Machine Learning Methods for Discovering Proof Tactics in Automated Proofs (Work in Progress)

Ekaterina Komendantskaya

School of Computing, University of Dundee

STP'11,
17 June 2011

Outline

Outline

1. Motivation and wide perspective
2. Machine Learning sequential proof tactics

Outline

1. Motivation and wide perspective
2. Machine Learning sequential proof tactics
3. Pattern-recognition in Coinductive Proof trees

Outline

Proof in Coq - Demo.

- Not all proofs can be done automatically. That is, automatic tactics fail.
- In bigger industrial proofs there might be thousands of lemmas and theorems needing proofs, with about 5-20% needing programmer's intervention.
- Can statistical methods help us to analyse why these fail (The models of "Why", Cliff Jones)? and perhaps even suggest combination of tactics to try instead?
- Note on discovering "Why?s": my experience with Coq in INRIA was that the experts often justify those "Why's" statistically rather than conceptually. ("Coq Art")
- Note on combinations of tactics, like `intro`, `induction`, `simpl`, etc. in Higher-order interactive provers. Even very complicated proofs use about 50-100 tactics only; BUT their combinations can be very clever. Again, is there room for statistical analysis?

Overall feeling

- These are hard questions to solve.
- If solved, may change the way we look at proofs (automated and manual)...
- May enrich the methods of machine learning and proof theory.

Lessons learnt:

I have tried to implement several logic algorithms in Neural nets

- Semantic operators for first-order logic programs and many-valued logic programs;
- First-order Unification algorithm;
- First-order term-rewriting;
- Inductive definitions akin inductive dependent types.

Overall conclusion

It is inefficient to apply statistical methods to logic algorithms "as they are" — because statistical methods cannot compete with logical methods on the same grounds. Where can they compete?

Why is machine learning **UN**-suitable for Formal methods:

- Many logic algorithms have a precise, rather than statistical nature.

Example

Two formulae `list(x)` and `list(nil)` are unifiable: `x/nil`. We mean exactly this, and do not want it to be substituted by some approximate such as `nol`. (Although humans would tolerate this mis-spelling had it appeared in a written text...)

Why is machine learning **UN**-suitable for Formal methods:

- Many logic algorithms have a precise, rather than statistical nature.

Example

Two formulae `list(x)` and `list(nil)` are unifiable: `x/nil`. We mean exactly this, and do not want it to be substituted by some approximate such as `nol`. (Although humans would tolerate this mis-spelling had it appeared in a written text...)

- Many important logic algorithms are sequential, e.g. unification.

Example

If I have a goal: `list(cons(x,y))` $\wedge$ `list(x)`, my proof will never succeed — `x` will get substituted by some `nat` term, e.g. `0` or `S(0)`, which will make the second formula invalid. Note that the proof would have succeed had it been concurrent.

Machine learning may yet be useful in Logic...

...on several conditions

- we either use concurrent algorithms, or re-formulate existing algorithms into concurrent form,
- we are clear about what sort of statistical features in automated proofs we wish to learn.

Machine learning may yet be useful in Logic...

...on several conditions

- we either use concurrent algorithms, or re-formulate existing algorithms into concurrent form,
- we are clear about what sort of statistical features in automated proofs we wish to learn.

Boom in research into coalgebraic methods

- ... by this I mean coalgebra in either category theory sense, logic sense (e.g modal logic), or computational sense (as in ITPs).
- in most cases - considers concurrent models; e.g. Milner's CCS and π-calculus;
- because it is devoted to infinite processes/proofs/objects/... — it is based on the idea of repeating patterns (rather than final answers or terminating computations). E.g., the notion of productiveness. These have good chances of yielding statistical analysis.

Plan for the talk

In this talk, I will try to demonstrate this idea:

On example of logic programs:

- I will show sequential proofs analysed by machine learning methods;
- Show how they can be transformed into coinductive proofs;
- Compare their efficiency
- Hopefully — will persuade you that this example proofs the concept.

Plan for the talk

In this talk, I will try to demonstrate this idea:

On example of logic programs:

- I will show sequential proofs analysed by machine learning methods;
- Show how they can be transformed into coinductive proofs;
- Compare their efficiency
- Hopefully — will persuade you that this example proofs the concept.

- Logic programming is a simplified model compared to modern ITPs — but it is elegant, simple and yet we can learn a lot from it;
- The coinductive proofs part of my talk is based on joint work with J.Power, published in CALCO'11, CSL'11 - available on my webpage.

First-order syntax

We fix the *alphabet* **A** to consist of

- Signature Σ:
  - constant symbols a, b, c, possibly with finite subscripts;
  - function symbols $f, g, h, f_1 \ldots$, with arities;
- variables $x, y, z, x_1 \ldots$;
- predicate symbols $P, Q, R, P_1, P_2 \ldots$, with arities;

Term:

$$t = a \mid x \mid f(t)$$

Atomic Formula: $At = P(t_1, \ldots, t_n)$, where P is a predicate symbol of arity n and t_i is a term.

Example

Signature: $\{O, S\}$ - for natural numbers; {a,b,c} for a graph with 3 nodes.
Atoms: nat(x), edge(a,b), etc...

Logic programs

A first-order logic program consists of a finite set of clauses of the form

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

where A and the A_i's are atomic formulae, typically containing free variables; and $A_1, \ldots, A_n$ is to mean the conjunction of the A_i's.

Definition

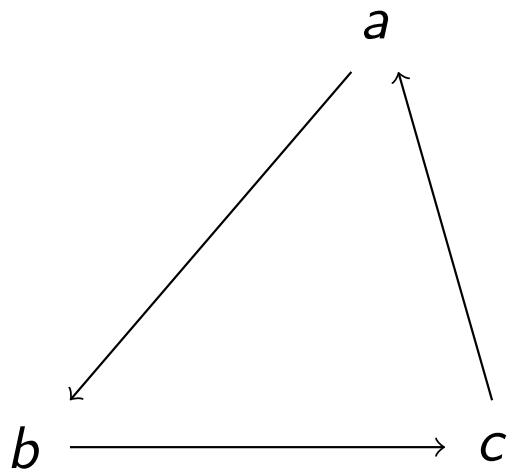
Let a goal G be $\leftarrow A_1, \ldots, A_m, \ldots, A_k$ and a clause C be $A \leftarrow B_1, \ldots, B_q$. Then G' is *derived* from G and C using mgu θ if the following conditions hold:

- A_m is an atom, called the *selected* atom, in G.
- θ is an *mgu* of A_m and A.
- G' is the goal $\leftarrow (A_1, \ldots, A_{m-1}, B_1, \ldots, B_q, A_{m+1}, \ldots, A_k)\theta$.

Logic program: Graph connectivity

Example

```
connected(x,x)  ←
connected(x,y)  ←  edge(x,z),
connected(z,y).
     edge(a,b)  ←
     edge(b,c)  ←
     edge(c,a)  ←
```



Example: Goal $\leftarrow$ connected(a, c).

Example

```
connected(x,x)  ←
connected(x,y)  ←  edge(x,z),
connected(z,y)
     edge(a,b)  ←
     edge(b,c)  ←
     edge(c,a)  ←
```

$$\leftarrow \texttt{connected}(\texttt{a}, \texttt{c})$$

$$\Big|$$

$$\leftarrow \texttt{edge}(\texttt{a}, \texttt{x}), \texttt{connected}(\texttt{x}, \texttt{c})$$

Example: Goal $\leftarrow$ connected(a, c).

Example

```
connected(x,x)  ←
connected(x,y)  ←  edge(x,z),
connected(z,y)
     edge(a,b)  ←
     edge(b,c)  ←
     edge(c,a)  ←
```

$$\leftarrow \texttt{connected}(a, c)$$
$$\Big|$$
$$\leftarrow \texttt{edge}(a, x), \texttt{connected}(x, c)$$
$$\Big|$$
$$\leftarrow \texttt{connected}(b, c)$$

Example: Goal $\leftarrow$ connected(a, c).

Example

```
connected(x,x)  ←
connected(x,y)  ←  edge(x,z),
connected(z,y)
     edge(a,b)  ←
     edge(b,c)  ←
     edge(c,a)  ←
```

$$\leftarrow \texttt{connected}(\texttt{a},\texttt{c})$$
$$\Big|$$
$$\leftarrow \texttt{edge}(\texttt{a},\texttt{b}), \texttt{connected}(\texttt{b},\texttt{c})$$
$$\Big|$$
$$\leftarrow \texttt{connected}(\texttt{b},\texttt{c})$$
$$\Big|$$
$$\leftarrow \texttt{edge}(\texttt{b},\texttt{x}), \texttt{connected}(\texttt{x},\texttt{c})$$

Example: Goal $\leftarrow$ connected(a, c).

Example

```
connected(x,x)  ←
connected(x,y)  ←  edge(x,z),
connected(z,y).
     edge(a,b)  ←
     edge(b,c)  ←
     edge(c,a)  ←
```

$\leftarrow$ connected(a, c)

|

$\leftarrow$ edge(a, b), connected(b, c)

|

$\leftarrow$ connected(b, c)

|

$\leftarrow$ edge(b, x), connected(x, c)

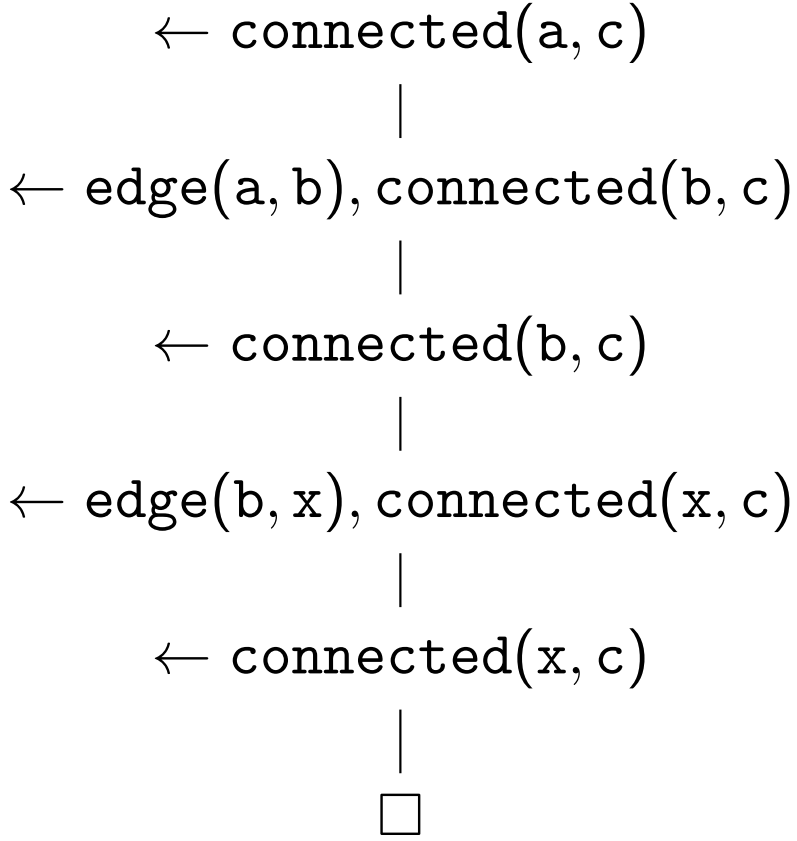
|

$\leftarrow$ connected(x, c)

Example: Goal $\leftarrow$ connected(a, c).

Example

```
connected(x,x)   ←
connected(x,y)   ←   edge(x,z),
connected(z,y).
      edge(a,b)  ←
      edge(b,c)  ←
      edge(c,a)  ←
```



Recursion and Corecursion in Logic Programming

Example

```
nat(0)  ←
nat(s(x))  ←  nat(x)
list(nil)  ←
list(cons x y)  ←  nat(x), list(y)
```

Example

```
bit(0)  ←
bit(1)  ←
stream(cons (x,y))  ←  bit(x),stream(y)
```

Machine-learn patterns of sequential proofs

This example mimics similar research done for ITPs in the PhD thesis of Hazel Duncan, University of Edinburgh.

Example

1. `nat(0)` ←
2. `nat(s(x))` ← `nat(x)`
3. `list(nil)` ←
4. `list(cons x y)` ← `nat(x), list(y)`

Given correct and incorrect sequences of the numbers 1, 2, 3, 4, 5, 6 how likely is it that we can train a neural network to recognise correct and incorrect proofs?

Example

Correct

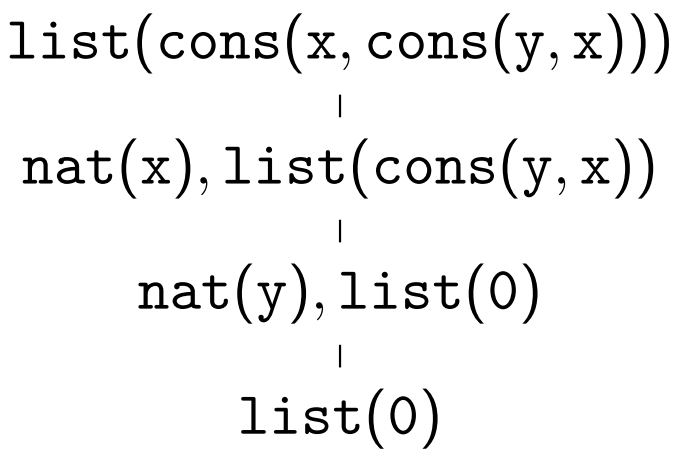
4, 1, 4, 1, 5.

$$\texttt{list}(\texttt{cons}(\texttt{x}, \texttt{cons}(\texttt{y}, \texttt{x})))$$
$$\texttt{nat}(\texttt{x}), \texttt{list}(\texttt{cons}(\texttt{y}, \texttt{x}))$$
$$\texttt{nat}(\texttt{y}), \texttt{list}(0)$$
$$\texttt{list}(0)$$

Example

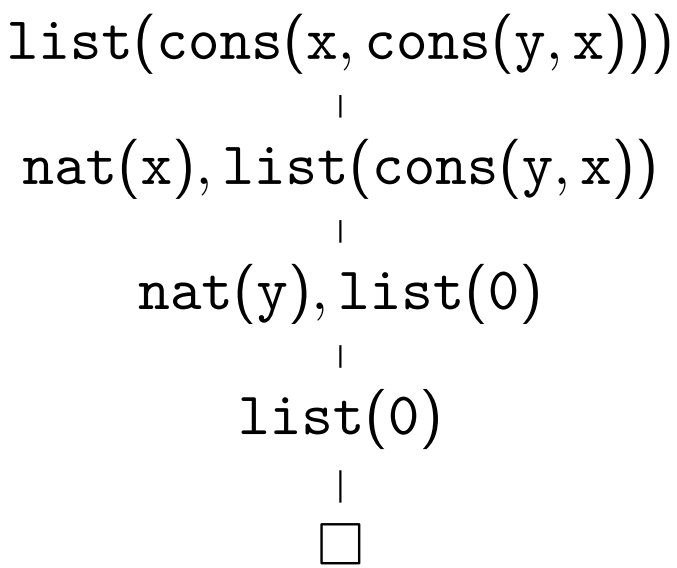
Correct

4, 1, 4, 1, 5.



Incorrect

4, 1, 4, 1, 6.



Example 2

Correct

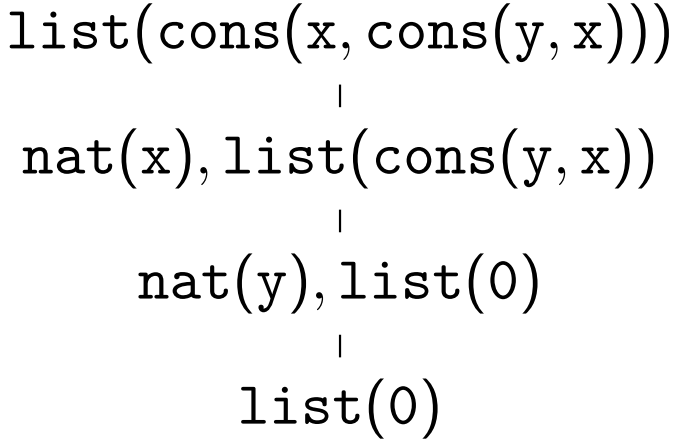
4, 1, 4, 1, 5.

$$\texttt{list(cons(x, cons(y, x)))}$$
$$\texttt{nat(x), list(cons(y, x))}$$
$$\texttt{nat(y), list(0)}$$
$$\texttt{list(0)}$$

Example 2

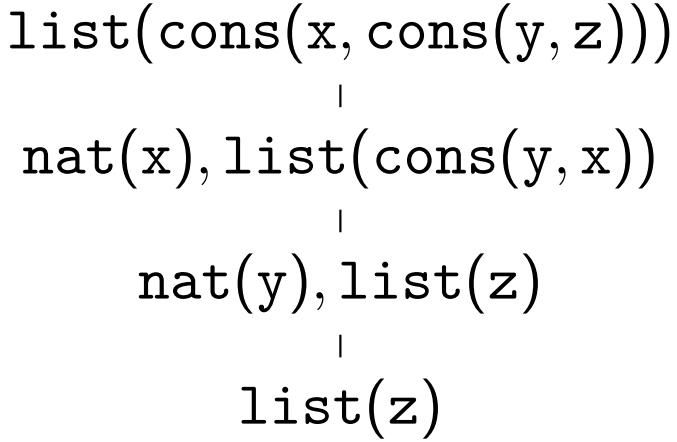
Correct

4, 1, 4, 1, 5.



Incorrect

4, 1, 4, 1, 5.



Conclusion

If the size of the training data set (= set of the examples used for training) is big and representative, derivations with "tactics" 4, 1, 4, 1, 5 are equally likely to be correct and incorrect.

What are the coinductive trees?

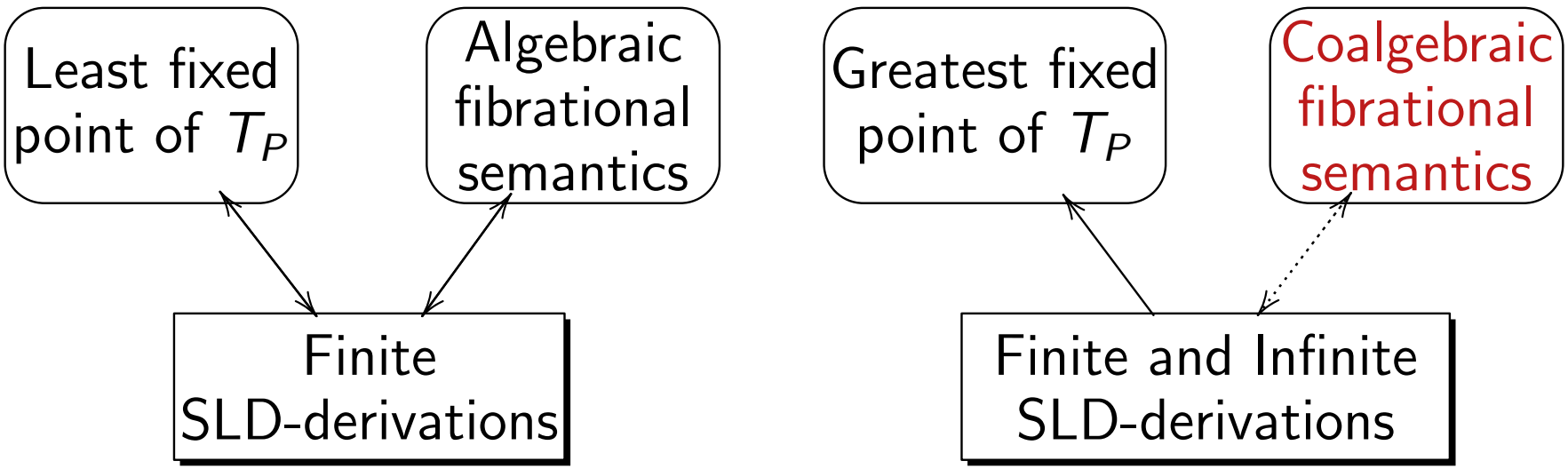
Lesson learnt

- The sequence of deductive rules alone does not help; and actually hides the structure of the proof.
- We need more "structural" representation of proofs.

Coinductive trees:

- They arose from coalgebraic semantics for derivations in logic programs, [Komendantskaya,Power CALCO'2011].
- They offer a proof method for recursive and corecursive logic programs.
- They also allow for concurrency.
- They offer very structured approach to automated proofs, as we will see shortly.

Algebraic and coalgebraic semantics for LP



Definition of coinductive trees

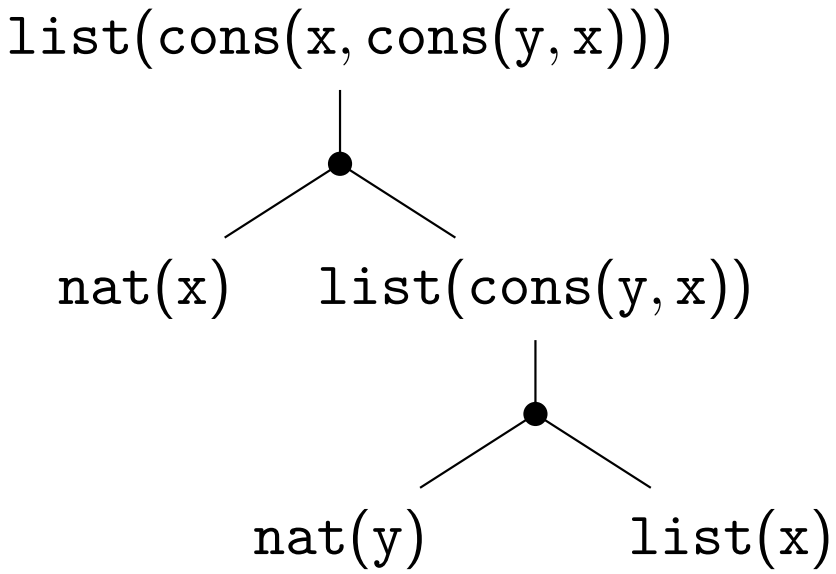
Definition

Let P be a logic program and $G = \leftarrow A$ be an atomic goal. The *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, 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 $B_1^i\theta_i, \ldots, B_{n_i}^i\theta_i$.

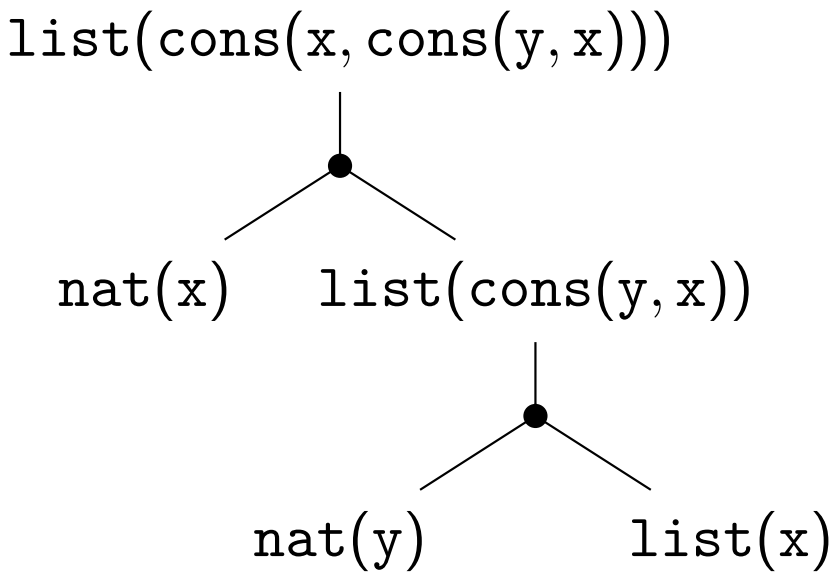
Examples of first-order coinductive trees:

Correct

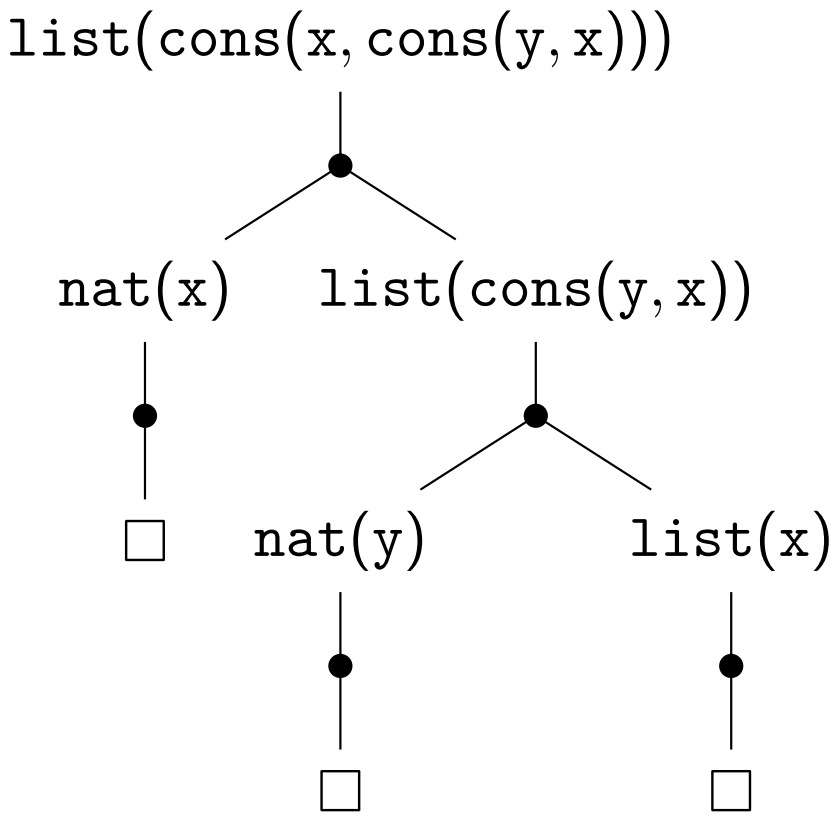


Examples of first-order coinductive trees:

Correct



Incorrect

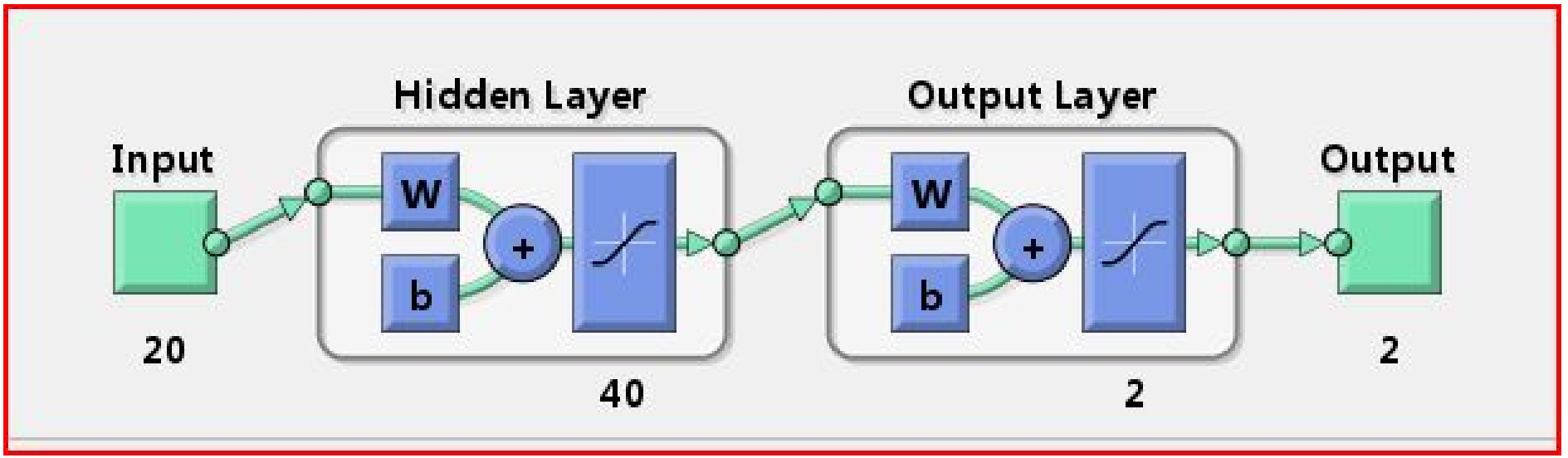


Representation in vectors

	list	nat	•	□
cons(x, cons(y, x))	- |cons(x, cons(y, x))|	0	2	0
cons(y, x))	- |cons(y, x))|	0	2	0
x	-1	-1	1	0
y	-1	0	1	0
z	0	0	0	0

And then it is further flattened into a vector.

A matrix of such vectors forms an input to the neural network:

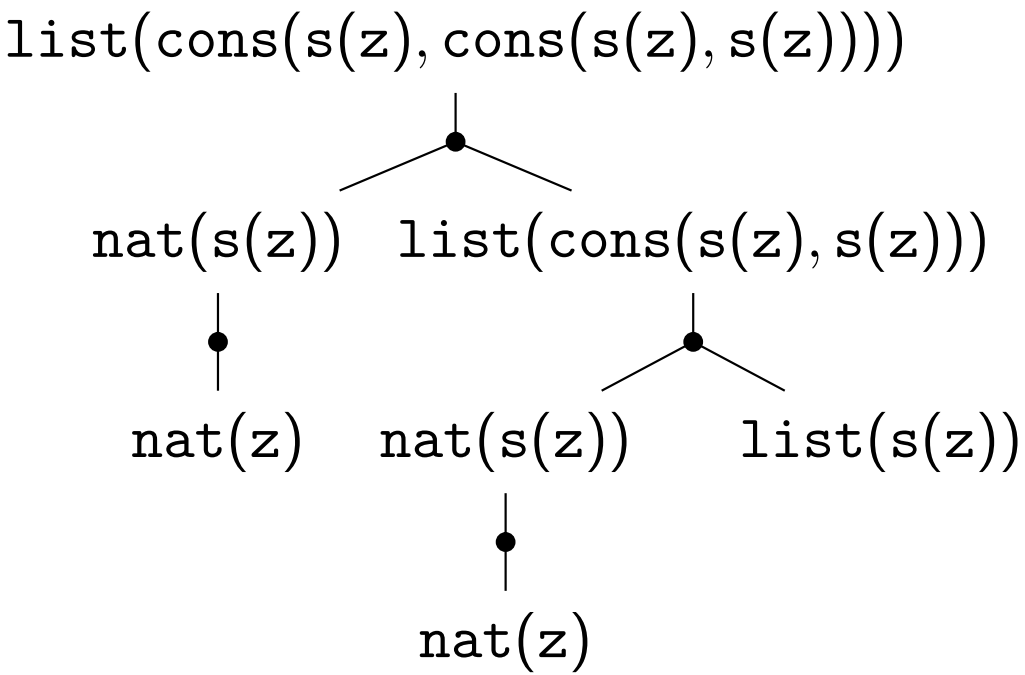


It's a two-layer feed-forward network, with sigmoid hidden and output neurons, that can classify vectors arbitrarily well, given enough neurons in its hidden layer.

The network was trained with scaled conjugate gradient back-propagation.

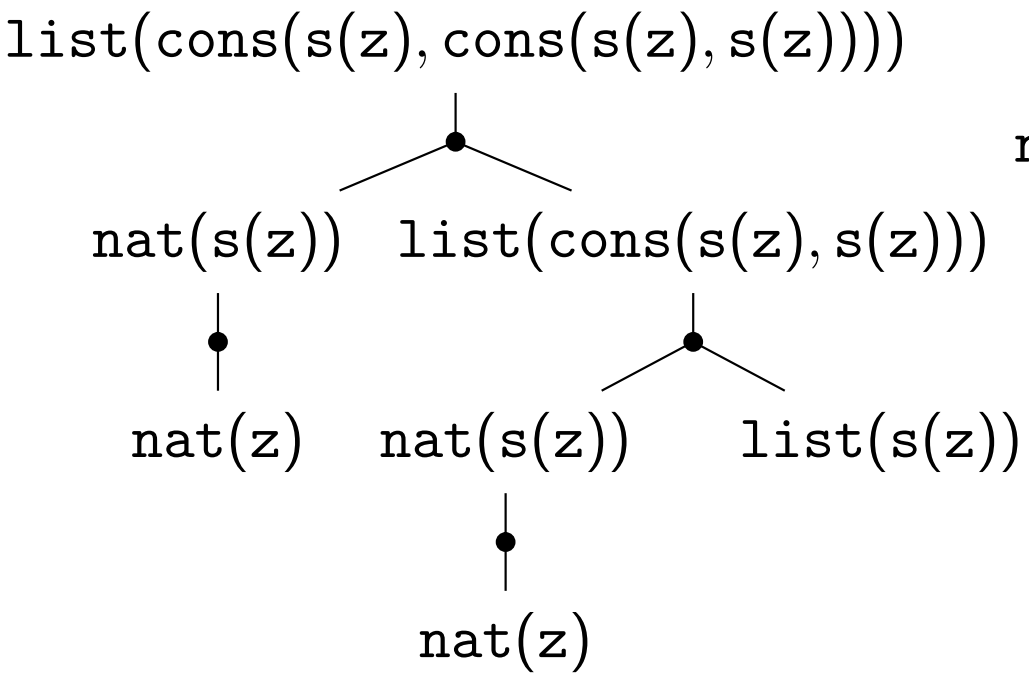
More examples coinductive trees:

Correct

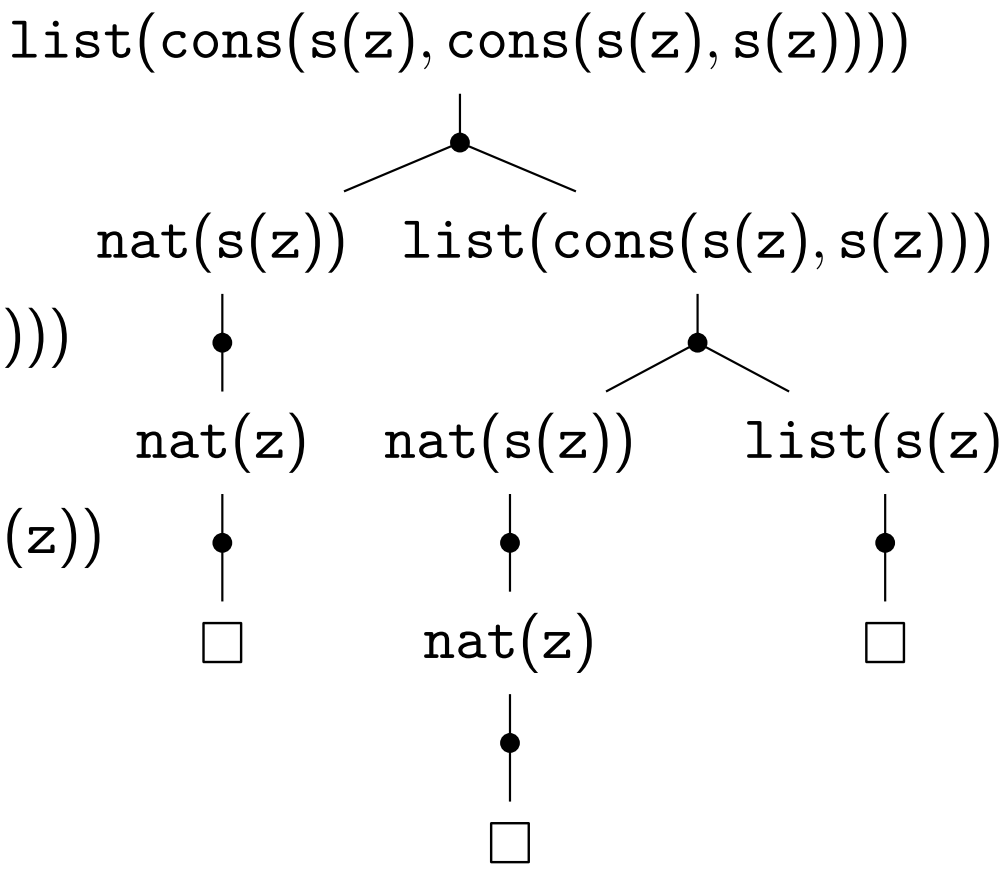


More examples coinductive trees:

Correct

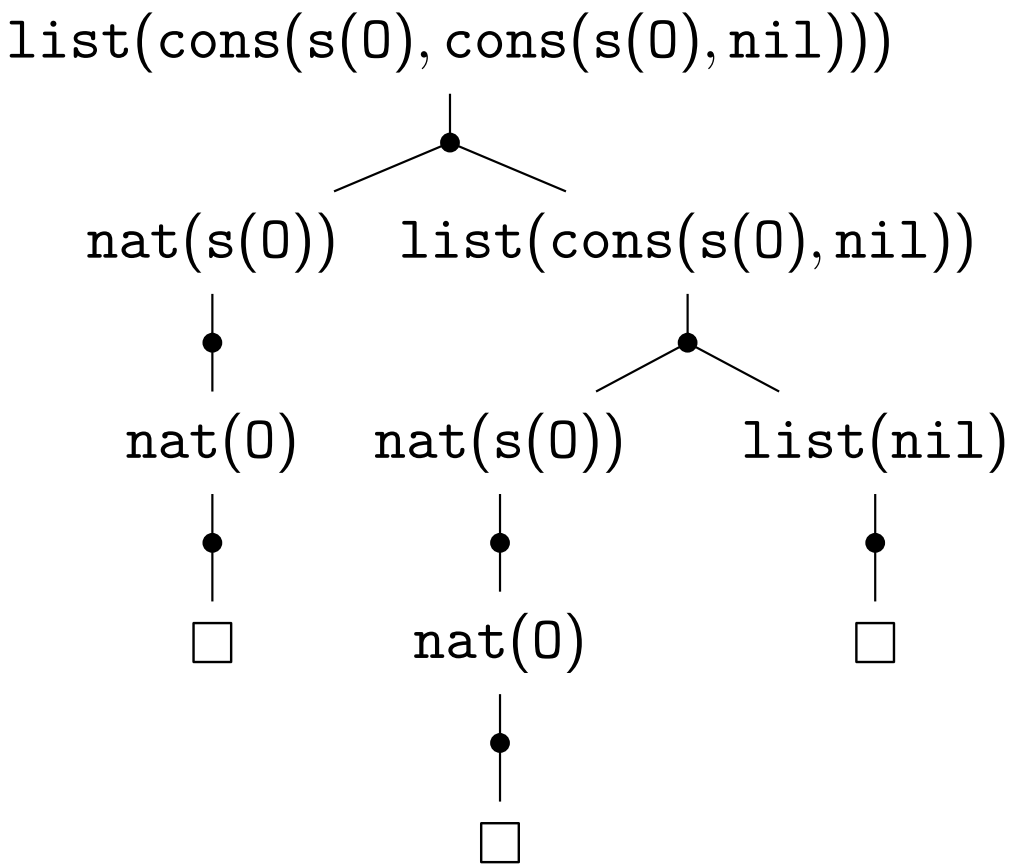


Incorrect



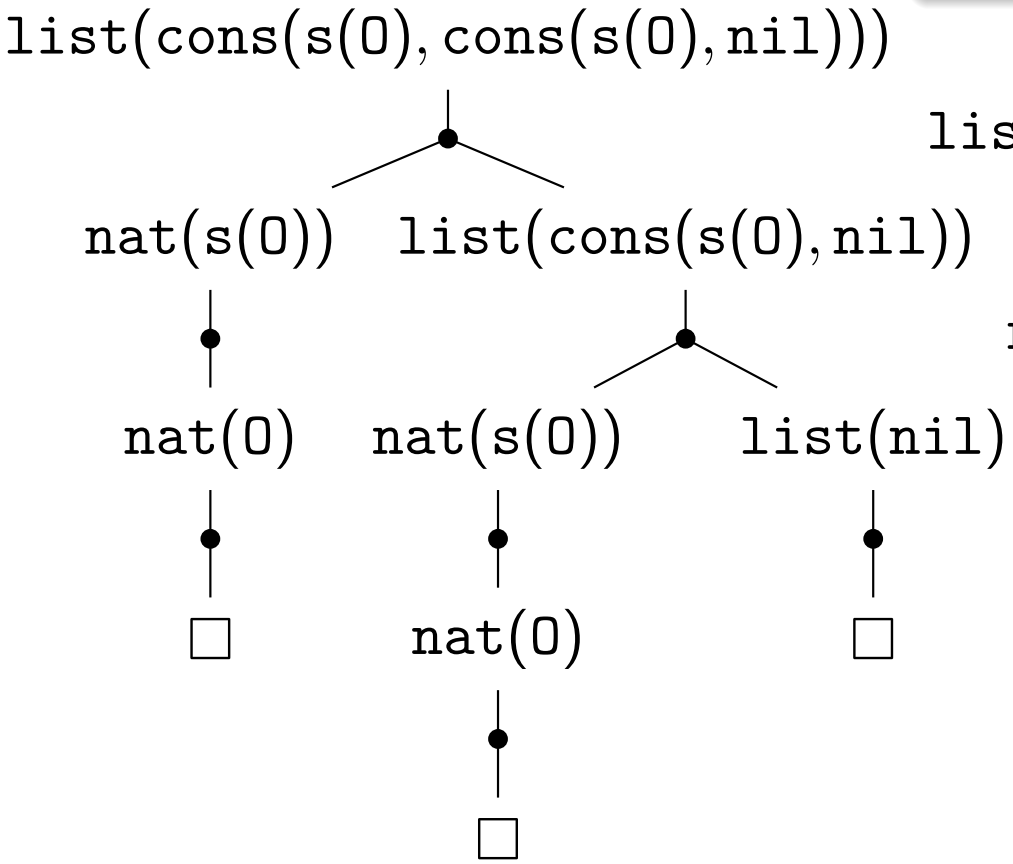
More examples of coinductive trees:

Correct

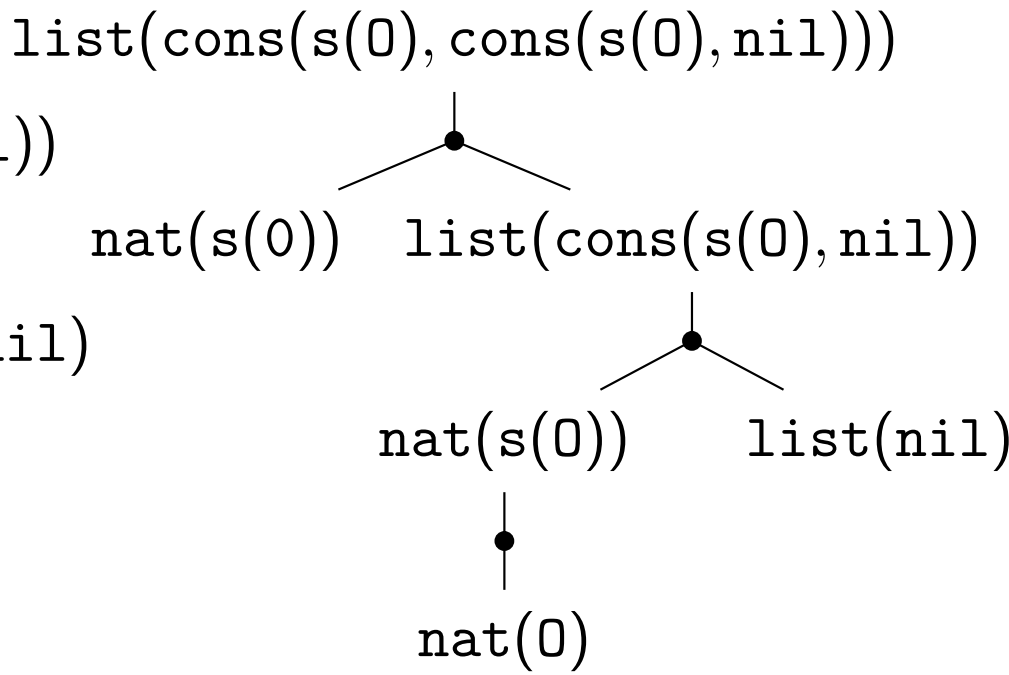


More examples of coinductive trees:

Correct



Incorrect



Test examples: Is this a correct CPT?

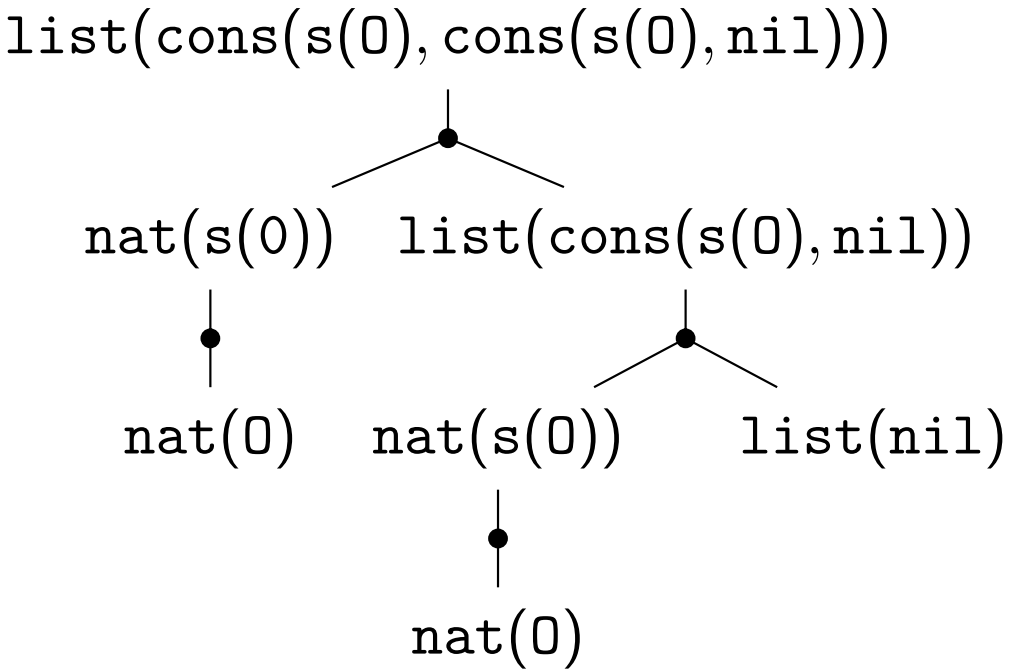
$$\texttt{nat}(\texttt{cons}(\texttt{s}(\texttt{0}), \texttt{cons}(\texttt{s}(\texttt{0}), \texttt{nil})))$$

Test examples: Is this a correct CPT?

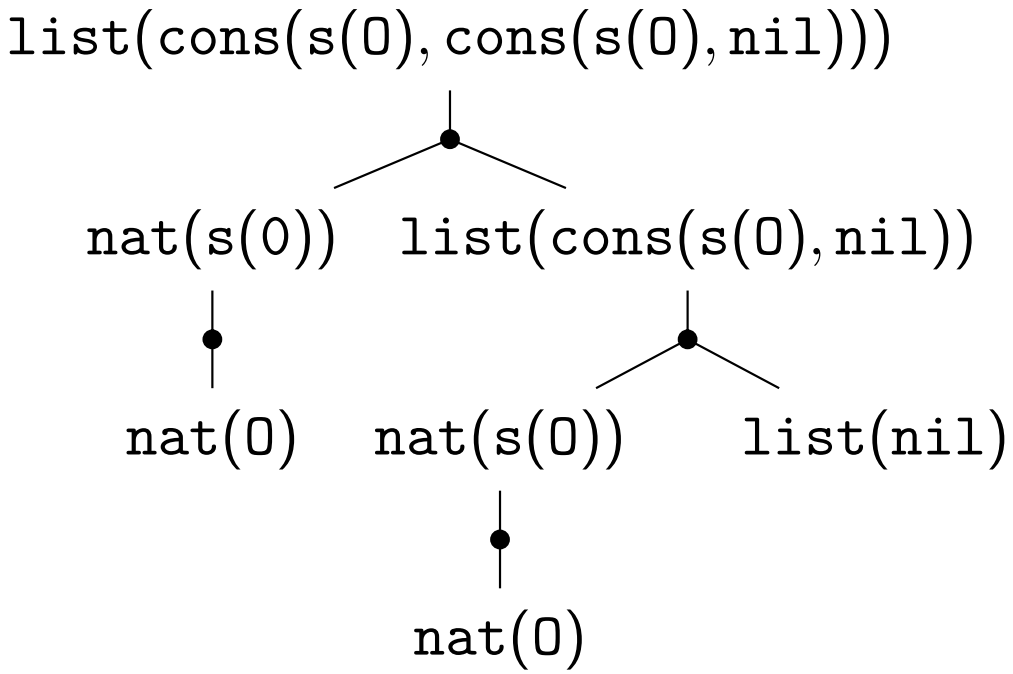
$$\texttt{nat}(\texttt{cons}(\texttt{s}(\texttt{0}), \texttt{cons}(\texttt{s}(\texttt{0}), \texttt{nil})))$$

Yes.

Test examples: Is this a correct CPT?

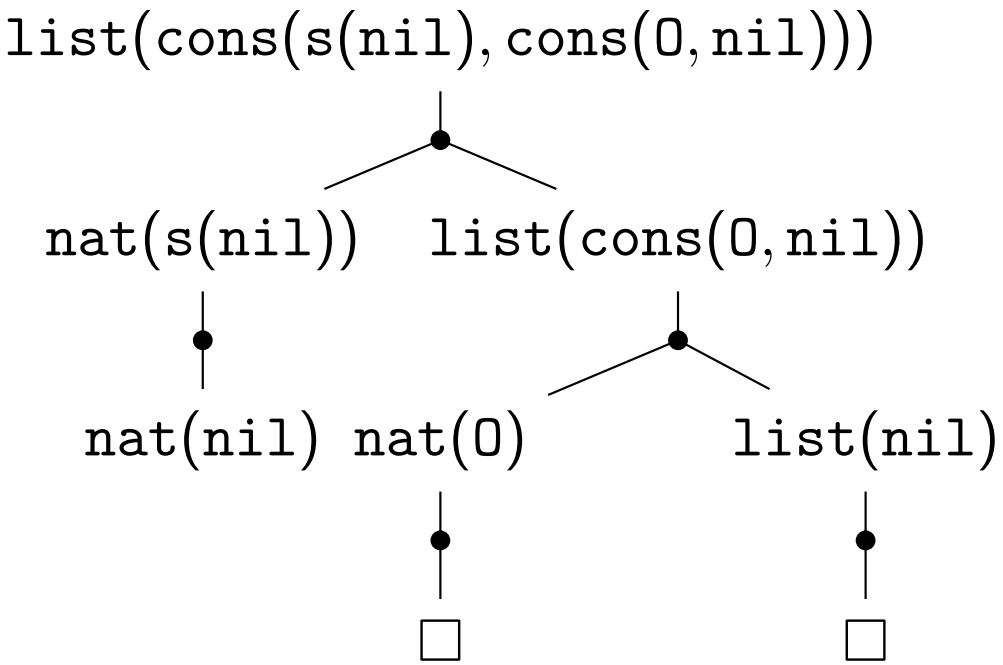


Test examples: Is this a correct CPT?

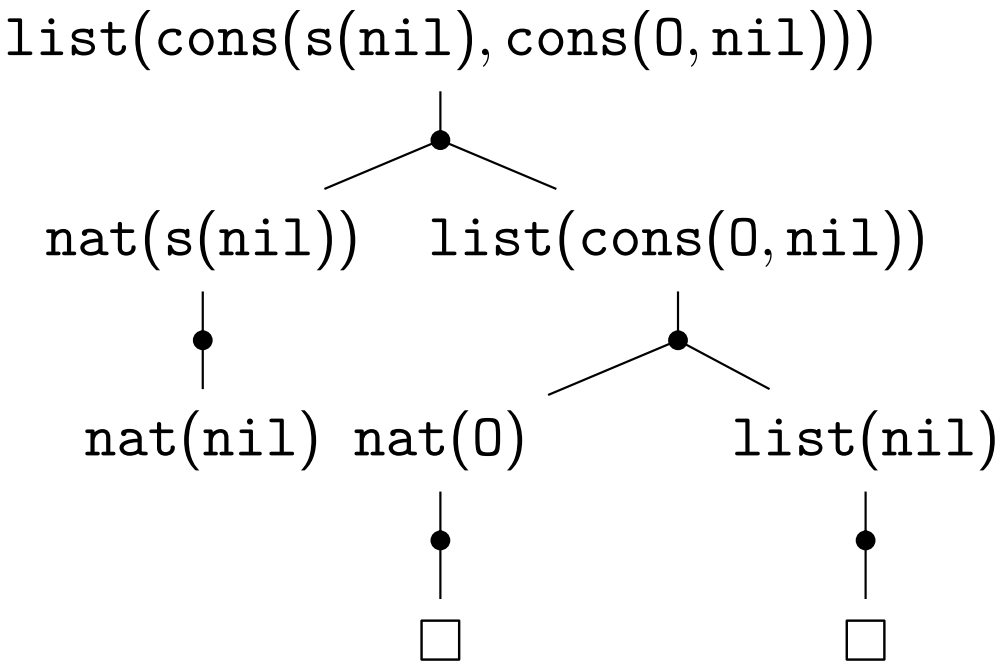


No.

Test examples: Is this a correct CPT?

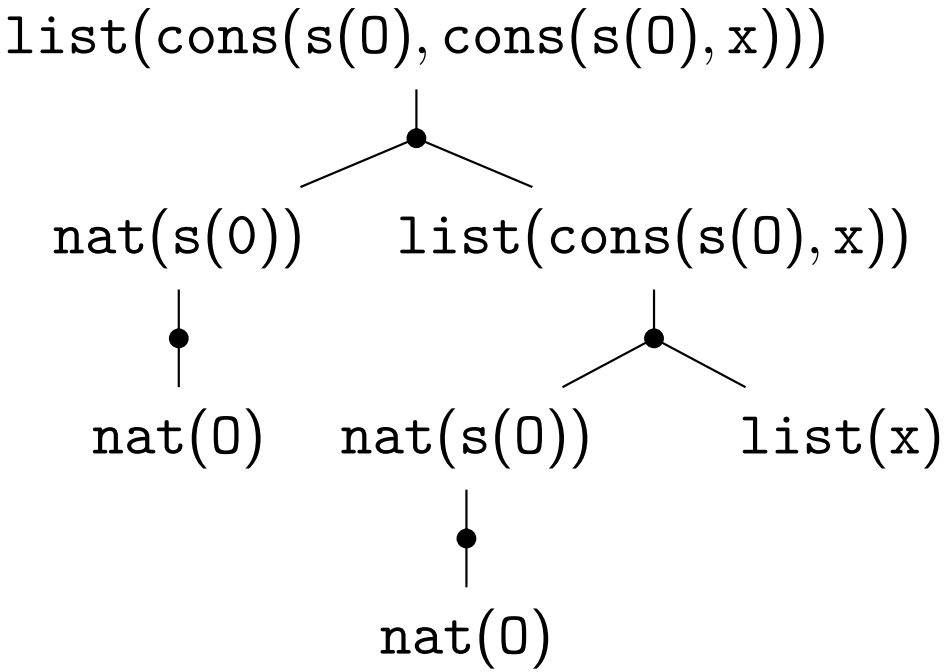


Test examples: Is this a correct CPT?

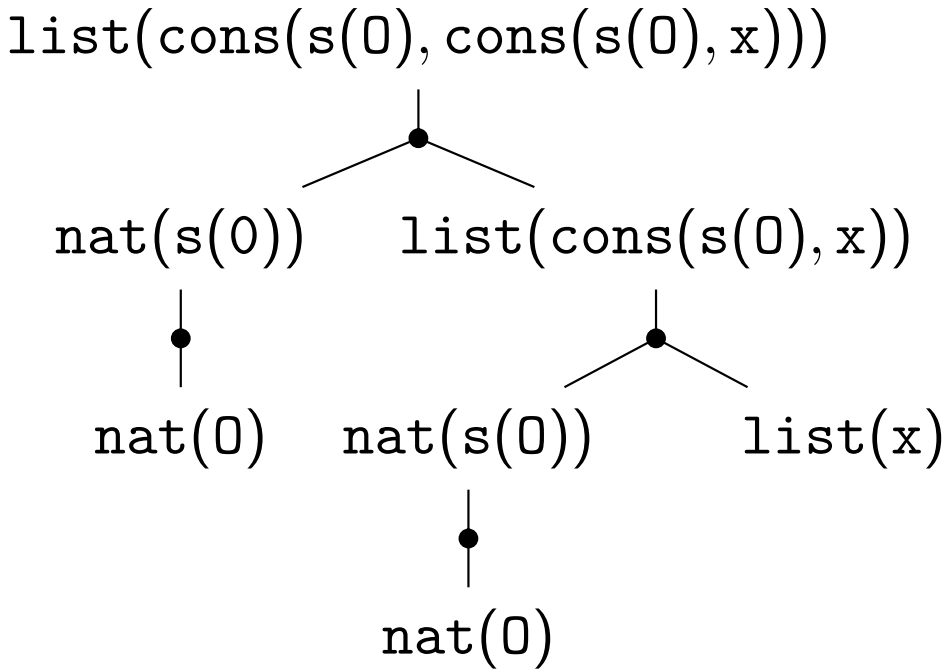


Yes.

Test examples: Is this a correct CPT?



Test examples: Is this a correct CPT?



No.

Our Result Versus Neural Nets result

AI4FM results — audience of computer scientists with good background in logic

Training set: 30 %
Testing examples: 50 %.

Our Result Versus Neural Nets result

AI4FM results — audience of computer scientists with good background in logic

Training set: 30 %
Testing examples: 50 %.

Neural Network

Training set: - roughly 82%.
(in the demo - 94%)
Testing examples: - 85 — 100%.

Conclusions:

- Many proofs, especially by (co-)induction, especially using constructors (like `nil` or `cons`), share some common structure (= follow some patterns in machine learning terms) that can be detected using statistical learning;
- We have tried to machine learn concurrent (coalgebraic) algorithms — e.g. coinductive proof trees [Komendantskaya,Power CALCO'2011 and CSL'11].
- Coalgebraic derivations look more promising than traditional sequential derivations.
- We have learned *both* from positive and negative examples. (models of "Why?")
- Big goal: move on to tactics in ITPs.

Questions?

(Please contact me katya@computing.dundee.ac.uk if they arise later!)