Neural Networks for Proof-Pattern Recognition

Ekaterina Komendantskaya and Kacper Lichota

School of Computing, University of Dundee, UK [⋆]

Abstract. We propose a new method of feature extraction that allows to apply pattern-recognition abilities of neural networks to data-mine automated proofs. We propose a new algorithm to represent proofs for first-order logic programs as feature vectors; and present its implementation. We test the method on a number of problems and implementation scenarios, using three-layer neural nets with backpropagation learning.
Key words: Machine learning, pattern-recognition, data mining, neural networks, first-order logic programs, automated proofs.

1 Introduction

Automated theorem proving has been applied to solve sophisticated mathematical problems (e.g., verification of the Four-Colour Theorem in Coq), and for industrial-scale software and hard-ware verification (e.g., verification of microprocessors in ACL2). However, such "computer-generated" proofs require considerable programming skills, and overall, are time-consuming and hence expensive.

Programs in automated provers may contain thousands of theorems of variable sizes and complexities. Some proofs will require programmer's intervention. In this case, a manually found proof for one problematic lemma may serve as a template for several other lemmas needing a manual proof. Automated discovery of common proof-patterns using tools of statistical machine learning such as neural networks could potentially provide the much-sought automatisation for statistically similar proof-steps; as was argued e.g. in [1–3, 11–13].

As was classified in [1], applications of machine-learning assistants to mechanised proofs can be divided into *symbolic* (akin e.g. Inductive logic programming), *numeric* (akin neural networks or Kernels), and *hybrid*. In this paper, we focus on neural networks. The advantages of the numeric methods over symbolic is tolerance to noise and uncertainty, as well as availability of powerful learning functions. For example, the standard multi-layer perceptrons with error back-propagation algorithm are capable of approximating any function from finite-dimensional vector space with arbitrary precision. In this case, it is not the power of the learning paradigm, but the feature selection and representation method that sets the limits. Consider the following example.

Example 1. Let `ListNat` be a logic program defining lists of natural numbers:
1. `nat(0)` $\leftarrow$; 2. `nat(s(x))` $\leftarrow$ `nat(x)`;

[⋆] The work was supported by EPSRC grants EP/F044046/2 and EP/J014222/1.

3. `list(nil)` $\leftarrow$; 4. `list(cons x y)` $\leftarrow$ `nat(x), list(y)`

For `ListNat` and a goal $G_0 =$ `list(cons(x, cons(y, z)))`, SLD-resolution produces a sequence of subgoals: $G_1 =$ `nat(x),list(cons(y, z))`, $G_2 =$ `list(cons(y,z))`, $G_3 =$ `nat(y),list(z)`, $G_4 =$ `list(z)`, $G_5 = \Box$. If we consider applications of each of the clauses 1-4 as proof tactics, then the proof steps above could be presented as a training example (feature vector) 4,1,4,1 to a neural network. However, we cannot statistically generalise this to future examples, as with some frequency, the same sequence of "tactics" will fail, e.g. take the goal $G_0 =$ `list(cons(x,cons(y,x)))`. This happens because the given features do not capture the essential proof context – given by the term structure, unification procedure, and other parameters.

Recursive logic programs, such as the program above, are traditionally problematic for neural network representation, as they cannot be soundly propositionalised and represented by the vectors of truth values, but see e.g. [6, 8].

The method we present here is designed to steer away from these problems. It covers (co-)recursive first-order logic programs. To manage (co-)recursion efficiently, we use the formalism of coinductive proof trees for logic programs, see [9]. The coinductive trees possess more regular structure than e.g. SLD-trees. In Section 2, we propose an original feature extraction algorithm for arbitrary proof-trees. It allows to capture intricate structural features of the proof-trees such as branching, dependencies between the terms and predicates; as well as internal dependencies between structures of terms appearing at different stages of the proof. We implement the feature extraction algorithm: see [7].

The main advantages of the feature selection method we propose are its accuracy, generality and robustness to changes in classification tasks. In Section 3, we test this method on a range of classification tasks and possible implementation scenarios with very high rates of success. All our experiments involving neural networks were made in MATLAB Neural Network Toolbox (*pattern-recognition package*), with a standard three-layer feed-forward network, with sigmoid hidden and output neurons. The network was trained with *scaled conjugate gradient back-propagation*. Such networks can classify vectors arbitrarily well, given enough neurons in the hidden layer, we tested their performance on 40, 50, 60, 70, 90 hidden neurons for all experiments. All the software, datasets, and detailed reference manual are available in [7].

2 Feature Extraction Method and Algorithm

In this Section, we will assume familiarity with first-order logic programming [10]. The definitions of the signature Σ, the alphabet A, the first-order language $\mathcal{L}$, and logic programs P are standard, see also Appendices A and B.

For our experiments, we chose coinductive proof trees [9]: they resemble the and-or trees used in concurrent logic programming [5]; but have more structured approach to unification, by restricting it to term-matching. Moreover, they allow

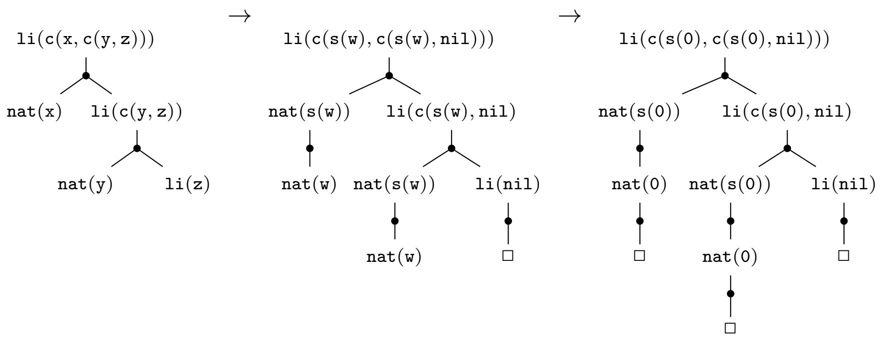

Fig. 1. Coinductive trees for `ListNat`. We abbreviate `cons` by `c` and `list` by `li`. The last tree is a *success tree* which implies that the sequence of derivation steps succeeded.

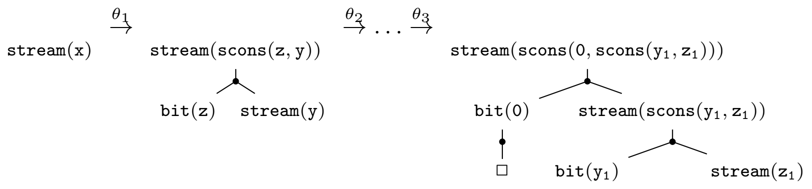

Fig. 2. Coinductive derivation for the goal $G =$ `stream(x)` and the program `Stream`.

to treat (co-)recursive derivations in lazy (finite) steps. But note that the feature extraction we present applies to many other kinds of proof trees.

Definition 1. *Let P be a logic program and $G = \leftarrow A$ be an atomic goal. The* coinductive tree *for A is a 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 atomic formula.*
- *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 $A' = B_1\theta_1 = ... = B_m\theta_m$, 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 $B_1^i\theta_i, \ldots, B_{n_i}^i\theta_i$.*

Example 2. The following corecursive program `Stream` defines infinite streams of binary bits. The program will induce infinite derivations if the SLD-resolution algorithms is applied, but only finite coinductive proof trees, see Figure 2.

```
bit(0) ← ; bit(1) ←
stream(scons (x,y)) ← bit(x),stream(y)
```

We represent coinductive trees as feature vectors. Several ways of representing graphs as matrices are known from the literature: e.g., *incidence matrix* and *adjacency matrix*. However, these traditional methods obscure some patterns found in coinductive trees; but see [12] for adjacency matrix encoding of logic formulae represented as trees. We propose a new method as follows.

1. Numerical encoding of the signature Σ and terms. Define a one-to-one function $[\![\ .\ ]\!]$ that assigns a numerical value to each function symbol appearing in the given tree T, including nullary functions. Assign -1 to any variable occurring in T. Gödel numbering is one of the classical ways to show that first-order language can be enumerated. But in our case, the signature of each given program is restricted, and we use a simplified version of enumeration for convenience. In the method we present, the choice of the function $[\![\ .\ ]\!]$ is not crucial for proof classification.

Complex terms are encoded by simple concatenation of the values of the function symbols and variables. If a term $t = (f(x_1, \dots x_n))$ contains variables $x_1, \dots x_n$, the numeric values $[\![x_1]\!] \dots [\![x_n]\!]$ are negative. In this case, the positive values $|[\![x_1]\!]| \dots |[\![x_n]\!]|$ are concatenated, but the value of the whole term is assigned a negative value.

Example 3. For program ListNat, $[\![\texttt{0}]\!] = 6$, $[\![\texttt{S}]\!] = 5$, $[\![\texttt{cons}]\!] = 2$, $[\![\texttt{nil}]\!] = 3$, $[\![\texttt{x}]\!] = [\![\texttt{y}]\!] = [\![\texttt{z}]\!] = -1$. Thus, $[\![\texttt{cons(x,cons(y,x))}]\!] = -21211$.

2. Matrix representation of the proof trees. For a given tree T, we build a matrix M as follows. The size of M is $(n+2) \times m$, where n and m are the number of distinct predicates and terms appearing in T.

The entries of M are computed as follows. For the predicate R_i, and the term t_j, the ijth matrix entry is $[\![t_j]\!]$ if $R(t', \dots, t_j, \dots, t'')$ is a node of T, and 0 otherwise. The columns marked as $\bullet$ and $\Box$ are added to allow to trace tree-branching factors and patterns of *success* leaves. For the $n+1$ column and the term t_j, if every node containing t_j has exactly k children given by or-nodes, then the $(n+1)j$th entry in M is equal to k; if the parameter k for t_j varies, then the $(n+1)j$th entry is -1; and it is 0 otherwise. For the $n+2$ column and the term t_j, if all children of the node $Q(t)$, for some $Q \in P$ are given by or-nodes, such that all these or-nodes have children nodes $\Box$, then $(n+2)j$th entry is 1; if some but not all such nodes are $\Box$, then the $(n+2)j$th entry is -1; and it is 0 otherwise. See Appendix C for the algorithm in pseudocode, Figure 3 for examples, and [7] for implementation and more examples.

3. Vector representation. The matrix M is then flattened into a vector.

Example 4. The matrix M_1 above will be given by $V_1 = [-21211, -211, 0, 0, -1, 0, 0, -1, -1, 0, 2, 2, 0, 0, 0, 0, 0, 0, 0, 0]$.

Proposition 1 (Properties of the encoding).

1. For a given matrix M, there may exist more than one corresponding coinductive tree T; i.e., the mapping from the set $\mathcal{T}$ of coinductive trees to the set $\mathcal{M}$ of the corresponding matrices is not bijective.

Matrix M_1	`list`	`nat`	●	□	Matrix M_2	`list`	`nat`	●	□
`cons(x, cons(y, z))`	- 21211	0	2	0	`cons(s0, cons(s0, nil))`	2562563	0	2	0
`cons(y, z))`	- 211	0	2	0	`cons(s0, nil))`	2563	0	2	0
x	0	-1	0	0	`s0`	0	56	1	0
y	0	-1	0	0	`0`	0	6	1	1
z	-1	0	0	0	`nil`	3	0	1	1

Fig. 3. Matrices M_1 and M_2 encode the left-most and right-most trees in Figure 1.

	Neural Net - `ListNat`	Neural net - `Stream`	SVM - `Stream`
Problem 1	76.4%	84.3 %	89 %
Problem 2	96.3%	99.1 %	88 %
Problem 3	86 %	n/a	n/a
Problem 4	n/a	85.7 %	90%
Problem 5	82.4 %	82.4%	n/a

Fig. 4. Summary of the best-average results (positive recall) of classification of coinductive proof trees for the classification problems of Section 3 and proof trees for programs `ListNat` and `Stream`, performed in neural networks and SVMs.

2. If there exist two distinct root nodes F_1 and F_2 whose coinductive trees are encoded by matrices M_1 and M_2 such that $M_1 \equiv M_2$, then F_1 and F_2 differ only in variables, however, F_1 and F_2 are not necessarily α-equivalent.

3 Evaluation in Neural Networks

Here, we test generality, accuracy, and robustness of the method on a range of classification tasks and implementation scenarios.

For the experiments of this section, we used data sets of various sizes - from 120 to 400 examples of coinductive trees for various experiments; we sampled trees produced for several distinct logic programs – such as `ListNat` and `Stream` above, see [7]. Finally, we repeated all experiments using three-layer neural networks of various sizes, and compared the results with those given by the SVMs with kernel functions.

Note that we deliberately did not tune the learning functions in neural networks to fit our symbolic data; but see e.g. [1, 12, 13].

3.1 Tests on various classification tasks

Problem 1. Classification of well-formed and ill-formed proofs. Figures 1, 2, 7, 8 show well-formed trees. Trees that do not conform to Definition 1 are ill-formed, see Appendix D or [7] for more examples. This task is one of the most difficult for pattern-recognition, due to a wide range of possible erroneous proofs compared to the correct ones, the results are summarised in Figure 4.

Problem 2. Discovery of proof families.

Definition 2. *Given a logic program P, and an atomic formula A, we say that a tree T belongs to the* family of coinductive trees determined by *A, if*
– T is a coinductive tree with root A' and;
– there is a substitution θ such that $A\theta = A'$.

Example 5. The three trees in Figure 1 belong to the family of proofs determined by `list(cons(x,cons(y,z)))`; similarly for Figure 2.

The results of pattern-recognition for proof families are exceptionally robust; cf. Figure 4. Determining whether a given tree belongs to a certain proof family has practical applications. For Figure 1, knowing that the right-hand side tree belongs to the same family as the left-hand side tree would save the intermediate derivation step. Note that unlike [12], it is not the shape of a formula, but proof patterns it induces when the program is run, that influence classification.

Problem 3. Discovery of potentially successful proofs. Significance of this classification problem is asserted in the proposition below.

Definition 3. *We say that a proof-family F is a* success family *if, for all $T \in F$, T generates a proof-family that contains a success subtree (cf. Definition 4).*

Proposition 2. *Given a coinductive tree T, there exists a success family F such that $T \in F$ if and only if T has a successful derivation.*

This problem has solutions only for inductive definitions. Trees like the ones given in Figure 7 were negative examples for the training purposes, and trees akin Figure 1 were given as positive examples. Bearing in mind subtlety of the notion of a success family, the accuracy of classification was astonishing (86%), cf. Figure 4.

Problem 4. Discovery of ill-typed proofs in a proof family. In our examples, an ill-typed formula would e.g. be `nat(cons(x,y))`, as by definition, the term `cons(x,y)` must be of type `list`. A tree is ill-typed if it contains ill-typed terms.

When it comes to coinductive logic programs like `Stream`, detection of success families is impossible, see Figures 2 and 8. In such cases, detection of well-typed and ill-typed proofs within a proof family will be an alternative. Figures 7 and 8 show ill-typed proof families induced by the trees from Figure 1 and 2.

Problem 5. Discovery of ill-typed proofs. The problem is similar to Problem 4, however, the restriction that all proofs belong to the same proof family is lifted.

Problems 4 and 5 have conceptual significance for future applications, see [4]; our experiments show high accuracy in recognition of such proofs. We will experiment with Problem 5 in later sections. Overall, the proposed method works well, and applies to the variety of classification tasks.

X-Y — Problem	Initial accuracy for X	Test 1 on Y	Test 2	Test 3	X-Y Mixed data
List-Stream — P1	76.4%	44.2%	51.9%	63.9%	67.1%
Stream-List — P1	84.3%	36.7%	44%	67%	67.1%
List-Stream — P5	82.4%	65.6%	80.%	99%	80.1%
Stream-List — P5	79%	43.5%	63.5%	85.9%	80.1%

Fig. 5. Gradual adaptation to new types of proofs. Letters X and Y stand for logic programs `ListNat` and `Stream` interchangeably; P1 and P5 stand for Problems 1 and 5. First logic program X is taken, and neural network's accuracy is shown in the first column. Then these trained networks were used to classify examples of the proofs for a new logic program Y. The accuracy drops at the start, see the "Test 1" column. Further columns show how the neural network regains its accuracy as it is trained and tested on more examples of type Y. For comparison, the last column shows batch learning on mixed data without gradual adaptation.

3.2 Testing on a range of implementation scenarios

In real-life extensions of this technique, the neural network proof assistant (*NN-tool*) will be used on a wide variety of problems. How robust will be neural network learning when it comes to less regular and more varied data structures? We designed three **Implementation Scenarios (IS)** below to test the method:

IS 1 *NN-tool can create a new neural network for every new logic program.*

Example 6. Using our running examples, a separate sets of feature vectors for `ListNat` and `Stream` can be used to train two separate neural networks, see also Figure 4 for experiments supporting this approach.

The obvious objection to such approach is that creating a new neural network for every new fragment of a big program development may be cumbersome; it will not capture possible common patterns across different programs and fragments, but also, it will handle badly the cases where some apparently disconnected programs are bound by a newly added definition. The next two implementation scenarios address these problems.

IS 2 *NN-tool can use only one neural network, and re-train it irrespective of the changes in program clauses, new predicates or proof structures.*

In this case, the main question is how quickly the neural network will adapt to new patterns determined by a new logic program. We designed an experiment to test it, see Figure 5. It shows that gradual adaptation of previously trained neural network is at least as efficient as training on a mixed data. In fact, for Problem 5, it is more successful than training on mixed data! This suggests that there are common patterns in well-formed proofs for two different logic programs.

IS 3 *NN-tool re-defines and extends feature vectors to fit all available programs.*

Matrix M_3	`listream`	`stream`	`bit`	`list`	`nat`	●	□
`cons(x,x)`	- 211411	0	0	-211	0	2	0
`scons(y,z))`	- 211411	-411	0	0	0	2	0
x	0	0	0	-1	-1	0	0
y	0	0	-1	0	0	0	0
z	0	-1	0	0	0	0	0

	Prob 1	Prob 5
Merged matrices	84.3%	82%
`Listream`	76.3%	88.6%
Merged-`Listream`	51.2%	64.9%

Fig. 6. Left: Feature matrix for coinductive tree for `listream(cons(x,x), cons(y,z))`. **Right:** Accuracy of pattern recognition for feature matrices extended to encode trees for both `ListNat` and `Stream` ("Merged matrix" row); proofs for extended program `Listream` ("`Listream`" row); and experiment on first training neural networks on "Merged Matrix", and testing the trained network on `Listream` (last row).

Example 7. Suppose the NN-tool was used to work with proofs constructed for two programs – `ListNat` and `Stream`; and maintains two corresponding neural networks. However, a new clause is added by the user:
`listream(x,y)` $\leftarrow$ `list(x), stream(y).`

This new program `Listream` subsumes also `ListNat` and `Stream`.

The old neural networks will not accept the changed feature vectors for new proofs, as additional new predicate will infer the change in size of the feature vectors. In this case, it is possible to extend matrices, and thus treat the proof features from different programs as features of one meta-proof. An example of an extended feature matrix and results of tests are given in Figure 6. Note that accuracy for `Listream` exceeds accuracy for `Stream` and `List` separately (cf. Figure 4), despite of the growth of the feature vectors, which is exceptional.

Finally, as Figure 6 shows, we tried to mix the Scenarios 2 and 3. It is encouraging that for Problem 5, training on merged-matrix features over-performed simple mixing of data sets (as in Figure 5). When working with extended feature vectors, `Listream` over-performed the simpler merged-matrix data training. This shows that the feature-selection method we present allows extensions that capture significant and increasingly intricate proof-patterns.

4 Conclusions

The advantage of the learning method presented here lies in its ability to capture intricate relational information hidden in proof trees, such as patterns arising from interdependencies of predicate type, term structure, branching of proofs and ultimate proof success. This method allows to apply neural networks to a wide range of data mining tasks; and universality of the method is its another advantage. We implemented it in [7]. The future work is to integrate the neural network tool into one of the existing theorem provers. Another direction is to apply the method to other kinds of contextually-rich data, such as e.g. web-pages.

References

1. J. Denzinger, M. Fuchs, C. Goller, and S. Schulz. Learning from previous proof experience: A survey. Technical report, Technische Universitat Munchen, 1999.

2. J. Denzinger and S. Schulz. Automatic acquisition of search control knowledge from multiple proof attempts. *Inf. Comput.*, 162(1-2):59–79, 2000.

3. H. Duncan. *The use of Data-Mining for the Automatic Formation of Tactics*. PhD thesis, University of Edinburgh, 2002.

4. G.Grov, E.Komendantskaya, and A.Bundy. A statistical relational learning challenge - extracting proof strategies from exemplar proofs. In *ICML'12 worshop on Statistical Relational Learning, Edinburgh, 30 July 2012*, 2012.

5. G. Gupta and V. Costa. Optimal implementation of and-or parallel prolog. In *Conference proceedings on PARLE'92*, pages 71–92, NY, 1994. Elsevier.

6. P. Hitzler, S. Hölldobler, and A. K. Seda. Logic programs and connectionist networks. *Journal of Applied Logic*, 2(3):245–272, 2004.

7. E. Komendantskaya. ML-CAP home page, 2012. http://www.computing.dundee.ac.uk/staff/katya/MLCAP-man/.

8. E. Komendantskaya, K. Broda, and A. S. d'Avila Garcez. Neuro-symbolic representation of logic programs defining infinite sets. In *ICANN (1)*, volume 6352 of *LNCS*, pages 301–304. Springer, 2010.

9. E. Komendantskaya and J. Power. Coalgebraic derivations in logic programming. In *CSL'11*, 2011.

10. J. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, 2nd edition, 1987.

11. J. Lloyd. *Logic for Learning: Learning Comprehensible Theories from Structured Data*. Springer, Cognitive Technologies Series, 2003.

12. E. Tsivtsivadze, J. Urban, H. Geuvers, and T. Heskes. Semantic graph kernels for automated reasoning. In *SDM'11*, pages 795–803. SIAM / Omnipress, 2011.

13. J. Urban, G. Sutcliffe, P. Pudlák, and J. Vyskocil. Malarea sg1- machine learner for automated reasoning with semantic guidance. In *IJCAR*, LNCS, pages 441–456. Springer, 2008.

A First-order Logic Programs and Coinductive derivations

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*. Terms and substitution are defined in a standard way [10].

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* $\mathcal{L}$ given by an alphabet consists of the set of all formulae constructed from the symbols of the alphabet.

Given a first-order language $\mathcal{L}$, 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*. A *goal* is given by $\leftarrow B_1, \ldots B_n$, where $B_1, \ldots B_n (\ n \geq 0)$ are atoms.

The algorithm of SLD-resolution [10] is a sequential proof-search algorithm. It 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 [10]. The algorithm uses Horn-clause logic, with variable substitution determined universally to make a selected atom in G agree with the head of a clause in P, then proceeding inductively.

B Coinductive derivations

Definition 1 introduced coinductive trees, below are formal explanations of how they can be used to build derivations.

The notion of a successful proof is captured by the definition of *success subtrees* [9], they correspond to refutations in SLD-resolution.

Definition 4. *Let P be a logic program, A be a goal, and T be the coinductive derivation tree determined by P and A. A subtree T' of T is called a* success subtree *of T if it satisfies the following conditions:*

- *the root of T' is the root of T;*
- *if an and-node belongs to T', and the node has k children in T given by or-nodes, only one of these or-nodes belongs to T'.*
- *if an or-node belongs to T', then all its children given by and-nodes in T belong to T'.*
- *all the leaves of T' are and-nodes represented by $\Box$.*

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.

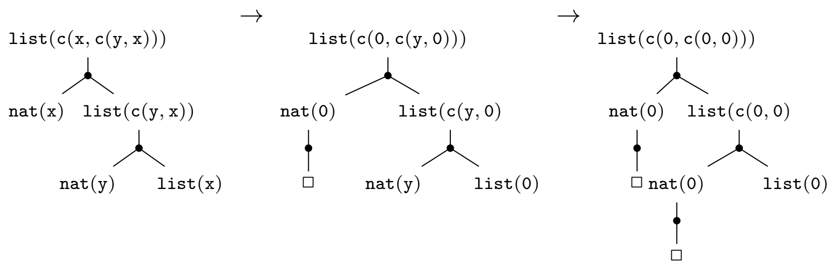

Fig. 7. The unsuccessful derivation and ill-typed proof family for the program ListNat and the goal `list(cons(x,cons(x,x)))`. We abbreviate `cons` by `c`.

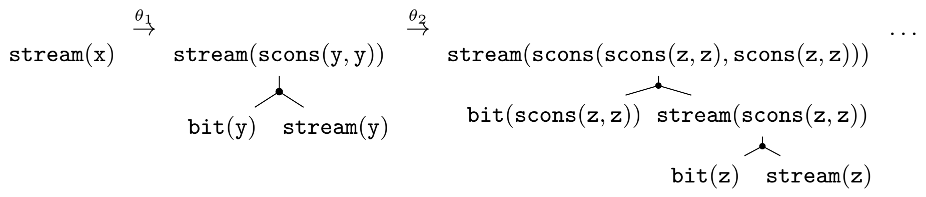

Fig. 8. Ill-typed derivation for the goal $G =$ `stream(x)` and the program `Stream`.

Definition 5. *Let G 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 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 $A\theta$.*

Definition 6. *A* coinductive derivation *of $P \cup \{G\}$ consists of a sequence of goals $G = G_0, G_1, \ldots$ called* coinductive resolvents *and a sequence $\theta_1, \theta_2, \ldots$ of mgus such that each G_{i+1} is derived from G_i using θ_{i+1}. A* coinductive refutation *of $P \cup \{G\}$ is a finite coinductive derivation of $P \cup \{G\}$ such that its last goal contains a success subtree. If G_n contains a success subtree, we say that the refutation has length n.*

Programs like `Stream` or `ListNat` always give rise to *finite* coinductive trees. This applies equally to any potentially infinite data defined using *constructors*, such as `scons` in `Stream` or `cons` and `nil` in `ListNat`. If a logic program P defines data in a guarded manner , we call it a *well-founded Logic Program.* So one may view infinite coinductive trees as indicating "bad" cases, in which (co)recursion is *not guarded by constructors.*

C Algorithm for feature extraction: proof trees to matrices

Here, we give the feature extraction algorithm in pseudocode, see Algorithm 1.

Algorithm 1 Feature extraction: proof trees to matrices

```
Require: $T$ – finite coinductive tree.
  $n =$ number of distinct predicates in $T$.
  $m =$ number of distinct terms appearing in the nodes of $T$.
  Construct a $(n+2) \times m$ matrix $M$, as follows:
  for $i = 1, \dots n$ do
      for $j = 1, \dots, m$ do
          if $P_i(t', \dots, t_j, \dots, t'')$ is a node of $T$ then
              $M_{ij} = [\![t_j]\!]$
          else $M_{ij} = 0$
          end if
      end for
  end for
  for $i = n+1$ do
      for $j = 1, \dots, m$ do
          if every node containing $t_j$ has branching factor $k$ then
              $M_{ij} = k$
          else if nodes containing $t_j$ have different branching factors then
              $M_{ij} = -1$
          else $M_{ij} = 0$
          end if
      end for
  end for
  for $i = n+2$ do
      for $j = 1, \dots, m$ do
          if all nodes containing $t_j$ have children given by $\Box$ then
              $M_{ij} = 1$
          else if some nodes containing $t_j$ have children given by $\Box$, and some - con-
  taining other formulas then
              $M_{ij} = -1$
          else $M_{ij} = 0$
          end if
      end for
  end for
      return $M$.
```

D Examples of the trees from the training set for Problem 1.

We present a sample of various negative and positive examples of coinductive trees for Problem 1, as, unlike other four problems, these examples were not shown in previous sections. The full datasets covering all results described in this paper are available in [7].

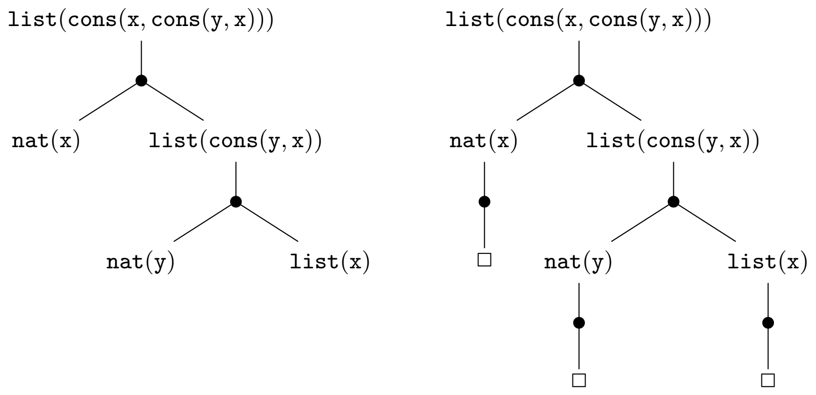

Fig. 9. An example of the training set for Problem 1. Left-hand-side tree is a positive example, and the right-hand-side — negative.

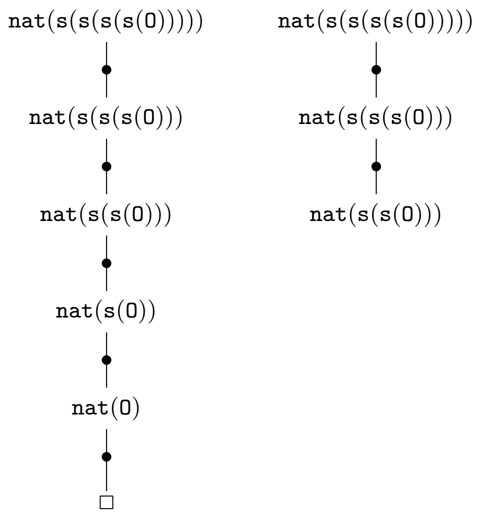

Fig. 10. An example of the training set for Problem 1. Left-hand-side tree is a positive example, and the right-hand-side — negative.