ is made of the list of its maximal roots followed by the list of its nonmaximal ones. By the previous claim this is true of D'. Therefore, by Lemma 23, $D = H \otimes_\xi T$ and $D' = H' \otimes_{\xi'} T'$, where H and H' are the respective heads of D and D', T and T' their respective tails, ξ_T and $\xi'_{T'}$ are empty, ξ_H and $\xi'_{H'}$ are total and surjective. It follows that $H \otimes_\xi T =_o H' \otimes_{\xi'} T'$.
   We can now build new bijections o_h and o_t that witness the isomorphisms between heads and tails, respectively. Since ξ_T is empty and ξ_H is surjective, the roots of $H \otimes_\xi T$ are those of H. Likewise, the roots of $H' \otimes_{\xi'} T'$ are those of H'. Hence o restricts to a one-to-one-mapping from the roots of H to those of H'. By property (2) of isomorphisms, o restricts to a one-to-one mapping from the vertices of $H \otimes_\xi T$ that are vertices of H to the vertices of $H' \otimes_{\xi'} T'$ that are vertices of H', and satisfies (2). To extend o_h to the sprouts of H and H', it suffices to notice that they are equally many by (2). Since o satisfies (2), so does o_h, hence $H =_{o_h} H'$.
   Next, we define o_t by removing o_h from o. The roots of T and T' correspond to the sprouts of H, H'. Since the roots of T and T' are in one-to-one correspondence with the sprouts of H and H', respectively, o_t restricts to the roots of T, T'. Also, since ξ_H and $\xi'_{H'}$ are surjective, the sprouts of T and T' are those of $H \otimes_\xi T$ and $H' \otimes_\xi T'$, respectively. Hence, o_t restricts to the sprouts of T, T'. Since o satisfies (2), so does o_t, hence $T =_{o_t} T'$.
2. The other case, where the maximal roots do not all come first, reduces to the previous one by considering two drags A and B, where A (B, respectively) is obtained from D (D') by permuting the roots thanks to a permutation of $[1..|\mathcal{R}(D)| = |\mathcal{R}(D')|]$, so that the maximal roots of A (B) come first. Composing the permutation with o yields a witness that A and B are isomorphic. By the previous case, the conclusion holds for A, B. It is then easy to transport the result to D, D', by composing o_h with the inverse permutation, o_t remaining the same. □

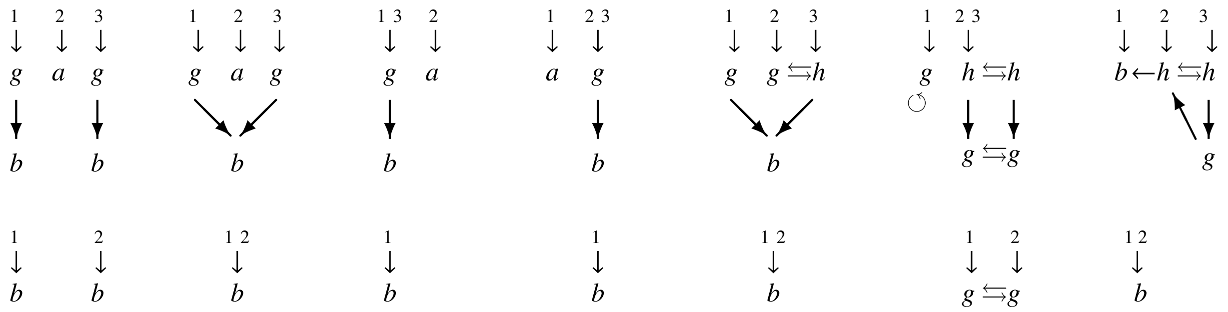

Figure 3: Successive subdrags of the drags of Figure 1

□

Example 25. *The successive tails of the drags of Figure 1 are displayed in Figure 3. The numbering displayed above the roots of each tail specifies the order of roots in their list of roots. These tails, represented on the first row, have all the same number of roots, which is determined by their head f. The next row represents the next level of tails, which here have one or two roots, depending on their own head. The last tail in that row has two roots, a nonmaximal one in its father drag, and a new one.*

The fundamental property of a drag implied by Theorem 24 is that its decomposition into head and tail faithfully represents it. This property holds true because drags are multirooted. Uni-rooted drags cannot represent horizontal sharing faithfully, because several roots may pop-up at the same time when taking tails. This contrasts with vertical sharing, which can *sometimes* be preserved with uni-rooted drags. Consider for example, the second drag (2) of Figure 1. Its tail, shown in Figure 3, has two roots, which cannot be decomposed into two different subdrags having a single root each.

Given a drag D, we can therefore partition its set of vertices into those belonging to a connected component of the vertices belonging to its head, and, recursively, the partition associated with its tail. The successor function of D can then be seen as a successor function between the elements of the partition, hence defining a multigraph which is acyclic, hence is a dag (actually, a multi-rooted multi-dag) which is a *dag decomposition* of D [4]:

Definition 26. *Let C be all maximal strongly connected components of drag D. The* dag decomposition *of D has vertices C and as edges those that cross components. The head $H(D)$ of the drag comprises the components containing the maximal roots. The tail $T(D)$ is the rest. The decomposition is $\{H(T^i(D))\}$.*

It is argued in [4] that dag decompositions are extremely useful to speed up graph algorithms, in particular those that require searching the input graph. So, this property is not only useful for our understanding of a drag's structure, but also for developing good drag algorithms. One such application, currently under exploration, is to drag matching and unification, since both require some search. In the case of trees, matching relies on first searching for the head of a left-hand side L of rule in a given tree s. In the case of arbitrary graphs, there is no head to search for. In the case of drags, the dag structure of L provides the head to search for, and the dag decomposition of s provides the way to organize the search.

Indeed many algorithms developed for trees or dags should be reusable by applying them to the drag structure of the drags given as inputs. We use this idea in [16], where two drags are compared with the recursive path order (RPO) initially developed for trees [13]. In this case, all we needed to do is to define an appropriate order on drag heads and then use the recursive structure of RPO applied to the dag decomposition of the graphs to be compared.

5. Drag rewriting

Rewriting with drags is very similar to rewriting with trees: we first select an instance of the left-hand side L of a rule in a drag D – this is drag matching, then replace it by the corresponding right-hand side R. Selecting a left-hand side L in a drag D amounts to expressing the drag D as a composition of some drag W with L via a switchboard ξ. Replacement amounts to computing the composition of W with the right-hand side R via the same switchboard ξ. A very important condition for the result to be a drag is, accordingly, that the left- and right-hand sides of rules have the same number of roots and the same variables in equal number for each. Our definition will therefore always avoid the creation of ill-formed drags, in particular, of dangling edges.

Definition 27 (Pattern)**.** *A pattern is a drag all of whose internal vertices are accessible.*

In other words, a pattern is a drag that becomes clean when removing its inaccessible sprouts.

Definition 28 (Rules)**.** *A* drag rewrite rule *is a pair of patterns written* $L \to R$ *such that (i)* $|\mathcal{R}(L)| = |\mathcal{R}(R)|$, *(ii)* $\mathcal{V}ar(L) = \mathcal{V}ar(L^\sharp)$, *and (iii)* $\mathcal{M}var(L) = \mathcal{M}var(R)$. *A* graph rewrite system *is a set of drag rewrite rules.*

Conditions (iii) and (i) ensure that L and R have a perfect fit with any (same) environment – that is, can both be composed with any extension of L, and condition (ii), that there are no useless variables in the rule. This is needed since patterns may have arbitrarily many inaccessible sprouts. All together, these conditions imply that $\mathcal{V}ar(R^\sharp) \subseteq \mathcal{V}ar(L^\sharp)$.

Trees and drags are particular uni-rooted drags. Because they have a single root, term-rewrite rules and dag-rewrite rules satisfy the first condition. They satisfy the second trivially. The third is not needed for trees or dags, which are acyclic graphs.

Rewriting drags uses a specific kind of switchboard, which allows one to "encompass" a drag U within drag D, so that all roots and sprouts of U disappear from the composition:

Definition 29 (Rewriting Switchboard)**.** *A switchboard ξ for W, U is a* rewriting *switchboard if ξ_W is linear and surjective and ξ_U is total. The pair $\langle W, \xi \rangle$ is called a* rewriting extension *of U.*

When $D = W \otimes_\xi U$, we say that D *matches* U, W and ξ being the *matching* context and switchboard. Matching is therefore the operation which, given D and U, computes a drag W and switchboard ξ such that $D = W \otimes_\xi U$.

We are now ready to define rewriting:

Definition 30 (Rewriting)**.** *Let $\mathcal{R}$ be a graph rewrite system. We say that a nonempty clean drag D* rewrites *to a clean drag D', and write $D \longrightarrow_{\mathcal{R}} D'$, iff $D = W \otimes_\xi L$ and $D' = (W \otimes_\xi R)^\sharp$ for some drag rewrite rule $L \to R \in \mathcal{R}$ and clean rewriting extension $\langle W, \xi \rangle$ of L.*

All assumptions on ξ play a rôle.

First, because ξ is a rewriting switchboard, ξ_W must be linear. This implies that the variables labeling the sprouts of W that are not already sprouts of D must all be different. Second, ξ_C must be surjective, implying that the roots of L (hence those of R by condition (i)) disappear in the composition. Third, ξ_L must be total, implying that the sprouts of L (hence those of R by condition (iii)) disappear in the composition. Fourth, D must be nonempty, implying that the roots of W do not all disappear in the composition; hence ξ_L could not be surjective.

Note finally that we use the condition $D = W \otimes_\xi L$, and state that W must be clean. This may seem contradicting the fact that L has unreachable sprouts, but it is not, since no variable of L can label unreachable sprouts only. This property is not true of the right-hand side of the rule, hence garbage collection may be needed to clean $W \otimes_\xi R$. On the other hand, duplication is banned by the drag rewriting model, which therefore generalizes shared rewriting, but not term rewriting per se. We shall come back to this issue in the conclusion.

Example 31. *In the three examples of Figure 2, rewrite rules are pictured in red, while contexts are in blue. Rewriting can be visualized by using the same red and blue colours in the input drag and the resulting one.*

In the first two examples, the term to be rewritten is the same cyclic drag.

Example (shrinking) uses the rewrite rule $f(y') \to y'$, whose left-hand and right-hand sides are the drags $\langle \{f_1, y_2\}, f_1, \{f_1 \mapsto_L f, y_2 \mapsto_L y'\}, f_1 \mapsto_X y_2, y_2 \rangle$ and $\langle f_1, f_1, f_1 \mapsto_L y', \varnothing, f_1 \rangle$, respectively. The drag $D = \langle \{f_1, f_2\}, f_1, \{f_1 \mapsto_L f, f_2 \mapsto_L f\}, \{f_1 \mapsto_X f_2, f_2 \mapsto_X f_2\} \rangle$ rewrites to the drag $D' = \langle \{f_1, x_2\}, f_1, \{f_1 \mapsto_L f, x_2 \mapsto_L x'\}, f_1 \mapsto_X x_2, x_2 \rangle \otimes_{\{x' \mapsto f_1, y' \mapsto 1\}} \langle y_1, y_1, y_1 \mapsto_L y', \varnothing, y_1 \rangle = \langle f_1, f_1, f_1 \mapsto_L f, f_1 \mapsto_X f_1 \rangle$. A cycle of length 2 has been rewritten into one of length 1.

Example (breaking) rewrites the same drag D to the drag $D'' = \langle \{f_1, x_2\}, f_1, \{f_1 \mapsto_L f, x_2 \mapsto_L x'\}, f_1 \mapsto_X x', x_2 \rangle \otimes_{x' \mapsto 1} \langle a_1, a_1, a_1 \mapsto_L a, \varnothing \rangle = \langle \{f_1, a_2\}, f_1, \{f_1 \mapsto_L f, a_2 \mapsto_L a\}, f_1 \mapsto_X a_2 \rangle$ with the rule $f(y') \mapsto a$, which

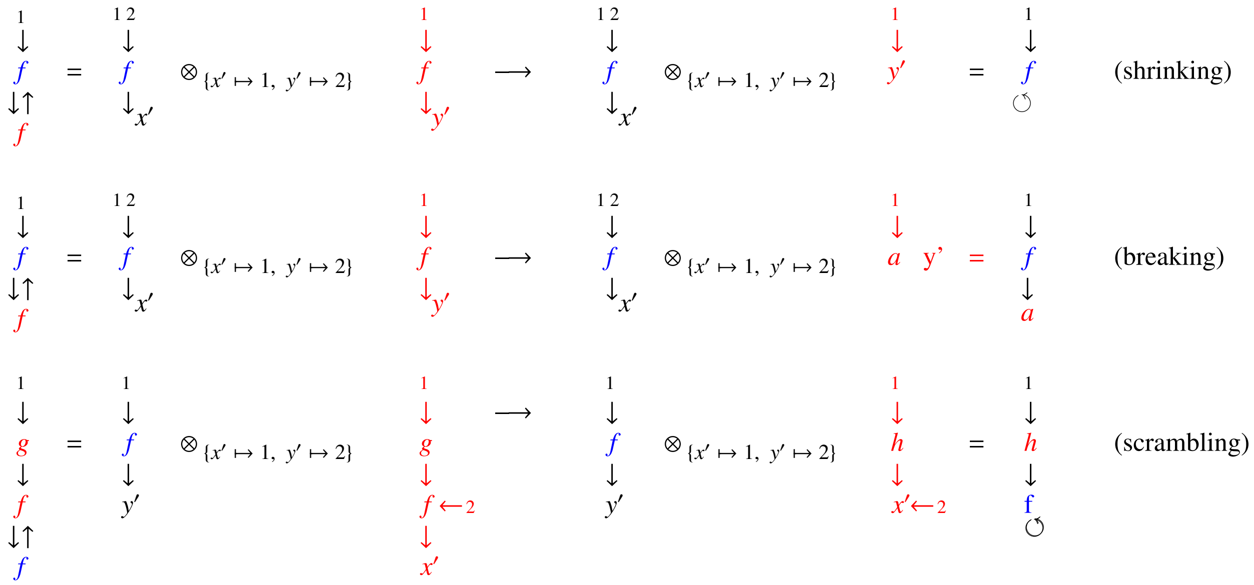

Figure 4: Rewriting and cycles.

is written $f(y') \to a \quad y'$, where vertices f and a only are roots, in order to satisfy the conditions for being a drag rewrite rule. Note the essential rôle of the unreachable sprout y' in the right-hand side of the rule, which will connect to the root number 2 of the context and make it disappear.

Example (scrambling) uses the rewrite rule $g(f(x')) \to h(x')$ (whose roots are g and f on the left-hand side and h and y' on the right-hand side), which applies across the cycle of the term to be rewritten, here a vertex labelled g followed by the same cycle as before. We can see that the g has been changed to an h, while the cycle has shrunk as in (shrinking).

Lemma 32. *If $U \longrightarrow_{\mathcal{R}} U'$, then $\mathcal{V}ar(U') \subseteq \mathcal{V}ar(U)$ and $|\mathcal{R}(U)| = |\mathcal{R}(U')|$.*

Proof. Both properties are true of rewrite rules and preserved by composition with rewriting switchboards as a consequence of Lemma 13. □

Note that it is important here to allow non-clean drags in rewrite rules. If not, then rules could hardly preserve the multiset of their variables, and then, the number of roots could not be an invariant of drag rewriting.

Lemma 33. *Assume U, U', W are three drags such that U rewrites to U', and $\xi = (\xi_W, \xi_U)$ is a switchboard for W, U. Then, $\xi' = (\xi_W, \xi_{U'})$ – also denoted ξ, where $\xi_{U'}$ is the restriction of ξ_U to $\mathcal{V}ar(U')$, is a switchboard for W, U'.*

Proof. Because the lists of roots of U and U' have the same length, and because the sprouts of U, U' are labeled by the same variables that are here identified with the sprouts themselves, ξ_U requires no change. □

6. Drag extensions

The notion of composition lead in Section 4 to a canonical way of decomposing a drag, into its head and tail. In this section, we investigate further ways of decomposing a drag, possibly breaking its cycles, which is of course important in the context of drag rewriting. More precisely, we investigate now the structure of the rewriting extensions of a given drag U.

The idea is that the rewriting context W splits into three parts: the vertices of W that are accessible from some sprout of U and can access some of its roots define a drag B (such a drag B cannot exist in the case of trees, unless it is an identity); those that are accessible from some sprout of U but cannot access any of its roots define a drag C that

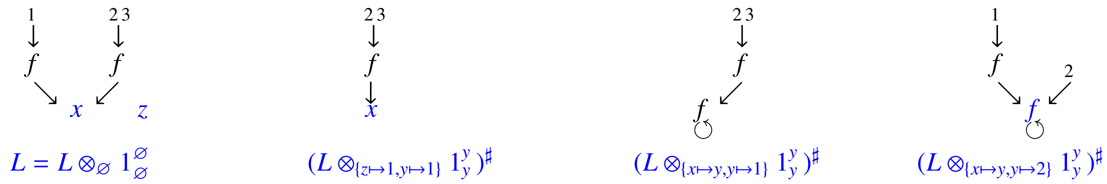

Figure 5: Identity cyclic extensions.

is therefore (approximately) a substitution extension of U; those that are not accessible from the sprouts of U form the remaining part A, which is therefore (approximately) a context extension of U.

A particular context extension of U is strict identity extension, which, by Lemma 15, leaves U unchanged.

Particular substitution extensions of U are *identity sharing extensions* $\langle 1^z_z, \xi \rangle$ such that $\xi(x_1) = \cdots = \xi(x_n) = z$, where $x_1, \ldots, x_n$ are variables labeling sprouts of U, and, of course, $z \notin \mathcal{D}om(\xi)$. Then, the drag resulting from the composition of U with that extension is the same as U, except that all its sprouts labeled by the variables $x_1, \ldots, x_n$ are now merged into a single sprout labeled z. The rôle of these extensions is to modify the structure of U by introducing sharing among its sprouts. Identity sharing extensions are complete, in the sense that any sharing that is introduced by a substitution extension can be achieved by an identity sharing extension.

We are left with the task of defining the kind of extension that B belongs to:

Definition 34 (Cyclic Extension)**.** *An extension $\langle B, \xi \rangle$ of a pattern U is* cyclic *if B is generated by $\mathcal{I}m(\xi_U)$, $(B \otimes_\xi U)^\sharp$ is non-empty, and for all $t \in \mathcal{I}nt(B)$, there exists $s \in \mathcal{D}om(\xi_B)$ such that $t X^*_B\, s$. The extension is said to be* clean *if so is its* result *$B \otimes_\xi U$.*

The conditions for being a cyclic extension impose that ξ_U is onto $\mathcal{R}^\bullet(B)$ so as to generate B. The roots of the resulting drag are therefore originating from either B or U (possibly via a transfer), but there must be sufficiently many of them so that $(B \otimes_\xi U)^\sharp$ is nonempty.

Identity cyclic extensions of U are of the form $\langle 1^Z_Y, \iota \rangle$, where the variables in $Z \subseteq Y$ are one-to-one with those in $\mathcal{D}om(\iota_U) \subseteq \mathcal{V}ar(U)$, and $\iota_1 : Y \to [1 \,..\, |R|]$ is an arbitrary map. The rôle of cyclic extensions is to modify the structure of U by connecting some of its sprouts to some of its roots. Unfortunately, identity cyclic extensions do not suffice for that purpose, as identity-sharing extensions do for sharing, unless in special cases.

Example 35. *Let L be the drag made of two copies of the tree $f(x)$ sharing the variable x. Four identity extensions of L are represented in Figure 5. The first is the trivial one-way extension. The second eats the first root of L, which disappears by cleaning. The last two yield clean drags. The third maps the variable x of L to the only root of the identity drag, which is its sprout y, and the only sprout y of the identity drag to the first root of L. The fourth instead maps y to the second root of L, giving a slightly different result from the third.*

Notice that the result of a cyclic extension is not necessarily a recycling drag.

Identity cyclic extensions allow one to change the structure of a drag by adding new edges without changing its internal nodes (some may be deleted, though). If the drag has a single root, it is easy to see that identity extensions are enough to predict all shapes that a drag may take under composition with a cyclic extension. This is no longer true with multirooted drags, since identity cyclic extensions cannot reach two different roots from the same sprout. They cannot therefore suffice unless no cyclic extension can reach two different roots either, in other words, when the vocabulary contains only constants and unary symbols.

The following property is at the heart of drag rewriting:

Lemma 36 (Decomposition)**.** *Let U be a pattern and $\langle W, \xi \rangle$ be a clean extension of U such that $W \otimes_\xi U$ is clean and $\mathcal{R}^\bullet(U) \subseteq \mathcal{I}m(\xi_W)$. Then, there exist drags A, B, C and switchboards ζ, θ such that*

1. *$W \otimes_\xi U \simeq A \otimes_\zeta ((U \otimes_\xi B) \otimes_\theta C)$;*
2. *$\langle A, \zeta \rangle$ is a clean context extension of $(U \otimes_\xi B) \otimes_\theta C$;*
3. *$\langle B, \langle \xi_B, \xi_{U \to B} \rangle \rangle$ is a clean cyclic extension of U denoted $\langle B, \xi \rangle$;*
4. *$\langle C, \theta \rangle$ is a clean substitution extension of $U \otimes_\xi B$;*

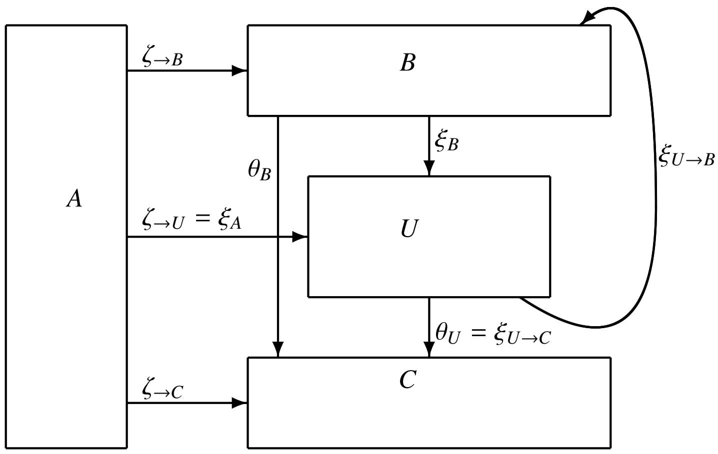

Figure 6: Decomposition of a composition $(W \otimes_\xi U)^\sharp$ for which ξ_W is onto $\mathcal{R}^\bullet(U)$.

Furthermore, C is empty if all internal nodes of W reach one of its sprouts, and $(B \otimes_\xi U)^\sharp$ is recycling if C is empty and ξ_B is onto $\mathcal{R}^\bullet(U)$.

Proof. The lemma is depicted in Figure 6.

Let first W' be the subdrag of W generated by the vertices of W which are accessible from $\mathcal{R}^\bullet(U)$ in $W \otimes_\xi U$. By its definition, W' is clean. By Lemma 20, $W \simeq A \otimes_\zeta W'$, where the domain of the one-way switchboard ζ is the set of those sprouts of A which are not sprouts of W. Therefore, $W \otimes_\xi U \simeq (A \otimes_\zeta W') \otimes_\xi U$. Since ζ and ξ are compatible switchboards, $W \otimes_\xi U \simeq A \otimes_\zeta (W' \otimes_\xi U)$ by Lemma 17. Note further that A is clean since W is clean.

Consider now the subdrag C of W' that is generated by those vertices from which no root of U is accessible. By its definition, C is clean. By Lemma 20, $W' \simeq C \otimes_\theta B$, where the domain of the one-way switchboard θ is the set of those sprouts of B that were not sprouts of W'. Therefore, $W' \otimes_\xi U \simeq (C \otimes_\theta B) \otimes_\xi U$. Since ξ and θ are compatible switchboards, $W' \otimes_\xi U \simeq C \otimes_\theta (B \otimes_\xi U)$. Note further that B is clean since W' is clean.

Putting things together, we get $W \otimes_\xi U \simeq A \otimes_\zeta (C \otimes_\theta (B \otimes_\xi U)) \simeq A \otimes_\zeta ((B \otimes_\xi U) \otimes_\theta C)$ by using Lemma 14.

We are left showing that (B, ξ) is a clean cyclic extension of U. We already know that B is clean. Let now $t \in \mathcal{I}nt(B)$. Then, t must access a sprout of B in the domain of ξ_B, since otherwise, t would belong to C. Hence (B, ξ) is a cyclic extension of U.

Conditions 5 and 6 follow easily. □

A consequence of Lemma 33 is that properties of extensions are preserved by rewriting. Further, the decomposition lemma (Lemma 36) itself is also preserved when left-hand and right-hand sides of rules have the same sets of accessible variables, but must be slightly modified otherwise:

Lemma 37. *Let $U \longrightarrow_{L\to R} U'$. Then, $U = A \otimes_\zeta ((B \otimes_\xi L) \otimes_\theta C)$ and $U' = A \otimes_\zeta ((B \otimes_\xi R) \otimes_\theta C)$ for some A, B, C and ζ, ξ, θ such that*

1. *$\langle C, \theta\rangle$ is a substitution extension for both $B \otimes_\xi L$ and $B \otimes_\xi R$;*
2. *$\langle A, \zeta\rangle$ is a context extension for both $(B \otimes_\xi L) \otimes_\theta C$ and $(B \otimes_\xi R) \otimes_\theta C$;*
3. *$\langle B, \xi\rangle$ is a cyclic extension for L, and for R as well if $\mathcal{V}ar(R) = \mathcal{V}ar(L)$. In the other case, $\mathcal{V}ar(R) \subsetneq \mathcal{V}ar(L)$, $B = A' \otimes_\delta B'$ for some drags A', B' such that $\langle B', \xi\rangle$ is a cyclic extension of R and $\langle A', \delta\rangle$ is a context extension of $B' \otimes_\xi R$.*

Proof. By definition of rewriting, $U = W \otimes_\xi L$ and $U' = (W \otimes_\xi R)^\sharp$, where $\langle W, \xi\rangle$ is a rewriting extension of L. By the definition of a rewriting extension, Lemma 36 applies; hence, $W \otimes_\xi L = A \otimes_\zeta (B \otimes_\xi L) \otimes_\theta C$. By the definition of rewriting, $B \otimes_\xi L \longrightarrow B \otimes_\xi R$, hence $\mathcal{V}ar(B \otimes_\xi R) \subseteq \mathcal{V}ar(B \otimes_\xi L)$ and $|\mathcal{R}(B \otimes_\xi R)| = |\mathcal{R}(B \otimes_\xi L)|$ by Lemma 32. It follows that θ is a switchboard for $C, B \otimes_\xi R$.

$$\underset{\circlearrowleft}{\overset{1}{\underset{f}{\downarrow}}} \;=\; \overset{1\,2}{\underset{x'}{\downarrow}} \;\otimes_{\{x'\mapsto 1,\ y'\mapsto 2\}}\; \underset{\downarrow y'}{\overset{1}{\underset{f}{\downarrow}}} \;\longrightarrow\; \overset{1\,2}{\underset{x'}{\downarrow}} \;\otimes_{\{x'\mapsto 1,\ y'\mapsto 2\}}\; \overset{1}{\underset{y'}{\downarrow}} \;=\; \overset{1}{\underset{x'}{\downarrow}}$$

Figure 7: Rewriting with an ill-behaved switchboard.

Claim (1) follows. The proof of (2) is similar. We are left with (3).

By Lemma 33, $\langle B, \xi \rangle$ is an extension of R, and if $\mathcal{V}ar(R) = \mathcal{V}ar(L)$, it must be a cyclic extension of R because L and R have the same number of roots, hence the images of ξ_L and ξ_R define the same roots of B which implies that $\langle B, \xi \rangle$ is a cyclic extension of R. In case $\mathcal{V}ar(R) \subsetneq \mathcal{V}ar(L)$, we use the same reasoning as in Lemma 36 to define A', B', and δ. □

The key notion of this section is decomposition of drags. Singling out a subset of its vertices – be it by matching the left-hand side of a rewrite rule or by other means – yields a subdrag that imposes a tripartitie structure on the remaining vertices (context, cyclic extension, substitution), divides the edges connecting them into several switchboards linking their interfaces. Furthermore, the essence of that structure is preserved by rewriting.

7. Variations on the drag model

The reader should by now have a clear idea of our drag model and its capabilities. It is therefore time to discuss the model, its potential weaknesses and strengths.

Its main strength is that it appears as a nice generalization of basic rewriting notions, with rules having variables and roots that allow one to hook them inside a context via what we call a switchboard. The switchboard appears therefore as a central notion of the theory of drags.

A switchboard takes two drags and gives a way to compose them by redirecting the edges ending up in (some of) the sprouts of both drags to the vertices of the other drag. The redirected sprouts therefore disappear. If a sprout of one drag is redirected to a rooted-sprout of the other drag which is itself redirected, the redirection process must then continue until an internal vertex is found, or a sprout which is not redirected. This is the essence of Definition 9, the *target* calculation. The fact that the target calculation eventually terminates is ensured by condition (3) of Definition 7: switchboards must be well-behaved.

We could think of allowing ill-behaved switchboards, but then, the above redirection process ends up redirecting a sprout to a rooted-sprout, to a rooted-sprout, without ending up as previously. We would then need to define which rooted-sprout is considered to be the target of the process. This has to be one of these rooted-sprouts, of course, and the one chosen can only be a mater of convention. It turns out however that any convention leads to more ill behavior: were we to allow ill behaved switchboards, then the example displayed in Figure 7 shows that rewriting may generate fresh variables (taken from the rules themselves). In this example, the drag s made of a loop on the rooted vertex f is rewritten with the rule $f(y') \to y'$. To this end, s is written as indicated, by composing the drag x' (which has two roots) with the left-hand side of the rule. Since we need the ill-behaved switchboard $\{x' \mapsto 1, y' \mapsto 2\}$, we need to choose which one among x', y' is the target. Let's choose x', the other choice can be carried our by the reader. Then, composition of x' with y' yields the drag reduced to the target x'. So, the fresh variable x' introduced by the matching process has become the result of the rewrite. This is of course unacceptable; hence condition (3) of Definition 7 cannot be weakened.

We now consider the other two conditions of a switchboard, namely (1) and (2) in Definition 7.

Slightly more liberal conditions could have been chosen for defining a switchboard, by replacing vertex equality $\xi_D(s) = \xi_D(t)$ in (1) and $\xi_{D'}(s) = \xi_{D'}(t)$ in (2) by drag isomorphism: $R_{|\xi_{D'}(s)} \simeq R_{|\xi_{D'}(t)}$ in in (1) and $R'_{|\xi_D(s)} \simeq R'_{|\xi_D(t)}$ in in (2). These two conditions can even be weakened by using quasi-isomorphism instead of strict isomorphism. These alternative choices are possible, the theory carries out with cosmetic changes only. They have one major practical drawback, though: checking that a particular ξ is a switchboard would involve drag isomorphism, which is expensive although polynomial in the case of strict isomorphism, since we have ordained fixed arities. (Were one to allow variadic vertices, strict isomorphism would be quasi-polynomial [3].) Quasi-isomorphism, on the other hand, is non-polynomial. Yet they should increase the expressivity of the drag model, this is briefly discussed in conclusion.

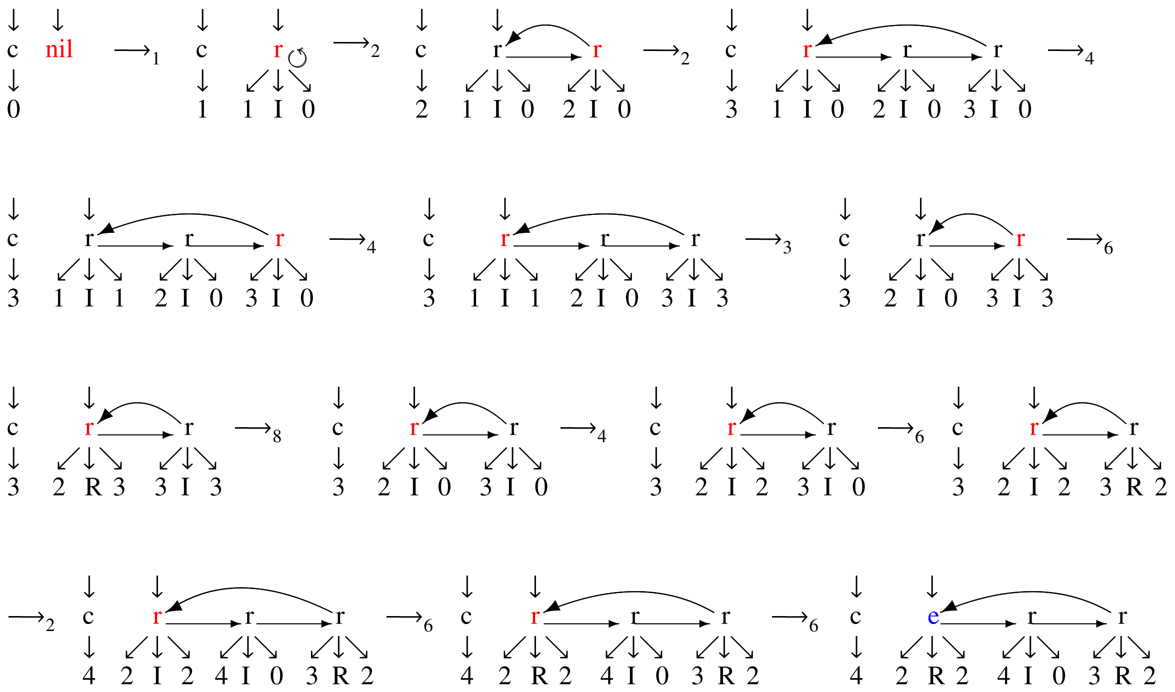

Figure 8: Election of a leader. The rule number for each step is noted. At each step, the second root of the rule applies to the vertex of the ring indicated in red.

Another possible variant that we tried to develop unsuccessfully so far, is the possibility to have a set of roots, each root being available for connection as many times as needed. In this model, switchboards would not be injective, and roots would never be exhausted. Such a model looks simpler, but would not have the same practical relevance for concurrency as the one chosen here since it would imply that resources are unlimited. An intermediate variant would be to have a multiset of roots. This won't be a good model for programs, however, since it would not allow us to observe sharing.

Lastly, our framework is mostly based on clean drags. This shows up first in our definition of isomorphic drags. Likewise, rewriting operates on clean drags too. However, as we have observed, rewrite rules can't be clean in general. Restricting rewriting to clean drags has one important consequence: non-clean cyclic extensions don't show up in Lemma 36; otherwise, they would, and the resulting rewrite relation would be slightly different, even on clean drags.

8. Example: leader election in the drag model

In order to anticipate on the comparison of the drag model with other existing frameworks for graph rewriting, we specify now a variant of the distributed leader election algorithm of Chang and Roberts [9]. For its precise specification together with a specification using the double pushout approach, we refer to [35].

Example 38 (Leader Election)**.** *First, the ring itself and its management: There are two connected components, each one being a rooted drag whose root is the name of the data structure used for implementing the protocol: one for the ring of processes, and one for the counter. We will use the symbol* self *to indicate an edge going back to the root of the data structure (here, the ring). Beware that self is a convention, not a constant of the language. Its use allows us to write the leader election rules as algebraic expressions standing for drags whose root, another convention, is associated to the outermost symbol of the expression.*

The vocabulary uses four sorts (with the usual algebraic meaning): natural numbers generated by 0 *and s for representing process IDs, flag, ring, and counter. Messages are process IDs, hence are also represented by natural numbers. Natural numbers are assumed, as well as flags which can take two values: I for idle and R for ready.*

Function symbols obey the following sort declarations:

$$\begin{array}{rcrcl} nil & : & & \to & ring \\ r & : & nat \times bool \times nat \times ring & \to & ring \\ e & : & nat \times bool \times nat \times ring & \to & ring \\ c & : & nat & \to & counter \end{array}$$

The arguments in $r(x, f, m, z)$ (or in $e(x, f, m, z)$) have the following meaning: x is the process ID; f is the value of the flag; m is a process ID circulating in the ring to be checked for minimality; z is (a pointer to) the rest of the ring. The use of the constructor e indicates that the process x has been elected.

Initially, the ring is empty, hence equal to nil and the shared counter holds the initial value 0*.*

Each process can communicate its ID to the next process in the ring, and messages can then be passed along the ring past processes whose ID is larger. Otherwise, the message may be deleted. When and if a message returns to its sender, she becomes the leader. Any process can quit the ring at any time, unless she has been elected leader. Note that all arguments of a process are private.

Although we have not defined conditional rules, the following rules use conditions, which should be interpreted in the very same way as conditional term-rewriting rules normally are. (Or they may be viewed as a schema defining its instances. Or the conditions could be eliminated at the price of adding additional rules to implement the comparison between natural numbers.)

$$\begin{array}{llrcll}
1.\ \text{start} & : & c(0) \quad nil & \to & c(s(0)) \quad r(s(x), I, 0, self) & \\
2.\ \text{enter} & : & c(y) \quad r(x, b, m, z) & \to & c(s(y)) \quad r(x, b, m, r(s(y), I, 0, z)) & \\
3.\ \text{quit} & : & r(x, f, m, z) & \to & z & \textbf{if } f \neq E \\
4.\ \text{send} & : & r(x, I, m, z) & \to & r(x, I, x, z) & \\
5.\ \text{pass} & : & r(x, f, m, self) & \to & r(x, R, m, self) & \%\textit{passes the message to herself} \\
6.\ \text{pass} & : & r(x, f, m, r(y, f', n, z)) & \to & r(x, f, m, r(y, R, m, z)) & \textbf{if } x \geq m \\
7.\ \text{elect} & : & r(x, R, x, z) & \to & e(x, R, x, z) & \\
8.\ \text{ignore} & : & r(x, f, m, z) & \to & r(x, I, 0, z) &
\end{array}$$

Note that the application of a rule does not leak any unnecessary information to the processes in the ring about the state of the other processes that are in the ring. In particular, an elected process blocks the transmission of messages, but nobody knows who is blocking except the elected process.

The rules are nonterminating since processes can enter the ring without restriction, even if a leader has been elected already. All processes, but an already elected leader, can also quit. As a result, the ring may become empty, and then grow again. An example run is represented in Figure 8.

This example demonstrates that the drag model is quite easy to use, thanks to the presence of variables. It is quite expressive too, although this example uses only a little bit of its capabilities. The use of self allows us to write drag expressions as if they were trees. We expand on this idea in the next section.

9. Drag expressions

It is indeed easy to give a general notation for drags in the style of Ariola and Klop's notation for λ-graphs [1] by allowing for multi-rooted expressions. The main idea is to view the vertices of a drag as variable names and an incoming edge to a vertex v labeled f as an equation of the form $v = f(\overline{u})$ when v has outgoing edges to the list of vertices $\overline{u}$. The whole drag becomes therefore a set of such equations, in which each internal vertex v appears exactly once on the left of an equation. Note that these equations correspond to Prolog substitutions with occurs check. Then, a scoping operator allows us to give structure to the whole set of equations; more precisely, it allows one to reveal the dag structure of the drag by moving the scoping operator down to the leaves of the drag as long as it is allowed by the scoping rules.

There is a vast literature around this idea, where the binding operator is sometimes called μ because of its greatest-fixpoint semantics [49, 1, 2], while the expressions themselves are sometimes called term graphs because they implicitly have a single root and no cycle [45]. Our binder has a different semantics, which justifies a different binder name.

These languages of graph expressions were used by Ariola and Klop [1], who studied confluence problems in graph rewriting, and by Goubault [23], who studied termination. All these languages, however, fail to account for horizontal sharing, because term-graphs have a single root. Our language of expressions is a generalization that includes multiple roots. A syntax for composing expressions whose operations have multiple outputs was proposed by James Kajiya [34]. Our own syntax is the simplest possible, we believe, extending Ariola's and Klop's language.

We use overlining as a general way to form lists from expressions of any kind, for example terms, but also equations.

Definition 39 (Drag Expressions)**.** *The set of* expressions *is defined by the grammar:*

$$s, t \quad ::= \quad x \quad \Big| \quad (\bar{t}) \quad \Big| \quad f(\overline{s}) \quad \Big| \quad [\overline{x = f(\overline{s})}]t$$

with $x \in \Xi$, $f \in \Sigma$, $\overline{x}$ is a list of distinct variables in $[\overline{x = f(\overline{s})}]$, and $/(\overline{s})/_r = |f|$, where the root-length $/s/_r$ *of a drag expression s (the number of its roots) is defined by induction as follows:*

$$/x/_r = 1, \quad /(\bar{t})/_r = \sum\nolimits_{t \in \bar{t}} /t/_r, \quad /f(\overline{s})/_r = 1, \quad \textit{and} \quad /[\overline{x = f(\overline{s})}]t/_r = /t/_r,$$

Expressions are categorized as follows:

- *x is a variable; we shall see that it corresponds to a rooted sprout.*
- *$(\bar{t})$ is a list of drags, which includes as a particular case the empty drag expression* ( ) *of root-length* 0.
- *$f(\overline{s})$ is* root-algebraic*; its subexpressions are arbitrary terms.*
- *$[\overline{x = f(\overline{s})}]t$ is an* abstraction *whose* assignment *$\overline{x = f(\overline{s})}$ is a (possibly empty) set of equalities, and* body *t, an arbitrary expression.*

Abstractions are expressions formed with the binder "$[-, \ldots, -]_n-$" of arity $n \geq 0$, where n is usually omitted. The use of parentheses for lists of expressions is necessary to obtain a nonambiguous grammar that can be parsed efficiently.

Note that assignments $[\overline{x = f(\overline{s})}]$ are not drag expressions. They could be. There is no particularly good reason for this choice, other than that it fits with [1]. When giving semantics to drag expressions, we shall implicitly give them semantics too.

The root-length $/(\bar{t})/_r$ of a list $(\bar{t})$ should not be confused with the length of $\bar{t}$ as a list, sometimes denoted by $|\bar{t}|$. They are in general different. By convention, we usually characterize a list of length 1 by writing (t) instead of $(\bar{t})$.

The first three grammar rules define uni-rooted drags, the last grammar rule being necessary to allow for multi-rooted drags. The first two grammar rules suffice for trees. Abstractions are needed for representing sharing. In the case of dags, the third grammar rule should be restricted so as to avoid cycles, whose presence can be detected by computing the occurs-check relationship to be defined later. Our definition eliminates only cycles among the variables themselves by equating variables in assignments to root-algebraic terms only. This crucial syntax restriction relates to our definition of well-behaved switchboards, as we shall realize when giving semantics to drag expressions.

In the following, we sometimes identify $[S](t)$ with $[S]t$, $[](\bar{t})$ with $(\bar{t})$, $[]t$ with t and $[S]()$ with $()$. These identifications are part of the congruence on drag expressions to be introduced in Section 11.

Definition 40 (Free and Bound Variables)**.** *A variable x occurs* free *in a term t iff any of the following holds:*

1. *$t = x$, or*
2. *$t = (\overline{u})$ and x occurs free in some $v \in \overline{u}$, or*
3. *$t = f(\bar{t})$ and x occurs free in $\bar{t}$, or*
4. *$t = [\overline{y = u}]w$, x occurs free in some $v \in \overline{u}$ or in w, and $x \notin \overline{y}$.*

Otherwise, it is bound*. We denote by $\mathcal{V}ar(t)$ the set of variables occurring free in t. The set of ground expressions with no free variables is denoted $\mathcal{G}(\Sigma)$. We say that a drag expression d is* pure *if no variable of d occurs both free and bound, nor does it occur bound more than once.*

Drag	Drag Expression
(1)	$f(g(b), a, g(b))$
(2)	$[x = b]f(g(x), a, g(x))$
(3)	$[x = g(b)]f(x, a, x)$
(4)	$f(a, [x = g(b)](x, x))$
(5)	$[x = b]f(g(x), [y = g(z), z = h(y, x)](y, z))$
(6)	$[x = g(x), y = h(y', z), z = h(y, z'), y' = g(z'), z' = g(y'), x' = f(x, y, y)]x'$
(7)	$[x = b]f(x, [y = h(x, z), z = h(y, [z' = g(y)]z')](y, z))$

Table 1: Drag expressions for the seven drags shown in Figure 1.

As for any other algebraic language, with or without scoping, drag expressions can take substitutions. Our notation for substitutions takes the form $\{x_1 \mapsto s_1, \dots, x_n \mapsto s_n\}$, where the variables $x_1, \dots, x_n$ are distinct. We won't use substitution much in this article, except for renaming variables, so that no confusion with the switchboard notation is to be feared. Application of a substitution σ to a drag expression s is denoted $s\sigma$, using postfix notation.

Definition 41 (Roots)**.** *The list of* roots $\mathcal{R}(t)$ *of a pure drag expression t is defined by induction as follows:*

1. $\mathcal{R}(x) = x$;
2. $\mathcal{R}((u_1, \dots, u_n)) = \mathcal{R}(u_1) \cup \cdots \cup \mathcal{R}(u_n)$;
3. $\mathcal{R}(f(\bar{t})) = f(\bar{t})$;
4. $\mathcal{R}([\overline{x = f(\overline{u})}]t) = \mathcal{R}(t)$.

It can be easily verified that the root-length of a drag expression t is just the length of its list of roots.

Example 42. *The seven drags shown in Figure 1 have expressions (among many others) shown in Table 1.*

It is easy to match a drag with its expression. Note first that bound variables are used to name vertices, in particular shared vertices for which they are necessary. The first drag (1) is a tree; no abstraction is needed. The second (2) and third (3) specify the sharing in an abstraction coming first, which gives a quite readable expression, which we call flat. *In both cases, using an outside abstraction is necessary. The fourth drag (4) is expressed differently, using the fact that its shared subdrag occurs at two contiguous places, which allows one to push the abstraction inside. The fifth (5) combines an inside and outside abstraction. The sixth drag (6) uses an outside abstraction including an extra variable naming the (single here) root; we call it* flattened. *Flattened expressions do not reflect the drag structure, but are easy to calculate for any given drag. Finally, in the last expression (7), abstractions are pushed inside as far as possible, reflecting best the drag structure. On the other hand, we can see that the description of its big cycle is split into two different abstractions, which actually correspond to the fact that the cycle has itself an internal cycle whose vertex labeled z' is not a root.*

Assume now that the first drag (1) of Figure 1 is modified by adding the vertex labeled a as the second root. Its expression becomes $[x = a](f(g(b), x, g(b)), x)$. Hence, the presence of nonmaximal roots impacts the expression of a drag in the same way as sharing does, since it corresponds to adding a fake new root whose successors are all existing roots of the drag.

10. Semantics of drag expressions

We address here two related questions: Are all expressions drag representations (soundness)? Can all drags be represented (completeness)?

First, we describe the construction of a drag $[\![d]\!]$ from an expression d with possibly free variables, thereby giving semantics to our language of drag expressions, and answering therefore the first question. This construction ensures the *root invariant property*, namely, that the roots of $[\![d]\!]$ are one-to-one with the roots of d.

With no loss of generality, we may assume that the expression is pure: no variable is bound twice in d, nor free and bound.

Definition 43. *Given a pure drag expression d, the drag $[\![d]\!]$ is built by induction on the expression d as per the following cases:*

- $d \in \Xi$. *Then the drag $[\![d]\!]$ is reduced to a rooted sprout labeled x, that is, 1_x^x;*
- $d = (s_1, \ldots, s_n)$. *Then $[\![d]\!] = [\![s_1]\!] \oplus [\![s_2]\!] \oplus \cdots \oplus [\![s_n]\!]$;*
- $d = f(s_1, \ldots, s_n)$, *where $m = |f| = /(s_1, \ldots, s_n)/_r$. Then $[\![d]\!] = f(x_1, \ldots, x_m) \otimes_{\{x_i \mapsto i\}_{i=1}^m} ([\![s_1]\!] \oplus \cdots \oplus [\![s_n]\!])$;*
- $d = [x_1 = f_1(\overline{s_1}), \ldots, x_n = f_n(\overline{s_n})]\, t$. *Then, $[\![d]\!] = \left((d_1^{q_1} \oplus \cdots \oplus d_n^{q_n}) \otimes_{\{x_i^{p_i} \mapsto i, y_i^{p_i} \mapsto i\}} 1_{y_1^{p_1} \ldots y_n^{p_n}}^{y_1^{p_1} \ldots y_n^{p_n}}\right) \otimes_{\langle \varnothing\, ;\, x_i \mapsto i\rangle} [\![t]\!]$, where*
  - $p_i + 1$ *denotes the number of occurrences of the variable x_i in $[x_1 = f_1(\overline{s_1}), \ldots, f_1(\overline{s_1})]$, the 1 standing for its occurrence to the left of the equal sign in the elementary assignment $x_i = f_i(\overline{s_i})$.*
  - $q_i + 1$ *denotes the number of occurrences of the variable x_i in d (hence including t this time).*
  - $d_i^{q_i}$ *denotes the drag $d_i = [\![f_i(\overline{s_i})]\!]$ in which the root (corresponding to the vertex f_i) is repeated q_i times. This allows us to share the drag d_i a number of times equal to the number of occurrences (minus 1) of the variable x_i in d. As a matter of convenience, we improperly refer to all roots of d_i by the same number i (but two different sprouts y_i must of course connect two different roots of d_i).*
  - y_i^p *is the repetition of y_i for p times; the y_i's duplicate the x_i's from the abstraction part and inherit their multiplicities as root multiplicities for the unit drag. To minimize the notational burden, we refer by the same number i (in the target of $x_i \mapsto i$) to all occurrences of the variable y_i in the list $y_1^{p_1} \ldots y_n^{p_n}$.*

Before showing that this definition makes sense, we work out an example.

Example 44. *Let d be $[x_1 = f(x_2), x_2 = a](x_1, x_1, x_2)$. In order to use a repeated variable x as a sprout name in switchboards, let's distinguish its occurrences by writing them x, x', x'', and so on. So, $d = [x_1 = f(x_2), x_2' = a](x_1', x_1'', x_2')$, the variable labeling a sprout being then obtained by removing the primes.*

Since the variables x_1 and x_2 both occur three times in d, we need to first construct the two drags corresponding to the expressions $f(x_2)$ and a, which are just $f(x_2)$ and a, both with a repeated top root. We indicate these roots by lists of numbers in the superscript of the corresponding symbol; we get here $f^{1,2}(x_2)$ and $a^{1,2}$. We then consider the drag $f^{1,2}(x_2) \oplus a^{1,2}$, which computes to the drag

$$f^{1,2}(x_2) \quad a^{3,4}.$$

We now need to compute $(f^{1,2}(x_2), a^{3,4}) \otimes_{\{x_2' \mapsto 1, y_2 \mapsto 3\}} 1_{y_2}^{y_2}$, which yields the drag $f^{1,2}(a^3)$. Note that x_1 does not appear in the switchboard, since that variable has a single occurrence (on the left of the "$=$" sign) in $[x_1 = f(x_2), x_2' = a]$.

The list (x_1', x_1'', x_2') is interpreted as the drag $1_{x_1}^{x_1} \oplus 1_{x_1}^{x_1} \oplus 1_{x_2}^{x_2}$, that is, the drag

$$x_1'^1 \quad x_1''^1 \quad x_2'^1.$$

Note that there is no sharing yet.

We are left with computing $f^{1,2}(a^3) \otimes_{\langle \varnothing; x_1' \mapsto 1, x_1'' \mapsto 2, x_2' \mapsto 3\rangle} (x_1'\ x_1''\ x_2')$, which yields the drag $f^{1,2}(a^3)$ as expected, the roots of the result being obtained by transfer of the three roots of the drag

$$x_1'^1 \quad x_1''^2 \quad x_2'^3.$$

Lemma 45. *For every pure drag expression d, $[\![d]\!]$ is a well-defined drag, and it has the same number of roots as d.*

Proof. By induction on the structure of d.

- $d \in \Xi$. This case is straightforward.
- $d = (s_1, \ldots, s_n)$. Straightforward use of the induction hypothesis.

- $d = f(s_1, \ldots, s_n)$. First, $[\![s_1]\!], \ldots, [\![s_n]\!]$ satisfy the induction hypothesis. Notice that the switchboard is one-way, hence well-behaved. Composition with the switchboard will then put f on top of $[\![s_1]\!], \ldots, [\![s_n]\!]$ with the required edges.

- $d = [x_1 = f_1(\overline{s_1}), \ldots, x_n = f_n(\overline{s_n})]\, t$. First, $[\![f(\overline{s_1})]\!], \ldots, [\![f(\overline{s_n})]\!], [\![t]\!]$ satisfy the induction hypothesis. Then, $d_1^{p_1} \oplus \cdots \oplus d_n^{p_n}$ constructs a drag for the expression $(f(\overline{s_1})^{p_1}, \ldots, f(\overline{s_n})^{p_n})$, where the exponents indicate the number of times that the (root of the) expression is repeated. By definition of an assignment, each d_i is headed by a function symbol; hence the switchboard $\{x_i \mapsto y_i, y_i \mapsto i\}$ is well-behaved. The effect of the composition $(d_1^{p_1} \oplus \cdots \oplus d_n^{p_n}) \otimes_{\{x_i \mapsto i\}} 1_Y^Y$ is to redirect all edges incoming a variable x_i in d_i to a (specific) root of d_i. Therefore, the number of remaining roots corresponding to x_i matches now the number of occurrences of x_i in t. Hence the second composition makes sense, and the obtained number of roots of the whole drag matches that of d. $\square$

Before addressing the second question, we need to define some basic notions and vocabulary:

Definition 46 (Occurs Check)**.** *Given an assignment $\overline{x = s}$, we define the following:*

1. *The* occurs-check *binary relation $<_{oc}$ is the least transitive relation between variables in $\overline{x}$ such that $y <_{oc} z$ if $z = v \in \overline{x = s}$ and $y \in \mathcal{V}ar(v)$.*
2. *The set $\circlearrowleft y$ of* occurs-check *variables of $y \in \overline{x}$ (for given $\overline{x = s}$) is the set of variables $\{z : z <_{oc} y <_{oc} z\}$.*
3. *The set of* occurs-check *variables of $\overline{x = s}$ is the set $\{y : \circlearrowleft y \neq \varnothing\}$.*
4. *An occur-check variable $y \in \overline{x}$ is* maximal *if $x <_{oc} y \Rightarrow y <_{oc} x$ for all $x \in \overline{x}$.*

Note that the meaning of the occurs-check relationship $y <_{oc} z$ (as is the case in unification for logic programming) is that the vertex y is further away from a maximal root than vertex z; hence the drag expression associated with vertex y is smaller in size than that associated with vertex z. Our definition is made simpler than usual thanks to the assumption that terms in $\overline{s}$ are root-algebraic, hence are not variables. Note also that a variable x that occurs only once in an assignment $\overline{x = s}$ has an empty set of occurs-check variables: the occurs-check relationship need not be reflexive.

Definition 47 (Flat Assignment)**.** *An assignment $\overline{x = s}$ is* flat *if (i) the variables in $\overline{x}$ are distinct, and (ii) all expressions in $\overline{s}$ are* flat*, that is, are either variables or of the form $f(\overline{y})$, with $y \in \overline{x}$ for all $y \in \overline{y}$. A drag expression of the form $[S](\overline{y})$ is* flat *if S is flat.*

Flat expressions are always pure.

Definition 48 (Clean Expression)**.** *A flat expression $[\overline{x = s}](\overline{y})$ is* clean *if $\forall x \in \overline{s}.\, \exists y \in \overline{y}.\, (y = x \vee x <_{oc} y)$.*

Note that $\overline{y}$ is, therefore, the list of all roots (whether maximal or not) of the flattened expression $[S](\overline{y})$. Note also the particular case where the flat expression has the form $[\,](\overline{y})$, in which case we will write $(\overline{y})$, or even $\overline{y}$

Constructing a flat drag expression $\ulcorner D \urcorner$ from a drag D is easy: it suffices to name each internal vertex of the drag by a fresh variable and each sprout by its variable label, then write down all equations that describe the relationship between an internal vertex and its outgoing neighbors, and finally build the expression as an abstraction whose roots can be easily listed since all vertices of the drag have a name. This gives a description of the drag that is flat. Moreover, it is easy to see that the obtained drag expression is clean if and only if the starting drag is.

To show adequacy of the semantics with respect to the syntax, we need a technical lemma:

Lemma 49. *Let $d = [x_1{=}s_1, \ldots, x_m{=}f(\ldots, x_j, \ldots)](x_1, \ldots, x_n)$ and $g = [x_1{=}s_1, \ldots, x_m{=}f(\ldots, y, \ldots)](x_1, \ldots, x_n, y)$, where y is a fresh variable. Then, $[\![d]\!] = [\![g]\!] \otimes_{\{y \mapsto 1, z \mapsto n+1\}} 1_z^z$.*

Proof. Using the notation $d_i^{p_i}$ of Definition 43, we get $[\![d]\!] = (d_1^{p_1} \oplus \ldots \oplus d_m^{p_m}) \otimes_{\{x_i \mapsto i, y_i \mapsto i\}} 1_Y^Y \otimes_{\langle \varnothing\, ;\, x_i \mapsto i \rangle} (1_{x_1}^{x_1} \oplus \cdots \oplus 1_{x_m}^{x_m})$, where $Y = x_1^{p_1} \ldots x_m^{p_m}$. By Lemma 15, $[\![d]\!] = (d_1^{p_1} \oplus \cdots \oplus d_m^{p_m}) \otimes_{\{x_i \mapsto i, y_i \mapsto i\}} 1_Y^Y$. Let $d'_m = [\![f(\ldots, y, \ldots)]\!]$. Since $d_m = d'_m \otimes_{\{y \mapsto 1\}} x_j$, we get $d = (d_1^{p_1} \oplus \ldots \oplus (d'^{p_m}_m \otimes_{\{y \mapsto 1\}} x_j)) \otimes_{\{x_i \mapsto i, y_i \mapsto i\}} 1_Y^Y = (d_1^{p_1} \oplus \ldots \oplus d'^{p_m}_m) \otimes_{\{x_i \mapsto i, y_i \mapsto i, y \mapsto j\}} 1_Y^Y = (d_1^{p_1} \oplus \cdots \oplus d'^{p_m}_m) \otimes_{\{x_i \mapsto i, y_i \mapsto i\}} 1_Y^Y \otimes_{\{y \mapsto 1, z \mapsto i\}} 1_z^z = g \otimes_{\{y \mapsto 1, z \mapsto i\}} 1_z^z$. $\square$

Theorem 50 (Adequacy)**.** *Let D be a clean closed drag. Then, $[\![\ulcorner D \urcorner]\!] \simeq D$.*

Proof. Let $d = \ulcorner D \urcorner = [S](\overline{x})$. The proof that $\llbracket d \rrbracket \simeq D$ is by induction on the number n of edges of D relating internal vertices.

- If D is empty, then $d = ()$, and the result holds.
- If $n = 0$ and D has p roots, then, since D is clean, it must be made of internal vertices labeled by constants or of sprouts labeled by variables. Since D is their (disjoint) union, we proceed by induction on the number of roots.

  Base case 1: D is reduced to a single sprout labeled x. Then, $d = x$ and $\llbracket x \rrbracket$ is clearly isomorphic to D.

  Base case 2: D is a lone internal vertex labeled a. Then $d = [x = a](x)$ and, by Definition 43, $D' = a \otimes_{y \mapsto 1} 1^y_y$, which reconstructs the drag $a \simeq D$ by transferring the root of 1^y_y to a.

  Induction step: the only (minor) difficulty is to take care that a new root may correspond to a vertex belonging to the rest of the list.
- Otherwise, $n > 0$. Let p be the number of roots of D. Let now v, w be a pair of (possibly identical) vertices of D with an edge going from v to w. By the definition of d, let x_i, x_j be the variables of d corresponding to v, w. We now define the drag G obtained from D by replacing this edge by one going from v to a new sprout labeled by a fresh variable y, and adding w as the $(p+1)$-th root. By the definition of G, we have $D \simeq G \otimes_{y \mapsto 1, z \mapsto p+1} 1^z_z$.

  Let $g = \ulcorner D \urcorner$. By the induction hypothesis, $\ulcorner G \urcorner = g$ and $\llbracket g \rrbracket \simeq G$.

  Let $d = \ulcorner D \urcorner = [x_1{=}s_1, \ldots, x_i{=}f(\ldots, x_j, \ldots)](\overline{x})$, where $|\overline{x}| = p$. By using, without loss of generality, the same variable names for g and d, we have $g = [x_1{=}s_1, \ldots, x_i{=}f(\ldots, y, \ldots)](\overline{x}, y)$. By Lemma 49, $\llbracket d \rrbracket = \llbracket g \rrbracket \otimes_{\{y \mapsto 1, z \mapsto p+1\}} 1^z_z \simeq G \otimes_{\{y \mapsto 1, z \mapsto p+1\}} 1^z_z$, since vertical products preserve isomorphisms. We are done. □

We have therefore shown the adequacy of clean, flattened expressions with drags. Of course, there are drag expressions that are neither flattened nor clean. We show in the next section that those are actually semantically equivalent to flattened, clean expressions, thereby extending the adequacy theorem to all expressions.

11. Canonical expressions

We address a complementary question in this section: the language of drag expressions being redundant, can we exhibit an equational theory that characterizes drag isomorphism? The main tool used for answering this question is the definition of normal forms by the means of convergent rewriting systems [14].

Whereas there is a single way to associate the drawing of a drag to a given drag expression by the previous construction, there are many ways to associate a drag expression to a given drag. The easiest method is the one already mentioned, which associates a variable to each vertex of the drag, writes all equations of the form $\overline{x = f(\overline{y})}$ expressing that the vertex x labeled by f has the vertices $\overline{y}$ as successors, and outputs the expression $[\overline{x = f(\overline{y})}](\overline{z})$, where $\overline{z}$ is the list (with repetitions) of variables denoting root vertices. But our language of expressions being richer than needed, there are other ways: as we have seen with the examples, a drag may be denoted by many (actually infinitely many) different expressions. We therefore introduce an equational theory on drag expressions aimed at capturing drag isomorphism.

Definition 51 (Convertible Expressions)**.** *Two drag expressions s, t are* convertible*, written $s \simeq t$, iff they may be obtained from each other by using the three sets of equations given in Table 2.*

The equations in the table play the following rôles: the first three (E) speak for themselves. The next four (S) serve to clean drag expressions. The last four (F) serve to move assignments up or down while preserving the scoping rules. The many side conditions have a single purpose, namely, ensuring that the cleaning rules don't apply on either side.

Theorem 52 (Soundness)**.** *The equations are sound with respect to the semantics of drag expressions.*

Proof. Tedious, but easy, checking, using the semantics of drag expressions defined in Section 10. □

$[\overline{x=u}]\,t$	$=$	$[\overline{z=u\{\overline{x}\mapsto\overline{z}\}}]\,t\{\overline{x}\mapsto\overline{z}\}$	if $\overline{z}\cap \mathcal{V}ar(\overline{x},\overline{u},t)=\varnothing$	α	
$[\overline{x=u}][\overline{y=v}]t$	$=$	$[\overline{x=u},\overline{y=v}]t$	if $\overline{x}\cap\overline{y}=\varnothing$	A	E
$[\ldots,x=u,\ldots,y=s,\ldots]\,t$	$=$	$[\ldots,y=s,\ldots,x=u,\ldots]\,t$		C	
(t)	$=$	t		$()\curvearrowright$	
$[\overline{x=u}]()$	$=$	$()$		$()$	
$[\,]\,t$	$=$	t		$[\,]$	S
$[\overline{x=u},\overline{y=v}]t$	$=$	$[\overline{x=u}]t$	if $\overline{y}\cap\mathcal{V}ar(\overline{u},\bar{t})=\varnothing$ and $\overline{y}\neq\varnothing$	$\ominus$	
$[\overline{x=u},\overline{y=s}](\overline{v},t,\overline{w})$	$=$	$[\overline{x=u}](\overline{v},[\overline{y=s}]t,\overline{w})$	if $\overline{y}\cap\mathcal{V}ar(\overline{u},\overline{v},\overline{w})=\varnothing,\overline{y}\cap\mathcal{V}ar(t)\neq\varnothing$, $\overline{vw}\neq\varnothing$ and $\overline{x}\cap\mathcal{V}ar(\overline{s},\overline{v},\overline{w},t)\neq\varnothing$	$\curvearrowright\curvearrowright()$	
$[\overline{x=u},\overline{y=s}]\,f(\overline{v},t,\overline{w})$	$=$	$[\overline{x=u}]\,f(\overline{v},[\overline{y=s}]t,\overline{w})$	if $\overline{y}\cap\mathcal{V}ar(\overline{u},\overline{v},\overline{w})=\varnothing,\overline{y}\cap\mathcal{V}ar(t)\neq\varnothing$, and $\overline{x}\cap\mathcal{V}ar(\overline{s},\overline{v},\overline{w},t)\neq\varnothing$	$\curvearrowright\curvearrowright\Sigma$	F
$[\overline{x=u},y=f(\overline{s})]\,y$	$=$	$[\overline{x=u}]\,f(\overline{s})$	if $y\notin\mathcal{V}ar(\overline{u},\overline{s})$ and $\overline{x}\cap\mathcal{V}ar(\overline{s})\neq\varnothing$	$\Uparrow$	
$[\overline{x=u},\overline{y=v},z=w]\,t$	$=$	$[\overline{x=u},z=[\overline{y=v}]\,w]\,t$	if $\overline{y}\cap\mathcal{V}ar(\overline{u},t)=\varnothing,\overline{y}\cap\mathcal{V}ar(w)\neq\varnothing$, $\overline{x}\cap\mathcal{V}ar(\overline{v},\overline{w},t)\neq\varnothing$, $z\notin\mathcal{V}ar(v)$ and $z\in\mathcal{V}ar(\overline{u},t)$	$\curvearrowright$	

Table 2: Equations of expression convertibility.

To decide convertibility, we are now going to transform this set of equations into a rewrite system that defines normal forms. It turns out that there are at least two ways to turn these equations into rewrite rules.

In both cases, the first three equations, made of α-conversion, merge of contiguous assignments (called A) and permutations inside assignments (called C), define the equational (modulo) part $AC\alpha$. Indeed, A expresses the fact that concatenation of assignments is associative, and C that it is commutative. This was hidden by the fact that bracketing was defined as a varyadic operator. Note further that the condition in the equation A is a *purity* condition, which does not prevent merging the assignments since the expression can always be purified by using α-conversion.

The next four equations are oriented from left to the right, yielding the set S of simplifiers, which serves for eliminating useless information, hence defining the *clean normal forms*. There are two choices for the last four equations (F): the *factorization* system FA oriented them from left to right, while the *flattening* system FL orients them from right to left, and adds two rules that are needed to obtain the unique normal property:

$(\overline{v},[\overline{y=s}]t,\overline{w})$	$\longrightarrow$	$[\overline{y=s}](\overline{v},t,\overline{w})$	$\curvearrowright()$
$f(\overline{v},[\overline{y=s}]t,\overline{w})$	$\longrightarrow$	$[\overline{y=s}]f(\overline{v},t,\overline{w})$	$\curvearrowright\Sigma$

Rules in FA and FL are supposed to be applied to clean normal forms. Such a rewriting system (R,S,E) coming into three parts – viz. a set of equations E, a set of simplifiers S that is convergent (terminating and Church-Rosser) modulo E, and a set of rules R, such that the rules in R apply to fully simplified expressions – is called *normal* [31].

For termination arguments, we will interpret a drag expression d by a pair $\langle n,|d|\rangle$ of natural numbers, where the second component is the size of the expression d, and the first component n is a weight that will be defined along the way, when needed. Pairs are compared lexicographically, and, therefore, the size $|d|$ will come into play only when the two triples under comparison have the same weight n. Note that both sides of all the equations in E have the same interpretation provided the weights of the left- and right-hand sides of each equation are equal. We shall have to enforce this property when defining the weight n, as well as the following one, which will serve for the simplifiers in S: weights must be the same for $t,(t)$, and $[]t$.

Theorem 53. *The rewriting system S is convergent modulo E.*

Proof. Checking convergence of rewriting modulo E is described in [30]. Termination is clear: E leaves the order on the interpretation invariant, while S decreases it strictly under our assumptions. Furthermore, the critical pairs (modulo E) are all joinable (modulo E). Confluence follows [30]. ☐

Theorem 54. *The normal rewriting system* (FA, S, E) *is convergent, hence defining the* factorized *normal forms.*

Proof. Checking convergence of normal rewriting systems is described in [31]. First, we need to prove that FA/E is a convergent rewriting system modulo, using the same termination order as for S so as to ensure that the union of both rewrite relations is indeed terminating modulo E. To this end, we now define n as a multiset of natural numbers, one for each non-empty assignment, where assignments are assumed flattened ($[\overline{x = u y = v}]$ is the assignment $[\overline{x = u}, \overline{y = v}]$, after possibly renaming the variables to ensure purity), and contribute for the number of elementary assignments that they contain. It is easy to check that this definition satisfies our requirements.

We are left with showing that the critical pairs of FA strictly inside S and the critical pairs of FA with itself are joinable modulo E. There are none of the former since an abstraction cannot be unified with an assignment. The latter can only be obtained by unifying left-hand sides of two rules in FA, unification at subterms being impossible here. Self-overlaps of a rule with itself resolve by commuting the applications. The rule $\upuparrows$ can only be unified with the rule $\curvearrowright$ because of the variable y, which must be a variable bound by its left abstraction. Moreover, $\curvearrowleft\!\!\curvearrowright()$ cannot unify with $\curvearrowleft\!\!\curvearrowright\Sigma$ because a list of length 1 cannot unify with a list of length at least 2 (the condition $\overline{vw}$ was added for that purpose). This leaves three pairs of rules that overlap. Due to the various constraints on the variables, we obtain five critical pairs in total:

1. $\curvearrowleft\!\!\curvearrowright()$ with $\curvearrowright$: a first most general overlap is $d = [\overline{x = u}, \overline{y = v}, z = w, \overline{y = s}](\overline{v'}, t, \overline{w'})$, which gives a commuting diagram:
   $d \longrightarrow_{\curvearrowright} [\overline{x = u}, z = [\overline{y = v}]w, \overline{y = s}](\overline{v'}, t, \overline{w'}) \longrightarrow_{\curvearrowleft\!\!\curvearrowright()} [\overline{x = u}, z = [\overline{y = v}]w](\overline{v'}, [\overline{y = s}]t, \overline{w'})$, and
   $d \longrightarrow_{\curvearrowleft\!\!\curvearrowright()} [\overline{x = u}, z = [\overline{y = v}]w](\overline{v'}, [\overline{y = s}]t, \overline{w'}) \longrightarrow_{\curvearrowright} [\overline{x = u}, z = [\overline{y = v}]w](\overline{v'}, [\overline{y = s}]t, \overline{w'})$.
2. $\curvearrowleft\!\!\curvearrowright()$ with $\curvearrowright$: another most general overlap is $[\overline{y = v}, \overline{x = u}, \overline{y' = s'}, z = w](\overline{v'}, t, \overline{w'})$, which rewrites
   – using rule $\curvearrowright$ to $[\overline{x = u}, \overline{y' = s'}, z = [\overline{y = v}]w](\overline{v'}, t, \overline{w'})$, which now rewrites with rule $\curvearrowleft\!\!\curvearrowright()$ to the expression $[\overline{x = u}](\overline{v'}, [\overline{y' = s'}, z = [\overline{y = v}]w]t, \overline{w'})$
   – using rule $\curvearrowleft\!\!\curvearrowright()$ to $[\overline{y = v}, \overline{x = u}](\overline{v'}, [\overline{y' = s'}, z = w]t, \overline{w'})$, which rewrites successively with rules $\curvearrowleft\!\!\curvearrowright()$ again and $\curvearrowright$ (modulo A) to $[\overline{x = u}](\overline{v'}, [\overline{y = v}][\overline{y' = s'}, z = w]t, \overline{w'})$ and $[\overline{x = u}](\overline{v'}, [\overline{y' = s'}, z = [\overline{y = v}]w]t, \overline{w'})$, joining the critical pair.

3,4. $\curvearrowleft\!\!\curvearrowright\Sigma$ with $\curvearrowright$: we get the same two cases as above with $f(\overline{v'}, t, \overline{w'})$ as body of the abstraction instead of $(\overline{v'}, t, \overline{w'})$.

5. $\upuparrows$ with $\curvearrowright$: the only possible most general unifier is $[\overline{x = u}, \overline{y = v}, z = f(\overline{s})]z$, which rewrites
   – using rule $\curvearrowright$ to $[\overline{x = u}, z = [\overline{y = v}]f(\overline{s})]z$, which then rewrites with rule $\upuparrows$ to $[\overline{x = u}][\overline{y = v}]f(\overline{s})$;
   – using rule $\upuparrows$ to $[\overline{x = u}, \overline{y = v}]f(\overline{s})$, joining the critical pair with an A step. ☐

Theorem 55. *The normal rewriting systems* (FL, S, E) *is convergent, hence defines* flat *normal forms.*

Proof. The roadmap is the same as for the proof of the previous theorem.

For termination, we now define n in the measure $\langle n, |d| \rangle$ of expressions d to be another multiset of natural numbers, each assignment (in flattened form) contributing its distance to the root of d, where this distance ignores parentheses. The rules of FL all move assignments outwards in the expression.

We proceed with the Church-Rosser check, that is, with the computation of critical pairs. There are many rules and critical pairs that we are not going to compute. Instead, we show that normal forms are flat expressions, that is, that a rule always applies to any non-flat drag expression in S_E-normal form. The fact that the rules do never erase equations, nor any function symbol, implies then that different normal forms issuing from a same term in S_E-normal form must be equal modulo permutations of the elementary assignments in the abstraction.

Showing that non-flat drag expressions in S_E-normal can always be rewritten is an easy task. ☐

We illustrate below the computation of the flat normal-form expression, by rewriting modulo the equations, of the expression given earlier for drag (7) of Table 1:

$$
\begin{array}{ll}
 & [x = b]f(x, [y = h(x,z), z = h(y, [z' = g(y)]z')](y,z)) \\
\longrightarrow_{\curvearrowright\Sigma} & [x = b]f(x, [y = h(x,z), z = [z' = g(y)]h(y,z')](y,z)) \\
\longrightarrow_{\curvearrowright} & [x = b]f(x, [y = h(x,z), z' = g(y), z = h(y,z')](y,z)) \\
\longrightarrow_{\curvearrowright\!\!\curvearrowright\Sigma} & [x = b, y = h(x,z), z' = g(y), z = h(y,z')]f(x,y,z') \\
\longrightarrow_{\Uparrow} & [x = b, y = h(x,z), z' = g(y), z = h(y,z'), y' = f(x,y,z')]y'
\end{array}
$$

We now compute the factorized expression from the obtained flat expression, and observe that there are more steps in this direction:

$$
\begin{array}{ll}
 & [x = b, y = h(x,z), z' = g(y), z = h(y,z')]f(x,y,z') \\
\longrightarrow_{\curvearrowright\!\!\curvearrowright\Sigma} & [x = b]f(x, [y = h(x,z), z' = g(y), z = h(y,z')](y,z)) \\
\longrightarrow_{\curvearrowright} & [x = b]f(x, [y = h(x,z), z = [z' = g(y)]h(y,z')](y,z)) \\
\longrightarrow_{\curvearrowright\!\!\curvearrowright\Sigma} & [x = b]f(x, [y = h(x,z), z = []h(y, [z' = g(y)]z')](y,z)) \\
\longrightarrow_{[\,]} & [x = b]f(x, [y = h(x,z), z = h(y, [z' = g(y)]z')](y,z))
\end{array}
$$

We can also observe that factorized normal forms are quite economical: all bound variables denote vertices of the graph that all have several incoming edges, hence are shared and must therefore be named. In addition, every variable assignment in an abstraction is located as close as possible to its intended body, that is, at the vertex that is the closest common ancestor of the vertices denoted by its bound variables.

Because the equations of E need to be built into the matching process, rewriting with these systems could be expensive. This is not in fact the case. Firstly, α-conversion can be efficiently implemented. Secondly, left-hand sides of rules of S are linear, and matching modulo AC can be done efficiently in that case. Thirdly, all rules in FL are left-linear, hence the same remark applies. For FA, there is a single rule whose left-hand side is nonlinear, $\Uparrow$. For that rule, however, it suffices to match first the variable y in root position, and then the rest of the left-hand side, which makes it even easier. The normal-form computations can therefore be efficiently implemented in both cases.

These normal forms can be used in many ways. We use them here as a theoretical tool, to show that convertibility is a sound and complete axiomatization of drag isomorphism:

Theorem 56. *Two closed drags are isomorphic iff their drag expressions are convertible.*

Proof. Two drags are isomorphic iff there is a one-to-one map between their sets of vertices that identifies their respective lists of roots and commutes with their successor functions. We can therefore assume without loss of generality that their sets of vertices are identical. For convenience, we name them by variables. Given two isomorphic drags whose vertex names are now identified, their corresponding drag expressions are both of the form $[\overline{x} = \overline{f(\overline{y})}](\overline{z})$, hence can only differ by a permutation of their variable assignments. They are therefore convertible. Conversely, two convertible drag expressions have the same flat normal form (up to renaming of their variables and permutation of their assignments) by Theorem 55. The drag construction yields the same drag for both, up to the names of their vertices. $\square$

The normal forms we have defined have many other uses. In particular, it is easy to characterize the following from both normal forms $d{\downarrow} = [\overline{x = f(\overline{y})}](\overline{z})$ of a drag d: the connected components of d; the head of d; the tail of d.

We can now also prove a final result, a sort of dual for Theorem 50:

Theorem 57. *Let d be a drag expression. Then, $\ulcorner [\![d]\!] \urcorner$ and d are convertible.*

Proof. Let d be a drag expression, and $D = [\![d]\!]$. By Theorem 50, $[\![\ulcorner D \urcorner]\!] \simeq D$, that is $[\![\ulcorner [\![d]\!] \urcorner]\!] \simeq [\![d]\!]$. By Theorem 56, $\ulcorner [\![d]\!] \urcorner$ and d are convertible. $\square$

12. Related frameworks

The literature on graph rewriting is huge. There are actually four different trends, which are of course interdependent, which we will review in turn:

1. The standard double-pushout (DPO) approach initiated by Hartmut Ehrig.

2. The study of instances of the DPO approach to specific classes of graphs, such as dags, jungles and term-graphs.
3. The use of graph-based models of computation in category theory and the theory of operads, under the name of string diagrams [51].
4. The use of graph-based models of computation in concurrency as well as the definition of languages for processes.

On the language level, there isn't much more to say than what we have already said, except in the context of concurrency, which we shall briefly discuss. Our language is directly inspired by that of Ariola and Klop [1]. An early work that possibly inspired them is that of Rose [49], which introduces a notion of cyclic substitution. That name becomes a bit misleading in the context of drags, since it actually introduces a notion of iterated substitution whose goal is to approximate infinite terms by unfolding (possibly mutually) recursive definitions without destroying sharing. This is of course different from our approach in which circular terms are handled as first-class citizens, although related, as explained in [1].

12.1. DPO

Composition of terms, that is, substitution, can be viewed in two different ways. The first, traditional way is to view composition as a homomorphism. This uses the fact that terms are freely generated. In this view, variables denote terms use to define the homomorphism. The second is to interpret variables as pointers. In this graphical view, composition amounts to redirecting pointers. This view fits very well indeed with dags, whose algebraic structure is not freely generated. It is commonly adopted by developers for efficiency reasons, and this is why term-rewriting packages are often based on dags rather than trees. It is also the essence of interaction nets, originating from linear logic, which were introduced by Lafont long ago [36].

Our notion of drag composition coincides with this latter viewpoint: roots are input ports, sprouts are output ports, and composition is a redirection of pointers from output ports to input ports. This process-algebra jargon reflects the fact that variables do not denote graphs, but channels connecting precisely two processes, in the process algebra tradition, rather than graph hyper-edges that would connect many processes all together.

As we have already said, this view fits very well with dags, a bit less so with trees. (See, however, the discussion in the conclusion.) We believe that the theory of term rewriting scales to drags very smoothly, as it does to dags, and we have demonstrated it with the framework first (here), and also with the design of syntactic rewrite orderings in [15]. A specific, important benefit of our framework is that rewriting drags cannot create dangling pointers.

The traditional view of composition of graphs initiated by Ehrig et al. [20] (see also [48] for a slightly different technical definition) takes the first route. Even though graphs are not freely generated, graph homomorphisms lead the way, rules being defined as pairs of homomorphisms preserving some interface graph included in the left-hand side, rewriting being then defined as a double graph pushout preserving that interface, one for the left-hand side homomorphism, and one for the right-hand side homomorphism. Called for that reason the double-pushout approach (DPO), it can be described either in categorical terms or by means of a graph-gluing construction. An easy-to-read recent survey is [35], in which the gluing construction is explained in all its (many) details.

In this view, the interface graph can be anything (provided it is included in the left-hand side of the rule), and as a result, rewriting can result in dangling pointers. Such rewrites may be either forbidden, or otherwise require the elimination of those edges. The rewriting definition comes therefore along with various side conditions that have to be checked when calculating the pushouts. This aspect resulted in several variations of the initial framework, some of which are thoroughly analyzed in [25]. In particular, there is a debate whether the right-hand side homomorphism should be injective or not. For implementation purposes, injectivity is of course much more comfortable.

One way to specify an interface for term-graphs that forces the absence of dangling edges is to indicate roots and sprouts; this is considered in [46]. In that work, matching imposes the restriction that roots of the rule coincide with those of the graph, something that we do not want. Other ways have been attempted for restricting the interface graph in order to minimize the checks generated by a rewriting step. In an implementation of graph rewriting used in various applications [21, 43], the interface graph is given as a set of connected vertices called ports and edges (the corresponding channels).

Because rewriting is based on graph homomorphisms, termination methods based on interpretations can be applied without too much difficulty, as is demonstrated, in particular, in [6]. Testing local confluence, on the other hand, is much more delicate: it is in general undecidable, but is decidable under the termination assumption [7]. Further, it leads to the computation of possibly infinitely many critical pairs for restricted decidable cases [44, 19, 37, 22]. A recent advance shows that it becomes decidable when enriching slightly the framework of graph rewrite rules [5].

In the traditional view of graph rewriting, there are no variables. On the one hand, variables are not really necessary, since the application of a homomorphism can be implemented by graph gluing. It follows that variables have been absent from most works on graph rewriting of which we are aware. We believe that variables are a user-friendly feature (apart from the possibility to quantify over them). This is probably why variables appear in some variations of DPO [42, 47, 26]. In these two works, hypergraphs are considered instead of graphs, and rewriting is again defined via a double pushout of hypergraphs. Variables, however, stand for edges of hypergraphs, with a given incoming and outgoing arity. This looks a little bit like our switchboard device, with the difference that a switchboard is a collection of one-way channels, which allows us to have variables as labels of vertices without successors, as in trees or dags, and to control easily how nodes must be merged (or glued, in DPO jargon). Another recent approach to including variables is [5], where it is shown that confluence of graph rewrite rules equipped with variables becomes decidable (under termination assumption) whereas it is undecidable in the usual variable-free DPO approach, a situation comparable with term rewrite rules, for which ground confluence is undecidable.

Plump and Habel [26] investigate matching and unification with hyperedges. They argue, "The matching concept suggests a new graph rewriting approach which is very simple to describe and which generalizes the well-known double-pushout approach." This work is however restricted to acyclic graphs, hence does not really generalize DPO.

Given the notion of drags, we can now explain drag rewriting in DPO terms: a drag rewrite rule is a pair of drag homomorphisms. This follows from Lemma 37, the interface graph being restricted to the list of roots and the set of sprouts of the left-hand side drag. This choice of interface, also advocated by Kahl in a more restricted setting [33], has ideal behavior, by ensuring the absence of dangling pointers and also making the double pushout construction really easy. The novelty is therefore in the notion of drag and its algebra, which allows us to define rewriting in a particularly simple way, one that does not need to refer to any explicit pushout construction, but uses instead graph composition.

12.2. DPO for dags, jungles and term graphs

As we have seen all along, terms, dags and jungles are just special cases of drags, without the added complications of cycle management. In particular, the extension of RPO to dags defined in [45] looks like a particular case of its extension to drags [16].

12.3. Graphs in the representation theory of symmetric monoidal categories

In order to compare the drag model with other similar models found in category theory, we shall now view the drag structure as a category. The idea is that horizontal and vertical composition of drags does not really operate on the drags D and D' themselves – their successor functions do not change – but on their interfaces. We therefore first normalize these interfaces. Let us assume that vertex names are natural numbers, variables are $x_1, \dots x_n$, which we identify with their natural number index, and that the roots are the vertices $1, \dots, m$. Then, the interface of a graph D belongs to the set $\{(l, m) : l \in L(n), m \in M(p)\}$, where $L(n)$ denotes the set of complete lists with repetitions over $[1 .. n]$ and $M(p)$ the set of complete finite multisets over $[1 .. p]$. By complete, we mean that all natural numbers in $[1 .. n]$ and $[1 .. p]$ must occur in the elements of $L(n)$ and $M(p)$, respectively.

Given an interface (l, m), an extension $\langle D, \xi\rangle$ can operate on that interface, yielding the new interface $(l, m) \otimes_\xi (\mathcal{R}(D), \mathcal{S}(D))$, where $\mathcal{R}(D)$ and $\mathcal{S}(D)$ need of course to be normalized. In this view of the drag structure as a category, the objects are interfaces, and the arrows are drag extensions. To have a category, composition of arrows needs to be associative:

Lemma 58. *Let $\langle D, \xi\rangle : (l, m) \longrightarrow (l', m')$ and $\langle D', \xi'\rangle : (l', m') \longrightarrow (l'', m'')$. Then, $\langle D \otimes_{\xi'} D', \xi\rangle : (l, m) \longrightarrow (l'', m'')$.*

Proof. By assumption, $(l', m') = (l, m)\otimes_\xi(\mathcal{R}(D), \mathcal{S}(D))$ and $(l'', m'') = (l', m')\otimes_{\xi'}(\mathcal{R}(D'), \mathcal{S}(D'))$. Therefore, $(l'', m'') = ((l, m) \otimes_\xi (\mathcal{R}(D), \mathcal{S}(D))) \otimes_{\xi'} (\mathcal{R}(D'), \mathcal{S}(D'))$. Since the switchboards ξ and ξ' operate on the pairs of interfaces (l, m) and $(\mathcal{R}(D), \mathcal{S}(D))$ for ξ, and (l', m') and $(\mathcal{R}(D'), \mathcal{S}(D'))$ for ξ', they are compatible. Hence, by Lemma 17, $(l'', m'') = (l, m) \otimes_\xi ((\mathcal{R}(D), \mathcal{S}(D)) \otimes_{\xi'} (\mathcal{R}(D'), \mathcal{S}(D')))$. Therefore, $\langle D \otimes_{\xi'} D', \xi\rangle : (l, m) \longrightarrow (l'', m'')$. □

All properties of the drag algebra lift without difficulty to the above category of drags, making it a monoidal category, hence showing that our notion of composition is functorial. Monoidal categories play an essential rôle in computational models, such as models of concurrency, because they with compositionality [8]. The use of drags in this area should therefore be investigated. Note that our category is actually symmetric, since the symmetric group action on interfaces is hidden in our definition of a switchboard (in the usual presentation of monoidal categories, the composition operator is not indexed, as it is here; instead, ξ would be normalized as the identity map, and drags identified up to a permutation of their sprouts). Finally, our category should also be traced, since we allow for cycles, but hav'nt proved it. Therefore, it appears to be a traced symmetric monoidal category [32]. Note, on the other hand, that it is not cartesian, since we can observe sharing.

Traced monoidal categories play an important role in semantics: they have in particular be studied long ago by Hasegawa [27] to model the cyclic lambda calculus [1].

Let us now consider the monoidal category of string diagrams, where the objects are natural numbers and the arrows are string diagrams, which plays an important rôle in various areas, in particular in the theory of operads. A string diagram of m inputs and n outputs can be identified with a drag whose roots form a list of length m without repetitions, and sprouts form a set of n vertices labeled by distinct variables $x_1, \ldots, x_n$. Horizontal composition is the same for both monoidal categories. On the other hand, the vertical composition of string diagrams corresponds to the composition $D \otimes_\xi D'$ with a specific switchboard from D to D', which is total, one-way and surjective (and taken as the identity map, as explained above). This switchboard is necessary in our model because the sprouts of D form a multiset (a set in the particular case of string diagrams), rather than a list. So the rôle of the switchboard is to order the sprouts of D so as to materialize their one-to-one correspondence with the list of roots of D'.

String diagrams are graphical representations of computations, initiated by Feynman and Penrose in the realm of theoretical physics, but they are also found in other areas like control theory and concurrency where they arise quite naturally [8, 52]. What we have shown is that the above particular category of string diagrams can be embedded in that of drags. We suspect that many (possibly traced) symmetric monoidal categories found in the literature and their string diagram languages [51], can be embedded in the structure of drags, provided the function symbols of the signature have output arity 1. The model closest to ours seems to be that of cyclic networks [28]. A restriction of our model is that each function symbol has a single output that can be shared. Whether we can have (possibly shared) multiple outputs remains to be investigated, as well as the precise relationship of our models with all these.

12.4. Graphs in models of concurrent and distributed computations

Meseguer and Montanari were the first to give semantics for Petri nets as monoids [39], before their study of their semantics via their underlying categorical structure [40, 41]. This work later developed in order to compare the expressivity of various concurrent models, such as Petri nets under various token management policies, event structures, concurrent transition systems, interaction categories, etc., both on the syntactic level using rewriting logic [38], and on the semantic level using the framework of symmetric monoidal categories and their representation languages [8]. The drag model of computation appears to be one more model of concurrent and distributed computations. Its monoidal structure should allow easy comparisons with these many other models. One tool used for these comparisons is a language of expressions in normal forms that denote concurrent processes [8]. This language is built by composition from a finite set of constructors denoting elementary processes, such as creation of a channel, duplication, etc., in the style of string diagrams. Although the authors point out similarities with the language of Ariola and Klop [2], no precise comparison was made. Our language must therefore have similarities with theirs as well. Whether both have the same expressivity, and whether our normal forms have any relationship to theirs, is beyond the scope of this paper.

13. Conclusion

We have invented drags, a very general class of multi-rooted multigraphs. This has resulted in a new, simple, and very effective model for graph rewriting, based upon a composition operation over drags: given a composed drag $D = G \otimes L$ and a rewrite rule $L \to R$, the resulting drag is nothing but $G \otimes R$, up to removal of inaccessible vertices. At the heart of drag rewriting, therefore, is the decomposition of input drag D into constituents G and L, in a fashion that generalizes term matching.

Once equipped with composition, drags can be seen as the arrows of a monoidal category. This insight reveals unexpected relationships with the semantics of concurrent processes, as well as, more generally, with traced monoidal categories and their string diagrams' languages. These relationships merit in-depth investigation.

Furthermore, it is the first time, to our knowledge, that a composition operation for graph structures with arbitrary cycles has been defined, one which is capable of constructing cycles by composing acyclic graphs. Of course, construction has an inverse operation: decomposition. We have already exploited one possible decomposition of a drag to build total well-founded orders over graphs. Again, investigations of all sorts of graph decompositions that become possible in this framework should be pursued.

The drag model bridges the gap between the cycle-free models of graph rewriting and the general graph model. It generalizes the dag (jungle) models by allowing for cycles without any sort of restriction. But one can ask whether it can capture term rewriting per se? There is at least one simple way in which that can be achieved. Looking at rewriting as a relation between drags, the fact that variables that have multiple occurrences in drag rewrite rules are implemented with a copying or sharing operation is entirely determined by the definition of the switchboard. Using equality in conditions (1,2) of Definition 7 yields sharing. Using isomorphism allows for copying. We could therefore index variables in right-hand sides of rules, so that variables having different indexes could not be shared; on the other hand, variables having the same index could possibly be shared. The absence of index would conventionally be index 0. Variables in left-hand sides would not be indexed, allowing them to be shared if necessary. Details remain to be worked out.

In [15], we use this model to show that termination techniques that have been developed for terms, specifically, the recursive path order, scale to drags. The obtained order is generated from an order on heads (which are recycling drags), in the same way as the recursive path order is generated from an order on function symbols. In turn, the order on heads is generated from an order on their function symbols. This order must enjoy a specific monotonicity property, when heads grow by adding new vertices. This work shows that the difficulty with rewrite orderings for graphs lies in monotonicity with respect to cyclic extensions: breaking, forming or growing cycles is indeed a new phenomenon that shows up with our drag model. This order is not total, and, we believe it cannot be.

We have not started to investigate confluence in our drag model. For terminating term-rewriting systems, confluence reduces to the joinability of critical pairs. This is true too for dag rewrite rules [44], that is, terms with sharing but without cycles. The presence of cycles raises two difficulties: first, unification, but that should be the easy part. The second difficulty is the reduction of local peaks (pairs of rewrites of the form $u \longrightarrow v$ and $u \longrightarrow w$ to critical pairs obtained by overlapping left-hand sides of rules. In the tree case, this reduction depends heavily on the tree structure: two left-hand sides in a term are either: (i) at parallel positions, (ii) at related positions without overlapping, and (iii) at related positions with overlapping. With drags, these situations are no longer exclusive. We believe that the key to solving this problem lies in using the tree decomposition of a drag and Lemma 36.

A pleasant extension of our framework would be to allow for symbols of varyadic arity in order to facilitate rewriting modulo associativity and commutativity. This would be of interest, for example, for algebraic operads. We have not worked this out yet.

Our framework has the potential to be further extended with abstraction and application, in order to obtain a language for λ-terms with sharing and back-arrows, also called *lambda graphs* [1]. So far, we have designed a language for drags inspired from this latter work by introducing the possibility of having many roots. Although introducing also application and abstraction is now easy, generalizing our drag model so as to obtain a meaningful λ-calculus for drags may conceal some unforeseen difficulties.

Acknowledgements

We thank Alfons Geser, who suggested the example that disclosed ill-behaved switchboards, Samuel Mimram for a fruitful discussion about symmetric monoidal categories, and the referees for their manifold suggestions and pointers.

References

[1] Z. M. Ariola, J. W. Klop, Cyclic lambda graph rewriting, in: Proceedings of the Ninth Annual Symposium on Logic in Computer Science (LICS '94), Paris, France, IEEE Computer Society, 1994.
URL http://dx.doi.org/10.1109/LICS.1994.316066

[2] Z. M. Ariola, J. W. Klop, Equational term graph rewriting, Fundam. Inform. 26 (3/4) (1996) 207–240.
URL `http://dx.doi.org/10.3233/FI-1996-263401`

[3] L. Babai, Graph isomorphism in quasipolynomial time [Extended abstract], in: Proceedings of the 48th Annual ACM SIGACT Symposium on Theory of Computing (STOC 2016), Cambridge, MA, 2016.
URL `http://doi.acm.org/10.1145/2897518.2897542`

[4] D. Berwanger, E. Grädel, L. Kaiser, R. Rabinovich, Entanglement and the complexity of directed graphs, Theor. Comput. Sci. 463 (2012) 2–25.
URL `https://doi.org/10.1016/j.tcs.2012.07.010`

[5] F. Bonchi, F. Gadducci, A. Kissinger, P. Sobociński, F. Zanasi, Confluence of graph rewriting with interfaces, in: H. Yang (ed.), Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, vol. 10201 of Lecture Notes in Computer Science, Springer, 2017.
URL `https://doi.org/10.1007/978-3-662-54434-1_6`

[6] G. Bonfante, B. Guillaume, Non-simplifying graph rewriting termination, in: R. Echahed, D. Plump (eds.), Proceedings of the 7th International Workshop on Computing with Terms and Graphs (TERMGRAPH), Rome, Italy, 2013.
URL `https://hal.inria.fr/hal-00921053`

[7] H. J. S. Bruggink, R. Cauderlier, M. Hülsbusch, B. König, Conditional Reactive Systems, in: S. Chakraborty, A. Kumar (eds.), IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2011), vol. 13 of Leibniz International Proceedings in Informatics (LIPIcs), Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 2011.
URL `http://drops.dagstuhl.de/opus/volltexte/2011/3325`

[8] R. Bruni, F. Gadducci, U. Montanari, Normal forms for algebras of connections, Theor. Comput. Sci. 286 (2) (2002) 247–292.
URL `http://dx.doi.org/10.1016/S0304-3975(01)00318-8`

[9] E. Chang, R. Roberts, An improved algorithm for decentralized extrema-finding in circular configurations of processes, Commun. ACM 22 (5) (1979) 281–283.
URL `http://doi.acm.org/10.1145/359104.359108`

[10] H. Cirstea, D. Sabel (eds.), Proceedings Fourth International Workshop on Rewriting Techniques for Program Transformations and Evaluation, WPTE@FSCD 2017, Oxford, UK, September 2017, vol. 265 of EPTCS, 2018 (2018).
URL `http://arxiv.org/abs/1802.05862`

[11] B. Courcelle, Graph rewriting: An algebraic and logic approach, in: Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B), 1990, pp. 193–242.

[12] B. Courcelle, Graph rewriting: A bibliographical guide, in: H. Comon, J. Jouannaud (eds.), Term Rewriting, French Spring School of Theoretical Computer Science, Font Romeux, France, May 17-21, 1993, Advanced Course, vol. 909 of Lecture Notes in Computer Science, Springer, 1993.
URL `https://doi.org/10.1007/3-540-59340-3_6`

[13] N. Dershowitz, Orderings for term-rewriting systems, Theor. Comput. Sci. 17 (1982) 279–301.

[14] N. Dershowitz, J.-P. Jouannaud, Rewrite systems, in: Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics, Elsevier, 1990, pp. 243–320.

[15] N. Dershowitz, J.-P. Jouannaud, GPO: A path ordering for graphs, in: Proceedings of the Sixteenth International Workshop on Termination (WST 2018), Oxford, UK, 2018.

[16] N. Dershowitz, J.-P. Jouannaud, Graph path orderings, in: G. Barthe, G. Sutcliffe, M. Veanes (eds.), LPAR-22. 22nd International Conference on Logic for Programming, Artificial Intelligence and Reasoning, vol. 57 of EPiC Series in Computing, EasyChair, 2018.
URL `https://easychair.org/publications/paper/8DzT`

[17] N. Dershowitz, J.-P. Jouannaud, A simple algebraic framework for graph rewriting, in: M. Fernandez, I. Mackie (eds.), Proceedings of the 7th International Workshop on Computing with Terms and Graphs (TERMGRAPH), Oxford, England, 2018.
URL `https://hal.inria.fr/hal-01853836`

[18] V. Dotsenko, M. Bremner, Algebraic Operads: An Algorithmic Companion, CRC Press, 2016.

[19] H. Ehrig, A. Habel, L. Lambers, F. Orejas, U. Golas, Local confluence for rules with nested application conditions, in: H. Ehrig, A. Rensink, G. Rozenberg, A. Schürr (eds.), Proceedings of the 5th International Conference on Graph Transformations (ICGT 2010), Enschede, The Netherlands, vol. 6372 of Lecture Notes in Computer Science, Springer, 2010.
URL `https://doi.org/10.1007/978-3-642-15928-2_22`

[20] H. Ehrig, M. Pfender, H. J. Schneider, Graph-grammars: An algebraic approach, in: 14th Annual Symposium on Switching and Automata Theory, Iowa City, IA, IEEE Computer Society, 1973.
URL `http://dx.doi.org/10.1109/SWAT.1973.11`

[21] M. Fernández, H. Kirchner, B. Pinaud, Strategic port graph rewriting: an interactive modelling framework, Mathematical Structures in Computer Science to appear.

[22] U. Golas, L. Lambers, H. Ehrig, F. Orejas, Attributed graph transformation with inheritance: Efficient conflict detection and local confluence analysis using abstract critical pairs, Theor. Comput. Sci. 424 (2012) 46–68.
URL `https://doi.org/10.1016/j.tcs.2012.01.032`

[23] J. Goubault-Larrecq, A constructive proof of the topological Kruskal theorem, in: K. Chatterjee, J. Sgall (eds.), Proceedings of the 38th International Symposium on the Mathematical Foundations of Computer Science (MFCS 2013), vol. 8087 of Lecture Notes in Computer Science, Springer, 2013.
URL `http://dx.doi.org/10.1007/978-3-642-40313-2_3`

[24] A. Habel, H. Kreowski, D. Plump, Jungle evaluation, Fundam. Inform. 15 (1) (1991) 37–60.

[25] A. Habel, J. Müller, D. Plump, Double-pushout approach with injective matching, in: H. Ehrig, G. Engels, H. Kreowski, G. Rozenberg (eds.), Selected Papers of the 6th International Workshop on Theory and Application of Graph Transformations (TAGT '98), Paderborn, Germany,

vol. 1764 of Lecture Notes in Computer Science, Springer, 1998.
URL https://doi.org/10.1007/978-3-540-46464-8_8

[26] A. Habel, D. Plump, Unification, rewriting, and narrowing on term graphs, Electr. Notes Theor. Comput. Sci. 2 (1995) 110–117.
URL https://doi.org/10.1016/S1571-0661(05)80187-2

[27] M. Hasegawa, Recursion from cyclic sharing: Traced monoidal categories and models of cyclic lambda calculi, in: P. de Groote (ed.), Typed Lambda Calculi and Applications, Third International Conference on Typed Lambda Calculi and Applications, TLCA '97, Nancy, France, April 2-4, 1997, Proceedings, vol. 1210 of Lecture Notes in Computer Science, Springer, 1997.
URL https://doi.org/10.1007/3-540-62688-3_37

[28] M. Hasegawa, M. Hofmann, G. D. Plotkin, Finite dimensional vector spaces are complete for traced symmetric monoidal categories, in: A. Avron, N. Dershowitz, A. Rabinovich (eds.), Pillars of Computer Science, Essays Dedicated to Boris (Boaz) Trakhtenbrot on the Occasion of His 85th Birthday, vol. 4800 of Lecture Notes in Computer Science, Springer, 2008.
URL https://doi.org/10.1007/978-3-540-78127-1_20

[29] R. Heckel, G. Taentzer (eds.), Graph Transformation, Specifications, and Nets – In Memory of Hartmut Ehrig, vol. 10800 of Lecture Notes in Computer Science, Springer, 2018 (2018).
URL https://doi.org/10.1007/978-3-319-75396-6

[30] J.-P. Jouannaud, H. Kirchner, Completion of a set of rules modulo a set of equations, SIAM Journal of Computing 15 (4) (1986) 1155–1194.

[31] J.-P. Jouannaud, J. Li, Church-Rosser properties of normal rewriting, in: P. Cégielski, A. Durand (eds.), Computer Science Logic (CSL), vol. 16 of LIPIcs, Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, 2012.

[32] A. Joyal, R. Street, D. Verity, Traced monoidal categories, Mathematical Proceedings of the Cambridge Philosophical Society 119 (3) (1996) 447468.

[33] W. Kahl, Categories of coalgebras with monadic homomorphisms, in: M. M. Bonsangue (ed.), Revised Selected Papers of the 12th IFIP WG 1.3 International Workshop on Coalgebraic Methods in Computer Science (CMCS 2014), Colocated with ETAPS 2014, Grenoble, France, vol. 8446 of Lecture Notes in Computer Science, Springer, 2014.
URL https://doi.org/10.1007/978-3-662-44124-4_9

[34] J. Kajiya, Ramified expressions – expanding the syntax of expressions, unpublished report, Microsoft Research (2010).

[35] B. König, D. Nolte, J. Padberg, A. Rensink, A tutorial on graph transformation, in: Graph Transformation, Specifications, and Nets – In Memory of Hartmut Ehrig, vol. 10800 of Lecture Notes in Computer Science, Springer, 2018.
URL https://doi.org/10.1007/978-3-319-75396-6_5

[36] Y. Lafont, Interaction nets, in: F. E. Allen (ed.), Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages, San Francisco, CA, ACM Press, 1990.
URL http://doi.acm.org/10.1145/96709.96718

[37] L. Lambers, H. Ehrig, F. Orejas, Efficient conflict detection in graph transformation systems by essential critical pairs, Electr. Notes Theor. Comput. Sci. 211 (2008) 17–26.
URL https://doi.org/10.1016/j.entcs.2008.04.026

[38] J. Meseguer, Rewriting as a unified model of concurrency, OOPS Messenger 2 (2) (1991) 86–88.
URL http://doi.acm.org/10.1145/127070.127091

[39] J. Meseguer, U. Montanari, Petri nets are monoids, Inf. Comput. 88 (2) (1990) 105–155.
URL https://doi.org/10.1016/0890-5401(90)90013-8

[40] J. Meseguer, U. Montanari, V. Sassone, On the semantics of petri nets, in: R. Cleaveland (ed.), CONCUR '92, Third International Conference on Concurrency Theory, Stony Brook, NY, USA, August 24-27, 1992, Proceedings, vol. 630 of Lecture Notes in Computer Science, Springer, 1992.
URL https://doi.org/10.1007/BFb0084798

[41] J. Meseguer, U. Montanari, V. Sassone, Representation theorems for petri nets, in: C. Freksa, M. Jantzen, R. Valk (eds.), Foundations of Computer Science: Potential - Theory - Cognition, to Wilfried Brauer on the occasion of his sixtieth birthday, vol. 1337 of Lecture Notes in Computer Science, Springer, 1997.
URL https://doi.org/10.1007/BFb0052092

[42] F. Parisi-Presicce, H. Ehrig, U. Montanari, Graph rewriting with unification and composition, in: H. Ehrig, M. Nagl, G. Rozenberg, A. Rosenfeld (eds.), Proceedings of the 3rd International Workshop on Graph-Grammars and Their Application to Computer Science, Warrenton, VA, vol. 291 of Lecture Notes in Computer Science, Springer, 1986.
URL https://doi.org/10.1007/3-540-18771-5_72

[43] B. Pinaud, O. Andrei, M. Fernández, H. Kirchner, G. Melançon, J. Vallet, PORGY : a visual analytics platform for system modelling and analysis based on graph rewriting, in: F. L. Gandon, G. Bisson (eds.), 17ème Journées Francophones Extraction et Gestion des Connaissances, EGC 2017, 24-27 Janvier 2017, Grenoble, France, vol. E-33 of RNTI, Éditions RNTI, 2017.
URL http://editions-rnti.fr/?inprocid=1002327

[44] D. Plump, Critical pairs in term graph rewriting, in: I. Prívara, B. Rovan, P. Ruzicka (eds.), Proceedings of the 19th International Symposium on Mathematical Foundations of Computer Science (MFCS '94), Kosice, Slovakia, vol. 841 of Lecture Notes in Computer Science, Springer, 1994.
URL http://dx.doi.org/10.1007/3-540-58338-6_102

[45] D. Plump, Simplification orders for term graph rewriting, in: I. Prívara, P. Ruzicka (eds.), Proceedings of the 22nd International Symposium on Mathematical Foundations of Computer Science (MFCS'97), Bratislava, Slovakia, vol. 1295 of Lecture Notes in Computer Science, Springer, 1997.
URL https://doi.org/10.1007/BFb0029989

[46] D. Plump, C. Bak, Rooted graph programs, ECEASST 54.
URL http://journal.ub.tu-berlin.de/eceasst/article/view/780

[47] D. Plump, A. Habel, Graph unification and matching, in: J. E. Cuny, H. Ehrig, G. Engels, G. Rozenberg (eds.), Selected Papers of the 5th

International Workshop on Graph Grammars and Their Application to Computer Science, Williamsburg, VA, vol. 1073 of Lecture Notes in Computer Science, Springer, 1994.
URL https://doi.org/10.1007/3-540-61228-9_80

[48] J. Raoult, On graph rewritings, Theor. Comput. Sci. 32 (1984) 1–24.
URL http://dx.doi.org/10.1016/0304-3975(84)90021-5

[49] K. H. Rose, Explicit cyclic substitutions, in: M. Rusinowitch, J. Remy (eds.), Conditional Term Rewriting Systems, Third International Workshop, CTRS-92, Pont-à-Mousson, France, July 8-10, 1992, Proceedings, vol. 656 of Lecture Notes in Computer Science, Springer, 1992.
URL https://doi.org/10.1007/3-540-56393-8_3

[50] G. Rozenberg (ed.), Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations, World Scientific, 1997.

[51] P. Selinger, A survey of graphical languages for monoidal categories, 2010.

[52] J. C. Willems, Open and interconnected systems, IEEE Control Systems Magazine 27 (2007) 46–99.
URL https://ieeexplore.ieee.org/document/4384643