Well-founded Path Orderings for Operads

Nachum Dershowitz[1,2], Jean-Pierre Jouannaud[3,4], and Jianqi Li[5]

1 **School of Computer Science, Tel Aviv University, Tel Aviv, Israel**
2 **Institut d'Études Avancées de Paris, Paris, France**
3 **Université Paris-Saclay**
4 **LIX, École Polytechnique, Palaiseau, France**
5 **Tsinghua University, Beijing, China**

Abstract

The definition herein of the Graph Path Ordering (GPO) on certain graph expressions is inspired by that of the Recursive Path Ordering (RPO), and enjoys all those properties that have made RPO popular, in particular, well-foundedness and monotonicity on variable-free terms.

We are indeed interested in a generalization of algebraic expressions called operadic expressions, which are finite graphs each vertex of which is labelled by a function symbol, the arity of which governs the number of vertices it relates to in the graph. These graphs are seen here as terms with sharing and back-arrows. Operadic expressions are themselves multiplied (an associative operation) to form monomials, which are in turn summed up (an associative commutative operation) to form polynomials. Operadic expressions and their polynomials occur in algebraic topology, and in various areas of computer science, notably concurrency and type theory. Rewriting basic operadic expressions is very much like rewriting algebraic expressions, while rewriting their monomials and polynomials is very much like the Gröbner basis theory. GPO provides an initial building block for computing with operadic expressions and their polynomials.

1 Introduction

In the introduction to their book [7], Bremmer and Dotsenko write:

> Elements of algebras are trees. 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-flavoured methods to those structures.

We are indeed interested in first-order terms with sharing and back-arrows, that is, in graphs whose each vertex is labelled by a function symbol, the arity of which governs the number of vertices it relates to in the graph. 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, whose algebraic representation as terms we call *operadic expressions*, 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 occur: What rewriting relation do we use? Is there an efficient pattern matching algorithm? Is rewriting terminating? Is it confluent?

This paper describes the design of well-founded orderings that can be used to show termination of rewriting in an operad. Note that operadic expressions are structured: graphs, monomials, polynomials. This will somewhat ease our task, since, once a (possibly total) order is obtained for graphs, it can be easily extended to one for monomials, and then to one

for polynomials by standard techniques originating from Gröbner bases. Further, in case the graph is a collection of disconnected components, an order on connected components can be extended to the entire graph by using a multiset extension. Now, if the edges of the graph are not oriented, it is easy again to replace them by a pair of oriented edges. Finally, if the graph has no distinguished root, we can supply for root a new vertex whose set of successors is the set of vertices of the original graph whose set of predecessors is either empty or contains itself. These reductions are indeed the most straightforward one may think of, there are indeed better ones. So, the main question we address here is the design of a well-founded ordering for finite, connected, rooted, oriented graphs.

The most popular ordering for ordinary terms is the recursive path ordering (RPO) [5], which is recursively generated from an order on the set of function symbols, called the "precedence". One reason for its utility is that it enjoys all the good properties one may expect: it is compatible with the term structure; it is well-founded when the precedence is well founded; it is increasing when the precedence increases; and – last but not least – its behavior is intuitive and can be easily and efficiently implemented. One main aspect of its definition is that comparing two terms with the same head symbol reduces to a comparison of their respective subterms via a lexicographic or multiset extension of the order on their subterms organized as lists or multisets, respectively. The type of extension assigned to each head symbol is called its *status*.

Following a long tradition initiated by Rose [22] and continued by Ariola and Klop [1,2], who studied confluence, and later Goubault [10], who studied termination, we provide syntax to finite, rooted oriented graphs via a single binding operator. The variables bound by the fixpoint operator represent vertices in the graph, while their occurrences in expressions represent edges ending up in that node. Such expressions are sometimes called term-graphs [20], 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), or mixed sharing, a terminology due to Ariola and Klop [1]. To represent a graph as an expression, the expression must be ground, that is, all variables, considered as names for vertices, must be bound. We then design an ordering named GPO for expressions which is similar in spirit to RPO. GPO can finally be easily extended to arbitrary expressions of an operad as already sketched.

The main novelty of GPO is to build a certain congruence on graph expressions 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, but the computation of the subterms of an expression shares little resemblance with the case of free terms.

This idea actually germinated from a careful reading of Rubio's work [23]. Rubio introduced two novelties in his fully syntactic associative commutative path ordering ACPO, that he presents as equally important to ensure monotonicity with respect to equivalence classes of terms: a new status, and an extended set of subterms of a flattened term for terms headed by an associative commutative operator. We indeed believe that the second innovation is the key to the definition of a well-behaved RPO-like ordering working on equivalence classes of terms such as associativity and commutativity. The status innovation is merely a question of efficiency, prompted by the implementation of the order.

GPO has all the properties that are important for its use by "operaders". It is well-founded, total on graph expressions up to a congruence contained in graph isomorphism, and monotonic with respect to the term structure. It is therefore an answer to the question of finding such an order asked to the second author by Bruno Vallette [15].

2 Directed Rooted Labelled Graphs

Graph rewriting was considered very early on as the appropriate framework for shared rewriting. Initial attempts to develop a theory of graph rewriting were based on category theory, the result of a rewrite step being formalized as a (single or double) pushout in a category [8, 9, 21]. Following these lines, Detlef Plump developed support tools for both confluence [19] and syntactic termination [20] in so called *term-graphs*, which are terms with shared subterms.

Independently, Kurihara and Ohuchi, as well as others, developed a more operational framework, *marked terms*, which also captures sharing in directed acyclic graphs (dags) [14]. In [17], Ohlebusch generalizes their technique to handle marked graphs, and shows that rewriting marked graphs generalizes both term rewriting and graph rewriting.

Rose initiated the modern view of dags as terms with sharing, by adopting a syntactic representation by pairs made of a term s and an explicit idempotent substitution σ from a subset X of the variables of s, acting as a binder for the variables in X which are replaced by terms without variables from X. The substitution is then seen as a pointer from the vertices in the dag named by the variables in X to vertices further down in the dag [22]. This modern view was actually already implicit in Huet's unification algorithm [11], as well as in Nelson's congruence-closure algorithm [16]. Let us go back to the future!

This view was developed by Ariola and Klop to account for a larger class of directed graphs than dags, including back-arrows to capture potentially infinite structures. This is accomplished by using a fixpoint operator μ, allowing one to repeatedly unfold terms [1]. Since Ariola and Klop were actually interested in the pure lambda calculus, hence in lambda-graphs, they already had at their disposal a fixpoint operator that could be used for representing back-arrows. The syntax obtained, however, did not include arrows between sibling nodes, or more generally, between two vertices neither of which is an ancestor of the other [2]. The representation of arbitrary finite graphs came shortly after, by adding an explicit fixpoint operator μ in the syntax of terms to express substitutions which may be non-idempotent [3]. We follow here this tradition of using a calculus of explicit substitutions to represent finite graphs as term expressions with arbitrary sharing.

The class of graphs with which we deal here is that consisting of finite directed rooted graphs with labelled 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 shall call these ***d***irected ***r***ooted l***a***beled ***g***raphs, ***drag***s.

We presuppose a set of function symbols $\mathcal{F}$, whose elements are equipped with a fixed arity. (We have no associative-commutative symbols here.)

▸ **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 set of *vertices*, $r \in V$ is the *root*, L is the *labelling* function, and S is the *successor* function, such that, $S(v)$ is a list whose length equals the arity of $L(v)$.

The successor function defines both a *successor* relationship and an *ancestor* relationship. We shall use both. We denote by D_v the drag D in which the root has been moved to v, so that D is D_r if r is its root,

and by $Succ^i(v)$ the i-th successor of v in D. The word *root* is used both for the vertex r of the drag, and for its label $L(r)$.

By convention, the set of vertices is an initial segment of the natural numbers.

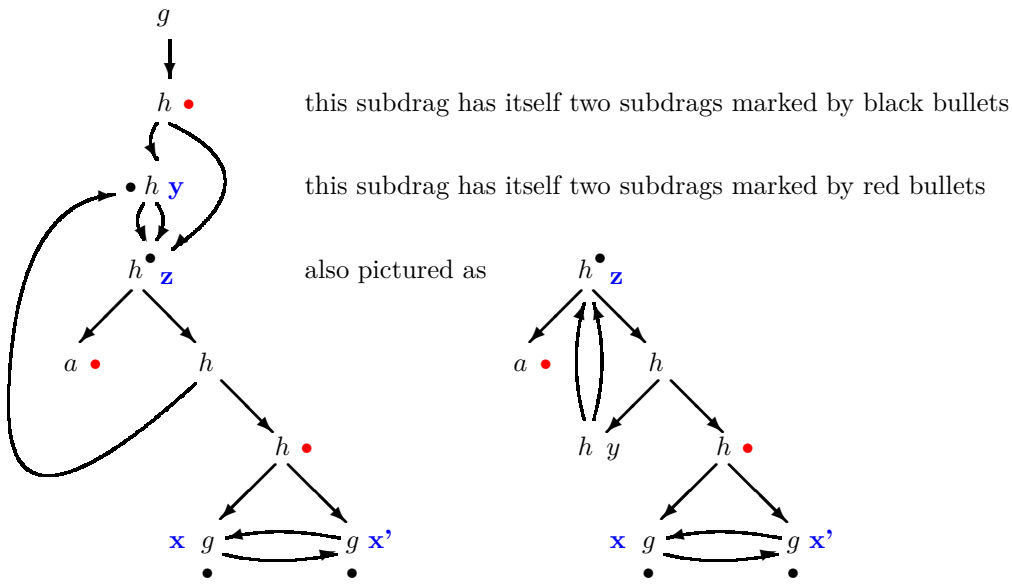

Figure 1 The drag expression $g([y = h(z,z), z = h(a,h(y,[x = g(x'), x' = g(x)]h(x,x')))]h(y,z))$ and its drag above on left. Immediate subdrags are marqued by bullets, alternating red and black. The drag displayed on the right has expression $[z = h(a,h(y,[x = g(x'), x' = g(x)]h(x,x'))), y = h(z,z)]z$. Note that variable z appears now before y in the assignement, a change explained later. This drag is an immediate subdrag of the drag above marked with a red bullet, but *not* of the one, of expression $[y = h(z,z), z = h(a,h(y,[x = g(x'), x' = g(x)]h(x,x')))]y$, marked with a black bullet.

2.1 Expressions

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, using $[\overline{x = s}]t$ for "let $\overline{x=s}$ in t":

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

where: in a *root-algebraic* term $f(\overline{s})$, the number of arguments is equal to the arity of $f \in \mathcal{F}$; and in an *abstraction* $[\overline{x = s}]t$, 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* t is any term. 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.

Such expressions are also called *mu-terms* in the literature when the (maximal) fixpoint operator μ is used instead of the brackets we are using. The scoping rules are those expected:

▸ **Definition 2** (Free and Bound Variables). A variable x occurs *free* in a term t iff any of the following holds (otherwise, it is *bound*):

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

We denote by $\mathcal{FV}ar(t)$ the set of free variables of the expression t.

A *drag expression* is a *ground* expression, that is, an expression in which no variable occurs free. The set of drag expressions is denoted $\mathcal{G}(\mathcal{F})$.

Substitutions operate on (open) expressions with free variables very much like on higher-order terms. *Renamings* are particular substitutions generated by bijections between finite subsets of $\mathcal{X}$. We denote a substitution of domain $\{x_i\}_i$ as in $\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$, and use the postfix notation for its application.

2.2 Canonical expressions

While there is a single way to associate the drawing of a drag to a given drag expression, that we do not explicit here for lack of space, there are many ways to associate a drag expression to a given drag. We will investigate these maps later, but introduce here an equational theory on drag expressions aiming at approximating graph isomorphism for drags. We need a couple of preliminary definitions:

▸ **Definition 3.** Given an assignment $\overline{x = s}$, we define:

1. the *occurs-check* binary relation $<_{oc}$ as the least transitive relation such that $y <_{oc} z$ in $\overline{x = s}$ if $y = u \in \overline{x = s}$, $z = v \in \overline{x = s}$ and $y \in \mathcal{V}ar(v)$;
2. the set $\circlearrowleft y(\overline{x = s})$ of *occurs-check* variables of y in $\overline{x = s}$, or simply $\circlearrowleft y$, assuming its second argument given, as the set of variables $\{z : z <_{oc} y <_{oc} z \text{ in } \overline{x = s}\}$.

Note that our definition of the occurs-check relationship is made simpler than usual thanks to the assumption that terms in $\overline{s}$ are not variables. Note also that a variable x which occurs only once in an assignment has an empty set of occurs-check variables: the occurs-check relation may not be reflexive.

▸ **Definition 4.** We say that a variable y is *proper* in an abstraction $u = [\overline{x = s}]t$ if $y \in \mathcal{FV}ar(t)$ or $z <_{oc} y$ for some variable z such that $z = u \in \overline{x = s}$ and z is proper in $[\overline{x = s}]u$. We denote by $\mathcal{PV}ar(u)$ the set of proper variables in u. The abstraction $u = [\overline{x = s}]t$ is *reduced* if $\mathcal{PV}ar(u) = \overline{x}$.

Improper variables are those variables in an abstraction which can be dropped, the abstraction remaining ground. Removing improper variables is one case of canonization:

▸ **Definition 5.** Two drag expressions s, t are *convertible*, written $s \simeq t$, iff they are obtained from each other by alpha-conversion, permutation of the equality assignments in their abstractions, moving abstractions up and down, and removing useless variable assignements:

$$
\begin{array}{rcll}
[\overline{x = s}]\, t & = & [\overline{z = s\{\overline{x} \mapsto \overline{z}\}}]\, t\{\overline{x} \mapsto \overline{z}\} & (\alpha) \\
 & \text{if} & \overline{z} \text{ is a vector of } |\overline{x}| \text{ pairwise distinct fresh variables} & \\
{[\overline{x = s}, x = s, \overline{y = u}, z = v, \overline{z = v}]}\, t & = & [\overline{x = s}, z = v, \overline{y = u}, x = s, \overline{z = v}]\, t & (\rightleftharpoons) \\
[\overline{x = s}]\, [\overline{y = u}]\, t & = & [\overline{x = s}, \overline{z = u\{\overline{y} \mapsto \overline{z}\}}]\, t\{\overline{y} \mapsto \overline{z}\} & (\curlyvee) \\
 & \text{if} & \overline{z} \text{ is a vector of } |\overline{x}| \text{ pairwise distinct fresh variables} & \\
{[\overline{y = u}, x = s, \overline{z = v}]}\, t & = & [\overline{y = u}, \overline{z = v}]\, t & (\ominus) \\
 & \text{if} & x \notin \mathcal{FV}ar(\overline{u}, \overline{v}, t) & \\
[\,]\, t & = & t & (\ominus) \\
[\overline{x = s}]\, f(\overline{u}, t, \overline{v}) & = & f(\overline{u}, [\overline{x = s}]\, t, \overline{v}) & (\downarrow) \\
 & \text{if} & \overline{x} \cap \mathcal{FV}ar(\overline{u}, \overline{v}) = \varnothing & \\
{[\overline{x = s}, y = u, \overline{z = v}]}\, y & = & [\overline{x = s}, \overline{z = v}]\, u & (\Downarrow) \\
 & \text{if} & y \notin \mathcal{FV}ar(\overline{s}, u, \overline{v}) &
\end{array}
$$

The latter four equations can be oriented from left to right to give a set of terminating rewrite rules. This allows us to defined several kinds of normal form for a drag expression t: $t\curlyvee$ for the normal form of t wrt ($\curlyvee$); $t{\downarrow_{\ominus}}$ for the normal form of t wrt ($\ominus$); $t{\downarrow_{red}}$ for the normal form of t wrt ($\curlyvee$, $\ominus$), yielding *reduced* expressions; and $t{\downarrow_{\alpha}}$ for the *canonical* form of t (modulo alpha-conversion) obtained by

(i) taking the (unique) normal form of t wrt ($\downarrow$,$\Downarrow$,$\curlyvee$,$\ominus$);

(ii) listing the variables $\overline{x}$ of an abstraction $[\overline{x = s}]\,t$ according to the traversal of the tree t first, and by recursively traversing the tree s when encountering for the first time a variable x such that $x = s \in \overline{x = s}$.

All drag expressions listed in example 1 are canonical.

▸ **Lemma 6.** *Two convertible drag expressions have canonical expressions that are variable renamings of each other.*

Proof. The rules ($\downarrow$,$\Downarrow$,$\curlyvee$,$\ominus$) are terminating modulo (α, $\rightleftharpoons$). Likewise, they are compatible with both. It therefore suffices to show that the critical pairs are joinable modulo (α, $\rightleftharpoons$) [12], which is routine check. ◂

Convertibility is of course included into graph isomorphism, but the above set of axioms defining convertibility is not a complete axiomatisation of graph isomorphism. Restricting the abstractions to be of the form $[\overline{x = s}]x$, where terms in $\overline{s}$ are either root-algebraic terms whose all subterms are variables in $\overline{x}$, or abstractions or the same form, would yield, we believe, a complete axiomatization of isomoprhism for the class of graphs that can be represented in our syntax.

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

3 The Graph Path Ordering

Like any path ordering, our *Graph Path Ordering (GPO)* makes use of a precedence, statuses, and a notion of subterm that we are going to introduce in turn.

3.1 Precedence

In practice, it may be useful to include some semantics in the precedence to compensate the purely syntactic nature of the recursive path ordering, an idea originating from [13, 18]. This is especially important for drag expressions, as we shall see later.

▸ **Definition 7.** A *precedence ordering* for an ordering $\succ$ is a quasi-ordering $\succeq$ of the set of drag expressions that has the following properties:

(i) well-foundedness: $\succ$ is well-founded,

(ii) congruence: $\doteq$ is a congruence on terms,

(iii) weak-monotonicity: $\forall f \in \mathcal{F} \cup \{[\,]\}, \overline{s}, \overline{t},\ \ u \succ v$ implies $f(\overline{s}, u, \overline{t}) \succeq f(\overline{s}, v, \overline{t})$.

Note that u, v being drag expressions, they have no free variable, hence cannot be usefully bound above.

For example, defining informally the *root* of a term as the root label of the associated drag, we can associate a precedence to any given well-founded quasi-ordering $\geq_{\mathcal{F}}$ of the signature $\mathcal{F}$ (usually called a *precedence*, hence our name for $\succeq$) by $t \succeq v$ iff $root(t) \geq_{\mathcal{F}} root(v)$.

Since terms are equivalent modulo $\doteq$ iff they have roots equivalent in $=_{\mathcal{F}}$, this equivalence is clearly a congruence, and $\succeq$ is weakly monotonic with respect to any ordering $\succ$.

Here are our first two assumptions:

$\geq$ is a precedence ordering of expressions such that $s =_\alpha t$ implies $s \doteq t$	(p1)
$\geq$ is weakly monotonic with respect to the order $\succ$ to be defined	(p2)

Although (p2) seems quite strong, precedences on function symbols satisfy (p2).

3.2 Statuses

An *extension* or *status* is a mapping that lifts a binary relation over some set S to a binary relation over some finite data structure of elements of S. Well-known statuses of interest here are the *lexicographic* status (left-to-right, right-to-left, etc.) for lists of elements of S of bounded length, and the *multiset* status for finite multisets of elements of S. All these statuses satisfy the following important properties: monotonicity with respect to the ordering on S; preservation of totality for the lexicographic status, and of totality in permutation classes of terms for the multiset status; preservation of well-foundedness; and preservation of quasi-orders. See, eg, [6].

We use the postfix notation $_t$ for the status associated with a drag expression t, assuming that $_s$ and $_t$ are identical statuses if $s \doteq t$, this is our assumption (s1).

3.3 Heads and subterms

A term is entirely defined by its head (or root) and its ordered set of immediate subterms, which are terms themselves. Our purpose is to define a notion of head (in general different from the root) and a notion of immediate subterm for a drag expression, which have the same property. These notions of head and subterm are the basis of a tree decomposition of a drag that will be used to define our path ordering.

We first define the immediate subterms of an expression t, whose list is denoted by ∇t.

▸ **Definition 8.** The list ∇t of immediate subterms of a reduced expression t is defined as

$$\begin{aligned}
\nabla f(\overline{t}) &\stackrel{\text{def}}{=} \overline{t} \\
\nabla[\overline{x = s}]\, f(\overline{t}) &\stackrel{\text{def}}{=} ([\overline{x = s}]\overline{t})\curlyvee \\
&\quad \text{where binders are extended to operate on sets in the obvious way.} \\
\nabla[\overline{x = s}]\, x &\stackrel{\text{def}}{=} \{[\overline{x = s}]v \in \nabla^+[\overline{x = s}]u : x = u \in \overline{x = s},\ [\overline{x = s}]v \text{ maximal},\ \mathcal{FV}ar(v) \cap \circlearrowleft x = \varnothing\}
\end{aligned}$$

Note that expressions are maintained in reduced normal form by our definition.

To show that the recursive call terminates in the third case, it suffices to observe that all variables in the body v of the abstraction are strictly smaller than x in the occur check relationship.

▸ **Example 9.** We illustrate the computation of all subterms of the drag expression $s \stackrel{\text{def}}{=} g([y = h(z,z), z = h(a, h(y, [x = g(x'), x' = g(x)]h(x,x')))]h(y,z))$ given at Figure 1. The first rule applies and gives $s_1 \stackrel{\text{def}}{=} [y = h(z,z), z = h(a, h(y, [x = g(x'), x' = g(x)]h(x,x')))]h(y,z)$. The second rules now applies and gives $s_2^1 \stackrel{\text{def}}{=} [y = h(z,z), z = h(a, h(y, [x = g(x'), x' = g(x)]h(x,x')))]y$ and $s_2^2 \stackrel{\text{def}}{=} [y = h(z,z), z = h(a, h(y, [x = g(x'), x' = g(x)]h(x,x')))]z$. We continue computing the successive subterms of s_2^1, those of s_2^2 being the same.

This time, the third rule applies, hence we need to compute the subterms of $[y = h(z,z), z = h(a, h(y, [x = g(x'), x' = g(x)]h(x,x')))]h(z,z)$ and filter our those that do not satisfy the variable condition. Applying the second subterm rule, we get two identical candidates equal to $[y = h(z,z), z = h(a, h(y, [x = g(x'), x' = g(x)]h(x,x')))]z$ which do not satisfy the variable condition. We continue with the first one, and use the third rule and

get the new candidate $[y = h(z,z), z = h(a, h(y, [x = g(x'), x' = g(x)]h(x,x')))]h(a, h(y, [x = g(x'), x' = g(x)]h(x,x')))$. Applying now the second rule, we get two candidates, $[y = h(z,z), z = h(a, h(y, [x = g(x'), x' = g(x)]h(x,x')))]a$ which reduces to a that satisfies the variable condition, and $[y = h(z,z), z = h(a, h(y, [x = g(x'), x' = g(x)]h(x,x')))]h(y, [x = g(x'), x' = g(x)]h(x,x'))$, who does not since it has y for free variable. Applying the second rule again, we get two candidates $[y = h(z,z), z = h(a, h(y, [x = g(x'), x' = g(x)]h(x,x')))]y$, which does not satisfy the variable condition, and $[y = h(z,z), z = h(a, h(y, [x = g(x'), x' = g(x)]h(x,x')))][x = g(x'), x' = g(x)]h(x,x')$ which reduces to $[x = g(x'), x' = g(x)]h(x,x')$ that satisfies the variable condition and is therefore a new subterm. In turn, this subterm will give us two immediate subterms $[x = g(x'), x' = g(x)]x$ and $[x' = g(x), x = g(x')]x'$, which differ from each other by a variable renaming.

The tree decomposition obtained from these calculations is pictured at Figure 2. Note that different drags may have the same tree decomposition, since sharing of subterms has disappeared. Avoiding this is possible, it suffices to define one subdrag with each cycle, for example the first encountered in the computation, which is also the first in the depth-first search of the drag expression.

This example shows that taking the transitive closure of the set of immediate subterms of $[\overline{x = s}]u$ in the third case is necessary, in case some immediate subterm has variables from $\circlearrowleft x$, requiring to search for a deeper subterm satisfying the variable condition. Then, the maximality condition (with respect to membership in the set of immediate subterms) will ensure that we keep only the immediate subterms. This maximality condition assumes that membership in that set is a well-founded relationship, which is proved at Lemma 11.

▸ **Lemma 10.** *Let $[\overline{x = s}]u \in \nabla[\overline{x = s}]t$. Then, $\mathcal{PV}ar(u) \subseteq \mathcal{PV}ar(t)$ if t is not a variable, and $\mathcal{PV}ar(u) \subset \mathcal{PV}ar(t)$ if t is a variable.*

Proof. The statement is clear if t is not a variable. In case t is a variable, then all free variables of $\mathcal{V}ar(u)$ are smaller in $<_{oc}$ than the variables in $\mathcal{V}ar(t)$ since u is a subterm of some s_i not containing any variable in $\circlearrowleft t$. Hence, at least the variable t has disappeared. ◂

▸ **Lemma 11.** *Let $t \triangleright s$ if $s \in \nabla t$. Then, $\triangleright$ is well-founded.*

Proof. The proof is immediate by induction on the set of proper variables of a term and use of Lemma 10. ◂

We can now define the head of a drag expression:

▸ **Definition 12.** The head $\widehat{t}$ of a reduced drag expression t is obtained by replacing the ith-subdrag s_i of the list of subdrags of t by a new constant $\Box_i$, called *hole*.

Heads are therefore trees of depth 1 or drags whose root is accessible from all inner vertices. To the price of introducing new variables (which we could have included in the convertibility relationship), canonical head expressions are described by the grammar:

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

Note that $\overline{v}$ is a list of boxes in case of a root. We shall consider here the drag head $f(\overline{v})$ as $[x = f(\overline{v}]x$, so that all expressions are now of the same form.

As announced, a reduced drag expression t is entirely characterized by its head $\widehat{t}$ and its list of subterms.

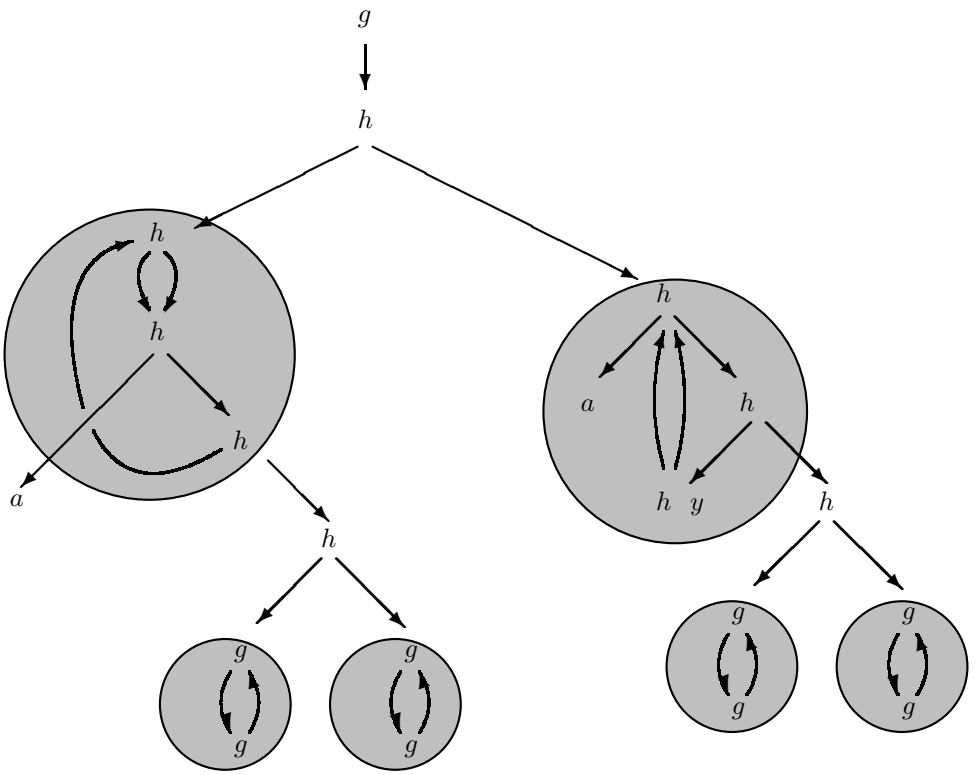

Figure 2 The tree decomposition of the drag given at Figure 1. Heads that differ from roots are inside a dark circle.

▸ **Lemma 13.** *Two reduced drag expressions s, t are convertible iff their heads are equivalent modulo variable renaming and permutations of variable assignemants as well as their lists of subdrag expressions.*

JPJ: This lemma is not true as it is. It becomes true with the other definition of subterms and head given later.

Proof. Note first that s, t have the same heads provided they have the list of same subterms. Assume s, t are convertible. Since they are reduced, be reduced, we only need to show, for the only if direction, that the statement holds true for $(\alpha, \rightleftharpoons, \downarrow, \Downarrow)$. The result is trivially true for the first two. It is true as well for $(\downarrow)$: the list of subterms of $[\overline{x = s}]f(\overline{u}, t, \overline{v})$ is the list of reduced terms (variables assignements disappear for $\overline{u}, \overline{v}$ which share no variable with $\overline{x}$) $\overline{u}$, $[\overline{x = s}]t$, and $\overline{v}$ in this order, that is, the same list of subterms as $f(\overline{u}, [\overline{x = s}]t, \overline{v})$. For $(\Downarrow)$, the list of subterms of $[\overline{x = s}, y = u, \overline{z = v}]y$ is by definition the same as the list of subterms of $[\overline{x = s}, y = u, \overline{z = v}]u$ in case $\mathcal{FV}ar(u) \cap \circlearrowleft y = \varnothing$, which is the case here. Then, reducing $[\overline{x = s}, y = u, \overline{z = v}]u$ under the condition that $y \notin \mathcal{FV}ar(\overline{s}, u, \overline{v})$ yields $[\overline{x = s}, \overline{z = v}]u$, and therefore both expressions have the same lists of subterms.

For the converse, we get that s, t have the same tree decomposition modulo variable renaming and permutations of variable assignements, as seen at Figure 2. ◂

3.4 GPO

In this section, we introduce the ordering GPO for expressions which operate on congruence classes of terms modulo conversion.

▸ **Definition 14.** Given two ground terms t, v, we define $t \succ v$ iff any of the following holds:

∇: $\nabla t \succeq v$

$\triangleright$: $t \triangleright v$ and $t \succ \nabla v$

$\doteq$: $t \doteq v$, $t \succ \nabla v$ and $\nabla t \succ_t \nabla v$

where $\succ$ is extended (asymmetrically) as usual to multisets on one side.

3.5 Examples

To define a precedence, we are going to use the *root* of a drag expression, defined by cases as follows: $root(f(\overline{t})) = f$, $root([\overline{x = f(\overline{s})}]g(\overline{t})) = g$ and $root([\overline{x = f(\overline{s})}, x = g(\overline{t})]x) = g$.

We first assume that the precedence compares roots of terms in a given well-founded order of the set of function symbols. Statuses are then associated with function symbols as usual.

Consider the goal $t \stackrel{\text{def}}{=} [x = f(x, x'), x' = g(a, b)]\, x \succ [y = f(f(y, z), z'), z = a, z' = b]\, y \stackrel{\text{def}}{=} s$, and assume that the order $\succeq$ compares roots with he order $a \succeq g \succeq f \succeq b$. Note that t has a single immediate subterm $g(a, b)$ while s has two, namely a, b. Since s, t have the same root f, Case $\doteq$ applies and we first recursively compare $t \succ \{a, b\}$. The comparison $t \succ b$ succeeds immediately by Case $\triangleright$, since $f \triangleright b$. On the other hand, since the root a of a is bigger than the root f of t, the comparison must proceed by Case ∇, which yields $g(a, b) \succ b$, which succeeds immediately by using Case ∇ again. We are then left comparing the subterms $\{g(a, b)\}$ and $\{a, b\}$, which succeeds easily with a multiset extension of the order $\succ$.

Consider now the converse goal $[y = f(f(y, z), z'), z = a, z' = b]\, y \succ [x = f(x, x'), x' = g(a)]\, x$.

Comparing roots is not enough here, since we cannot expect $\{a, b\}$ to be bigger than $g(a, b)$. We will instead compare the head of the drag expressions, obtained for our purpose here, by replacing their subdrags by a new constant $\Box$. Here, the head of s is $[y = f(f(y, z), z'), z = \Box, z' = \Box]\, y$ and that of t is $[x = f(x, x'), x' = \Box]\, x$.

Let now $\cdot\succeq$ compare first the root function symbol, in the same order as before, then the total length of the cycles of the head of the terms, which is equal to 2 for s, and 1 for t. Since $(f, 2)$ is bigger than $(f, 1)$, we now have $s \cdot\!\succ t$. The comparison is therefore by Case $\cdot\succ$, and we need to prove that $s = [y = f(f(y, z), z'), z = a, z' = b]\, y \succ g(a, b)$. Heads are now $(f, 2)$ and $(g, 0)$, hence we proceed again by Case $\triangleright$, yielding the subgoals $s = [y = f(f(y, z), z'), z = a, z' = b]\, y \succ a, b$, which both succeed by Case ∇.

3.6 Basic properties of GPO

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

▸ **Lemma 15** (Subterm). $\triangleright \subseteq \succ$.

Proof. By (∇), $t \succ \nabla t$. ◂

▸ **Lemma 16** (Subterm Transitivity). *Let $t \succeq v$. Then $t \succ \nabla v$.*

Proof. By induction on the definition of GPO. If $t = v$, we conclude by Case ∇ of GPO. If $\nabla t \succeq v$, then $\nabla t \succeq \nabla v$ follows by the induction hypothesis, and we conclude by Case ∇ of GPO. Otherwise, $t \succ \nabla v$ by definition of GPO. ◂

▸ **Lemma 17** (Transitivity). $\succ$ *is transitive.*

Proof. Assuming that $u \succ t \succ s$, we show that $u \succ s$ by induction on (u, t, s) compared in $(\triangleright)_{mul}$. Apart from the induction ordering itself, the proof is standard. ◂

3.6.1 Compatibility

We show compatibility of GPO with convertibility congruence classes. This result is of course important since it justifies our way of building compatibility into a path ordering via a careful definition of subterms instead of using a normal-form representation of congruence classes of terms.

▸ **Theorem 18** (Compatibility). *Assuming that the precedence is compatible with convertibility (p3), GPO is a strict ordering compatible with convertibility.*

Proof. Anti-symmetry follows from well-foundedeness proved next, and transitivity from Lemma 17. Compatibility follows from the fact that two alpha-convertible terms have alpha-convertible lists or multisets of subterms by Lemma 13. ◂

Note how easy the compatibility proof is with our method, while the only existing related order, by Goubault, is incompatible with alpha-convertibility [10]. There is indeed a rule in his order making a bound variable bigger than any ground term. It must be said that his purpose is different, and that his mu-terms have an interpretation very different from ours, in terms of formal languages.

3.6.2 Well-Foundedness

▸ **Lemma 19.** *Assume terms ∇t are SN. Then, $t = g(\overline{t})$ is SN.*

Proof. By definition of the SN predicate, t is SN iff all its reducts v are SN. We prove that a given reduct v of t is SN by induction on the triple $\langle t, \nabla t, v\rangle$, compared in the lexicographic composition $\ggg \stackrel{\text{def}}{=} (\succeq, \succ_t, \triangleright)_{lex}$. For this lexicographic composition to be well-founded, each component relation must be well-founded. First, $\succeq$ is well-founded by definition of a precedence. Second, $\triangleright$ is well-founded by Lemma 11. Third, all terms in ∇t are SN by assumption. And since statuses preserve well-foundedness, $\succ_t$ is well-founded on lists, sets or multisets of elements taken from the set of elements smaller or equal to elements in ∇t. Therefore, $\ggg$ is well-founded. There are two kinds of reducts whose triple are smaller than $\langle t, \nabla t, v\rangle$: the reducts of t smaller than v in $\triangleright$. And the reducts w of some v such that the pair $\langle v, \nabla v\rangle$ is smaller strictly than the pair $\langle t, \nabla t\rangle$ in the order $(\succeq, \succ_t)_{lex}$. By "hypothesis (n)", we mean to apply the induction hypothesis in the case where the n-th component of the triple has decreased.

We distinguish the three usual cases:

1. Case ∇: $\nabla t \succ v$. Since ∇t is SN by assumption, v is SN by definition of the SN predicate.
2. Case $\succ$: $t \succ v$ and $t \succ \nabla v$. By definition of $\triangleright$, $v \triangleright \nabla v$. Therefore ∇v is SN by hypothesis (3). Let now w be an arbitrary reduct of v. Then, w is SN by hypothesis (1). It follows that v is SN by definition of the SN predicate.

3. Case $\doteq$: $t \doteq v$, $t \succ \nabla v$, and $\nabla t \succ_t \nabla v$. By the same token as above, ∇v is SN. Let w be an arbitrary reduct of v. Then, w is SN by induction hypothesis (2). Hence, v is SN by definition of the SN predicate, and we are done. ◂

▸ **Theorem 20.** *Under the assumption that the precedence is well-founded, GPO is well-founded.*

Proof. Let $t = f(\overline{t})$. We show that t is SN by induction on $\triangleright$. Since $t \triangleright \nabla t$ by definition of $\triangleright$, terms in ∇t are SN by assumption. By Lemma 19, t is SN. The result follows. ◂

Note that this well-foundedness proof does not rely on the particular form of expressions, but only on the definition of the SN predicate, the well-foundedness of the precedence, the well-foundedness of subterm membership, and the preservation of well-foundedness by statuses.

3.6.3 Monotonicity

We now turn to monotonicity. Although the proof is not that difficult, it appears to be the one – with compatibility – that departs most from the traditional proofs. This is not surprising though: the monotonicity proofs in the AC case are very specific, too.

▸ **Theorem 21** (Monotonicity)**.** *GPO is monotonic over reduced expressions.*

Proof. Assuming that $u \succ v$ for some ground terms u, v, we show that $U = f(\overline{s}, u, \overline{t}) \succ f(\overline{s}, v, \overline{t}) = V$ for all $f \in \mathcal{F} \cup \{[\,]\}$ and terms $\overline{s}, \overline{t}$. Note that $U \geq V$ by the weak monotonicity property of the precedence $\gtrsim$ with respect to $\succ$. We therefore show first that $U \succ \nabla V$ by induction on pairs (U, V) compared in $(\triangleright, \triangleright)_{mul}$. There are three cases:

1. U, V are not abstractions. The result is clear, since the respective immediate subterms of U, V are $\overline{s}, u, \overline{t}$ and $\overline{s}, v, \overline{t}$; hence we can conclude by Case ∇ for all immediate subterms of V but v, for which we need in addition to use Lemma 16.
2. $U = [\overline{x = s}]u$ and $V = [\overline{x = s}]v$. Since u, v are graph expressions, they have no free variables, and therefore u, v are not variables. Hence, $\nabla V = [\overline{x = s}]\nabla v$. Since $u \succ \nabla v$ by Case ∇ and Lemma 16, we conclude that $U \succ \nabla V$ by Case ∇.
3. $U = [\overline{x = s}, y = u]w$ and $V = [\overline{x = s}, y = v]w$. By assumption, y is a proper variable in both U, V. (Note that we implicitly use the fact that the order of bound variables is irrelevant.) There are three cases:
   - **a.** $w \notin \mathcal{X}$. Then $\nabla V = [\overline{x = s}, y = v]\nabla w$. Let $w' \in \nabla w$. Then, $[\overline{x = s}, y = u]w' \in \nabla U$, and by the induction hypothesis, $[\overline{x = s}, y = u]w' \succ [\overline{x = s}, y = v]w'$. We conclude that $U \succ [\overline{x = s}, y = v]w'$ by Case ∇.
   - **b.** $w = x \neq y$. Let $x = s \in \overline{x = s}$. Then, $\nabla V = [\overline{x = s}, y = v]\nabla s$. Let $w' \in \nabla s$. Then, $[\overline{x = s}, y = u]w' \in \nabla U$, and by the induction hypothesis, $[\overline{x = s}, y = u]w' \succ [\overline{x = s}, y = v]w'$. We conclude that $U \succ [\overline{x = s}, y = v]w'$ by Case ∇.
   - **c.** $w = y$. Since u, v are ground expressions, $\nabla U = [\overline{x = s}, y = u]u$ and $\nabla V = [\overline{x = s}, y = v]v$. Now, $U \succ [\overline{x = s}, y = u]u$ by Lemma 15. By induction hypothesis, $[\overline{x = s}, y = u]u \succ [\overline{x = s}, y = u]v$. By the induction hypothesis again $[\overline{x = s}, y = u]v \succ [\overline{x = s}, y = v]v$. We conclude by transitivity.

If $U \gtrsim V$, we are done. Otherwise, $U \doteq V$ and we need that $\nabla U \geq_U \nabla V$, which we have by the monotonicity property of statuses. ◂

The proviso that expressions are reduced is necessary: otherwise, only weak monotonicity would be satisfied. This is so because the expressions u, v could be equalities $x = u$ and $x = v$

in the assignment part of the abstractions U and V. Were x an improper variable, then U and V would be equivalent even if $u \succ v$.

3.7 Totality

We start by showing that the ordering increases when the precdence increases:

▸ **Lemma 22** (Incrementality). *If $\succeq$ or $\doteq$ increase, then $\succ$ increases.*

Proof. Routine induction on the definition of GPO. ◂

Incrementality is very important for practice, because it helps automating the search for a precedence given a set of rules to be shown terminating. But it of course not enough to ensure that GPO can be made total, an important objective.

▸ **Lemma 23** (Totality). *If $\succeq$ is total on drag heads modulo convertibility, then $\succ$ is total on equivalence classes of drags modulo convertibility.*

Proof. Let t and v be two ground terms which are not equivalent. We prove that they are comparable by induction on the relation $\triangleright$. By the induction hypothesis, t and ∇v are comparable, as well as v and ∇t. If $\nabla v \succeq t$ or $\nabla t \succeq v$, then $v \succ t$ or $t \succ v$, respectively, by Case ∇ of GPO. Otherwise, $t \succ \nabla v$ and $v \succ \nabla t$. Since $\succeq$ is total on drags, there are three cases, of which two are symmetrical. If $t \succcurlyeq v$ or $v \succcurlyeq t$, then $t \succ v$ or $v \succ t$, respectively, by Case $\succcurlyeq$ of GPO. We are therefore left with the case where $t \doteq v$, for which we simply need to show that $\nabla t \succ_f \nabla v$ or $\nabla v \succ_v \nabla t$. By the induction hypothesis, all terms in these – be they lists, multisets or sets – are pairwise comparable. Since t and v are not equivalent, ∇t and ∇v are not equivalent either by Lemma 13. Since statuses preserve totality of non-equivalent lists or multisets of terms, then one multiset must be strictly bigger than the other, ensuring that one term is bigger than the other, ending the proof. ◂

3.8 Ordering drag heads

Our goal has therefore now shifted towards a total ordering on drag heads, allowing to define a precedence ofndrags by comparing their heads. Because canonical head expressions are of the form $[x = f(\overline{v}), \overline{y = f(\overline{v})}]x$, they are entirely characterized by their ordered list of variable assignements, which will be denoted by $\overline{x} = f(\overline{u})$ in the sequel, the first variable assignement defining the root of the drag head.

To get a total order on drags head it suffices to consider the variables in $\overline{x}$ as new constants $v_1, \ldots, v_n$, to order the augmented signature $\mathcal{F} \cup \{\Box_i\}_i \cup \{v_i\}_i$ with some total order $>$, and define the order on heads by comparing the tuple $\overline{s}\{\overline{x} \mapsto \overline{v}\}$ in the lexicographic extension of RPO. Since the order on the signature is total, so is RPO on expressions in $\overline{s}\{\overline{x} \mapsto \overline{v}$, and therefore on the tuples themselves.

We need to verify that the obtained precedence $[\overline{x = s}]x \succcurlyeq [\overline{y = t}]y$ iff $\overline{s}\{\overline{x} \mapsto \overline{v}\}$ $(>_{RPO})_{lex}$ $\overline{t}\{\overline{y} \mapsto \overline{v}\}$ satisfies our assumptions (p1,p2,p3), of which (p1) is trivially true. (p2) holds as well, since given two head expressions u, v, the expressions $f(\overline{s}, u, \overline{t})$ and $f(\overline{s}, v, \overline{t})$ have the same heads. For (p3), this follows from the fact that convertible canonical heads are identical up to variable renaming by Lemma 13.

We also need to satisfy (s1), which amounts to have the same status for two equivalent drag heads. This is true by assumption, provided we use the lexicographic status for all drag heads in order to GPO to be total on drag expressions.

3.9 Sharing

As shown at Figure 2, the tree decomposition does not reflect sharing. In order to keep sharing in the tree decomposition, we need to avoid duplicating subterms when they are shared. This impacts of course the definition of heads, since we need to be able to reconstruct a drag from its head and list of subterms in order to keep Lemma 13.

To define subterms, we introduce a second argument in the definition of the function ∇, a list L of terms used to collect all subterms found so far which is initialized with the empty list. Using "::" for list concatenation, we get:

▸ **Definition 24.** The list $\nabla(t, L)$ of immediate shared subterms of a reduced expression t wrt to L is defined as

$$\begin{aligned}
\nabla(f(\overline{t}), L) &\stackrel{\text{def}}{=} L :: \overline{t} \\
\nabla([\overline{x = s}]\, f(\overline{t}), L) &\stackrel{\text{def}}{=} Filter((([\overline{x = s}]\overline{t})\curlyvee, L) :: L \\
&\quad \text{where binders are extended to operate on sets in the obvious way.} \\
\nabla([\overline{x = s}]\, x, L) &\stackrel{\text{def}}{=} Filter(\{[\overline{x = s}]v \in \nabla^+([\overline{x = s}]u,) : x = u \in \overline{x = s},\ [\overline{x = s}]v \text{ maximal}, \\
&\qquad \mathcal{FV}ar(v) \cap \circlearrowleft x = \varnothing\}, L) :: L
\end{aligned}$$

where $Filter(L', L)$ returns the list L' from which all elements already in L have been removed.

Heads need be redefined as well. They becomes trees of depth one, or drags whose roots are accessible from inner vertices. Indeed, a head can now have multiple variable-bodies instead of a unique variable as body. The meaning of a head $[\overline{x = u}](\overline{y}$, with $\forall y \in \overline{y}\, y \in \overline{x}$, is that the cycle $[\overline{x = u}](\overline{y}$ has many incoming edges, one for each variable in the list $\overline{y}$, and the order of these variables is that of the incoming edges in the depth-first search of the covering of the initial drag from its root. The reader can verify that Lemma 13 still holds.

Comparing heads can now be achieved in a similar way as before, but the list of terms associated to a head depends on its multiple roots: first, the root values in the order given by the body of the head, then the terms given by the remaining assignements, in the same depth-first search order. As one may guess, a particular order is not important, we just choosed the most straighforward one.

The variant of our order can orient Toyama's example $f(x, x, x) \to f(x, x, x)$, in which x is shared. The heads of the lefthand and righthand sides are both equal to $f(\square_1, \square_2, \square_3)$ while the lists of subterms are $[y = x]y, a, b$ for the lefthand side and the single expression $[y = x]y$ for the righthand side. Here, x is of course considered as a constant.

4 Conclusion

We have designed an ordering which can be used for comparing drags. Several questions remain to be investigated, among which the extension to terms with free variables, and the precise conditions, apart from groundness, under which the order is monotonic.

We took advantage of the hierarchical structure of polynomials to reduce the design of an order for (quite general) operads to an order on finite directed rooted labelled graphs. Allowing some nodes in the graph to be associative, or associative commutative would surely be technically difficult (this is more general than polynomials over finite graphs). We believe however that it can be done by reusing the techniques developed by Rubio [23].

Our ordering has the potential to be further extended with an abstraction and an application operator as done in CPO [4], and obtain an ordering over λ-terms with sharing and

back-arrows, also called *lambda graphs* in [1]. This is not carried out here, and we presume that the problem conceals some difficulties.

In general, designing rewrite orders compatible with some equational theory satisfying some properties would be worth investigating. Our order is such one, and, as Rubio's ACRPO [23], it based on building the desired congruence on expressions via a definition of subterms which is invariant in the congruence. We believe that this is the right way to go.

References

1 Zena M. Ariola and Jan Willem Klop. Cyclic lambda graph rewriting. In *Proceedings of the Ninth Annual Symposium on Logic in Computer Science (LICS '94), Paris, France, July 4-7, 1994*, pages 416–425. IEEE Computer Society, 1994.

2 Zena M. Ariola and Jan Willem Klop. Equational term graph rewriting. *Fundam. Inform.*, 26(3/4):207–240, 1996.

3 Zena M. Ariola and Jan Willem Klop. Lambda calculus with explicit recursion. *Inf. Comput.*, 139(2):154–233, 1997.

4 Frédéric Blanqui, Jean-Pierre Jouannaud, and Albert Rubio. The computability path ordering. *Logical Methods in Computer Science*, 11(4), 2015.

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

6 Nachum Dershowitz and Jean-Pierre Jouannaud. Rewrite systems. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*, pages 243–320. Elsevier, 1990.

7 Vladimir Dotsenko and Murray Bremner. Algebraic operads: An algorithmic companion, 2015. Research monograph, version 0.999, xvii+365pp.

8 Hartmut Ehrig, Reiko Heckel, Martin Korff, Michael Löwe, Leila Ribeiro, Annika Wagner, and Andrea Corradini. Algebraic approaches to graph transformation – part II: single pushout approach and comparison with double pushout approach. In Grzegorz Rozenberg, editor, *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations*, pages 247–312. World Scientific, 1997.

9 Hartmut Ehrig, Michael Pfender, and Hans Jürgen Schneider. Graph-grammars: An algebraic approach. In *14th Annual Symposium on Switching and Automata Theory, Iowa City, Iowa, USA, October 15-17, 1973*, pages 167–180. IEEE Computer Society, 1973.

10 Jean Goubault-Larrecq. A constructive proof of the topological Kruskal theorem. In Krishnendu Chatterjee and Jirí Sgall, editors, *38th International Symposium MFCS 2013, 2013.*, volume 8087 of *Lecture Notes in Computer Science*, pages 22–41. Springer, 2013.

11 Gérard Huet. *Unification dans les langages d'ordre 1, ...,ω*. PhD thesis, Université Paris 7, Paris, France, 1976.

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

13 Sam Kamin and Jean-Jacques Lévy. Two generalizations of the recursive path ordering. Unpublished report, Department of Computer Science, University of Illinois at Urbana-Champaign, 1980.

14 Masahito Kurihara and Azuma Ohuchi. Modularity in noncopying term rewriting. *Theor. Comput. Sci.*, 152(1):139–169, 1995.

15 Jean-Louis Loday and Bruno Vallette. Algebraic operads, 2012. Springer Verlag.

16 Greg Nelson. *Techniques for Program Verification*. PhD thesis, Stanford University, Stanford, California, USA, June 1981.

17 Enno Ohlebusch. A uniform framework for term and graph rewriting applied to combined systems. *Inf. Process. Lett.*, 73(1-2):53–59, 2000.

18 David A. Plaisted, 1979. Personal communication.

19 Detlef Plump. Critical pairs in term graph rewriting. In Igor Prívara, Branislav Rovan, and Peter Ruzicka, editors, *Mathematical Foundations of Computer Science 1994, 19th International Symposium, MFCS'94, Kosice, Slovakia, August 22 – 26, 1994, Proceedings*, volume 841 of *Lecture Notes in Computer Science*, pages 556–566. Springer, 1994.

20 Detlef Plump. Simplification orders for term graph rewriting. In Igor Prívara and Peter Ruzicka, editors, *22nd International Symposium MFCS'97, 1997*, volume 1295 of *Lecture Notes in Computer Science*, pages 458–467. Springer, 1997.

21 Jean-Claude Raoult. On graph rewritings. *Theor. Comput. Sci.*, 32:1–24, 1984.

22 Kristoffer Høgsbro Rose. Explicit cyclic substitutions. In Michaël Rusinowitch and Jean-Luc Remy, editors, *CTRS-92, July 8-10, 1992*, volume 656 of *Lecture Notes in Computer Science*, pages 36–50. Springer, 1992.

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