Coalgebraic Logic Programming: implicit versus explicit resource handling

Ekaterina Komendantskaya[1], John Power[2] and Martin Schmidt[3]

[1] Department of Computing, University of Dundee, UK [⋆]
[2] Department of Computer Science, University of Bath, UK
[3] Institute of Cognitive Science, Osnabrueck University, Germany

Abstract. We compare approaches to implicit and explicit resource handling in coinductive and concurrent logic programming. We show various effects that implicit and explicit handling of resources have on implementation and semantics. In particular, we show that recently introduced coalgebraic logic programming [17] is a paradigm in which, in contrast to many other alternative systems, the aspects of logic and control are intertwined, and computational resources are handled implicitly.
Key words: Logic programming, Coinduction, Coalgebra, Resources.

1 Introduction

First-order logic programming is a language combining first-order syntax with efficient techniques of *unification* and *SLD-resolution* [18, 19, 25]. These two techniques underlie many modern computer tools, ranging from automated theorem provers and SAT/SMT solvers to type inference in declarative [20, 21] and imperative [2] languages.

Example 1. The program `Stream` defines infinite streams of binary bits. It is constructed using five atomic first-order formulas (atoms), arranged into three clauses, two of which are themselves atomic. The last clause has one atom in its *head* and two atoms in its *body*.

$$\begin{aligned} \texttt{bit}(0) &\leftarrow \\ \texttt{bit}(1) &\leftarrow \\ \texttt{stream(scons (X,Y))} &\leftarrow \texttt{bit(X)}, \texttt{stream(Y)} \end{aligned}$$

At the propositional (atomic) level, logic programs resemble, and indeed induce, transition systems or rewrite systems, hence coalgebras. This fact has been used to study their operational semantics, e.g. [5, 7]. 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.

⋆ The work was supported by EPSRC grants

Our main result was that if $C(P_fP_f)$ is the cofree comonad on P_fP_f, then, given a logic program P qua P_fP_f-coalgebra, the corresponding $C(P_fP_f)$-coalgebra structure characterises the parallel and-or derivation trees of P.

There have been several category theoretic models of first-order fragments 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* [1, 5, 6, 14], and that of most general unifiers (mgu's) as *equalisers* [4] or as *pullbacks* [6, 1].

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, in [16], we considered 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 $[\mathcal{L}_\Sigma^{op}, P_fP_f]$-coalgebra $p : At \longrightarrow P_fP_fAt$ 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. As we showed in [16], this does not work.

In fact, to make the theory work, we need to extend Set to $Poset$, natural transformations to *lax natural transformations*, and replace the outer instance of P_f by P_c - the countable powerset functor (as recursion generates countability). Subject to those replacements, $p : At \longrightarrow P_cP_fAt$ behaves as above, giving a $Lax(\mathcal{L}_\Sigma^{op}, P_cP_f)$-coalgebra structure on At. Thus, p determines a $Lax(\mathcal{L}_\Sigma^{op}, C(P_cP_f))$-coalgebra structure $\bar{p} : At \longrightarrow C(P_cP_f)(At)$.

In [17], we proved the adequacy of this coalgebraic semantics relative to SLD-resolution. We also showed that observationally, the coalgebraic semantics inspires a different type of derivation – *coalgebraic*. In this paper, we will consider the implications the new coalgebraic derivation algorithm has on resource handling in logic programming, paying special attention to the aspects of corecursion and concurrency. We also present an implementation of coalgebraic logic programming, [23].

There are two aspects of logic programming that are both desirable and problematic in practice — these are corecursive derivations and concurrent computations. For example, `Stream` is a coinductive definition, that is, proof search for the goal `stream(X)` will result in an infinite SLD-derivation. Programs like `Stream` can be given declarative semantics via the *greatest* fixed point of the semantic operator T_P. However the fixed point semantics is incomplete in general [19]: it fails for some infinite derivations.

Example 2. The program `Stream` is characterised by the greatest fixed point of the T_P operator, which contains `stream`$(scons^\omega(X, Y))$; whereas no infinite term can be computed via SLD-resolution.

There have been numerous attempts to resolve the mismatch between infinite derivations and greatest fixed point semantics [9, 12, 19, 24]. Notably, many solutions [9, 24] resort to explicit annotation of corecursive loops to terminate infinite derivations gracefully. One may also view such attempts as explicit corecursive resource handling, as we explain in Section 2.

Another distinguishing feature of logic programming languages is that they allow implicit parallel execution of programs [10, 22, 11]. However, many first-order algorithms are P-complete and hence inherently sequential [8, 13]. This especially concerns first-order unification and variable substitution in the presence of variable dependencies.

Example 3. The goal `stream(scons(X, scons(Y,X)))`, if processed sequentially, leads to a failed derivation (due to ill-typing). But, if the proof search proceeds in concurrent fashion, it may find substitutions for x in distinct parallel branches of the derivation tree.

Implementations of *parallel* SLD-derivations require keeping special records of previously made substitutions and hence involve additional data structures and algorithms that coordinate variable substitution in different branches of parallel derivation trees. Again, this can be seen as explicit resource handling, where *resources* are variables, terms, and substitutions, cf. Section 4.

Sections 2 and 4 also analyse how these common practices of explicit resource handling of corecursion and concurrency change with the introduction of coalgebraic logic programming [17], notably in favour of implicit resource-handling. In Section 6, we discuss an implementation [23] and draw conclusions.

2 Co-recursion: implicit versus explicit resource control

In this section, we consider applications of corecursion and concurrency [9, 24] that rely on explicit resource management. As Example 4 illustrates, standard SLD-derivation procedures of logic programming can be caught in infinite derivation chains for any proof involving the predicate `stream`.

Example 4. The goal `stream(X)` produces the following SLD derivation: $\texttt{stream(X)} \xrightarrow{X/scons(Y,Z)} \texttt{bit(Y), stream(Z)} \xrightarrow{Y/0} \texttt{stream(Z)} \rightarrow \ldots$.
It contains an infinite repetition of `stream(X)` for various variables X.

In [9, 24], the solution was to introduce a procedure that allows one to assert "`stream(X)` is proven" and terminate derivations whenever such a regular loop is detected. Extending this extra "rule" to inductive computations would lead to unsound results: in the inductive case, infinite loops normally indicate lack of progress in a derivation rather than "success". Thus, explicit annotation of predicates is required:

Example 5. Consider the annotated logic program below comprising both inductive and coinductive clauses:

$$
\begin{aligned}
\texttt{bit}^i(0) &\leftarrow \\
\texttt{bit}^i(1) &\leftarrow \\
\texttt{stream}^c(\texttt{scons}(X,Y)) &\leftarrow \texttt{bit}^i(X), \texttt{stream}^c(Y) \\
\texttt{list}^i(\texttt{nil}) &\leftarrow \\
\texttt{list}^i(\texttt{cons}(X,Y)) &\leftarrow \texttt{bit}^i(X), \texttt{list}^i(Y)
\end{aligned}
$$

Only infinite loops produced for corecursive goals (marked by c) are gracefully terminated; others are still be treated as "undecided" proof branches. In practice, this works as locks and keys in resource logics, allowing or disallowing infinite data structures. There are several drawbacks to this method:

★ some predicates may behave inductively or coinductively depending on the arguments provided, and such cases need to be resolved dynamically, and not statically; in which case mere predicate annotation fails.

★★ this new coinductive algorithm is not in essence a lazy infinite (corecursive) computation. Instead, it substitutes an infinite proof by a finite derivation, on the basis of guarantees of the data regularity in the corecursive loops. But such guarantees cannot always be given; consider computing the number π.

An alternative solution to the problem is given in [16, 17]. There, instead of introducing explicit annotations into the programming language, a new derivation algorithm is introduced; it uses lazy rewriting of coinductive trees. The definition of coinductive trees arises directly from the coalgebraic semantics [16] as sketched in the Introduction.

Definition 1. *Let P be a logic program and $G = \leftarrow A$ be an atomic goal. A* coinductive derivation tree *for A is a 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 and-node A' occurring in T, if there exist exactly $m > 0$ distinct clauses $C_1, \ldots, C_m$ in P (a clause C_i has the form $B_i \leftarrow B_1^i, \ldots, B_{n_i}^i$, for some n_i), such that $A' = B_1\theta_1 = \ldots = B_m\theta_m$, for some substitutions $\theta_1, \ldots, \theta_m$, then A' has exactly m children given by or-nodes, such that, for every $i \in m$, the ith or-node has n children given by and-nodes $B_1^i\theta_i, \ldots, B_{n_i}^i\theta_i$.*

In general case, a goal may induce an infinite family of coinductive trees - as there can be a countable number of substitutions $\theta_i', \ldots, \theta_i''$ that match a given goal with the clause C_i.

Definition 2. *Let P be a logic program and $G = \leftarrow A$ be an atomic goal. The* coinductive forest *F for A is the set of all coinductive trees for A. We say that the forest has* depth *n if the deepest tree in F has depth n. A coinductive forest F has* breadth *k if at most k distinct variables appear in all and-nodes of all of its trees together.*

The coinductive trees and forests mimic the action of the comonad $\bar{p} : At \longrightarrow C(P_c P_f)(At)$ on the atomic goals:

Theorem 1 (Adequacy). *For any logic program P and for any atom A generated by the predicate symbols of P and k distinct variables $x_1, \ldots, x_k$, $\bar{p}(k)(A)$ expresses precisely the same information as that given by the coinductive forest F for the goal A. That is, the following holds:*

- *$p_n(k)(A)$ is isomorphic to the coinductive forest of depth n and breadth k.*
- *F has finite depth n if and only if $\bar{p}(k)(A) = p_n(k)(A)$.*

The coalgebraic semantics can serve as a diagnostic tool for well-foundness of corecursion; and that provides an alternative to explicit atom labelling [9, 24]. In particular, two features will distinguish well-founded coinductive programs like `Stream` from ill-founded programs:

- finite size of the coinductive forests, in which the coinductive tree for every goal is uniquely determined;
- finite depth of the coinductive trees.

We now step back from the semantics and consider coinductive trees as a computational model. The first feature to note is that, comparing this with the SLD-resolution algorithm or co-LP [9, 24], the definition of coinductive derivation tree restricts unification to the case of *term matching*, i.e., the substitution θ unifying atoms A_1 and A_2 is applied only to one atom, e.g. $A_1 = A_2\theta$, whereas traditionally mgus satisfy $A_1\theta = A_2\theta$. This restriction in the unification algorithm provides a powerful tool for *implicit resource control*: it allows one to unfold coinductive trees lazily, keeping each individual tree at a finite size, provided the program is well-founded.

We can go further and introduce a new derivation algorithm that allows proof search using coinduction trees. We modify the definition of a goal by taking it to be a pair $< A, T >$, where A is an atom, and T is the coinduction tree determined by A, as in Definition 1, in which we restrict the choice of substitutions $\theta_1, \ldots \theta_m$ to the most general unifiers only, in which case T is uniquely determined by A. This restriction to mgus is the *second method for implicit resource handling* of coinductive trees.

Definition 3. *Let G be a goal given by an atom $\leftarrow A$, let T be the coinductive tree induced by A, and let C be a clause $H \leftarrow B_1, \ldots, B_n$. Then goal G' is* coinductively derived *from G and C using mgu θ if the following conditions hold:*

- *A' is a leaf atom, called the* selected *atom, in T.*
- *θ is an* mgu *of A' and H.*
- *G' is given by the atom $\leftarrow A\theta$ and the coinduction tree T' determined by $A\theta$.*

Coinductive derivations resemble *tree rewriting.* Figure 1 shows a coinductive derivation of length 3 for the goal $G =$ `stream(X)` and the program `Stream`

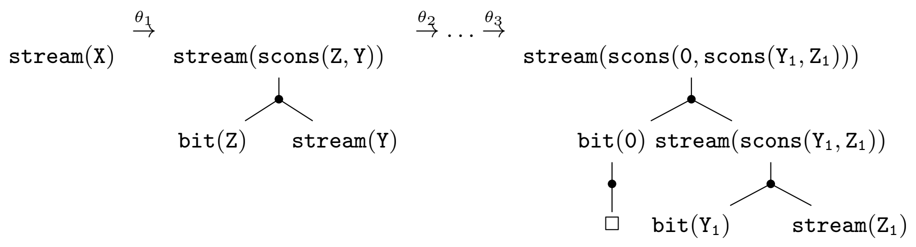

Fig. 1. A coinductive derivation of length 3 for the goal G = `stream(X)` and the program `Stream`, with $\theta_1 = X/scons(Z,Y)$ and $\theta_2 = Z/0, \theta_3 = Y/scons(Y_1,Z_1)$.

from Example 5. Coinductive derivations were proven to be sound and complete relative to the coalgebraic semantics [17]; see [23] for implementations.

One way to achieve well-foundness of corecursion in practice is to *guard* (co-)recursive function applications by *constructors*. In our case, this amounts to requiring that, *for the predicate appearing in the clause head, at least one of its arguments is the term formed by means of the function symbol.*[4] It is easy to see that `Stream` is a *guarded* definition. We will discuss a method to guard logic programs later. Note that guardedness is the *third way to ensure sound implicit handling* of corecursive resources.

Proposition 1. *Given a logic program P and a goal G, a coinductive tree determined by P and G has finite depth if and only if P is guarded.*

As Figure 1 shows, coinductive programs such as `Stream` may give rise to infinite derivations of coinduction trees, in which case implementation may prune the chain of derivations as [9, 24] suggest, or, if infinite production of new streams is desirable, let the coinductive derivations run lazily, stopping each time after generating a finite coinductive tree.

The advantages of this implicit method of handling (co-)recursive computational resources can be summarised as follows. It solves both difficulties that explicit coinductive resource management causes: in response to ★, the method uniformly treats inductive and coinductive definitions, and it can be used to detect non-well-founded cases in both; In response to ★★, it is a corecursive process in spirit. Thus, instead of relying on guarantees of loop regularity, it relies on well-foundness of every single coinductive tree in the process of lazy infinite derivations.

3 Case study 1. Coinductive trees: resource-handling of corecursion

In this case study, we consider the effects of coalgebraic programming on corecursive resource handling, using a classical example from [25].

[4] Note that function symbols can be nullary.

Example 6. [25] Let GC (for graph connectivity) denote the logic program

1. `connected(X,X)` $\leftarrow$
2. `connected(X,Y)` $\leftarrow$ `edge(X,Z)`, `connected(Z,Y)`.

Here, we use predicates "connected" and "edge", to make the intended meaning of the program clear. Additionally, there may be clauses that describe the data base; in our case - edges of a graph whose nodes are labeled by natural numbers, e.g. `edge(0,s(0))` $\leftarrow$, `edge(s(0),s(s(0)))` $\leftarrow$.

The example uses recursion in order to traverse all the connected nodes in a graph. Two kinds of infinite SLD derivations are possible: computing finite or infinite objects.

Example 7. Consider the logic program from Example 6. It is easy to facilitate infinite SLD-derivations by adding a clause that makes the graph cyclic: `edge(s(s(0)),0)` $\leftarrow$. Taking a query $\leftarrow$ `connected(0,Z)` as a goal may lead to an infinite SLD-derivation corresponding to an infinite path starting from `0` in the cycle. However, the object that is described by this program, the cyclic graph with three nodes, is finite.

In the standard practice of logic programming, where the ordering of the clauses is taken as above, the program behaves gracefully, giving finitely-computed answers, but potentially infinitely many times. However, this balance is fragile. For example, the following program (with different orderings of the clauses and of the atoms in the body) results in non-terminating derivations:

Example 8.

1. `connected(X,Y)` $\leftarrow$ `connected(Z,Y)`, `edge(X,Z)`
2. `connected(X,X)` $\leftarrow$

SLD-derivation loops as follows: $\texttt{connected(0,}Z\texttt{)} \rightarrow (\texttt{connected(}Y\texttt{,}Z\texttt{)},\texttt{edge(0,}Y\texttt{)}) \rightarrow (\texttt{connected(}Z_1\texttt{,}Y_1\texttt{)},\texttt{edge(}Y\texttt{,}Z_1\texttt{)},\texttt{edge(0,}Y\texttt{)}) \rightarrow \ldots$. It never produces an answer as it falls into an infinite loop irrespective of the particular graph in question.

There are two details we should note here. First, in traditional LP, the burden of deciding which programs might result in loops like the one above falls completely to the programmer. The programmer is given no semantic tools to help: semantically, the two programs above are equivalent, despite being different observationally! Second, the explicit approach to co-recursion [9, 24] (cf. Section 2) does not handle such cases properly. If the atoms in the programs above are labelled as inductive, the behaviour of Co-LP [9, 24] is exactly as it is for SLD-resolution. If, on the contrary, the atoms are marked as coinductive, we may find the derivation loop terminated as "successful", whereas we should be warned of its being non-well-founded.

Consider the action of coalgebraic semantics [16] and coalgebraic derivations [17] on programs like GC. Two semantic properties will immediately distinguish GC from well-founded programs like `Stream`; compare Figure 2 with Figure 1. In particular, for a single goal, GC gives rise to

1. infinite-breadth coinductive forests.
   This generally happens whenever a program in question contains clauses that have variables in the bodies that do not appear in their heads.
2. infinite-depth coinductive-trees (and hence forests).
   This happens whenever the formulas in the clause heads are *not guarded by constructors* – that is, do not contain function symbols.

Note that, as shown in Figure 1, the coinductive `Stream` program gives rise only to single-tree forests of finite depth, for every given goal. Thus, the semantics can serve the diagnostic purpose of finding potential sources of non-well-foundness in computations.

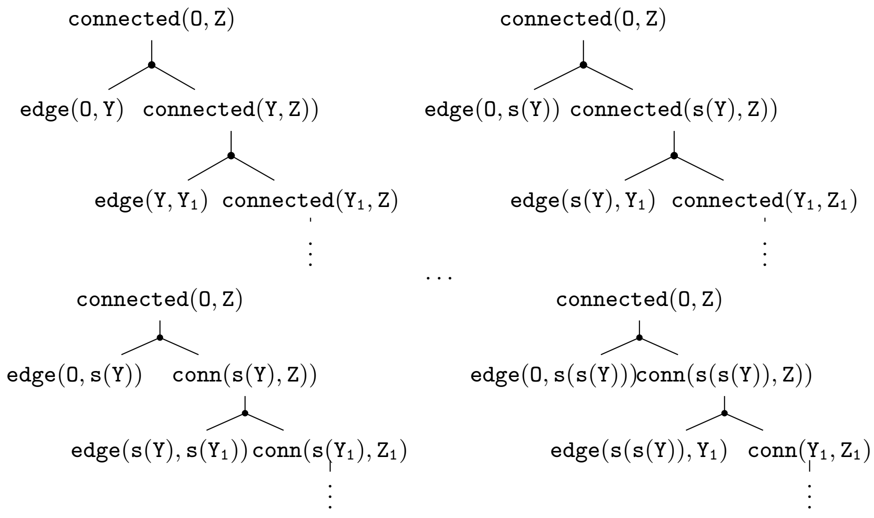

Fig. 2. The coinductive forest of infinite depth and breadth for the program GC and the goal `connected(0,Z)`; `conn` abbreviates `connected`.

On the level of coalgebraic derivations, the two problems are treated differently. The first problem can be solved by using mgus in Definition 1, as explained in Section 2. This allows one to determine coinductive trees uniquely for every goal. The second problem, however, has a deeper, (co-)recursive nature, solved by use of *guardedness*, as explained in Section 2. To make the GC example guarded, we have to reformulate it as follows, see also [23]:

```
connected (X,cons(Node,Path)) ← edge(X,Node), connected(Node,Path)
             connected (X,nil) ←
                     edge(0,0) ←
                  edge(X,s(X)) ←
```

The coinductive derivation for it is shown in Figure 3; and features trees and forests of finite size.

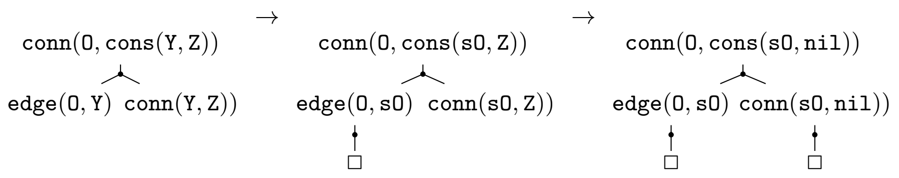

Fig. 3. Finite Coinductive derivations for a guarded variant of the program GC; `conn` abbreviates `connected`; `s0` abbreviates `s(0)`.

4 Concurrent Derivations: Logic versus Control

One of the distinguishing features of logic programming languages is that they allow implicit parallel execution of programs. In the last two decades, an astonishing variety of parallel logic programming implementations have been proposed, see [11] for a detailed survey. The three main types of parallelism used in implementations of logic programs are *and-parallelism*, *or-parallelism* and their combination.

Or-parallelism arises when more than one clause unifies with the goal atom — the corresponding bodies can be executed in Or-parallel fashion. Or-parallelism is thus a way of efficiently searching for solutions to a goal by exploring alternative solutions in parallel. Although in theory this is the most obvious way to parallelize logic programs, in practice, the variable binding needs to be propagated sequentially from root nodes of proof trees down through the leaves, and the dependencies often span several parallel branches.

(Independent) And-parallelism arises when more than one atom is present in the goal, and the atoms do not share variables. That is, given a goal $G = \leftarrow B_1, \dots B_n$, an *And-parallel algorithm* of SLD resolution looks for SLD

derivations for each B_i simultaneously. *Dependent and-parallelism* aims to extend independent and-parallelism by introducing more algorithms for sharing and binding common variables and substitutions.

Predominantly, the existing parallel implementations of logic programming follow Kowalski's principle [18]:

$$\textit{Programs} = \textit{Logic} + \textit{Control}.$$

This principle separates the control component (backtracking, occur check, goal ordering/selection, parallelisation) from the logical specification of a problem (first-order Horn logic, SLD-resolution, unification). Thus the control of program execution becomes independent of programming semantics.

With many parallel solutions on offer, some form of resource handling and process scheduling are inevitable ingredients of parallel logic programming: this is due to the fact that the algorithms of unification and SLD- resolution underlying logic programming are P-complete [26, 13] and cannot themselves be parallelized in the general case. The existing trend for parallel implementations of PROLOG is to hide all additional control-handling algorithms at the level of implementation, away from program specification or semantics [11].

For lack of space, we will illustrate this trend using just one example. One way to express And-Or parallelism in logic programs is through *and-or trees* [10], which consist of *or-nodes* and *and-nodes*. We start with the parallelisable case of variable-free (ground) logic programs, for which and-or trees and coinductive trees coincide, cf. [15].

Definition 4. *[10] Let P be a ground logic program and let $\leftarrow A$ be an atomic goal (possibly with variables). The* and-or parallel 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 $C_i = B^i \leftarrow B^i_1, \ldots, B^i_n$, then the ith or-node has n children given by and-nodes $B^i_1\theta_i, \ldots, B^i_n\theta_i$.*

A naive extension of Definition 4 to the full first-order case yields inconsistent derivations, cf. Example 3, so variable dependencies need to be handled by additional tools. A solution proposed in [10] was given by *composition (and-or parallel) trees.* Composition trees contain a special kind of *composition node.* A 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. Additionally, special binding arrays are kept to synchronize substitutions in different but related branches.

We would like to highlight several properties of this example implementation that are also shared by many other parallel implementations:

$\star$ Although the implementation is called "implicit parallelism" in the literature [11], it boils down to explicit resource handling at a compiler level: this includes both annotating the syntax and maintaining special schedulers/arrays/hash tables to synchronize variable substitutions computed by different processes; these are separated from the language and semantics.

$\star\star$ Issues of logic and control are separated to the point that parallel PROLOG systems are usually built as speed-ups to SLD-resolution and have neither "logic" algorithms nor semantics of their own. In the example of *composition trees*, they are implemented by adding extra features to SLD-resolution. Specifically, composition nodes are handled by binding arrays at the compiler level.

Returning to coalgebraic logic programming (cf. Definitions 1, 3), note that coinductive trees allow for concurrency in that every proof branch in a coinductive tree is computed independently of the others. We can notice that this distinguishes two approaches: *Parallel LP = and-or trees + explicit resource handling* and *Coalgebraic LP = coinductive trees + implicit resource handling.* This gives an altogether different view of resource handling in concurrent logic programming:

1. We avoid explicit resource handling (either at "logic" or "control" level); instead, we uses the *three implicit methods* (cf. Section 2) to control resources. Note that in particular, we have restricted unification to term-matching: in contrast to the inherently sequential unification algorithm [8], it is parallelisable. As a result, parallel proof-search in separate branches of a coinductive tree does not require explicit synchronization of variables.

2. The issues of logic and control are now bound together: coinductive derivation trees provide both logic specification and resource control. Moreover, coalgebraic logic programming [17] comes with its own coalgebraic semantics that accounts for observational behavior of coinductive derivations.

5 Case-study 2. Coalgebraic logic programming and resources for concurrency.

In this case study, our focus is resource-handling of concurrency in logic programming. We start by illustrating ground cases of concurrent derivations: these can be parallelised straightforwardly, and coinductive trees [17] and and-or trees [10] coincide. We will take an inductive logic program `ListNat` as a running example, although a similar case-study could be done with a coinductive logic program such as `Stream`.

Example 9. Let ListNat denote the logic program

```
          nat(0) ←
       nat(s(X)) ← nat(X)
       list(nil) ←
list(cons(X,Y)) ← nat(X), list(Y)
```

Consider the parallel and-or tree [10] for the program `ListNat` and goal `list(cons(0,cons(0,nil)))` in Figure 4. As was discussed in Section 4, it is unification and variable dependencies that make the parallelisation of logic programming hard and require special attention to resource-handling. To illustrate this, consider the naive extension of Definition 4 of and-or trees to the full first-order case, see Figure 5. It shows an unsound derivation for the goal $\texttt{list}(\texttt{cons}(\texttt{X}, \texttt{cons}(\texttt{Y}, \texttt{X})))$ when variable dependencies were omitted and substitutions computed in parallel.

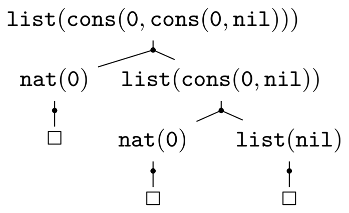

Fig. 4. An and-or parallel derivation for the goal `list(cons(0,cons(0,nil)))`.

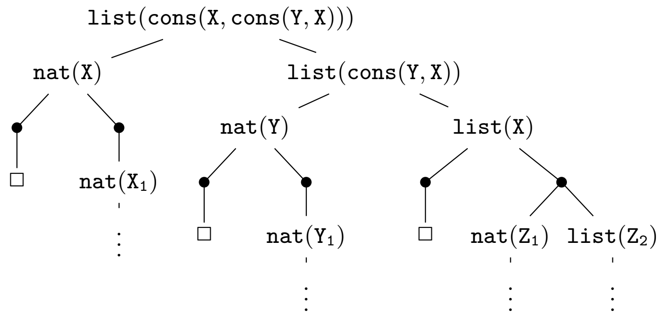

Fig. 5. An unsound refutation by an and-or parallel tree, with $\theta = \{X/0, Y/0, X/nil\}$
.

Real-life implementations of parallel PROLOG [11] employ various mechanisms to synchronise variable substitutions (cf. Section 4), and these typically introduce sequentiality and explicit resource handling at the implementational level. The coalgebraic approach [16, 17] re-establishes the balance in favour of concurrency and implicit handling of resources (by which we mean variables and substitutions in this case). Consider the coinductive derivation for the goal list(cons(X, cons(Y, X))) given in Figure 6. In contrast to the and-or tree, and owing to the restriction of unification to term-matching, every coinductive tree in the derivation pursues fewer variable substitutions than the corresponding and-or tree does (cf. Figure 5). This allows one to keep variables synchronised while pursuing parallel proof-branches in the tree.

Example 10. The coinductive trees from Figure 6 agree with the first part of the and-or parallel tree for list(cons(X, cons(Y, X))) in Figure 5.

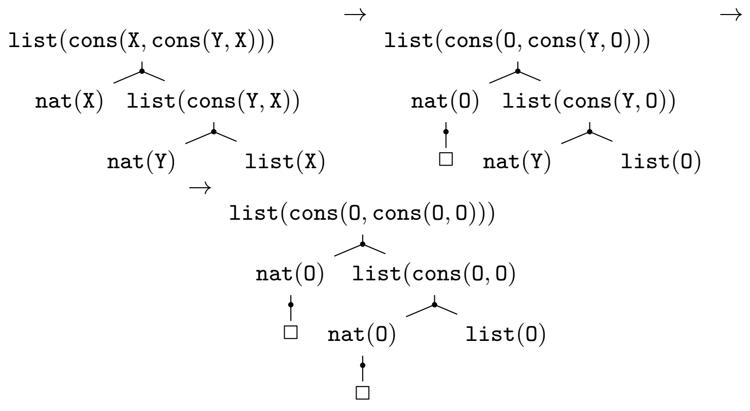

Fig. 6. A coinductive derivation for the goal `list(cons(X,cons(Y,X)))`.

Note that coinductive trees not only allow one to achieve soundness where parallelism normally is not sound, but also they achieve this without any kind of explicit resource handling.

6 Implementation

The first minimal implementation [23] to show the feasibility of the coalgebraic logic programming approach has been developed in Prolog. Constructing derivations for success trees for a given input is modelled as a uniform cost search through the graph of coalgebraic derivation trees connected by the derivation operation. A derivation step here is constrained to first order unification of the first unifiable open leaf that has the lowest level in the tree; cf. Definition 3 and Figures 1,6.

Each expansion of a tree e.g. `list(X)` to `list(nil)` or to $\texttt{list(cons(X}_1\texttt{,Y}_1\texttt{))}$ and then $\texttt{list(cons(s(X}_2\texttt{),Y}_1\texttt{))}$ involves a unification that acts as an implicit barrier to this process. This bounds the resources a tree uses at each derivation step. Only a very thin layer of control for the search is needed on top of the resource handling in the form of a priority search queue.

Using the substitution length of all the substitutions used in the derivation chain as priority ranking, we gain an enumeration order even for an infinite lazy derivation process. Therefore, while an infinite number of success trees can in principle be produced for the goal `list(X)`, the algorithm returns `list(nil)`, `list(cons(0,nil))` and then `list(cons(s(0),nil))` in a finite number of time-steps and keeps producing finite success trees thereafter. The substitution lengths for the first three trees are 1 $\{X/nil\}$, 3 $\{X/cons(X_1, Y_1), X_1/0, Y_1/nil\}$ and 4 $\{X/cons(X_1, Y_1), X_1/s(X_2), X_2/0, Y_1/nil\}$.

The use of term-matching to traverse and expand trees allows for parallelization without synchronization while operating directly on trees. On the search level of the algorithm it is possible to simultaneously dequeue, check and expand derivation trees to introduce further parallelization.

7 Conclusions and Future Work

The main feature of the coalgebraic logic programming approach is its generality: it is suitable for both inductive and coinductive logic programs, for programs with variable dependencies or not, and for programs that are unification-parallelisable or inherently sequential. Many distinctions that led to a variety of engineering solutions in the design of corecursive and concurrent logic programs [11, 9, 24] are erased here, with resource-handling delegated to a *logic* algorithm; and issues of logic and control, semantics and execution, become inseparable.

In the future, we plan to investigate the integration of coalgebraic LP with methods of resource handling in state-of-the-art coinductive LP [11, 9, 24], as well as in modern concurrent logic programming systems [11]. Furthermore, we would like to investigate whether Coalgebraic LP has potential to play a positive role in type inference, cf. [3].

References

1. G. Amato, J. Lipton, and R. McGrail. On the algebraic structure of declarative programming languages. *Theor. Comput. Sci.*, 410(46):4626–4671, 2009.
2. D. Ancona and G. Lagorio. Idealized coinductive type systems for imperative object-oriented programs. *RAIRO - Theoretical Informatics and Applications*, 45(1):3–33, 2011.
3. D. Ancona, G. Lagorio, and E. Zucca. Type inference by coinductive logic programming. In *TYPES*, volume 5497 of *LNCS*, pages 1–18, 2009.
4. A. Asperti and S. Martini. Projections instead of variables: A category theoretic interpretation of logic programs. In *ICLP*, pages 337–352, 1989.
5. F. Bonchi and U. Montanari. Reactive systems, (semi-)saturated semantics and coalgebras on presheaves. *Theor. Comput. Sci.*, 410(41):4044–4066, 2009.

6. R. Bruni, U. Montanari, and F. Rossi. An interactive semantics of logic programming. *TPLP*, 1(6):647–690, 2001.
7. M. Comini, G. Levi, and M. C. Meo. A theory of observables for logic programs. *Inf. Comput.*, 169(1):23–80, 2001.
8. C. Dwork, P. Kanellakis, and J. Mitchell. On the sequential nature of unification. *J. Logic Prog.*, 1:35–50, 1984.
9. 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.
10. G. Gupta and V. Costa. Optimal implementation of and-or parallel prolog. In *Conf. proc. on PARLE'92*, pages 71–92. Elsevier, 1994.
11. G. Gupta, E. Pontelli, K. Ali, M. Carlsson, and M. Hermenegildo. Parallel execution of prolog programs: a survey. *ACM Trans. Computational Logic*, 2012. 126 pages, in print.
12. M. Jaume. On greatest fixpoint semantics of logic programming. *J. Log. Comput.*, 12(2):321–342, 2002.
13. P. C. Kanellakis. Logic programming and parallel complexity. In *Foundations of Deductive Databases and Logic Prog.*, pages 547–585. Morgan Kaufmann, 1988.
14. Y. Kinoshita and J. Power. A fibrational semantics for logic programs. In *Proc. Int. Workshop on Extensions of Logic Programming*, volume 1050 of *LNAI*, 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*, 2010.
16. E. Komendantskaya and J. Power. Coalgebraic derivations in logic programming. In *CSL*, LIPIcs, pages 352–366. Schloss Dagstuhl, 2011.
17. E. Komendantskaya and J. Power. Coalgebraic semantics for derivations in logic programming. In *CALCO*, LNCS, pages 268–282. Spinger, 2011.
18. R. Kowalski. *Logic for problem Solving.* Elsevier, Amsterdam, 1979.
19. J. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, 2nd edition, 1987.
20. R. Milner. A theory of type polymorphism in programming. *J. Comput. Syst. Sci.*, 17(3):348–375, 1978.
21. M. Odersky, M. Sulzmann, and M. Wehr. Type inference with constrained types. *TAPOS*, 5(1):35–55, 1999.
22. E. Pontelli and G. Gupta. On the duality between or-parallelism and and-parallelism in logic programming. In *Euro-Par*, pages 43–54, 1995.
23. M. Schmidt and E. Komendantskaya. Coalgebraic logic programming: Implementation, 2012. www.computing.dundee.ac.uk/staff/katya/MLCAP-man.
24. L. Simon and et al. Co-logic programming: Extending logic programming with coinduction. In *ICALP*, volume 4596 of *LNCS*, pages 472–483. Springer, 2007.
25. L. Sterling and E. Shapiro. *The art of Prolog.* MIT Press, 1986.
26. J. D. Ullman and A. V. Gelder. Parallel complexity of logical query programs. *Algorithmica*, 3:5–42, 1988.