Guarding (Co)Recursion in Coalgebraic Logic Programming

JÓNATHAN HERAS
School of Computing, University of Dundee, UK
(*e-mail:* jonathanheras@computing.dundee.ac.uk)

EKATERINA KOMENDANTSKAYA
School of Computing, University of Dundee, UK
(*e-mail:* katya@computing.dundee.ac.uk)

MARTIN SCHMIDT
Institute of Cognitive Science, University of Osnabrück, Germany
(*e-mail:* martisch@uos.de)

submitted 1 January 2003; revised 1 January 2003; accepted 1 January 2003

Abstract

Coalgebraic Logic Programming (CoALP) is a dialect of Logic programming designed to work with coinductive definitions of infinite objects. Its main goal is to introduce guarded lazy corecursion akin functional theorem provers into logic programming. In this paper, we give the first full formal account of guarded corecursion in CoALP, and present its implementation.

KEYWORDS: Lazy Languages, Eager Languages, Corecursion, Termination, Guardedness, Loop Detection.

1 Introduction

Logic Programming (LP) was conceived as a *recursive* programming language for first-order logic. Prolog and various other implementations of LP feature *eager* derivations, and therefore *termination* has been central for logic programming (de Schreye and Decorte 1994).

In the last decade, another trend has been developed within the declarative programming community: the theory of coinduction and corecursion (Coquand 1994; Abel et al. 2013). A representative example of a coinductive programming is to reason about an infinite data structure, for example an infinite stream of bits. In LP notation, the `Stream` definition would look like this:

Example 1.1 (BitStream)

$$\begin{aligned} 1.\,bit(0) &\leftarrow \\ 2.\,bit(1) &\leftarrow \\ 3.\,stream(scons(X,Y)) &\leftarrow bit(X), stream(Y) \end{aligned}$$

BitStream matches similar definitions in e.g. Haskell, Coq or Agda.

The tradition (Coquand 1994) has a dual notion to termination for well-behaving *corecursion* – and that is *productivity*. If termination imposes the condition that any call to an inductively

defined predicate like `bit` must terminate, then productivity says that every call to a coinductive predicate like `stream` must *produce* some partially *observed* structure in a finite number of steps. E.g. calling `stream(X)?`, the program must compute an answer `scons(0,Y)` observing the component `0` in finite time. Moreover, the productivity imposes a second condition: the computation must be able to proceed corecursively, e.g. `Y` must be an infinite productive datastructure. This situation is explained in e.g. (Abel et al. 2013; Bertot and Komendantskaya 2008).

This distinction between the terminating recursion and productive corecursion in practice disallows non-well-founded cases from the languages where total evaluation of terms is important (e.g. Agda and Coq). For example, in LP notation, the following (co)recursive logic program would be neither terminating, nor productive:

Example 1.2 (BadStream) 1. $badStream(X) \leftarrow badStream(X)$

Thus, some languages simply disallow such definitions, by imposing a system of (static) syntactic checks, sometimes called *guards*. Unlike BitStream, BadStream is lacking a guarding constructor like `scons` that allowed the finite observations. The central *"guarding" property* of the method is: *if a program is guarded by constructors, it is productive*. Termination being an undecidable property in the general case, the reverse would not normally hold; thus guardedness banishes some perfectly productive functions; see e.g. (Bertot and Komendantskaya 2008).

There are two principle ways of treating corecursive computations in functional languages: lazy evaluation (Coquand 1994; Abel et al. 2013) and loop-detection/equation solving (Jeannin et al. 2013), including the work on *modes* (Rohwedder and Pfenning 1996). The former was successfully implemented in lazy functional languages like Haskell, Coq, Agda, and the latter two – in eager functional languages like ML or higher-order LP.

Matching this trend, coinductive logic programming (CoLP) has been proposed (Gupta et al. 2007; Simon et al. 2007), featuring a combination of eager evaluation and loop analysis. In simplified terms, the loop detection would allow to return an answer `X=scons(0,X)` for a goal `stream(X)?`; by noticing the *"regular"* calling of Clause 3 in the derivations.

Termination by loop detection – be it in functional or logic programming – is inevitably restricted by the power of loop detecting algorithms; and they can always be tricked.

Example 1.3 (FibStream) *The program FibStream defines the stream of Fibonacci numbers, and allows to access the nth element of the stream by calling* fib.

$$
\begin{aligned}
1.\, fibs(X,Y,cons(X,S)) &\leftarrow add(X,Y,Z),\ fibs(Y,Z,S)\\
2.\, nth(0,cons(X,S),X) &\leftarrow\\
3.\, nth(s(N),cons(X,S),Y) &\leftarrow nth(N,S,Y)\\
4.\, fib(N,X) &\leftarrow fibs(0,s(0),S),\ nth(N,S,X)
\end{aligned}
$$

In CoLP, calls to e.g. `fib(s(s(0)),X)` fall into infinite computations that are not handled by the loop detection procedure. Examples like FibStream show that there is a potential place for lazy corecursion in LP. It is intuitively clear that, to find the *n*th element of such a stream should take a finite number of derivation steps. The only obstacle is: LP is an eager language.

In (Komendantskaya and Power 2011; Komendantskaya et al. 2014a), we gave an abstract, semantics-driven description of Coalgebraic Logic Programming (CoALP) – a language with inherent lazy behaviour; we will briefly remind its main properties in Section 2. At the heart of CoALP was the construction of coinductive trees – a finitely-sized structure – that provided the

finitely observable transition steps in a *potentially infinite* computation. To ensure productivity, first steps have been made in (Komendantskaya et al. 2014a) to impose a system of *guards*; but it was observed to be incomplete and the guarding property was not proven.

Initial formulation of CoALP was lacking some important components: a complete set of guards, the proof of the guarding property; and finite decision procedure for deciding soundness of potentially infinite derivations. This paper solves the first two problems. Section 3 introduces static guardedness checks (similar to (Komendantskaya et al. 2014a)) and Section 4 introduces additional, dynamic guardedness checks for CoALP, and proves termination and soundness of the CoALP guardedness checks. Finally, Section 5 discusses the language implementation, available at (Komendantskaya et al. 2014b), and concludes the paper.

2 Coalgebraic Logic Programming

CoALP uses the standard syntax of Horn-clause logic programming cf. (Lloyd 1987) or Appendix A, but it substitutes the standard mechanism of SLD-resolution by coalgebraic derivations. We start with the definition of CoALP's derivation procedure, as was introduced in (Komendantskaya et al. 2014a). We also introduce some changes to CoALP definitions: we introduce predicate annotations as in CoLP (Gupta and Costa 1994); and optimise the derivation procedure (Definitions 2.3 and 2.4).

In CoALP, subgoals in derivations are given by coinductive trees, not by atoms.

Definition 2.1 *Let P be a logic program and $G = \leftarrow A$ be an atomic goal. The* coinductive 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 (an atom) or an or-node (given by $\bullet$).*
- *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 mgus $\theta_1, \ldots, \theta_m$, then A' has exactly m children given by or-nodes, such that, for every $i \in \{1, \ldots, m\}$, the ith or-node has n_i children given by and-nodes $B_1^i\theta_i, \ldots, B_{n_i}^i\theta_i$.*
  In such a case, we say C_i and θ_i are internal resolvents *of A'.*

It is worth mentioning the resemblance of coinductive trees to And-Or-trees (Gupta and Costa 1994), further discussed in (Komendantskaya et al. 2014a; Komendantskaya et al. 2014c); and the restriction of unification to term matching in $A' = B_1\theta_1 = \ldots = B_m\theta_m$ that distinguishes coinductive trees from SLD-trees or And-Or-trees. Note also that the coinductive tree T is uniquely determined by its root A, cf. (Komendantskaya et al. 2014a).

Example 2.1 *Figure 1 shows three coinductive trees for various goals in BitStream.*

We are making one crucial assumption throughout this section: *every coinductive tree has a finite size*. For BitStream and FibStream, it is satisfied trivially by term-matching, but it is not a trivial matter in the general case. The next two sections will be devoted to this issue.

Now we are ready to define derivations between trees. In the initial formulations of CoALP, the derivations were made without annotations of predicates as "inductive" or "coinductive" as it is done in CoLP (Gupta et al. 2007; Simon et al. 2007). However, this annotation-free approach does not work equally well for all programs. For BitStream, the tree transitions work well with no

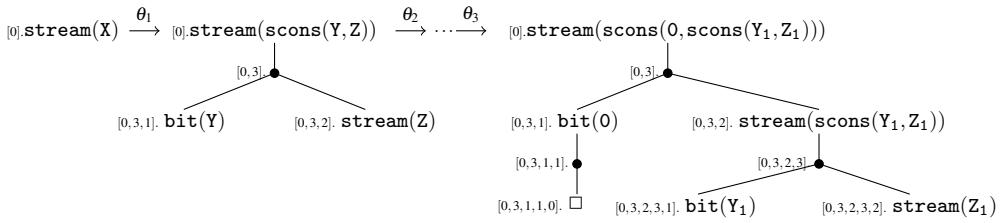

Fig. 1. *The three coinductive trees representing a coinductive derivation for the goal G = stream(X) and the program BitStream, with* $\theta_1 = \mathtt{X}/\mathtt{scons}(\mathtt{Y},\mathtt{Z})$, $\theta_2 = \mathtt{Y}/0$ *and* $\theta_3 = \mathtt{Z}/\mathtt{scons}(\mathtt{Y}_1,\mathtt{Z}_1)$. *The nodes are annotated with their values for the function N.*

annotations, see Figure 1; whereas for FibStream, equal treatment of inductive and coinductive predicates makes for very inefficient computations, as coinductive nodes tend to expand faster than the inductive nodes return, see Figure 2 and Appendix B. This is why, we annotate all coinductive predicates.

An atom can appear several times in different and-nodes of a coinductive tree; the function N is used to distinguish the nodes.

Definition 2.2 *Let P be a logic program with enumeration of its clauses $(C_1,\ldots,C_n)$, G be an atomic goal, and T be the coinductive tree determined by P and G. The function N : nodes $\to$ List$\mathbb{N}$ assigns to every node n of T a unique list of natural numbers as follows:*

- *if n is a root node; then, $N(n) = [0]$ where $[0]$ denotes the list with 0 as its only element.*
- *if n is an or-node; then, $N(n) = N(p) +\!\!+ [i]$ where p is the parent node of n, i is the number of the clause C_i that was used to derive n from p, and $+\!\!+$ denotes the concatenation of lists.*
- *if n is an and-node; then, $N(n) = N(p) +\!\!+ [j]$ where p is the parent node of n, and j is the number of the body atom of the clause used to obtain n.*

Definitions 2.3 and 2.4 formalise the notion of coinductive derivations; and are modified versions of similar definitions in (Komendantskaya et al. 2014a). In particular, they impose constraints on the coinductive tree transitions (resolvents): cf. items $*$ and $**$.

Definition 2.3 *Let $G = \langle A, T\rangle$ be a goal given by an atom $\leftarrow A$ and the coinductive tree T 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 the mgu θ if the following conditions hold:*

★ *$N(Q(\bar{t}))$ is a node in T.*
★★ *θ is an* mgu *of $Q(\bar{t})$ and H.*
★★★ *G' is given by the (coinductive) tree $T\theta$ with the root $A\theta$.*

Generally, $T\theta$ may differ from the coinductive tree for $A\theta$ (this happens when θ affects variables in T but not in A). When $T\theta$ expands, CoALP first makes substitution in the nodes, and then extends every node according to the definition of the coinductive tree. We will abuse the term convention, and call all such $T\theta$'s coinductive trees.

Figure 1 shows one possible coinductive derivation for BitStream. Coinductive derivations resemble *tree rewriting*. They produce the "lazy" corecursive effect: derivations are given by potentially infinite number of steps, where each individual step is executed in finite time.

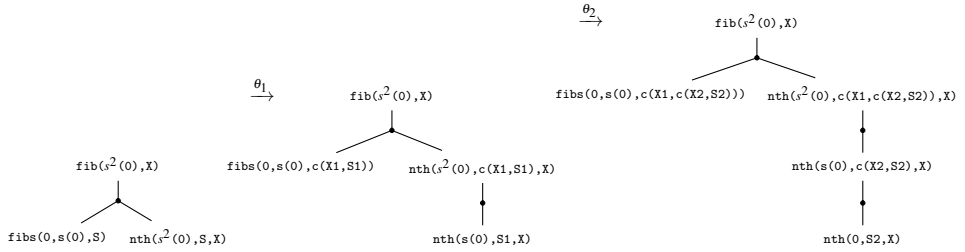

Fig. 2. *First steps of the derivation for the goal* `fib(s(s(0)),X)` *–* s^2 `(0)` *denotes* `s(s(0))` *– and the program FibStream, with* $\theta_1 =$ `S/cons(X1,S1)` (`c(X1,S1)` *denotes* `cons(X1,S1)`) *and* $\theta_2 =$ `S1/cons(X2,S2)`. *As* `nth` *is an inductive predicate, and* `fibs` *is coinductive; resolvents for* `nth` *nodes are given priority. The full derivation is shown in Appendix B.*

Definition 2.4 *A* coinductive derivation *of* $P \cup \{G\}$ *consists of a sequence of goals* $G = G_0, G_1, \ldots$ *and a sequence* $\theta_1, \theta_2, \ldots$ *of mgus such that each* G_{i+1} *is derived from a node* $N(A) \in T_i$ *and a clause* C *using a non-empty substitution* θ_{i+1}. *In this case,* $< N(A), C, \theta_{i+1} >$ *is called a* resolvent.

$*$ *Moreover, no resolvent is allowed to appear more than once in the coinductive derivation.*

$**$ *We additionally require that no coinductive node* $C \in T_i$ *gives rise to a derivation step if there is an inductive node* $I \in T_i$ *that could give rise to a coinductive derivation step satisfying the above conditions.*

Example 2.2 *Figure 2 shows the first steps in the coinductive derivation for FibStream. Note that* $**$ *of Definition 2.4 forces prioritisation of inductive resolvents over coinductive resolvents.*

As coinductive trees have and-or parallel branches, a successful derivation needs only certain subtrees of a coinductive tree to be successful, the next definition formalises this idea.

Definition 2.5 *Let* P *be a logic program,* G *be an atomic goal, and* T *be the coinductive tree determined by* P *and* G. *A subtree* T' *of* T *is called a* coinductive subtree *of* T *if it satisfies the following conditions:*

- *the root of* T' *is the root of* T *(up to variable renaming);*
- *if an and-node of* T *belongs to* T', *then one of its children belongs to* T'.
- *if an or-node of* T *belongs to* T', *then all its children belong to* T'.

A finite coinductive (sub)tree is called a success (sub)tree *if its leaves are empty goals.*

A *coinductive refutation* of $P \cup \{G\}$ is a finite coinductive derivation in which the last tree contains a success subtree. The derivation stops with mark "failure" if, for some goal G_i and the corresponding tree T_i, T_i does not contain success subtrees and no further resolvent can be found. The approach to success and failure of derivations corresponds to the inductive approach of the SLD-resolution. Deciding validity of coinductive derivations, like e.g. the ones in Figures 1 and 2 is a separate task, which would require extra productivity conditions. The following two examples illustrate the nature of the problem.

Example 2.3 (Comember) *The predicate* `comember` *is true if and only if the element* `X` *occurs an infinite number of times in the stream* `S`.

```
1.drop(H,scons(H,T),T)   ←
2.drop(H,scons(H1,T),T1) ←  drop(H,T,T1)
3.comember(X,S)          ←  drop(X,S,S1), comember(X,S1)
```

To check the validity of a query in Comember for an arbitrary stream, would require satisfaction of two conditions: 1). finding an element to `drop` in a finite time, 2). finding guarantees that this finite computation will be repeated an infinite number of times for the given stream. CoLP (Gupta et al. 2007; Simon et al. 2007) would handle such a case for all streams that consist of a regular finite repeating pattern and will not be able to handle cases when the input stream is not regular. CoLP would fail to derive true or falsity of e.g. the query `comember(0,nats)`, where `nats` is a stream of natural numbers, as CoLP falls into an infinite non-terminating computation and fails to produce any response to the query. CoALP in its current implementation will handle any case of corecursion, including `comember(0,nats)`, but in its lazy, and therefore partial, style; see Example 3.1. Similarly, FibStream falls into an infinite loop with CoLP, but unfolds lazily with CoALP, see Figure 2 and Appendix B.

Example 2.4 *In the following program `stream2` defines pairs of streams, in which the first stream consists of a successor of a natural number, and the other is a copy of the tail of the first stream; `stream3` defines a similar pair of streams where one is a full copy of another:*

```
1.stream2(scons(s(X),Y),Z)  ←  nat(X), stream2(Z,Y)
2.stream3(scons(s(X),Y),Z)  ←  nat(X), stream3(Z,scons(s(X),Y))
```

where `nat` is an inductively defined predicate for natural numbers by constructors `0` and `s`.

For the goal `stream2(X,Y)` CoLP would give the answers `X = scons(s(0),X), Y = X`, concluding the truth for that variable instantiation. For the goal `stream3(X,Y)`, CoLP computes substitution `X = scons(s(0),Z1), Y = scons(s(0),Z1)`. Suddenly, the tails of `X` and `Y` look like finite structures! – although `stream3` fails inductively in SWI-Prolog.

Similarly to the way in which the absence of a regular loop in FibStream did not signify that queries for FibStream were false, the possibility to obtain the two sets of "true" answers above does not mean we have a consistent analysis of the stream structure, productivity, and size. This illustrates the challenge of formalisation, testing and certification of conditions for truth and falsity of queries in the presence of infinite data structures. We will concentrate only on termination issues in the rest of the paper.

3 Static Guardedness Checks

Definitions of the previous section were relying on the *productivity assumption* that every coinductive tree is of finite size. We devote this section to the formal discussion of this property.

Definition 3.1 *Let P be a logic program, P is* productive *if for any goal G, the coinductive tree for $P \cup \{G\}$ has a finite size.*

Note that coinductive trees serve both to provide a measure for lazy steps, and for defining productivity for CoALP. If a program is productive, the coinductive derivations can be potentially infinite, but each coinductive tree in the derivation is of finite size. Comember (Example 2.3) is un-productive for e.g. the query `comember(X,S)` as `comember` lacks a guarding constructor.

We start introducing guardedness checks to ensure productivity. We start by defining guards for individual clauses, and then extend this inductively to handle composite programs. The intuitive idea is to ensure that every coinductive program behaves like BitStream: BitStream is guarded by

the coinductive constructor (or "guard") `scons`; and hence all coinductive trees for it are finite, see Figure 1.

Even if we have one inductive clause to check, deciding on multiple guards for it in the general case is not as simple as for the BitStream case, where there was no other choice for the guard but `scons`. Generally, there may be various cases when multiple guards arise, for several different reasons. Two (or more) guards may arise when we nest functions symbols in one argument of the head, as `s` and `scons` in Example 2.4. Two (or more) guards may arise within different arguments in the head, and reduce in different atoms of the body: cf. Example 2.4. Two (or more) guards may arise from different clauses containing same predicate in the head, as e.g. `0` and `s` are both constructors of `nat`. The definition of *guard* is constructed to allow all such cases.

Definition 3.2 (GC1. Static Guardedness Check) *Given a program P and a clause $C \in P$ of the form $Q(\bar{t}) \leftarrow \overline{body}$, where $\bar{t}$ is the list of all arguments $t_1, \ldots, t_n$ of Q and $\overline{body}$ is a list of atoms, we say that C and Q* have a guard *(or* are guarded*) if, for every occurrence of $Q(\overline{t'})$ in $\overline{body}$, there is a term $t_i \in \bar{t}$, and there is a function symbol f, satisfying the following conditions:*

- *f occurs in t_i exactly j times, ($j \geq 1$);*
- *f occurs exactly m times in the ith term in $Q(\overline{t'})$ (call it t'_i), with $m < j$;*
- *if $f^j \in t_i$ has arguments containing variables $\overline{x_i}$, and $f^m \in t'_i$ has arguments containing variables $\overline{x'_i}$, then $\overline{x'_i} \subseteq \overline{x_i}$. (The rule applies to f^0, in which case $\overline{x'_i}$ are all variables in t'_i.)*

In such case, we say that C and Q have a guard f in position *i; or sometimes simply f* is a guard of *Q. We also say that variables in $\overline{x_i}$ are* guarded by *f.*

Example 3.1 (Guarded Comember) *The definition of Comember used in CoLP (Example 2.3) does not pass GC1, as `comember` is missing a constructor. We propose the following guarded definition instead, thereby simplifying it and reducing an extra argument to `drop`.*

```
1.drop(H,scons(H,T))   ←
2.drop(X,scons(H,T))   ←  drop(X,T)
3.gcomember(X, scons(H,T))  ←  drop(X,scons(H,T)), gcomember(X,T)
```

The goal `gcomember(0,nats)` will lazily search for 0 *in an infinite stream of natural numbers, but it never falls into un-productive coinductive trees, as CoLP would do.*

Definition 3.2 allows a choice of constructors and a choice of argument positions where to check constructor reductions; and there generally may be more than one guard for every given clause or predicate. But the definition is strict with respect to the argument positions under which the constructors are reducing. The `stream2` and `stream3` of Example 2.4 do not pass GC1, because of the variable swap. The programmer will be prompted to reformulate the program, and the following alternative will pass GC1 in CoALP.

Example 3.2 (Guarded copy of the program from Example 2.4)

```
1.gstream2(scons(s(X),Y),Y)  ←  nat(X), gstream2(Y,scons(s(X),Y))
2.gstream3(scons(s(X),Y),Z)  ←  nat(X), gstream3(Y,scons(s(X),Y))
```

In CoALP, the program passes GC1; `Y` gives the tail of the initial stream in `gstream2`, and `Z` – the whole copy of the initial stream in `gstream3`. CoLP does not distinguish these programs and gives same answer for `stream2`, `gstream2` and `gstream3`; see Example 2.4.

We assume that at the pre-processing time, each predicate is assigned a set of guards, as specified in the definition above. Definition 3.2 would be a sufficient guardedness check for some programs, like BitStream, where there is only one (co)inductive clause; but not in the general case. LP in general is not compositional, that is, composing two programs may yield a program that has semantic properties not present in the initial programs. Same rule applies in CoALP: if both P_1 and P_2 are productive, their composition is not guaranteed to be a productive program.

Example 3.3 *The below program passes GC1 for each individual clause, but is un-productive.*

```
1.stream2(X,Y)  ←  stream3(X,Y)
2.stream3(X,Y)  ←  stream2(X,Y)
```

We now turn to guardedness checks for such "composed" programs.

Definition 3.3 *Let P be a logic program, G be an atomic goal, and T be the coinductive tree determined by P and G. A subtree T' of T is called a* coinductive tree branch *(or just* coinductive branch*) of T if it satisfies the following conditions:*

- *the root of T' is the root of T (up to variable renaming);*
- *if an and-node belongs to T', then one of its children belongs to T'.*
- *if an or-node belongs to T', then one of its children belong to T'.*

We say a coinductive branch is a success branch*, if its last node is a $\Box$.*

Definition 3.4 *Given a logic program P, a goal G, and the coinductive tree T for $P \cup \{G\}$, we say T* contains a loop *if there exists a coinductive branch B of T, such that:*
there exists a predicate $Q \in P$ that appears at least twice in B, with $Q(\bar{t'})$ being a child of $Q(\bar{t})$; and, there exists a clause C in P, and C was an internal resolvent for both of these nodes. If additionally the clause $Q(\bar{t}) \leftarrow Q(\bar{t'})$ is guarded, we call the loop $(Q(\bar{t}), Q(\bar{t'}))$ guarded*; otherwise, we say the loop is* un-guarded*. A coinductive tree is un-guarded if it contains an un-guarded loop.*

Definition 3.5 (GC2. Composite Static Guardedness Check) *Given a program P, for every clause C of the form $A \leftarrow B_1, \ldots, B_n$, construct a coinductive tree with the root A. If at least one clause gives rise to an un-guarded coinductive tree, the program P is un-guarded.*

Proposition 3.1 *For any logic program P, Composite Static Guardedness check terminates.*

Proof. Non-terminating GC2 will mean there is an infinite coinductive tree arising from at least one clause head. But, since there is a finite number of predicates in a program, it would need to contain a loop. If the loop is guarded, then the constructors (or guards) will reduce, and that reduction will have an end. Then it must contain an un-guarded loop; but then it will terminate by condition of GC2. $\Box$

GC1–GC2 handle programs similar to Example 3.3, but they are still insufficient. The following un-productive program (semantically equal to `stream2` and `gstream2`) passes the checks.

Example 3.4 (Un-productive Program (see Figure 3) that passes GC1-GC2)

```
1.stream2'(scons(s(X),Y),Z)  ←  nat(X), stream-aux(scons(s(X),Y),Z)
2.stream-aux(X,scons(s(Y),Z))  ←  nat(Y), stream2'(X,scons(s(Y),Z))
```

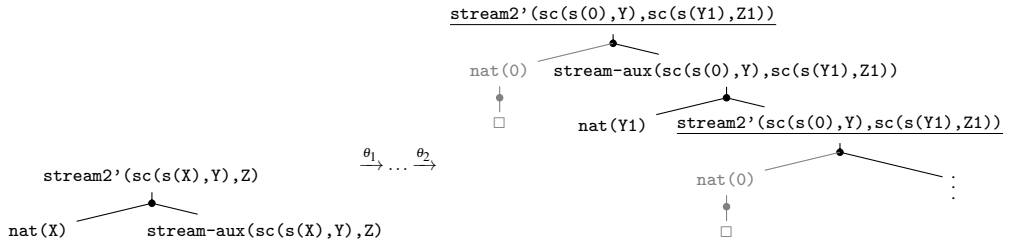

Fig. 3. *Coinductive derivation of `stream2'(sc(s(0),Y),sc(s(Y1),Z1))` (where `sc` denotes `scons`) and the program from Example 3.4 producing an infinite coinductive tree, with $\theta_1 =$ `X/0` and $\theta_2 =$ `Z/sc(s(Y1),Z1)`. The figure also represents one GC-derivation generated during GC3, in grey are the isolated branches. GC3 detects the un-guarded loop; see the underlined atoms.*

4 Dynamic Guardedness checks

The failure of the static guardedness checks for Example 3.4 is symptomatic, and can occur in different guises. It is a direct consequence of the fact that term-matching acting instead of full unification within one coinductive tree omits some important goal instantiations. So far, the restricted nature of term-matching was serving our purposes of guarded (co)-recursion in CoALP; and now we have to pay the bills and compensate the restriction with some extra unification steps. This idea underlies the dynamic guardedness check we introduce in this section.

We note here that productivity is a general semantic property, and as such is formulated with respect to any goal that may be formed using the given syntax. Intuitively, in the worst case, one has to check all atoms that may arise from the program syntax, to make sure that all possible goals have been checked for un-guarded loops. Such computation would be very inefficient. The dynamic guardedness check we introduce here restricts the set of all possible derivations for an atom to a finite subset of derivations; and our main task is to ensure that the subset we construct is capable of finding un-guarded loops (if such may arise) by finite means.

Definition 4.1 *Given a coinductive tree T, and its coinductive branch B, we say B is* isolated *if: either B contains a guarded loop, or B is a success branch.*

A *GC-derivation* is a coinductive derivation in which no resolvent comes from an isolated branch.

Definition 4.2 (GC3. Dynamic Guardedness Check) *Suppose P is a program containing clauses $C_1,\ldots,C_m$. For every clause $C_i = A \leftarrow B_1,\ldots B_k$ construct the set of all GC-derivations $D = D_1,\ldots D_n$ for A, where each D_j is a distinct GC-derivation for A; imposing the following termination conditions. For every GC-derivation $D_j = [(G_0 = A),\ldots,G_m]$, terminate the derivation D_j if, for some $T_i \in D_j$: either T_i contains an un-guarded loop, in which case, report un-guardedness; or every branch of T_i is isolated.*

Note that GC3 is essentially a pruned breadth-first search for all resolvents: the breadth-first character comes from considering all possible coinductive derivations arising from a given goal tree. The isolation of some branches makes sure that certain parts of the breadth-first search are not recomputed without need and terminates as early as possible.

GC3 is able to detect un-productive cases like Example 3.4, see Figure 3. Various conditions on GC-derivations are set to make sure that CoALP's GC-derivations are efficient. For example, for BitStream, there is an infinite number of infinite derivations arising from the goal

`stream(scons(X,Y))`. However, for the same program, there will only be a finite number of finite GC-derivations for that goal. A fragment of one possible GC-derivation for `stream(scons(X,Y))` is shown in Figure 1. In the right-most coinductive tree the success branches and the guarded loop $\texttt{stream}(\texttt{scons}(0,\texttt{scons}(\texttt{Y}_1,\texttt{Z}_1))) \leftarrow \ldots \leftarrow \texttt{stream}(\texttt{scons}(\texttt{Y}_1,\texttt{Z}_1))$ are isolated. Therefore, that GC-derivation will terminate on the next step, after computing a resolvent for $\texttt{bit}(\texttt{Y}_1)$, which in turn will be isolated. It now remains to formally prove the properties of GC1-GC3.

Lemma 4.1 *Given a Logic Program P, Guardedness Checks GC1-GC3 terminate.*

Proof. Note that for any program P, the number of clauses and distinct predicates occurring in P is always finite and fixed. Suppose GC1-GC3 do not terminate, that is, there is a clause $A \leftarrow B_1,\ldots,B_n$, such that a set of GC-derivations for A contains a GC-derivation D_j, such that D_j is infinite and has the shape $A = G_0, G_1, G_2, \ldots$.

The construction of coinductive trees for each of these subgoals is necessarily finite, by Proposition 3.1. It remains to consider GC3. Consider the limit of the GC-derivation D_j. It must be an infinite tree, as by definition of the GC-derivation, derivation steps lead to tree expansion. A GC-derivation will not include resolvents arising from loops or success branches. Then D_j must have resolvents for atoms that themselves were not part of a looping branch, but were created by a looping predicate, like e.g. `bit(X)` may be called repeatedly when BitStream is executed. However, this is impossible. By construction of GC-derivations, all branches containing looping predicates have been isolated, therefore cannot give rise to further derivation steps involving terms from looping predicates; in the same time, the non-looping predicates will exhaust the range of their possible resolvents in finite time. □

Theorem 4.1 *Given a Logic Program P, P is productive if P is guarded.*

Proof. Suppose P is un-productive, that is, there exists a goal with an atom G such that the coinductive tree T_G is of infinite depth. Then it must contain an infinite un-guarded loop $Q(\bar{t}) \leftarrow \ldots \leftarrow Q(\bar{t'})$ for some predicate Q and some terms $\bar{t}$ and $\bar{t'}$. Also, G cannot be equivalent (up to variable renaming) to any of the clause heads in P – otherwise, the loop would have been detected by GC1-GC2. Now, suppose GC1-GC3 did not detect P being un-guarded. We have two cases.

Case 1. Un-guarded loop cannot be detected by GC1-GC3 because it cannot be found by coinductive derivations in principle. (We make a stronger statement and assume that GC3 is not restricted by branch isolation.) Coinductive derivations generally would not compute certain term instantiations; e.g. `bit(cons(X,Y))` would never occur in coinductive derivations for BitSream. Suppose breadth-first CoALP did not compute an instance of G, but exactly that instance of G caused the un-guarded loop to appear.

Is it possible that term instantiations found by the set of all possible coinductive derivations (cf. GC3), fail to find an instance of the loop caused by G? If G caused the above un-guarded loop, there must be a clause $Q(\bar{t''}) \leftarrow body$ in P that triggers the loop, for the particular instantiations of terms, caused by G. This can only happen if the loop is *composite* of several clauses (say $C_i,\ldots,C_k$), otherwise GC1 would have found its more general instance. It also must mean that G's term instances make sure all the clauses $C_i,\ldots,C_k$ match the calls started by $Q(\bar{t})$, whereas other terms tried by coinductive derivations of $GC3$ failed to match them to give $Q(\bar{t}) \leftarrow \ldots \leftarrow Q(\bar{t'})$. But that is impossible, as coinductive derivations of GC3 will try all resolvents that unify $body$ with program clauses, including $C_i,\ldots,C_k$, and will do the same for each of $C_i,\ldots,C_k$, inductively. Therefore, if there was a chain of unifiers that could in principle unify $body$ with

$C_i, \ldots, C_k$ to create a loop $Q(\bar{t}) \leftarrow \ldots \leftarrow Q(\bar{t'})$, it would have been found, possibly in a more general instantiation than that given by particular term instances of G.

Case 2. [GC-derivations stopped too early to detect the loop, due to branch isolation used in GC3.] Suppose the un-guarded loop was not discovered due to isolation of a branch with a (different) guarded loop $R(\bar{s}) \leftarrow \ldots \leftarrow R(\bar{s'})$ or a branch $\Box \leftarrow \ldots \leftarrow G\theta^*$. Suppose a node $N(U)$ of that isolated branch could potentially give rise to a resolvent leading to a loop in the next subgoal. But then there must be a clause $U^* \leftarrow body$ in P that may potentially lead to a loop. However, there would be a separate set of derivations constructed for this clause, and such loop would be detected then. $\Box$

Note that the checks GC1-GC3 we have introduced here are pre-processing (compile-time) mechanism of CoALP. Once the program passed the guardedness checks, it does coinductive derivations lazily, similar to lazy functional languages; and does not require any loop-detection procedures at the run-time. Termination issues are handled solely by lazy steps (finite coinductive trees). In line with e.g. Coq or Agda, if a program fails GC1-GC3, the programmer will be asked to re-formulate the definitions. Similarly to the above languages, guardedness checks may exclude some productive programs, e.g. `stream2` in Example 2.4 is productive but not guarded. Termination being an un-decidable property, it is in principle not possible to formulate a set of finite checks that would exactly define productivity. Instead, the above programming languages use a conservative approximation of productivity given by guardedness.

5 Implementation, Conclusions and Future Work

CoALP has had a number of prototype implementations, starting with SWI-Prolog implementation, that was sequential and had no guardedness checks. Similar to CoLP, it was working for all guarded programs, and would fall into infinite coinductive trees otherwise. As CoALP semantically has both parallel and corecursive features, it was re-implemented as a parallel language in Go (Summerfield 2012); see (Komendantskaya et al. 2014c) for more details. The Go version featured *automatic* mode for inductive (or eager) computations; and interactive mode for lazy computations. The difference was whether the coinductive derivations were constructed automatically, or the user was prompted after each coinductive tree construction.

Implementing results of this paper, the Go implementation has been enriched with full set of guardedness checks; the coinductive predicate annotations and search heuristics (* and ** of Definition 2.4). Since GC3 checks can be done for every clause and every GC-derivation independently, we make use of the parallel features of Go implementation of CoALP. Table 1 shows that "GC-time" compares favourably with execution times of CoALP and CoLP, and looses only to SWI-Prolog, which is a fully-tuned mature programming language, as opposed to our implementation of CoALP in Go from scratch.

Examples may be hand-crafted where Guardedness checks can take more time than finding the answer to a query; however, even in that case this process always terminates and the programmer is rewarded with guarantees of program productivity for any future call. Note that GC1-GC3 are run only once at compile time, and do not need to be repeated at the runtime.

Termination checking of CoALP (GC1-GC3) does not rely on predicate annotation. However, predicate annotation helps as a heuristic mechanism to identify eager search strategies for inductive resolvents, and lazy strategies for coinductive resolvents. For examples like FibStream such heuristics are indispensable; as fully eager computation would result in an infinite coinductive derivation (similarly to CoLP); whereas fully lazy coinductive derivation would unfold coinduc-

	CoALP		CoLP	SWI-Prolog
FibStream†	Yes	GC time: 0.0006s runtime: lazy execution	No	No
Comember†	Yes	GC time: 0.0011s runtime: lazy execution	Yes$^?$ (0.0001s)	No
SumFirstn†	Yes	GC time: 0.0013s runtime: lazy execution	No	No
Takenats†	Yes	GC time: 0.0002s runtime: lazy execution	No	No
Takerepeat†	Yes	GC time: 0.0009s runtime: lazy execution	Yes (0.0001s)	No
Infinite Automata†	Yes	GC time: 0.0011s runtime: lazy execution	Yes (0.0001s)	No
NQueens	Yes	GC time: 0.004s runtime: 0.404s	Yes (0.03s)	Yes (0.0005s)
Knights	Yes	GC time: 0.225s runtime: 3.002s	Yes (1.13s)	Yes (0.012s)
Finite Automata	Yes	GC time: 0.0011s runtime: 0.0023s	Yes (0.04s)	Yes (0.0005s)
Ackermann	Yes	GC time: 0.001s runtime: 13.23s	Yes (7.692s)	Yes (3.192s)

Table 1. *Execution of different programs in CoALP, CoLP and SWI-Prolog (see also Appendix C). Examples marked with † involve both inductive and coinductive predicates. In the table, "No" means that the system runs forever without returning an answer: CoLP can only handle the coinductive programs that contain regular patterns and fails otherwise (in which case we mark it as Yes$^?$); on the contrary, CoALP, in its lazy style, works for any program (See also Comember discussion in Examples 2.3 and 3.1). Takenat (irregular input) and Takerepeat (regular input) show a similar effect. "Infinite Automata" example refers to automata with regular structure only. .*

tive nodes (`fibs`) sooner than substitutions for inductive nodes like `nth` and `add` are returned; thus annotations help to find the happy medium, see Figure 2 and Appendix B.

Conclusions and Future work. We have given the full description of *guarded* coinductive derivations in CoALP; and for the first time, we have stated a full set of guardedness conditions. We have proven that they are sufficient to guarantee productiveness of logic programs.

The construction of coinductive trees provides a unit of lazy computations in potentially infinite coinductive derivations. For efficiency reasons, we now adopt labelling of predicates as inductive/coinductive similar to CoLP (Gupta et al. 2007; Simon et al. 2007). The major difference between our approach and CoLP is that such labelling is not used for termination purposes in CoALP, whereas it was instrumental for loop detection and termination in CoLP. The early versions of CoALP (Komendantskaya et al. 2014a) advocated that lazy corecursion in LP was possible without any sort of loop detection; as lazy derivation steps were handling termination, similar to lazy functional languages. This still remains true for run-time coinductive derivations as we define them here; however, we now use a loop detection procedure at the pre-processing stage of guardedness checks GC1–GC3. So, one can notice certain convergence of techniques of lazy (CoALP) and eager (CoLP) corecursion in LP, which is an interesting phenomenon deserving future investigations.

Finally, we note that CoALP indeed extends the class of coinductive programs we can now handle by finite means in LP (cf. Table 1). This is modulo the discussion of Section 2 about the need for a new approach to decide the soundness of coinductive derivations. Our future work will be devoted to developing such an approach.

References

ABEL, A. ET AL. 2013. Copatterns: programming infinite structures by observations. In *POPL'13*. 27–38.

BERTOT, Y. AND KOMENDANTSKAYA, E. 2008. Inductive and coinductive components of corecursive functions in Coq. *ENTSC 203,* 5, 25–47.

COQUAND, T. 1994. Infinite objects in type theory. In *TYPES'93*. LNCS, vol. 806. 62–78.

DE SCHREYE, D. AND DECORTE, S. 1994. Termination of logic programs: the never-ending story. *J. of Logic Programming 19–20, Supplement 1*, 199–260. Special Issue: Ten Years of Logic Programming.

GUPTA, G. ET AL. 2007. Coinductive logic programming and its applications. In *ICLP'07*. LNCS, vol. 4670. 27–44.

GUPTA, G. AND COSTA, V. 1994. Optimal implementation of and-or parallel prolog. In *PARLE'92*. 71–92.

JEANNIN, J.-B. ET AL. 2013. Language constructs for non-well-founded computation. In *ESOP'13*. LNCS, vol. 7792. 61–80.

KOMENDANTSKAYA, E. ET AL. 2014a. Coalgebraic logic programming: from semantics to implementation. *Accepted J. Logic and Computation*.

KOMENDANTSKAYA, E. ET AL. 2014b. CoALP webpage: software and supporting documentation. http://staff.computing.dundee.ac.uk/katya/CoALP/.

KOMENDANTSKAYA, E. ET AL. 2014c. Exploiting parallelism in coalgebraic logic programming. *Accepted ENTCS*.

KOMENDANTSKAYA, E. AND POWER, J. 2011. Coalgebraic derivations in logic programming. In *CSL'11*. LIPIcs. Schloss Dagstuhl, 352–366.

LLOYD, J. 1987. *Foundations of Logic Programming*, 2nd ed. Springer-Verlag.

ROHWEDDER, E. AND PFENNING, F. 1996. Mode and termination checking for higher-order logic programs. In *ESOP'96*. LNCS, vol. 1058. 296–310.

SIMON, L. ET AL. 2007. Co-logic programming: Extending logic programming with coinduction. In *ICALP'07*. LNCS, vol. 4596. 472–483.

SUMMERFIELD, M. 2012. *Programming in Go: Creating Applications for the 21st Century*. Addison-Wesley.

Appendix A Logic Programming Background

CoALP uses standard syntax of logic programming cf. (Lloyd 1987), but it substitutes the standard mechanism of SLD-resolution by coalgebraic derivations. We fix here the standard notation of logic programming (Lloyd 1987).

A *signature* Σ consists of a set of *function symbols* $f, g, \ldots$ each equipped with a fixed *arity*. The arity of a function symbol is a natural number indicating the number of its arguments. Nullary (0-ary) function symbols are allowed: these 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 $x \in Var$; and, if f is an n-ary function symbol ($n \geq 0$) and $t_1, \ldots, t_n \in Ter(\Sigma)$, then $f(t_1, \ldots, t_n) \in Ter(\Sigma)$. Variables will be denoted x, y, z, sometimes with indices $x_1, x_2, x_3, \ldots$. A *substitution* is a map $\theta : Ter(\Sigma) \to Ter(\Sigma)$ which satisfies $\theta(f(t_1, \ldots, t_n)) \equiv f(\theta(t_1), \ldots, \theta(t_n))$ for every n-ary function symbol f.

We define an *alphabet* to consist of a signature Σ, the set *Var*, and a set of *predicate symbols* $P, P_1, P_2, \ldots$ each assigned an arity. Let P be a predicate symbol of arity n and $t_1, \ldots, t_n$ be terms. Then $P(t_1, \ldots, t_n)$ is a *formula* (also called an atomic formula or an *atom*). The *first-order language* $\mathscr{L}$ given by an alphabet consists of the set of all formulae constructed from the symbols of the alphabet.

Given a substitution θ and an atom A, we write $A\theta$ for the atom given by applying the substitution θ to the variables appearing in A. Moreover, given a substitution θ and a list of atoms $(A_1, \ldots, A_k)$, we write $(A_1, \ldots, A_k)\theta$ for the simultaneous substitution of θ in each A_m.

Let S be a finite set of atoms. A substitution θ is called a *unifier* for S if, for any pair of atoms A_1 and A_2 in S, applying the substitution θ yields $A_1\theta = A_2\theta$. 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 $\sigma = \theta\gamma$. If θ is an mgu for A_1 and A_2, moreover, $A_1\theta = A_2$, then θ is a *term-matcher*. If θ is trivial, e.g. x/x, we will call it an empty substitution, and denote it by ε.

Given a first-order language $\mathscr{L}$, a *logic program* consists of a finite set of clauses of the form $A \leftarrow A_1, \ldots, A_n$, where $A, A_1, \ldots, A_n$ ($n \geq 0$) are atoms. The atom A is called the *head* of a clause, and $A_1, \ldots, A_n$ is called its *body*. Clauses with empty bodies are called *unit clauses*. We call a term, a formula, or a clause *ground*, if it does not contain variables.

A *goal* is given by $\leftarrow B_1, \ldots B_n$, where $B_1, \ldots B_n$ ($n \geq 0$) are atoms.

Let S be a finite set of atoms. A substitution θ is called a *unifier* for S if, for any pair of atoms A_1 and A_2 in S, applying the substitution θ yields $A_1\theta = A_2\theta$. 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 $\sigma = \theta\gamma$. If θ is an mgu for A_1 and A_2, moreover, $A_1\theta = A_2$, then θ is a *term-matcher*.

Appendix B Derivation of the third member of the Fibonacci stream

Figures B 1 and B 2 show several steps of the coinductive derivation for the goal `fib(s(s(0)),X)` and the program FibStream.

Let us see the execution of the FibStream example (Example 1.3) in the Go implementation of CoALP. The program is invoked using the following call:

```
./goalp -showground fibs.logic "fib(s(s(0)),X)"
```

where `fibs.logic` contains the logic program of FibStream and `"fib(s(s(0)),X)"` is the query. The output in this case is (note the pausing at each coinductive step):

```
expand coinductive node [y/n]?
fibs(0,s(0),cons(X1,cons(X2,cons(X,S3)))) in fib(s(s(0)),X)->
        [fibs(0,s(0),cons(X1,cons(X2,cons(X,S3))))?,nth(0,cons(X,S3),X)] y


expand coinductive node [y/n]? fibs(s(0),s(0),cons(X2,cons(X,S3))) in
fib(s(s(0)),X)->[fibs(0,s(0),cons(0,cons(X2,cons(X,S3))))->
   [add(0,s(0),s(0)),fibs(s(0),s(0),cons(X2,cons(X,S3)))?],
   nth(0,cons(X,S3),X)] y


expand coinductive node [y/n]? fibs(s(0),s(Z...),cons(X,S3)) in
  fib(s(s(0)),X)->[fibs(0,s(0),cons(0,cons(s(0),cons(X,S3))))->
  [add(0,s(0),s(0)),fibs(s(0),s(0),cons(s(0),cons(X,S3)))->
  [add(s(0),s(0),s(Z...))->
  [add(0,s(0),Z...)?],fibs(s(0),s(Z...),cons(X,S3))?]],nth(0,cons(X,S3),X)] y

ground tree root: fib(s(s(0)),s(0))

expand coinductive node [y/n]? fibs(s(s(0)),s(X4...),S3) in fib(s(s(0)),s(0))->
[fibs(0,s(0),cons(0,cons(s(0),cons(s(0),S3))))->
[add(0,s(0),s(0)),fibs(s(0),s(0),cons(s(0),cons(s(0),S3)))->
[add(0,s(0),s(0)),fibs(s(0),s(s(0)),cons(s(0),S3))->
[add(s(0),s(s(0)),s(X4...))->
[add(0,s(s(0)),X4...)?],fibs(s(s(0)),s(X4...),S3)?]]],nth(0,cons(s(0),S3),s(0))] n
```

As can be seen from the above execution (the output produced by CoALP is a bit different, in particular variable names are not the ones presented here but something like V409432480; we changed the names to show the correspondence with Figures B 1 and B 2), the inductive derivation steps (1,2,3,5 and 7 in in Figures B 1 and B 2) are not shown to the user since they are performed eagerly. On the contrary, the coinductive derivation steps (Derivations 4,6 and 8) always request user interaction.

It is also worth mentioning that CoALP does not finish the execution when it outputs the ground tree root `fib(s(s(0)),s(0))` but it keeps prompting the user to expand the coinductive trees.

Appendix C Benchmark for CoALP

We present here the programs used as benchmark in Table 1. These programs can be downloaded from (Komendantskaya et al. 2014b).

Example Appendix C.1 (FibStream)

```
1. add(0,X,X)
2. add(s(X),Y,s(Z)) <- add(X,Y,Z).
3. fibs(X,Y,cons(X,S)) <- add(X,Y,Z), fibs(Y,Z,S)}
4. nth(0,cons(X,S),X) <-
```

```
5. nth(s(N),cons(X,S),Y) <- nth(N,S,Y)}
6. fib(N,X) <- fibs(0,s(0),S), nth(N,S,X)}
```

Example Appendix C.2 (SumFirstn)

```
1. add(0,Y,Y)  <-
2. add(s(X),Y,s(Z))  <-  add(X,Y,Z)
3. from(X,scons(X,Y))  <-  from(s(X),Y)
:- coinductive(from/2).
4. sum(0,scons(X,Y),X) <-
5. sum(s(X1),scons(X,Y),Z)  <-  sum(X1,Y,R), add(X,R,Z)
6. sumfirst(N,Y) <- from(0,Z), sum(N,Z,Y)
?- sumfirst(s(s(s(0))),X)
```

Example Appendix C.3 (Takenats)

```
1. from(X,scons(X,Y))  <-  from(s(X),Y)
:- coinductive(from/2).
2. take(0,scons(X,Y),nil) <-
3. take(s(X),scons(Y,Z),cons(Y,R))  <-  take(X,Z,R)
4. takeN(N,Y) <- from(0,Z), take(N,Z,Y)
?- takeN(s(s(s(0))),X)
```

Example Appendix C.4 (Takerepeat)

```
1. repeat(X,scons(X,Y)) <- repeat(X,Y).
:- coinductive(repeat/2).
2. take(0,scons(X,Y),nil) <-
3. take(s(X),scons(Y,Z),cons(Y,R))  <-  take(X,Z,R)
4. takeN(N,Y) <- repeat(0,Z), take(N,Z,Y)
?- takeN(s(s(s(0))),X)
```

Example Appendix C.5 (InfiniteAutomata)

```
1. trans(s0,a,s1) <-
2. trans(s1,b,s2) <-
3. trans(s2,e,s0) <-
4. trans(s2,c,s3) <-
5. trans(s3,d,s0) <-
6. automaton(cons(X,T), St) <- trans(St,X,NewSt), automaton(T,NewSt)
:- coinductive(automaton/2)
?- automaton(X,s0)
```

Example Appendix C.6 (NQueens)

```
1. solution(Queens) <- permutation([1,2,3,4,5,6,7,8], Queens), safe(Queens)
2. permutation(nil,nil) <-
3. permutation([Head|Tail],PermList) <- permutation(Tail,PermTail),
                                        del(Head,PermList,PermTail)
```

```
4. del(Item,[Item|List],List) <-
5. del(Item,[First|List],[First|List1])  <-  del(Item,List,List1)
6. safe(nil) <-
7. safe([Queen|Others]) <-  safe(Others), noattack(Queen,Others,s(0))
8. minus(X,0,X) <-
9. minus(0,X,0) <-
10. minus(s(X),s(Y),Z) <- minus(X,Y,Z)
11. notequal(0,X) <-
12. notequal(X,0) <-
13. notequal(s(X),s(Y)) <- notequal(X,Y)
14. noattack(X,nil,Y) <-
15. noattack(Y,[Y1|Ylist],Xdist) <- minus(Y1,Y,Z), notequal(Z,Xdist), minus(Y,Y1,Z1),
                                    notequal(Z1,Xdist), noattack(Y,Ylist,s(Xdist))
?- solution(X)
```

Example Appendix C.7 (Knights)

```
1. knights(Knights) <- knights1(6,[pair(1,1)|nil],Knights)
2. knights1(0,Knights,Knights) <-
3. knights1(s(M),[pair(X,Y)|Z],Knights) <- jump(pair(X,Y),pair(U,V)),
                                           notmem(pair(U,V),[pair(X,Y)|Z]),
                                           knights1(M,[pair(U,V),pair(X,Y)|Z],Knights)
4. jump(pair(A,B),pair(C,D)) <- jumpdist(X,Y), plus(A,X,C), 0 <= C, 5 < C, plus(B,Y,D),
                                0 <= D, 5 < D
5. jumpdist(1,2) <-
6. jumpdist(2,1) <-
7. jumpdist(2,n1) <-
8. jumpdist(1,n2) <-
9. jumpdist(n1,n2) <-
10. jumpdist(n2,n1) <-
11. jumpdist(n2,1) <-
12. jumpdist(n1,2) <-
?- knights(X)
```

Example Appendix C.8 (FiniteAutomata)

```
1. trans(s0,a,s1) <-
2. trans(s1,b,s2) <-
3. trans(s2,e,s0) <-
4. trans(s2,c,s3) <-
5. trans(s3,d,s0) <-
6. final(s2) <-
7. automaton(cons(X,T), St) <- trans(St,X,NewSt), automaton(T,NewSt)
8. automata(nil,St) <- final(St).
?- automaton(X,s0)
```

Example Appendix C.9 (Ackermann)

```
1. a(0,M,s(M)) <-
2. a(s(N),0,R) <- a(N,s(0),R).
3. a(s(N),s(M),R) <- a(s(N),M,R1),a(N,R1,R)
?- a(s(s(s(0))),s(s(s(s(s(0))))),X)
```

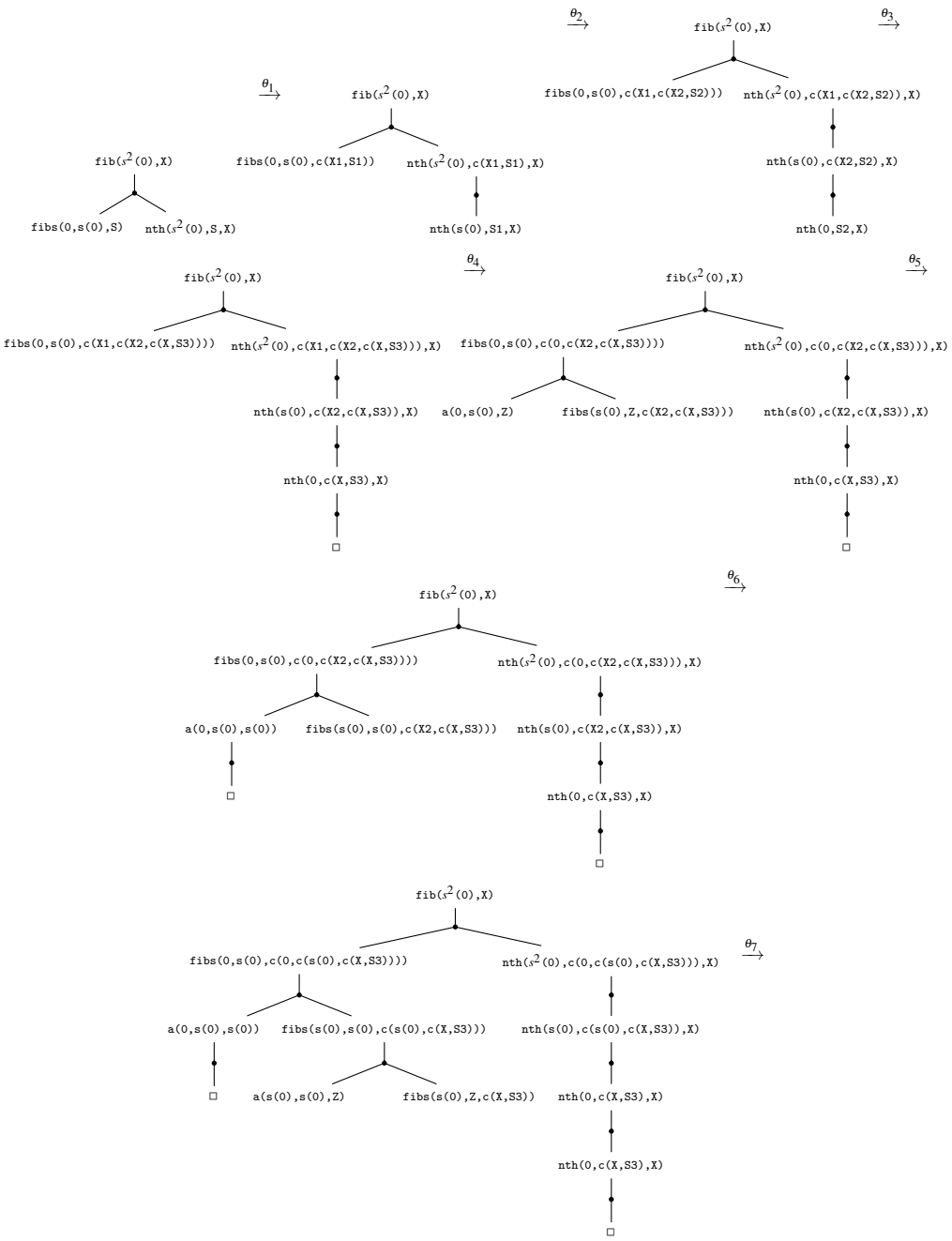

Fig. B 1. *First steps of the derivation for the goal* `fib(s(s(0)),X)` *and the program FibStream, with* $\theta_1 = S/cons(X1,S1)$, $\theta_2 = S1/cons(X2,S2)$, $\theta_3 = S2/cons(X,S3)$, $\theta_4 = X1/0$, $\theta_5 = Z/s(0)$, $\theta_6 = X2/s(0)$ *and* $\theta_7 = Z/s(s(0))$. *Derivation Steps 1, 2, 3, 5 and 7 are eager; and Derivation Steps 4 and 6 are lazy.*

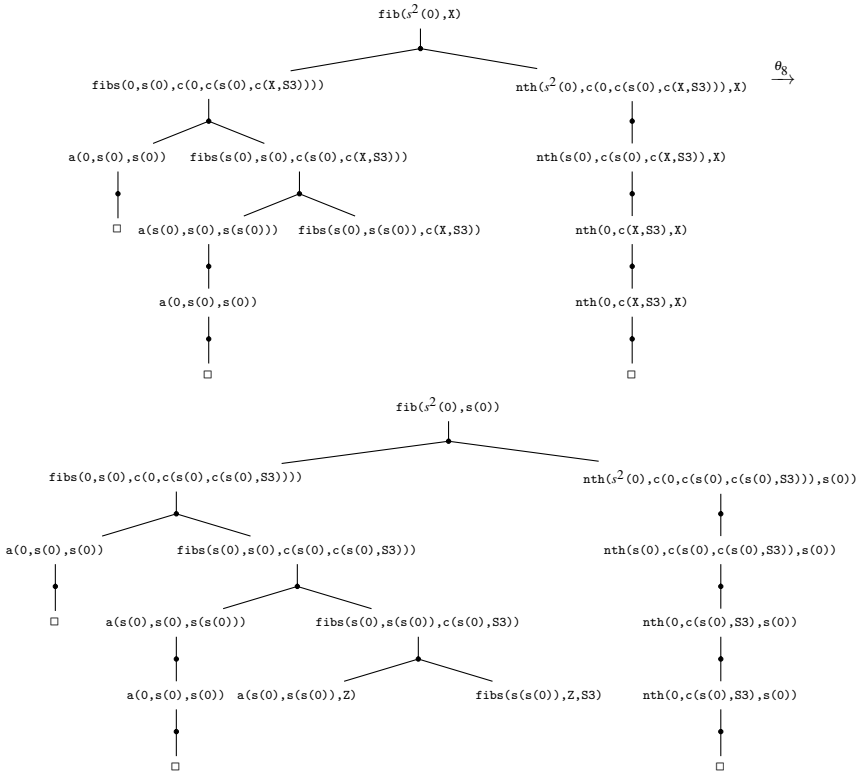

Fig. B 2. *Continuation of the derivation for the goal* `fib(s(s(0)),X)` *and the program FibStream, with* $\theta_8 = X/s(0)$. *Derivation Step 8 is lazy.*