Well-founded Path Orderings for Drags

Nachum Dershowitz, Tel Aviv U., I.E.A. Paris
Jean-Pierre Jouannaud, U. Paris-Saclay
Jianqi Li, Tsinghua U.

1 Introduction

In the introduction to their book [3], Bremmer and Dotsenko write:

> Elements of algebras are trees. Elements of operads are 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.

We are indeed interested in first-order terms with sharing and back-arrows, that is, in rooted multi-graphs each vertex of which is labeled by a function symbol, the arity of which governs the number of neighbors at the end of outgoing edges. These graphs—seen as expressions—may be multiplied to form monomials, multiplication being associative. In turn, these monomials can be added to form polynomials, addition being associative and commutative. These finite graphs and their polynomials are seen as algebraic expressions of an algebraic structure that we refer to generically as an *operad*. Rewriting terms of an operad is very similar to rewriting terms of an algebra. The same questions arise: What rewriting relation do we use? Is there an efficient pattern-matching algorithm? Is rewriting terminating? Is it confluent?

This work describes, first, the design of an algebra of such graphs, and, second, the design of a family of well-founded graph path orderings (in short, GPOs) that can be used to show termination of rewriting in an operad. Note that operadic expressions have structure: graphs, monomials, polynomials. This eases our task, since, once a total order is obtained for graphs, it can be extended to one for monomials, and then to one for polynomials by standard techniques originating from rewriting theory and Gröbner bases.

There is a long tradition of representing graphs by algebraic expressions. Our choice, in the "term tradition", is to view a multi-graph as a non-empty list of roots plus sets of equations $x = f(x_1, \ldots, x_n)$ describing the edges from the vertex x labeled by function symbol f (of arity n) and n vertices $x_1, \ldots, x_n$. This representation has two main advantages, as we shall see: it is invariant under isomorphism of multi-graphs, and there is a natural already available syntax for the case of a single root, which originates from the graph-rewriting community. The generalization of this syntax to multi-rooted multi-graphs is the key that allows us to compute the *dag decomposition* of a graph, which can be seen as a covering of the graph by a directed acyclic graph whose vertices

are multi-rooted cycles of the graph (empty cycles being vertices belonging to no cycle), and whose edges are those of the graph. Having multi-rooted cycles allows the representation to be faithful to sharing, which is not possible with uni-rooted cycles. The dag decomposition of a multi-graph provides us, then, with a natural notion of subgraph, namely, the subdags of the dag decomposition of the multi-graph.

A main novelty of GPO is to build the congruence on expressions representing isomorphic multi-graphs via the subterm rule. Then, the sets of subterms of two congruent expressions must contain pairwise congruent terms. This alone ensures that the order is compatible with the congruence on expressions. GPO has therefore the very same definition as RPO [2], but the computation of the subterms of an expression shares little resemblance with the case of free terms: it is obtained via the dag decomposition of the multi-graph representing the expression, implying that the congruence coincides with multi-graph isomorphism.

GPO has all the properties that are important for its use by "operaders". It is well-founded, total on graph expressions up to graph isomorphism, and has good monotonicity properties with respect to the term structure. It. therefore, serves as a partial answer to the challenge of finding such an order asked of the second author by Bruno Vallette [8]. The answer is still partial because function symbols here are free, while in many operads, the successor vertices of a node labelled by some function f may be permuted in some given way, and possibly in any possible way. On the other hand, drags have a more complex graph structure than in any operad or even properad existing as of today.

2 Finite, Directed, Labeled, Multi-rooted Multi-graphs

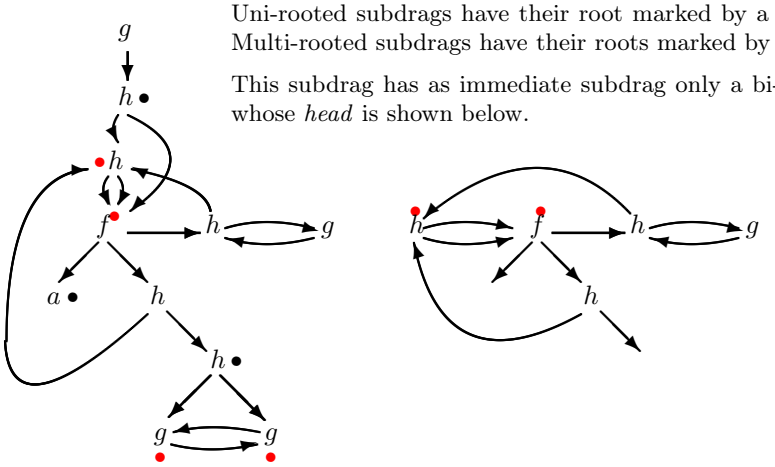
The class of graphs with which we deal here is that consisting of finite directed multi-rooted graphs with labeled vertices and allowing multiple edges between vertices. In the present work, we assume that the outgoing neighbors (vertices at the other end of outgoing edges) are ordered (from left to right, say) and that their number is fixed, depending solely on the label of the vertex: we presuppose a set of function symbols $\mathcal{F}$, whose elements are equipped with a fixed arity. (We have no associative-commutative symbols here.)

We shall call these finite ***d***irected multi-***r***ooted l***a***beled multi-***g***raphs, ***drag***s.

Definition 1 *A* drag *is a tuple* $\langle V, R, L : V \to \mathcal{F}, S : V \to list(V)\rangle$*, where* V *is the finite set of* vertices*,* R *is a list (with possibly repetitions) of vertices of* V *called* roots*,* L *is the* labeling *function, and* S *is the* successor *function, such that* $S(v)$ *is a list whose length equals the arity of* $L(v)$*.*

An isomorphism *from drag* $\langle V, R, L : V \to \mathcal{F}, S : V \to list(V)\rangle$ *to drag* $\langle V', R', L' : V' \to \mathcal{F}, S' : V \to list(V')\rangle$ *is a one-to-one mapping from* V *to* V' *that identifies their respective lists of roots and commutes with their successor functions.*

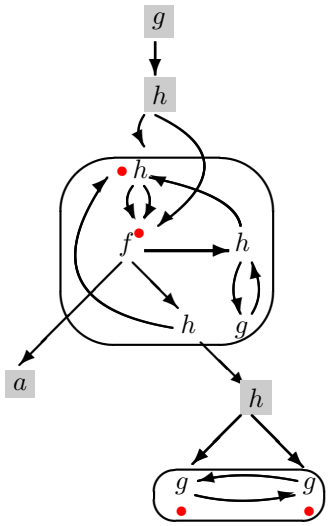
Our goal now is to represent a drag by a dag—whose vertices will be called *heads*—and whose subdags are the dag representations of its *subdrags*. For the drag of Figure 1, the successive subdrags are indicated by black bullets, or pairs of red bullets. The black bullet is used for a subdrag having a unique root.



Uni-rooted subdrags have their root marked by a black bullet. Multi-rooted subdrags have their roots marked by a red bullet.

This subdrag has as immediate subdrag only a bi-rooted cycle whose *head* is shown below.

Figure 1: Example of drag and head.



Grey boxes are used for heads reduced to their root.

Ovals are used for other heads, whose roots are indicated by a red bullet.

Figure 2: Dag decomposition of the drag shown in Figure 1.

A pair of bullets is used for subdrags having two roots. (We don't need more roots in this example.) The roots indicate where we enter the subdrag. In this example, we have pairs of roots for subdrags being headed by a cycle, but it may arise with non-cycles as well: two roots of a subdrag may coincide in case the subdrag has two incoming edges arriving at the same vertex. Whether they coincide or not, these roots form an ordered list, according to which is reached first. Note that the head shown on the picture is a drag with two roots and two outgoing edges that end nowhere, but are numbered according to a fixed search of the head starting at its first root (here, a depth-first search).

The dag-decomposition of this example is shown in Figure 2.

A fundamental property of a drag is that it is characterized—up to isomorphism—by its head and list of subdrags: the dag decomposition of a drag is a faithful representation of the drag. Note a very important fact: this property holds true because heads are multi-rooted. Uni-rooted heads cannot represent what Ariola and Klop dub *horizontal* sharing, in contrast to *vertical* sharing which can be preserved with uni-rooted heads.

We denote by $\widehat{t}$ and ∇t the head and list of subdrags, respectively, of a given

drag t. We further assume a well-founded quasi-order $\geq$ on drag heads. We can now define GPO:

Definition 2 *Given two drags s, t, we define $s \succ t$, iff any of the following holds:*

∇: $\nabla s \geq t$

$\succ$: $\widehat{s} \succ \widehat{t}$ *and* $s \succ \nabla t$

$\doteq$: $\widehat{s} \doteq \widehat{t}$, $s \succ \nabla t$ *and* $\nabla s \succ_{lex} \nabla t$

where $\succ$ is extended to lists of drags on the right by setting $s \succ T$ if $\forall t \in T.\ s \succ t$.

The rest of the paper is devoted to making these intuitions more precise. First, we describe our language of *drag expressions*. In Section 4, we give the equational theory characterizing drag isomorphism, thereby showing the adequacy of our syntax of drag expressions with its semantics in terms of drags. This justifies the abuse of language when using the word "drag" in place of "drag expression". We define the head and list of subexpressions of a drag expression in Section 5, and prove the representation theorem of a drag by its head and list of subterms that is the basis of the dag decomposition of a drag. Section 6 is devoted to the properties of GPO: transitivity, well-foundedness, compatibility with drag isomorphism all follow easily. Totality requires a total precedence on drag heads; we therefore provide one and give examples of ordering drags with this precedence. Monotonicity is more delicate. We prove it in some cases, but finding the most general syntactic condition on drags ensuring monotonicity remains to be achieved.

3 An Algebra of Drags

The main idea is to view the vertices of a drag as variable names, and the incoming edges to the vertex x labeled by f, if they originate from the list of vertices $\overline{x}$, as an equation of the form $x = f(\overline{s})$. A drag becomes then a set of such equations, in which each vertex x appears exactly once on the left. Note that these equations correspond to PROLOG's substitutions with occurs-check. Then, a scoping operator allows to give structure to the whole set of equations, more precisely to reveal the dag structure of the graph by moving the scoping operator down to the leaves of the graph as long as 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. Our binder has a different semantics, which justifies a different binder name. Such expressions are sometimes called term-graphs [6], or also terms with back arrows (edges going back on the path to the root), vertical sharing (edges ending up in some vertex further away from the root) and horizontal sharing (edges going sideways), a terminology due to Ariola and Klop [1].

These languages of graph expressions were used by Ariola and Klop [1], who studied confluence problems in graph rewriting, and by Goubault [4], who studied termination. All these languages however fail to account for horizontal sharing, because term graphs have a unique root. Our language of expressions is a generalization that includes multiple roots. (A syntax for terms with components having multiple roots was suggested informally by James Kajiya [5].)

We now presuppose a set $\mathcal{X}$ of *variables* disjoint from $\mathcal{F}$. A set of *terms* or

expressions, denoted $\mathcal{G}(\mathcal{F},\mathcal{X})$, is defined by the following grammar:

$$s,t := x \in \mathcal{X} \mid f(\overline{s}) \mid [\overline{x = f(\overline{s})}]\, t \mid [\overline{x = f(\overline{s})}](\overline{t}) \quad \text{where:}$$

the *root-length* of an expression is its number of roots: $|x| = |f(\overline{s})| = |[\overline{x = f(\overline{s})}]\, t| = 1$, and $|[\overline{x = f(\overline{s})}](\overline{t})| = \sum_{t \in \overline{t}} |t|$; in a *root-algebraic* term $f(\overline{s})$, $|\overline{s}|$ is equal to the arity of $f \in \mathcal{F}$; and in an *abstraction* $[\overline{x = f(\overline{s})}]\, t$ or in a *multi-abstraction* $[\overline{x = s}](\overline{t})$, the *assignment* $[\overline{x = s}]$ is made of a list of *variable assignments*, whose first arguments are pairwise distinct variables, and the seconds form a list of root-algebraic terms, while the *body* $\overline{t}$ of the multi-abstraction is any non-empty list of terms called *roots* (an abstraction is a multi-abstraction with a single root). Abstraction and multi-abstractions are therefore expressions formed with the abstraction operator "$[-,\dots,-]_{n}-$" of arity $n+1$, where $n = |\overline{x}|$ is usually omitted.

We denote by $\mathcal{V}ar(t)$ the set of variables of the term t. The scoping rules are as expected, an abstraction $[\overline{x = u}](\overline{v})$ binding the variables $\overline{x}$ in both $\overline{u}$ and $\overline{v}$. A *drag expression* is a *ground* expression, that is, one in which no variable occurs free. The set of drag expressions is denoted $\mathcal{G}(\mathcal{F})$. The drag shown in Figure 1 has the following *natural* drag expression (among others), in which binders have all been collected at the top:

$$[x_1 = g(x_2), x_2 = h(x_3,x_4), x_3 = h(x_4,x_4), x_4 = f(x_5,x_6,x_7), x_5 = a, x_6 = h(x_3,x_8),\\ x_8 = h(x_9,x_{10}), x_9 = g(x_{10}), x_{10} = g(x_9), x_7 = h(x_{11},x_3), x_{11} = g(x_7)]x_1$$

Conversely, it is possible to build the drag of Figure 1 when given the above drag expression. As can be easily guessed, the definition proceeds by induction over the syntax.

4 Canonical Drag Expressions

While there is a single way to associate the drawing of a drag to a given drag expression, 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, write all equations of the form $x = f(\overline{y})$ expressing that the vertex x labeled by f has the vertices $\overline{y}$ as successors. Assuming that $\overline{z}$ is the list of variables denoting the non-empty list of roots of the drag, its expression is then $[\overline{x = f(\overline{y})}](\overline{z})$.

Our language of expressions being richer than needed, we introduce an equational theory on drag expressions aiming at capturing drag isomorphism:

Definition 3 *Two drag expressions s,t are* convertible*, written $s \simeq t$, iff they are obtained from each other by using the following equations:*

$$[\overline{x = u}]\,(\overline{t}) = [\overline{z = u\{\overline{x} \mapsto \overline{z}\}}](\overline{t}\{\overline{x} \mapsto \overline{z}\}) \qquad (\alpha)$$

$$[\dots, x = u, \dots, y = s, \dots](\overline{t}) = [\dots, y = s, \dots, x = u, \dots](\overline{t}) \qquad (\rightleftharpoons)$$

$$[\overline{x = u}][\overline{y = v}](\overline{t}) = [\overline{x = u}, \overline{y = v}](\overline{t}) \quad \textit{if } \overline{x} \cap \overline{y} = \varnothing \qquad (\curlyvee)$$

$$[\overline{x = u}, \overline{y = s}](\overline{v}, t, \overline{w}) = [\overline{x = u}](\overline{v}, [\overline{y = s}]t, \overline{w}) \quad \textit{if } \overline{y} \cap \mathcal{FV}ar(\overline{u},\overline{v},\overline{w}) = \varnothing \qquad (\rightsquigarrow)$$

$$[\overline{x = u}, \overline{y = s}]\, f(\overline{v}, t, \overline{w}) = [\overline{x = u}]\, f(\overline{v}, [\overline{y = s}]t, \overline{w}) \quad \textit{if } \overline{y} \cap \mathcal{FV}ar(\overline{u},\overline{v},\overline{w}) = \varnothing \qquad (\rightsquigarrow)$$

$$[\overline{x = u}, y = s]\, y = [\overline{x = u}]\, s \quad \textit{if } y \notin \mathcal{FV}ar(\overline{u}, s) \tag{$\Uparrow$}$$

$$[\overline{x = u}, \overline{y = v}, y = w]\, t = [\overline{x = u}, y = [\overline{y = v}]\, w]\, t \quad \textit{if } \overline{y} \cap \mathcal{FV}ar(\overline{u}, t) = \varnothing \tag{$\rightsquigarrow$}$$

$$[\overline{x = u}, \overline{y = s}](\overline{t}) = [\overline{x = u}](\overline{t}) \quad \textit{if } \overline{s} \cap \mathcal{FV}ar(\overline{u}, \overline{t}) = \varnothing \tag{$\ominus$}$$

$$[\,]\, t = t \tag{$\ominus$}$$

where z denotes always a fresh variable and $\overline{z}$ a list of pairwise different fresh variables.

Convertibility is extended to lists of drag expressions in the obvious way.

The latter seven equations can be oriented from left to right to give a set of terminating rewrite rules. They serve pushing variable assignments downwards as long as is permitted by the scoping rule for abstractions (rules $\rightsquigarrow$) and our graph semantics (sharing must be preserved). This process may require merging contiguous abstractions (rule $\curlyvee$), as well as removing useless variable assignments from abstractions (rules $\Uparrow$ and $\ominus$). Using these rules may require renaming bound variables (equation α), and permuting variable assignments (equation $\rightleftharpoons$). We illustrate below the computation of the normal form expression, by rewriting modulo the equations, of the natural expression given earlier for the drag pictured in Figure 1. We give the input expression, the sequence of rules used, and the resulting expression:

$$[x_1 = g(x_2), x_2 = h(x_3, x_4), x_3 = h(x_4, x_4), x_4 = f(x_5, x_6, x_{10}), x_5 = a, x_6 = h(x_3, x_7),$$
$$x_7 = h(x_8, x_9), x_8 = g(x_9), x_9 = g(x_8), x_{10} = h(x_{11}, x_3), x_{11} = g(x_{10})]\, x_1$$
$$\Uparrow, \rightsquigarrow, \Uparrow, \rightsquigarrow, \Uparrow, \ominus, \rightsquigarrow, \rightsquigarrow, \rightsquigarrow, \rightsquigarrow, \ominus, \rightsquigarrow, \Uparrow, \rightsquigarrow, \ominus$$
$$g([x_3 = h(x_4, x_4), x_4 = f(a, h(x_3, [x_8 = g(x_9), x_9 = g(x_8)]h(x_8, x_9)),$$
$$[x_{10} = h(x_{11}, x_3), x_{11} = g(x_{10})]x_{10})]h(x_3, x_4))$$

The normal form of a drag expression is called *canonical* when variables in an abstraction are ordered according to a fixed traversal of the drag (for example, depth first search). This canonical form is quite economic: all its bound variables denote vertices of the graph that have several incoming edges, hence are shared and must therefore be named. Further, every variable assignment in an abstraction is located as close as possible from its intended body, that is, at the vertex which is the closest common ancestor of the vertices denoted by its bound variables.

The canonical form of an expression has a remarkable property: it is possible to read off the list of subterms of a drag expression from its canonical form: these are the ground subexpressions of the normal form expression that are headed by a function symbol or a (possibly) list of variables for the cycles, that is (with our favorite depth-first traversal order):

$$[x_3 =, x_4 = f(a, h(x_3, [x_8 = g(x_9), x_9 = g(x_8)]h(x_8, x_9)), \ldots)]h(x_3, x_4)\,;$$
$$[x_3 =, x_4 = f(a, h(x_3, [x_8 = g(x_9), x_9 = g(x_8)]h(x_8, x_9)), \ldots)](x_3, x_4)\,;$$
$$a\,;\ \ [x_8 = g(x_9), x_9 = g(x_8)]h(x_8, x_9)\,;$$
$$[x_8 = g(x_9), x_9 = g(x_8)](x_8, x_9).$$

We end up this section with the main property of convertibility.

Lemma 4 *Two drags are isomorphic iff their drag expressions are convertible, that is, iff their canonical forms are identical up to variable renaming.*

5 Dag Decomposition of a Drag

A term is entirely defined by its root symbol and its ordered set of immediate subterms, which are terms themselves. The same is true of drags, but the root must be replaced by a more complex structure, their head. The recursive decomposition of a drag into a head and a list of subterms is its *dag decomposition*. We give the flavor of these definitions via simple examples.

Consider for example the terms $f(a,b,b)$, $f(b,a,b)$, and $f(b,b,a)$, in which b is shared. All three will have the same two subterms, a and $[x = b](x,x)$, and, hence, different heads: $f(\square_1^1, \square_2^1, \square_2^2)$, $f(\square_1^1, \square_2^1, \square_1^2)$ and $f(\square_1^1, \square_1^2, \square_2^1)$. On the other hand, the same terms in which b is not shared will have different lists of three subterms, respectively, $(a,b,b), (b,a,b)$ and (b,b,a) and the same head $f(\square_1^1, \square_2^1, \square_3^1)$. In these expressions, $\square_i^j$ denotes therefore the jth occurrence of the ith subterm of a term in its head. In case the subterm is a cycle, its jth occurrence corresponds to the jth root of the cycle, sharing of a cycle being more complex than sharing of a term. (Remember than roots may be repeated.)

In the example of Figure 1, we obtain the successive heads which can be observed from the dag decomposition of Figure 2. Their expressions are:

$g(\square_1^1)$;
$h(\square_1^1, \square_1^2), [x = h(y,y), y = f(\square_1^1, h(x, \square_2^1), z), z = h(z', x), z' = g(z)](x,y)$;
a; $[x = g(x'), x' = g(x)]h(x,x')$

We now need to characterize drag heads as syntactic entities: they are either dags of height 1 or multigraphs whose all roots are accessible from all inner vertices. In both cases, the succession of leaves of a drag head d corresponds to the succession of subdrags of the drag t whose head is d. These subdrags will be abstracted by constants; only their order and repetitions matters. Head expressions can therefore be described by the grammar:

$$s \stackrel{\text{def}}{=} f(\overline{v}) \mid [x = f(\overline{v}), \overline{y = f(\overline{v})}](\overline{x}) \quad \text{where } f \in \mathcal{F} \text{ and } \forall v \in \overline{v}.\, v \in \mathcal{X} \cup \{\square_i\}$$

We shall identify the drag head $f(\overline{v})$ with $[x = f(\overline{v})]x$, so that all expressions have the same format. Note that the normal form used for drag heads differs slightly (but can easily be obtained) from that of canonical expressions.

Lemma 5 *Two drag expressions s, t are convertible, hence, isomorphic, iff they have convertible heads and lists of subterms, and therefore, the same* dag decomposition.

We can furthermore name the variables in a head's abstraction by the order in which they occur in our favorite depth-first search starting at the first root, using $\{x_i\}_i$ for variable names. Such heads are called *normal*. Then, two normal head expressions are convertible iff they are identical. This now allows us to build a total well-founded order on heads. Given a normal drag head $t \stackrel{\text{def}}{=} [\overline{x = s}](\overline{y})$ such that $\overline{y} \subseteq \overline{x}$ and $y_1 = x_1$, we define (a) the list $\overline{z} \stackrel{\text{def}}{=} \overline{y}, \overline{x}$ of length n; (b) the interpretation $[\![t]\!] \stackrel{\text{def}}{=} \langle n, \overline{t} \rangle$ such that $\forall i.\, t_i = s_j \Leftrightarrow z_i = x_j$.

We compare interpretations in the order $(>, \mathrm{RPO}_{lex})_{lex}$; this is our precedence $\gtrdot$, where the recursive path ordering RPO is generated by an arbitrary total well-founded order on the signature $\mathcal{F} \cup \{\square_i\}_i \cup \{x_i\}_i$. Note that RPO serves for comparing heads that are interpreted by lists of terms of the same length. Since these terms are ground (bound variables being considered as constants) and the signature is total, the comparison never fails.

6 Properties of GPO

We now state the main properties of GPO implying that it is a monotonic well-founded ordering of the set of expressions compatible with convertibility.

Theorem 6 *The order $\succ$ is transitive, compatible with convertibility, well-founded, and total when the precedence on $\mathcal{F} \cup \{\Box_i\}_i \cup \{x_i\}_i$ is total and monotonic with respect to replacement of drag expressions.*

Our order is defined on ground expressions, and is then monotonic. It is possible to define the order on expressions with free variables by the usual methods (least stable extension, or its approximation obtained by considering free variables as new constants that cannot be compared in the precedence to any other symbol). Monotonicity is then satisfied in case the context does not capture a free variable, that is, $s \succ t$ implies $u[s] \succ u[t]$ provided a variable free in s, t does not become bound in $u[s]$. Note also that in operads, variables are linear because outgoing edges (the variables) are in bijection with the roots, and pairwise independent. This may help to find good conditions for monotonicity in case a variable capture occurs, and also imply preservation of totality.

There is one more important question left for investigation: the action of the symmetric group on certain graph vertices labelled by some given function symbol f. We believe that the case where any action is possible, that is, the successors of a vertex labelled by f can be permuted arbitrarily, can be solved by replacing the RPO style comparison by Rubio's AC-RPO style comparison [7]. This is likely to work provided the graph is in *flattened form*, hence restricting the sharing of vertices labelled by f. Likewise, its successor vertices labelled by a variable should not be shared either, so that flattening becomes compatible with the symmetric group action. All properads we know of satisfy these conditions.

Acknowledgements. This Work is supported by NSFC grants 61272002.

References

[1] Zena M. Ariola and Jan Willem Klop. Cyclic lambda graph rewriting. In *Proc. LICS*, pp 416–425. IEEE Computer Society, 1994.

[2] Nachum Dershowitz. Orderings for term-rewriting systems. *Theor. Comput. Sci.*, 17:279–301, 1982.

[3] Vladimir Dotsenko and Murray Bremner. Algebraic operads: An algorithmic companion. Chapman & Hall/CRC Press, ISBN 9781482248562, April 2016.

[4] Jean Goubault-Larrecq. A constructive proof of the topological Kruskal theorem. In *Proc. MFCS*, *LNCS 8087*, pp 22–41. Springer, 2013.

[5] James Kajiya. Ramified expressions – expanding the syntax of expressions. Unpublished report, Microsoft reasearch, 2010.

[6] Detlef Plump. Simplification orders for term graph rewriting. In *Proc. MFCS*, *LNCS 1295*, pp 458–467. Springer, 1997.

[7] Albert Rubio. A fully syntactic AC-RPO. *Inf. Comput.*, 178(2):515–533, 2002.

[8] Bruno Valette. personnal communication, 2013.