Coalgebraic semantics for derivations in logic programming

Ekaterina Komendantskaya[1] and John Power[2]

[1] Department of Computing, University of Dundee, UK *
[2] Department of Computer Science, University of Bath, UK **

Abstract. Every variable-free logic program induces a $P_f P_f$-coalgebra on the set of atomic formulae in the program. The coalgebra p sends an atomic formula A to the set of the sets of atomic formulae in the antecedent of each clause for which A is the head. In an earlier paper, we identified a variable-free logic program with a $P_f P_f$-coalgebra on Set and showed that, if $C(P_f P_f)$ is the cofree comonad on $P_f P_f$, then given a logic program P qua $P_f P_f$-coalgebra, the corresponding $C(P_f P_f)$-coalgebra structure describes the parallel and-or derivation trees of P. In this paper, we extend that analysis to arbitrary logic programs. That requires a subtle analysis of lax natural transformations between *Poset*-valued functors on a Lawvere theory, of locally ordered endofunctors and comonads on locally ordered categories, and of coalgebras, oplax maps of coalgebras, and the relationships between such for locally ordered endofunctors and the cofree comonads on them.
Key words: Logic programming, SLD-resolution, Coalgebra, Lawvere theories, Lax natural transformations, Oplax maps of coalgebras.

1 Introduction

In the standard formulations of logic programming, such as in Lloyd's book [18], a first-order logic program P consists of a finite set of clauses of the form

$$A \leftarrow A_1, \ldots, A_n$$

where A and the A_i's are atomic formulae, typically containing free variables, and where $A_1, \ldots, A_n$ is understood to mean the conjunction of the A_i's: note that n may be 0.

SLD-resolution, which is a central algorithm for logic programming, takes a goal G, typically written as

$$\leftarrow B_1, \ldots, B_n$$

where the list of B_i's is again understood to mean a conjunction of atomic formulae, typically containing free variables, and constructs a proof for an instantiation of G from substitution instances of the clauses in P [18]. The algorithm uses

* The work was supported by the Engineering and Physical Sciences Research Council, UK; Postdoctoral Fellow research grant EP/F044046/2.

** John Power would like to acknowledge the support of SICSA for an extended visit to Dundee funding this research.

Horn-clause logic, with variable substitution determined universally to make the first atom in G agree with the head of a clause in P, then proceeding inductively.

A running example of a logic program in this paper is as follows.

Example 1. Let ListNat denote the logic program

$$\begin{aligned} \texttt{nat(0)} &\leftarrow \\ \texttt{nat(s(x))} &\leftarrow \texttt{nat(x)} \\ \texttt{list(nil)} &\leftarrow \\ \texttt{list(cons x y)} &\leftarrow \texttt{nat(x), list(y)} \end{aligned}$$

The program involves variables `x` and `y`, function symbols `0`, `s`, `nil` and `cons`, and predicate symbols `nat` and `list`, with the choice of notation designed to make the intended meaning of the program clear.

Logic programs resemble, and indeed induce, transition systems or rewrite systems, hence coalgebras. That fact has been used to study their operational semantics, e.g., [4, 6]. In [15], we developed the idea for variable-free logic programs. Given a set of atoms At, and a variable-free logic program P built over At, one can construct a $P_f P_f$-coalgebra structure on At, where P_f is the finite powerset functor: each atom is the head of finitely many clauses in P, and the body of each of those clauses contains finitely many atoms. Our main result was that if $C(P_f P_f)$ is the cofree comonad on $P_f P_f$, then, given a logic program P qua $P_f P_f$-coalgebra, the corresponding $C(P_f P_f)$-coalgebra structure characterises the parallel and-or derivation trees of P: see Section 2 for a definition and for more detail.

Modulo a concern about recursion, which can be addressed by extending from finiteness to countability, one can construct a variable-free logic program from an arbitrary logic program by taking all ground instances of all clauses in the original logic program. The resulting variable-free logic program is of equivalent power to the original one, but one has factored out all the analysis of substitution that appears in SLD-resolution. So, in order to model the substitution in the SLD-resolution algorithm, in this paper, we extend our coalgebraic analysis of logic programming from variable-free logic programs to arbitrary logic programs. In particular, we study the relationship between coalgebras for an extension of $P_f P_f$ and the coalgebras for the comonad induced by it.

There have been several category theoretic models of logic programs and computations, and several of them have involved the characterisation of the first-order language underlying a logic program as a *Lawvere theory*, e.g., [2, 4, 5, 14], and that of most general unifiers (mgu's) as *equalisers*, e.g., [3] or as *pullbacks*, e.g., [5, 2]. We duly adopt those ideas here, see Section 3.

Given a signature Σ of function symbols, let $\mathcal{L}_\Sigma$ denote the Lawvere theory generated by Σ. Given a logic program P with function symbols in Σ, we would like to consider the functor category $[\mathcal{L}_\Sigma^{op}, Set]$, extending the set At of atoms in a variable-free logic program to the functor from $\mathcal{L}_\Sigma^{op}$ to Set sending a natural

number n to the set $At(n)$ of atomic formulae with at most n variables generated by the predicate symbols in P. One can extend any endofunctor H on Set to the endofunctor $[\mathcal{L}_\Sigma^{op}, H]$ on $[\mathcal{L}_\Sigma^{op}, Set]$ that sends $F : \mathcal{L}_\Sigma^{op} \to Set$ to the composite HF. So we would then like to model P by the putative $[\mathcal{L}_\Sigma^{op}, P_f P_f]$-coalgebra $p : At \longrightarrow P_f P_f At$ that, at n, takes an atomic formula $A(x_1, \ldots, x_n)$ with at most n variables, considers all substitutions of clauses in P whose head agrees with $A(x_1, \ldots, x_n)$, and gives the set of sets of atomic formulae in antecedents, mimicking the construction for variable-free logic programs. Unfortunately, that does not work.

Consider the logic program ListNat of Example 1. There is a map in $\mathcal{L}_\Sigma$ of the form $0 \to 1$ that models the nullary function symbol 0. So, naturality of the map $p : At \longrightarrow P_f P_f At$ in $[\mathcal{L}_\Sigma^{op}, Set]$ would yield commutativity of the diagram

$$\begin{array}{ccc} At(1) & \longrightarrow & P_f P_f At(1) \\ \downarrow & & \downarrow \\ At(0) & \longrightarrow & P_f P_f At(0) \end{array}$$

There being no clause of the form $\mathtt{nat(x)} \leftarrow$ in ListNat, commutativity of the diagram would in turn imply that there cannot be a clause in ListNat of the form $\mathtt{nat(0)} \leftarrow$ either, but in fact there is one!

In order to model examples such as ListNat, we need to relax the naturality condition on p: if naturality could be relaxed to a subset condition, so that, in general,

$$\begin{array}{ccc} At(m) & \longrightarrow & P_f P_f At(m) \\ \downarrow & \geq & \downarrow \\ At(n) & \longrightarrow & P_f P_f At(n) \end{array}$$

need not commute, but rather the composite via $P_f P_f At(m)$ need only yield a subset of that via $At(n)$, it would be possible for $p_1(\mathtt{nat(x)})$ to be the empty set while $p_0(\mathtt{nat(0)})$ is non-empty in the ListNat example above.

In order to express such a lax naturality condition, we need to extend Set to $Poset$ and we need to extend P_f from Set to $Poset$. The category $Lax(\mathcal{L}_\Sigma^{op}, Poset)$ of strict functors and lax natural transformations is not complete, so the usual construction of a cofree comonad on an endofunctor no longer works directly. On the other hand, $Poset$ is finitely cocomplete as a locally ordered category, so we can adopt the subtle work of [13] on categories of lax natural transformations, which is what we do: see Section 4.

A mild problem arises in regard to the finiteness of the outer occurrence of P_f in $P_f P_f$. The problem is that substitution can generate infinitely many instances

of clauses with the same head. For instance, suppose one extends ListNat with a clause of the form $A \leftarrow \mathtt{nat(x)}$ with no occurrences of x in A. Substitution yields the clause $A \leftarrow \mathtt{nat(s^n(0))}$, for every natural number n, giving rise to a countable set of clauses with head A. We need to allow for possibilities such as this as the infiniteness arises even from a finite signature. So we extend from $P_f P_f$ to $P_c P_f$, where P_c extends the countable powerset functor.

Those are the key technical difficulties that we address in the paper. Note that, in contrast to [15], we do not model the ordering of subgoals and repetitions. These have been modelled in relevant literature, notably in Corradini and Montanari's landmark papers [7, 8], but we defer making precise the relationship with the ideas herein.

We end the paper by making a natural construction of a locally ordered endofunctor to extend $P_c P_f$ in Section 5, checking how coalgebra models our leading example, and comparing the trees we obtain with parallel and-or derivation trees.

2 Parallel and-or derivation trees and coalgebra

In this section, we briefly recall from [15] the definition of the parallel and-or derivation trees generated by an arbitrary logic program, and how, in the case of variable-free logic programs, they can be seen in terms of coalgebraic structure.

Key motivating texts for the definition of parallel and-or derivation tree are [9] and [12], as explained in [15]. We freely use the usual logic programming conventions for substitution and most general unifiers, see Section 3.

Definition 1. *Let P be a logic program and let $\leftarrow A$ be an atomic goal (possibly with variables). The* parallel and-or derivation tree *for A is the possibly infinite tree T satisfying the following properties.*

- *A is the root of T.*
- *Each node in T is either an and-node or an or-node.*
- *Each or-node is given by $\bullet$.*
- *Each and-node is an atom.*
- *For every node A' occurring in T, if A' is unifiable with only one clause $B \leftarrow B_1, \ldots, B_n$ in P with mgu θ, then A' has n children given by and-nodes $B_1\theta, \ldots B_n\theta$.*
- *For every node A' occurring in T, if A' is unifiable with exactly $m > 1$ distinct clauses $C_1, \ldots, C_m$ in P via mgu's $\theta_1, \ldots, \theta_m$, then A' has exactly m children given by or-nodes, such that, for every $i \in m$, if $C_i = B^i \leftarrow B^i_1, \ldots, B^i_{n_i}$, then the ith or-node has n_i children given by and-nodes $B^i_1\theta_i, \ldots, B^i_{n_i}\theta_i$.*

We now recall the coalgebraic development of [15].

Proposition 1. *For any set At, there is a bijection between the set of variable-free logic programs over the set of atoms At and the set of $P_f P_f$-coalgebra structures on At, where P_f is the finite powerset functor on Set.*

Proposition 2. *Let $C(P_fP_f)$ denote the cofree comonad on P_fP_f. Then, for $p : At \longrightarrow P_fP_f(At)$, the corresponding $C(P_fP_f)$-coalgebra is given as follows: $C(P_fP_f)(At)$ is a limit of a diagram of the form*

$$\ldots \longrightarrow At \times P_fP_f(At \times P_fP_f(At)) \longrightarrow At \times P_fP_f(At) \longrightarrow At.$$

Put $At_0 = At$ and $At_{n+1} = At \times P_fP_fAt_n$, and define the cone

$$p_0 = id : At \longrightarrow At(= At_0)$$
$$p_{n+1} = \langle id, P_fP_f(p_n) \circ p \rangle : At \longrightarrow At \times P_fP_fAt_n(= At_{n+1})$$

Then the limiting property determines the coalgebra $\overline{p} : At \longrightarrow C(P_fP_f)(At)$.

In [15], we gave a general account of the relationship between a variable-free logic program qua P_fP_f-coalgebra and the parallel and-or derivation trees it generates. Here we recall a representative example.

Example 2. Consider the variable-free logic program:

```
q(b,a) ←
s(a,b) ←
  p(a) ← q(b,a), s(a,b)
q(b,a) ← s(a,b)
```

The program has three atoms, namely `q(b,a)`, `s(a,b)` and `p(a)`. So $At = \{\texttt{q(b,a)}, \texttt{s(a,b)}, \texttt{p(a)}\}$. The program can be identified with the P_fP_f-coalgebra structure on At given by
$p(\texttt{q(b,a)}) = \{\{\}, \{\texttt{s(a,b)}\}\}$, where $\{\}$ is the empty set.
$p(\texttt{s(a,b)}) = \{\{\}\}$, i.e., the one element set consisting of the empty set.
$p(\texttt{p(a)}) = \{\{\texttt{q(b,a)}, \texttt{s(a,b)}\}\}$.

Consider the $C(P_fP_f)$-coalgebra corresponding to p. It sends `p(a)` to the parallel refutation of `p(a)` depicted on the left side of Figure 1. Note that the nodes of the tree alternate between those labelled by atoms and those labelled by bullets ($\bullet$). The set of children of each bullet represents a goal, made up of the conjunction of the atoms in the labels. An atom with multiple children is the head of multiple clauses in the program: its children represent these clauses. We use the traditional notation $\Box$ to denote $\{\}$.

Where an atom has a single $\bullet$-child, we can elide that node without losing any information; the result of applying this transformation to our example is shown on the right in Figure 1. The resulting tree is precisely the parallel and-or derivation tree for the atomic goal $\leftarrow$ `p(a)` as in Definition 1. So the two trees express equivalent information.

In the first-order case, direct use of Definition 2 yields inconsistent derivations, as explained e.g. in [12]. So *composition (and-or parallel) trees* were introduced [12]. Construction of composition trees involves additional algorithms that

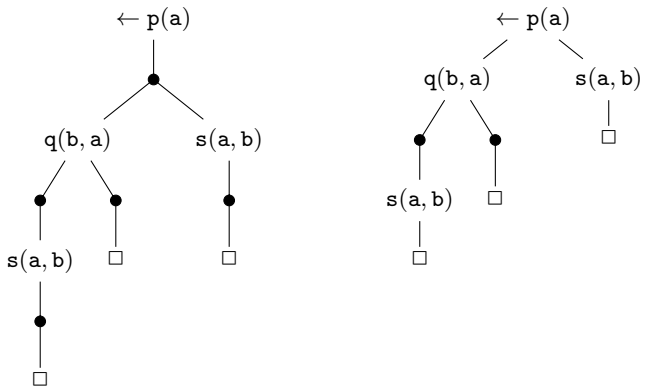


Fig. 1. The action of $\overline{p} : \mathrm{At} \longrightarrow C(P_f P_f)(\mathrm{At})$ on `p(a)`, and the corresponding parallel and-or derivation tree.

synchronise branches created by or-nodes. Composition trees contain a special kind of *composition node* used whenever both and- and or-parallel computations are possible for one goal. Every composition node is a list of atoms in the goal. If, in a goal $G = \leftarrow B_1, \ldots B_n$, an atom B_i is unifiable with $k > 1$ clauses, then the algorithm adds k children (k composition nodes) to the node G; similarly for every atom in G that is unifiable with more than one clause. Every such composition node has the form $B_1, \ldots B_n$, and n and-parallel edges. Thus, all possible combinations of all possible or-choices at every and-parallel step are given. In this paper, we do not study composition trees directly but rather suggest an alternative.

3 Using Lawvere theories to model first-order signatures and substitution

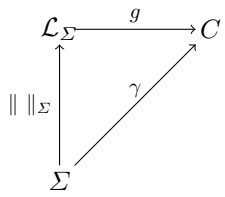
In this section, we start to move towards using coalgebra to model arbitrary logic programs by recalling the relationship between first-order signatures and Lawvere theories, in particular how the former give rise to the latter. Then we recall how to use that to model most general unifiers as equalisers.

A *signature* Σ consists of a set of *function symbols* $f, g, \ldots$ each equipped with a fixed *arity* given by a natural number indicating the number of arguments it is supposed to have. Nullary (0-ary) function symbols are allowed and are called *constants*. Given a countably infinite set Var of variables, the set $Ter(\Sigma)$ of *terms* over Σ is defined inductively:

- $x \in Ter(\Sigma)$ for every $x \in Var$.
- If f is an n-ary function symbol ($n \geq 0$) and $t_1, \ldots, t_n \in Ter(\Sigma)$, then $f(t_1, \ldots, t_n) \in Ter(\Sigma)$.

Definition 2. *Given a signature Σ and a category C with strictly associative finite products, an* interpretation *of Σ in C is an object X of C, together with, for each function symbol f of arity n, a map in C from X^n to X.*

Proposition 3. *Given a signature Σ, there exists a category $\mathcal{L}_\Sigma$ with strictly associative finite products and an interpretation $\| \ \|_\Sigma$ of Σ in $\mathcal{L}_\Sigma$, such that for any category C with strictly associative finite products, and interpretation γ of Σ in C, there exists a unique functor $g : \mathcal{L}_\Sigma \to C$ that strictly preserves finite products, such that g composed with $\| \ \|_\Sigma$ gives γ, as in the following diagram:*



Proof. Define the set $\mathtt{ob}(\mathcal{L}_\Sigma)$ to be the set of natural numbers.

For each natural number n, let $x_1, \ldots, x_n$ be a specified list of distinct variables. Define $\mathtt{ob}(\mathcal{L}_\Sigma)(n, m)$ to be the set of m-tuples $(t_1, \ldots, t_m)$ of terms generated by the function symbols in Σ and variables $x_1, \ldots, x_n$. Define composition in $\mathcal{L}_\Sigma$ by substitution. The interpretation $\| \ \|_\Sigma$ sends an n-ary function symbol f to $f(x_1, \ldots, x_n)$.

One can readily check that these constructions satisfy the axioms for a category and for an interpretation, with $\mathcal{L}_\Sigma$ having strictly associative finite products given by the sum of natural numbers. The terminal object of $\mathcal{L}_\Sigma$ is the natural number 0. The universal property follows directly from the construction.

Definition 3. *Given a signature Σ, the category $\mathcal{L}_\Sigma$ determined by Proposition 3 is called the* Lawvere theory *generated by Σ [17].*

One can describe $\mathcal{L}_\Sigma$ without the need for a specified list of variables for each n: in a term t, a variable context is always implicit, i.e., $x_1, \ldots, x_m \vdash t$, and the variable context is considered as a binder.

In contrast to the usual practice in category theory, sorting is not modelled by using a sorted finite product theory but rather by modelling predicates for sorts such as `nat` or `list` using the structure of the category $[\mathcal{L}_\Sigma, Set]$ or, more subtly, of $Lax(\mathcal{L}_\Sigma, Poset)$, as illustrated below: Lloyd's book [18] gives a representataive account of logic programming, and although category theorists may disapprove, it is not sorted.

Example 3. Consider `ListNat`. It is naturally two-sorted, with one sort for natural numbers and one for lists. Traditionally, category theory would not use Proposition 3 but rather a two-sorted version of it: see [16]. But `ListNat` is a legitimate untyped logic program and is representative of such.

The constants `0` and `nil` are modelled by maps from 0 to 1 in $\mathcal{L}_\Sigma$, `s` is modelled by a map from 1 to 1, and `cons` is modelled by a map from 2 to 1. The

term `s(0)` is therefore modelled by the map from 0 to 1 given by the composite of the maps modelling `s` and `0`; similarly for the term `s(nil)`, although the latter does not make semantic sense.

A key construct in standard accounts of SLD-resolution such as [18] is that of a most general unifier, which we now recall. It is typically expressed using distinctive notation for substitution. Note that the coalgebraic approach does not require us to model substitution by most general unifiers; it does not even require us to take syntax over Lawvere theories, as we may take it over more general categories: note the generality of Section 4 and see [14].

Definition 4. *A* substitution *is a function θ from Var to $Ter(\Sigma)$ that is the identity on all but finitely many variables. Each substitution canonically generates a function from $Ter(\Sigma)$ to itself defined inductively by the following:*

$$\theta(f(t_1, \ldots, t_n)) \equiv f(\theta(t_1), \ldots, \theta(t_n))$$

Following the usual convention in logic programming, we denote $\theta(t)$ by $t\theta$[18].

Definition 5. *Let S be a finite set of terms. A substitution θ is called a* unifier *for S if, for any pair of terms t_1 and t_2 in S, applying the substitution θ yields $t_1\theta = t_2\theta$. A unifier θ for S is called a* most general unifier *(mgu) for S if, for each unifier σ of S, there exists a substitution γ such that $\sigma = \theta\gamma$.*

The structure of $\mathcal{L}_\Sigma$ allows us to characterise most general unifiers in terms of equalisers as follows, cf [21], where they are modelled by coequalisers in the Kleisli category for a the monad T_Σ on Set induced by $\mathcal{L}_\Sigma$.

Proposition 4. *Given a signature Σ, for any pair of terms (`s`,`t`) with variables among $x_1, \ldots, x_n$, a most general unifier of `s` and `t` exists if and only if an equaliser of `s` and `t` qua maps in $\mathcal{L}_\Sigma$ exists, in which case the most general unifier is given by the equaliser.*

Example 4. A most general unifier of the terms `cons(x,nil)` and `cons(s(0),y)` of Example 3 exists and is given by the substitution $\sigma : \{\texttt{s(0)/x}, \texttt{nil/y}\}$.

4 Coalgebra on categories of lax maps

Assume we have a signature Σ of function symbols and, for each natural number n, a specified list of variables $x_1, \ldots, x_n$. Then, given an arbitrary logic program with signature Σ, we can extend our study of the set At of atoms for a variable-free logic program in [15] by considering the functor $At : \mathcal{L}_\Sigma^{op} \to Set$ that sends a natural number n to the set of all atomic formulae with variables among $x_1, \ldots, x_n$ generated by the function symbols in Σ and the predicate symbols appearing in the logic program. A map $f : n \to m$ in $\mathcal{L}_\Sigma$ is sent to the function $At(f) : At(m) \to At(n)$ that sends an atomic formula $A(x_1, \ldots, x_m)$ to $A(f_1(x_1, \ldots, x_n)/x_1, \ldots, f_m(x_1, \ldots, x_n)/x_m)$, i.e., $At(f)$ is defined by substitution.

As explained in the Introduction, we cannot model a logic program by a natural transformation of the form $p : At \longrightarrow P_f P_f At$ as naturality breaks down even in simple examples such as ListNat. We need lax naturality. In order even to define it, we first need to extend $At : \mathcal{L}_\Sigma^{op} \to Set$ to have codomain $Poset$. That is routine, given by composing At with the inclusion of Set into $Poset$. Mildly overloading notation, we denote the composite by $At : \mathcal{L}_\Sigma^{op} \to Poset$, noting that it is trivially locally ordered.

Definition 6. *Given locally ordered functors $H, K : D \longrightarrow C$, a* lax natural transformation *from H to K is the assignment to each object d of D, of a map $\alpha_d : Hd \longrightarrow Kd$ such that for each map $f : d \longrightarrow d'$ in D, one has $(Kf)(\alpha_d) \leq (\alpha_{d'})(Hf)$.*

Locally ordered functors and lax natural transformations, with pointwise composition and pointwise ordering, form a locally ordered category we denote by $Lax(D, C)$.

As explained in the Introduction, we need to extend the endofunctor $P_c P_f$ on Set rather than extending $P_f P_f$ as, even with finitely many function symbols, substitution could give rise to countably many clauses with the same head. So we need to extend $P_c P_f$ from an endofunctor on Set to a locally ordered endofunctor on $Lax(\mathcal{L}_\Sigma^{op}, Poset)$. A natural way to do that, while retaining the role of $P_c P_f$, is first to extend $P_c P_f$ to a locally ordered endofunctor E on $Poset$, then to consider the locally ordered endofunctor $Lax(\mathcal{L}_\Sigma^{op}, E)$ on $Lax(\mathcal{L}_\Sigma^{op}, Poset)$ that sends $H : \mathcal{L}_\Sigma^{op} \to Poset$ to the composite EH.

We shall return to the question of extending to $P_c P_f$ to $Poset$, but what about the cofree comonad $C(P_c P_f)$ on $P_c P_f$?

The locally ordered category $Lax(\mathcal{L}_\Sigma^{op}, Poset)$ is neither complete nor cocomplete, so it does not follow from the usual general theory that a cofree comonad on a locally ordered endofunctor on it need exist at all, let alone be given by a limiting construct resembling that of Proposition 2. Moreover, the laxness in $Lax(\mathcal{L}_\Sigma^{op}, Poset)$ makes the category of coalgebras for an endofunctor on it problematic, as the strictness in the definition of map of coalgebras does not cohere well with the laxness in the definition of map in $Lax(\mathcal{L}_\Sigma^{op}, Poset)$.

Using techniques developed by Kelly in Section 3.3 of [13], we can negotiate these obstacles. Rather than directly considering a cofree comonad on $Lax(\mathcal{L}_\Sigma^{op}, E)$, we can extend the comonad $C(P_c P_f)$ from Set to $Lax(\mathcal{L}_\Sigma^{op}, Poset)$, mimicking our extension of $P_c P_f$. We can then use a variant of the fact that, if it exists, a cofree comonad $C(H)$ on an arbitrary endofunctor H is characterised by a canonical isomorphism of categories

$$H\text{-}coalg \simeq C(H)\text{-}Coalg$$

where $-coalg$ stands for functor coalgebras while $-Coalg$ is for Eilenberg-Moore coalgebras. Although the categories of coalgebras and strict maps are problematic in the lax setting, categories of coalgebras and oplax maps do respect the laxness of $Lax(\mathcal{L}_\Sigma^{op}, Poset)$, allowing a suitable variant. The details are as follows.

Proposition 5. *Given a locally ordered comonad G on a locally ordered category C, the data given by $Lax(D,G) : Lax(D,C) \to Lax(D,C)$ and pointwise liftings of the structural natural transformations of G yield a locally ordered comonad we also denote by $Lax(D,G)$ on $Lax(D,C)$.*

The proof of Proposition 5 is not entirely trivial as it involves a mixture of the strict structure in the definition of comonad with the lax structure in the definition of $Lax(D,C)$. Nevertheless, with attention to detail, a proof is routine, and it means that, once we have extended the comonad $C(P_cP_f)$ to $Poset$, we can further extend it axiomatically to $Lax(\mathcal{L}_\Sigma^{op}, Poset)$.

Let E be an arbitrary locally ordered endofunctor on an arbitrary locally ordered category C. Denote by $E\text{-}coalg_{oplax}$ the locally ordered category whose objects are E-coalgebras and whose maps are oplax maps of E-coalgebras, meaning that, in the square

$$\begin{array}{ccc} X & \longrightarrow & Y \\ \downarrow & \leq & \downarrow \\ EX & \longrightarrow & EY \end{array}$$

the composite via EX is less than or equal to the composite via Y, with the evident composition and locally ordered structure. Since C and E are arbitrary, one can replace C by $Lax(D,C)$ and replace E by $Lax(D,E)$, yielding the locally ordered category $Lax(D,E)\text{-}coalg_{oplax}$. The following result is also not immediate, but it again follows from routine checking. It is an instance of a general phenomenon that allows laxness to commute exactly with oplaxness but not with any other variant of laxness such as laxness itself or strictness or pseudoness.

Proposition 6. *The locally ordered category $Lax(D,E)\text{-}coalg_{oplax}$ is canonically isomorphic to $Lax(D, E\text{-}coalg_{oplax})$.*

Proposition 6 gives us an easy way to make constructions with, and check claims regarding, $Lax(D,E)$-coalgebras : it characterises such coalgebras in terms of locally ordered functors into $E\text{-}coalg_{oplax}$; the latter locally ordered category, i.e., $E\text{-}coalg_{oplax}$, is simpler to study than $Lax(D,E)\text{-}coalg_{oplax}$ as it only involves one kind of laxness rather than two.

Definition 7. *Given a locally ordered comonad G on C, the locally ordered category $G\text{-}Coalg_{oplax}$ has objects given by (strict) G-coalgebras and maps given by oplax maps of coalgebras, where maps are defined as in $E\text{-}coalg_{oplax}$.*

With care, Proposition 6 can be extended from locally ordered endofunctors to locally ordered comonads, yielding the following:

Proposition 7. *Given a locally ordered comonad G, the locally ordered category $Lax(D,G)\text{-}Coalg_{oplax}$ is canonically isomorphic to $Lax(D, G\text{-}Coalg_{oplax})$.*

The analysis of [13], but expressed there in terms of laxness rather than oplaxness and in terms of monads rather than comonads, yields the following:

Theorem 1. *Given a locally ordered endofunctor E on a locally ordered category with finite colimits C, if $C(E)$ is the cofree comonad on E, then E-$coalg_{oplax}$ is canonically isomorphic to $C(E)$-$Coalg_{oplax}$.*

Combining Proposition 6, Proposition 7 and Theorem 1, we can conclude the following:

Theorem 2. *Given a locally ordered endofunctor E on a locally ordered category with finite colimits C, if $C(E)$ is the cofree comonad on E, then there is a canonical isomorphism*

$$Lax(D, E)\text{-}coalg_{oplax} \simeq Lax(D, C(E))\text{-}Coalg_{oplax}$$

Corollary 1. *For any locally ordered endofunctor E on $Poset$, if $C(E)$ is the cofree comonad on E, then there is a canonical isomorphism*

$$Lax(\mathcal{L}_\Sigma^{op}, E)\text{-}coalg_{oplax} \simeq Lax(\mathcal{L}_\Sigma^{op}, C(E))\text{-}Coalg_{oplax}$$

Corollary 1 provides us with the central axiomatic result we need to extend our analysis of variable-free logic programs in [15] to arbitrary logic programs. The bulk of the analysis of this section holds axiomatically, so that seems the best way in which to explain it although we have only one leading example, that determined by an extension of P_cP_f to $Poset$. In Section 5, we shall investigate such an extension.

5 Coalgebraic semantics for arbitrary logic programs

The reason we need to extend P_cP_f from Set to $Poset$ is to allow for lax naturality, and the reason for that is to take advantage of the partial order structure of the set $P_c(X)$: we neither need nor want to change the set $P_c(X)$ itself; we just need to exploit its natural partial order structure given by subset inclusion. Nor do we want to change the nature of the relationship between a variable-free logic program P and the associated coalgebra $p : At \longrightarrow P_fP_f(At)$: as best we can, we simply want to extend that relationship by making it pointwise relative to the indexing category $\mathcal{L}_\Sigma^{op}$.

In order to give a locally ordered endofunctor on $Poset$, we need to extend P_cP_f from acting on a set X to acting on a partially ordered set P, respecting the partial order structure. This leads to a natural choice as follows:

Definition 8. *Define $P_f : Poset \longrightarrow Poset$ by letting $P_f(P)$ be the partial order given by the set of finite subsets of P, with $A \leq B$ if for all $a \in A$, there exists $b \in B$ for which $a \leq b$ in P, with behaviour on maps given by image. Define P_c similarly but with countability replacing finiteness.*

As *Poset* is complete and cocomplete, and as P_cP_f has a rank, a cofree comonad $C(P_cP_f)$ necessarily exists on P_cP_f. Moreover, it is given by the transfinite (just allowing for countability) extension of the construction in Proposition 2.

By the work of Section 4, the $Lax(\mathcal{L}_\Sigma^{op}, P_cP_f)$-coalgebra structure, i.e., the lax natural transformation, $p : At \longrightarrow P_cP_fAt$ associated with an arbitrary logic program P, evaluated at a natural number n, sends an atomic formula $A(x_1, \ldots, x_n)$ to the set of sets of antecedents in substitution instances of clauses in P for which the head of the substituted instance agrees with $A(x_1, \ldots, x_n)$. Extending Section 2, this can be expressed as a tree of the nature of the left hand tree in Figure 1, interleaving two kinds of nodes.

Comparing these trees with the definition of parallel and-or derivation tree, i.e., with Definition 1, these trees are more intrinsic: parallel and-or derivation trees have most general unifiers built into a single tree, whereas, for each natural number n, coalgebra yields trees involving at most n free variables, then models substitution by replacing them by related, extended trees. We shall illustrate with our leading example.

The two constructs are obviously related, but the coalgebraic one makes fewer identifications, SLD-resolution being modelled by a list of trees corresponding to a succession of substitutions rather than by a single tree. We would suggest that this list of trees may be worth considering as a possible refinement of the notion of parallel and-or derivation tree, lending itself to a tree-rewriting understanding of the SLD-algorithm. Providing such an account is a priority for us as future research.

Example 5. Consider ListNat as in Example 3. Suppose we start with $A(x, y)\epsilon At(2)$ given by the atomic formula $\texttt{list}(\texttt{cons}(\texttt{x}, \texttt{cons}(\texttt{y}, \texttt{x})))$. Then $p(A(x, y))$ is the element of $P_cP_fAt(2)$ expressible by the tree on the left hand side of Figure 2.

This tree agrees with the first part of the parallel and-or derivation tree for $\texttt{list}(\texttt{cons}(\texttt{x}, \texttt{cons}(\texttt{y}, \texttt{x})))$ as determined by Definition 1. But the tree here has leaves `nat(x)`, `nat(y)` and `list(x)`, whereas the parallel and-or derivation tree follows those nodes, using substitutions determined by mgu's. Moreover, those substitutions need not be consistent with each other: not only are there two ways to unify each of `nat(x)`, `nat(y)` and `list(x)`, but also there is no consistent substitution for `x` at all.

In contrast, the coalgebraic structure means any substitution, whether determined by an mgu or not, applies to the whole tree. The lax naturality means a substitution potentially yields two different trees: one given by substitution into the tree, then pruning to remove redundant branches, the other given by substitution into the root, then applying p.

For example, suppose we substitute $s(z)$ for both x and y in $\texttt{list}(\texttt{cons}(\texttt{x}, \texttt{cons}(\texttt{y}, \texttt{x})))$. This substitution is given by applying At to the map $(s, s) : 1 \longrightarrow 2$ in $\mathcal{L}_\Sigma$. So $At((s, s))(A(x, y))$ is an element of $At(1)$. Its image under $p_1 : At(1) \longrightarrow P_cP_fAt(1)$ is the element of $P_cP_fAt(1)$ expressible by the tree on the right hand side of Figure 2. The laxness of the naturality of p is indi-

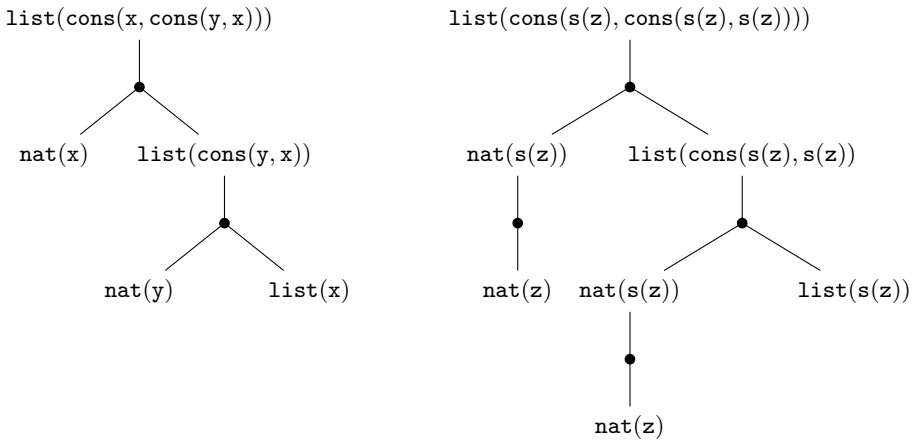


Fig. 2. The left hand tree represents $p(\texttt{list}(\texttt{cons}(\texttt{x}, \texttt{cons}(\texttt{y}, \texttt{x}))))$ and the right hand tree represents $pAt((s, s))(\texttt{list}(\texttt{cons}(\texttt{x}, \texttt{cons}(\texttt{y}, \texttt{x}))))$, i.e., $p(\texttt{list}(\texttt{cons}(\texttt{s}(\texttt{z}), \texttt{cons}(\texttt{s}(\texttt{z}), \texttt{s}(\texttt{z})))))$.

cated by the increased length, in two places, of the second tree when compared with the first tree. Observe that, before those two places, the two trees have the same structure: that need not always be exactly the case, as substitution in a tree could involve pruning if substitution instances of two different atoms yield the same atom.

Now suppose we make the further substitution of 0 for z. This substitution is given by applying At to the map $0 : 0 \to 1$ in $\mathcal{L}_\Sigma$. In Figure 3, we depict $p_1At((s, s))(A(x, y))$ on the left, repeating the right hand tree of Figure 2, and we depict $p_0At(0)At((s, s))(A(x, y))$ on the right.

Two of the leaves of the latter tree are labelled by $\Box$, but one leaf, namely $\texttt{list}(\texttt{s}(\texttt{0}))$ is not, so the tree does not yield a proof. Again, observe the laxness.

6 Conclusions and Further Work

Using sophisticated category theoretic techniques surrounding the notion of laxness, we have extended the coalgebraic analysis of variable-free logic programs in [15] to arbitrary logic programs. For variable-free logic programs, the cofree comonad on $P_f P_f$ allowed us to represent the parallel and-or derivation trees generated by a logic program. For arbitrary logic programs, the situation is more subtle, as coalgebra naturally gives rise to a list of trees determined by substitutions, whereas a parallel and-or derivation tree has all the information squeezed into one tree.

A natural question to arise in the light of this is whether the coalgebraic structure given here suggests a more subtle semantics for SLD-resolution than

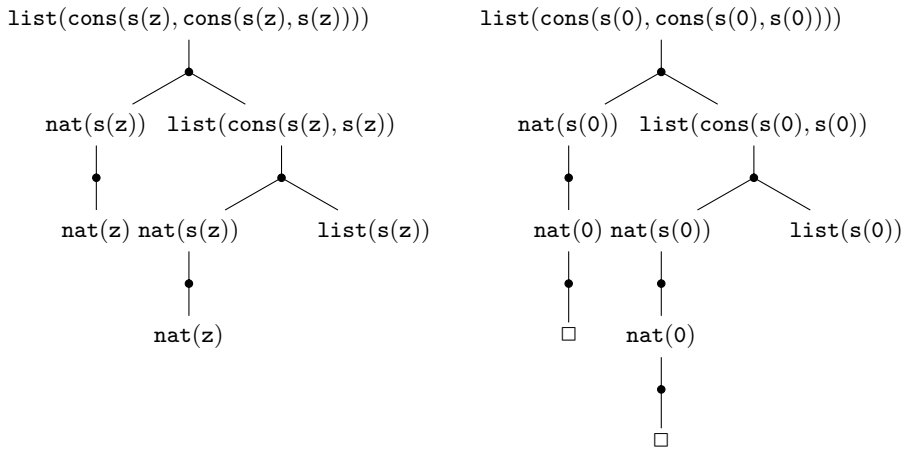


Fig. 3. On the left is the tree depicting $pAt((s,s))(\texttt{list}(\texttt{cons}(\texttt{x},\texttt{cons}(\texttt{y},\texttt{x}))))$ as also appears on the right of Figure 2, and on the right is the tree depicting $pAt(0)At((s,s))(\texttt{list}(\texttt{cons}(\texttt{x},\texttt{cons}(\texttt{y},\texttt{x}))))$

that given by parallel and-or derivation trees, perhaps one based upon tree-rewriting. That is one direction in which we propose to continue research.

The key fact driving our analysis has been the observation that the implication $\leftarrow$ acts at a meta-level, like a sequent rather than a logical connective. That observation extends to first-order fragments of linear logic and the Logic of Bunched Implications [10, 20]. So we plan to extend the work in the paper to logic programming languages based on such logics.

The situation regarding higher-order logic programming languages such as λ-*PROLOG* [19] is more subtle. Despite their higher-order nature, such logic programming languages typically make fundamental use of sequents. So it may well be fruitful to consider modelling them in terms of coalgebra too, albeit probably on a sophisticated base category such as a category of Heyting algebras.

More generally, the results of this paper can be applied to the studies of Higher-order recursion schemes, [1].

A further direction is to investigate the operational meaning of coinductive logic programming [11, 22]. That requires a modification to the algorithm of SLD-resolution we have considered in this paper. In particular, given a logic program that defines an infinite stream (similarly to our running example of `list`, but without the base case for `nil`), the interpreter for coinductive logic programs of this kind would be able to deduce a finite atom `stream(cons(x,y))` from the infinite derivations.

References

1. J. Adámek, S. Milius, and J. Velebil. Semantics of higher-order recursion schemes. *CoRR*, abs/1101.4929, 2011.
2. G. Amato, J. Lipton, and R. McGrail. On the algebraic structure of declarative programming languages. *Theor. Comput. Sci.*, 410(46):4626–4671, 2009.
3. A. Asperti and S. Martini. Projections instead of variables: A category theoretic interpretation of logic programs. In *ICLP*, pages 337–352, 1989.
4. F. Bonchi and U. Montanari. Reactive systems, (semi-)saturated semantics and coalgebras on presheaves. *Theor. Comput. Sci.*, 410(41):4044–4066, 2009.
5. R. Bruni, U. Montanari, and F. Rossi. An interactive semantics of logic programming. *TPLP*, 1(6):647–690, 2001.
6. M. Comini, G. Levi, and M. C. Meo. A theory of observables for logic programs. *Inf. Comput.*, 169(1):23–80, 2001.
7. A. Corradini and U. Montanari. An algebraic semantics of logic programs as structured transition systems. In *Proc. NACLP'90*. MIT Press, 1990.
8. A. Corradini and U. Montanari. An algebraic semantics for structured transition systems and its application to logic programs. *TCS*, 103:51–106, 1992.
9. V. S. Costa, D. H. D. Warren, and R. Yang. Andorra-I: A parallel prolog system that transparently exploits both and- and or-parallelism. In *PPOPP*, pages 83–93, 1991.
10. J.-Y. Girard. Linear logic. *Theor. Comput. Sci.*, 50:1–102, 1987.
11. G. Gupta, A. Bansal, R. Min, L. Simon, and A. Mallya. Coinductive logic programming and its applications. In *ICLP 2007*, volume 4670 of *LNCS*, pages 27–44. Springer, 2007.
12. G. Gupta and V. S. Costa. Optimal implementation of and-or parallel prolog. In *Conference proceedings on PARLE'92*, pages 71–92, New York, NY, USA, 1994. Elsevier North-Holland, Inc.
13. G. M. Kelly. Coherence theorems for lax algebras and for distributive laws. In *Category seminar*, volume 420 of *LNM*, pages 281 – 375, 1974.
14. Y. Kinoshita and A. J. Power. A fibrational semantics for logic programs. In *Proceedings of the Fifth International Workshop on Extensions of Logic Programming*, volume 1050 of *LNAI*. Springer, 1996.
15. E. Komendantskaya, G. McCusker, and J. Power. Coalgebraic semantics for parallel derivation strategies in logic programming. In *Proc. of AMAST'2010*, volume 6486 of *LNCS*, 2011.
16. E. Komendantskaya and J. Power. Fibrational semantics for many-valued logic programs: Grounds for non-groundness. In *JELIA*, volume 5293 of *LNCS*, pages 258–271, 2008.
17. W. Lawvere. *Functional semantics of algebraic theories*. PhD thesis, Columbia University, 1963.
18. J. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 2nd edition, 1987.
19. D. Miller and G. Nadathur. Higher-order logic programming. In *ICLP*, pages 448–462, 1986.
20. D. Pym. *The Semantics and Proof Theory of the Logic of Bunched Implications*, volume 26 of *Applied Logic Series*. Kluwer Academic Publishers, 2002.
21. D. Rydeheard and R. Burstall. *Computational Category theory*. Prentice Hall, 1988.
22. L. Simon, A. Bansal, A. Mallya, and G. Gupta. Co-logic programming: Extending logic programming with coinduction. In *ICALP*, volume 4596 of *LNCS*, pages 472–483. Springer, 2007.