Statistical Machine Learning for Theorem Proving: Automated or Interactive?*

Katya Komendantskaya and Jónathan Heras

University of Dundee

11 April 2013

*Funded by EPSRC First Grant Scheme

Outline

Outline

Motivation

- Automated Theorem Provers (ATPs) and SAT/SMT solvers are
  - . . . fast and efficient;
  - . . . applied in different contexts: program verification, scheduling, test case generation, etc.

Motivation

- Automated Theorem Provers (ATPs) and SAT/SMT solvers are
  - ... fast and efficient;
  - ... applied in different contexts: program verification, scheduling, test case generation, etc.
- Interactive Theorem Provers (ITPs) have been
  - ... enriched with dependent types, (co)inductive types, type classes and provide rich programming environments;
  - ... applied in formal mathematical proofs: Four Colour Theorem (60, 000 lines), Kepler conjecture (325, 000 lines), Feit-Thompson Theorem (170, 000 lines), etc.
  - ... applied in industrial proofs: seL4 microkernel (200, 000 lines), verified C compiler (50, 000 lines), ARM microprocessor (20, 000 lines), etc.

Challenges

- . . . size of ATPs and ITPs libraries stand on the way of efficient knowledge reuse;
- . . . manual handling of various proofs, strategies, libraries, becomes difficult;
- . . . team-development is hard, especially that ITPs are sensitive to notation;
- . . . comparison of proof similarities is hard.

Running example

Java Virtual Machine (JVM) is a stack-based abstract machine which can execute Java bytecode.

Running example

Java Virtual Machine (JVM) is a stack-based abstract machine which can execute Java bytecode.

Goal

- Model a subset of the JVM in Coq, defining an interpreter for JVM programs,
- Verify the correctness of JVM programs within Coq.

Running example

Java Virtual Machine (JVM) is a stack-based abstract machine which can execute Java bytecode.

Goal

- Model a subset of the JVM in Coq, defining an interpreter for JVM programs,
- Verify the correctness of JVM programs within Coq.

This work is inspired by:

H. Liu and J S. Moore. Executable JVM model for analytical reasoning: a study. Journal Science of Computer Programming - Special issue on advances in interpreters, virtual machines and emulators (IVME'03), 57(3):253–274, 2003.

Running example

Java code:

```
static int factorial(int n)
{
  int a = 1;
  while (n != 0){
    a = a * n;
    n = n-1;
    }
  return a;
}
```

Running example

Java code:

```
static int factorial(int n)
{
  int a = 1;
  while (n != 0){
    a = a * n;
    n = n-1;
    }
  return a;
}
```

Bytecode:

0	:	*iconst* 1
1	:	*istore* 1
2	:	*iload* 0
3	:	*ifeq* 13
4	:	*iload* 1
5	:	*iload* 0
6	:	*imul*
7	:	*istore* 1
8	:	*iload* 0
9	:	*iconst* 1
10	:	*isub*
11	:	*istore* 0
12	:	*goto* 2
13	:	*iload* 1
14	:	*ireturn*

Running example

Java code:

```
static int factorial(int n)
{
  int a = 1;
  while (n != 0){
    a = a * n;
    n = n-1;
    }
  return a;
}
```

Bytecode:

0	:	*iconst* 1
1	:	*istore* 1
2	:	*iload* 0
3	:	*ifeq* 13
4	:	*iload* 1
5	:	*iload* 0
6	:	*imul*
7	:	*istore* 1
8	:	*iload* 0
9	:	*iconst* 1
10	:	*isub*
11	:	*istore* 0
12	:	*goto* 2
13	:	*iload* 1
14	:	*ireturn*

JVM model:

counter:

0

stack:

				...

local variables:

5				...

Running example

Java code:

```
static int factorial(int n)
{
  int a = 1;
  while (n != 0){
    a = a * n;
    n = n-1;
    }
  return a;
}
```

Bytecode:

0	:	*iconst* 1
1	:	*istore* 1
2	:	*iload* 0
3	:	*ifeq* 13
4	:	*iload* 1
5	:	*iload* 0
6	:	*imul*
7	:	*istore* 1
8	:	*iload* 0
9	:	*iconst* 1
10	:	*isub*
11	:	*istore* 0
12	:	*goto* 2
13	:	*iload* 1
14	:	*ireturn*

JVM model:

counter:

1

stack:

1				...

local variables:

5				...

Running example

Java code:

```
static int factorial(int n)
{
  int a = 1;
  while (n != 0){
    a = a * n;
    n = n-1;
    }
  return a;
}
```

Bytecode:

0	:	*iconst* 1
1	:	*istore* 1
2	:	*iload* 0
3	:	*ifeq* 13
4	:	*iload* 1
5	:	*iload* 0
6	:	*imul*
7	:	*istore* 1
8	:	*iload* 0
9	:	*iconst* 1
10	:	*isub*
11	:	*istore* 0
12	:	*goto* 2
13	:	*iload* 1
14	:	*ireturn*

JVM model:

counter:

2

stack:

				...

local variables:

5	1			...

Running example

Java code:

```
static int factorial(int n)
{
  int a = 1;
  while (n != 0){
    a = a * n;
    n = n-1;
    }
  return a;
}
```

Bytecode:

0 : *iconst* 1
1 : *istore* 1
2 : *iload* 0
3 : *ifeq* 13
4 : *iload* 1
5 : *iload* 0
6 : *imul*
7 : *istore* 1
8 : *iload* 0
9 : *iconst* 1
10 : *isub*
11 : *istore* 0
12 : *goto* 2
13 : *iload* 1
14 : *ireturn*

JVM model:

counter:

3

stack:

5				...

local variables:

5	1			...

Running example

Java code:

```
static int factorial(int n)
{
  int a = 1;
  while (n != 0){
    a = a * n;
    n = n-1;
    }
  return a;
}
```

Bytecode:

0	:	*iconst* 1
1	:	*istore* 1
2	:	*iload* 0
3	:	*ifeq* 13
4	:	*iload* 1
5	:	*iload* 0
6	:	*imul*
7	:	*istore* 1
8	:	*iload* 0
9	:	*iconst* 1
10	:	*isub*
11	:	*istore* 0
12	:	*goto* 2
13	:	*iload* 1
14	:	*ireturn*

JVM model:

counter:

4

stack:

				...

local variables:

5	1			...

Running example

Java code:

```
static int factorial(int n)
{
  int a = 1;
  while (n != 0){
    a = a * n;
    n = n-1;
    }
  return a;
}
```

Bytecode:

0	:	*iconst* 1
1	:	*istore* 1
2	:	*iload* 0
3	:	*ifeq* 13
4	:	*iload* 1
5	:	*iload* 0
6	:	*imul*
7	:	*istore* 1
8	:	*iload* 0
9	:	*iconst* 1
10	:	*isub*
11	:	*istore* 0
12	:	*goto* 2
13	:	*iload* 1
14	:	*ireturn*

JVM model:

counter:

5

stack:

1				...

local variables:

5	1			...

Running example

Java code:

```
static int factorial(int n)
{
  int a = 1;
  while (n != 0){
    a = a * n;
    n = n-1;
    }
  return a;
}
```

Bytecode:

0	:	*iconst* 1
1	:	*istore* 1
2	:	*iload* 0
3	:	*ifeq* 13
4	:	*iload* 1
5	:	*iload* 0
6	:	*imul*
7	:	*istore* 1
8	:	*iload* 0
9	:	*iconst* 1
10	:	*isub*
11	:	*istore* 0
12	:	*goto* 2
13	:	*iload* 1
14	:	*ireturn*

JVM model:

counter:

6

stack:

5	1			...

local variables:

5	1			...

Running example

Java code:

```
static int factorial(int n)
{
  int a = 1;
  while (n != 0){
    a = a * n;
    n = n-1;
    }
  return a;
}
```

Bytecode:

0	:	*iconst* 1
1	:	*istore* 1
2	:	*iload* 0
3	:	*ifeq* 13
4	:	*iload* 1
5	:	*iload* 0
6	:	*imul*
7	:	*istore* 1
8	:	*iload* 0
9	:	*iconst* 1
10	:	*isub*
11	:	*istore* 0
12	:	*goto* 2
13	:	*iload* 1
14	:	*ireturn*

JVM model:

counter:

7

stack:

5				. . .

local variables:

5	1			. . .

Running example

Java code:

```
static int factorial(int n)
{
  int a = 1;
  while (n != 0){
    a = a * n;
    n = n-1;
    }
  return a;
}
```

Bytecode:

0	:	*iconst* 1
1	:	*istore* 1
2	:	*iload* 0
3	:	*ifeq* 13
4	:	*iload* 1
5	:	*iload* 0
6	:	*imul*
7	:	*istore* 1
8	:	*iload* 0
9	:	*iconst* 1
10	:	*isub*
11	:	*istore* 0
12	:	*goto* 2
13	:	*iload* 1
14	:	*ireturn*

JVM model:

counter:

8

stack:

				...

local variables:

5	5			...

Running example

Java code:

```
static int factorial(int n)
{
  int a = 1;
  while (n != 0){
    a = a * n;
    n = n-1;
    }
  return a;
}
```

Bytecode:

0	:	*iconst* 1
1	:	*istore* 1
2	:	*iload* 0
3	:	*ifeq* 13
4	:	*iload* 1
5	:	*iload* 0
6	:	*imul*
7	:	*istore* 1
8	:	*iload* 0
9	:	*iconst* 1
10	:	*isub*
11	:	*istore* 0
12	:	*goto* 2
13	:	*iload* 1
14	:	*ireturn*

JVM model:

counter:

9

stack:

5				...

local variables:

5	5			...

Running example

Java code:

```
static int factorial(int n)
{
  int a = 1;
  while (n != 0){
    a = a * n;
    n = n-1;
    }
  return a;
}
```

Bytecode:

0	:	*iconst* 1
1	:	*istore* 1
2	:	*iload* 0
3	:	*ifeq* 13
4	:	*iload* 1
5	:	*iload* 0
6	:	*imul*
7	:	*istore* 1
8	:	*iload* 0
9	:	*iconst* 1
10	:	*isub*
11	:	*istore* 0
12	:	*goto* 2
13	:	*iload* 1
14	:	*ireturn*

JVM model:

counter:

10

stack:

1	5			...

local variables:

5	5			...

Running example

Java code:

```
static int factorial(int n)
{
  int a = 1;
  while (n != 0){
    a = a * n;
    n = n-1;
    }
  return a;
}
```

Bytecode:

0	:	*iconst* 1
1	:	*istore* 1
2	:	*iload* 0
3	:	*ifeq* 13
4	:	*iload* 1
5	:	*iload* 0
6	:	*imul*
7	:	*istore* 1
8	:	*iload* 0
9	:	*iconst* 1
10	:	*isub*
11	:	*istore* 0
12	:	*goto* 2
13	:	*iload* 1
14	:	*ireturn*

JVM model:

counter:

11

stack:

4				...

local variables:

5	5			...

Running example

Java code:

```
static int factorial(int n)
{
  int a = 1;
  while (n != 0){
    a = a * n;
    n = n-1;
    }
  return a;
}
```

Bytecode:

0	:	*iconst* 1
1	:	*istore* 1
2	:	*iload* 0
3	:	*ifeq* 13
4	:	*iload* 1
5	:	*iload* 0
6	:	*imul*
7	:	*istore* 1
8	:	*iload* 0
9	:	*iconst* 1
10	:	*isub*
11	:	*istore* 0
12	:	*goto* 2
13	:	*iload* 1
14	:	*ireturn*

JVM model:

counter:

12

stack:

				...

local variables:

4	5			...

Running example

Java code:

```
static int factorial(int n)
{
  int a = 1;
  while (n != 0){
    a = a * n;
    n = n-1;
    }
  return a;
}
```

Bytecode:

0	:	*iconst* 1
1	:	*istore* 1
2	:	*iload* 0
3	:	*ifeq* 13
4	:	*iload* 1
5	:	*iload* 0
6	:	*imul*
7	:	*istore* 1
8	:	*iload* 0
9	:	*iconst* 1
10	:	*isub*
11	:	*istore* 0
12	:	*goto* 2
13	:	*iload* 1
14	:	*ireturn*

JVM model:

counter:

2

stack:

				...

local variables:

4	5			...

Running example

Java code:

```
static int factorial(int n)
{
  int a = 1;
  while (n != 0){
    a = a * n;
    n = n-1;
    }
  return a;
}
```

Bytecode:

...

JVM model:

...

Running example

Java code:

```
static int factorial(int n)
{
  int a = 1;
  while (n != 0){
    a = a * n;
    n = n-1;
    }
  return a;
}
```

Bytecode:

0	:	*iconst* 1
1	:	*istore* 1
2	:	*iload* 0
3	:	*ifeq* 13
4	:	*iload* 1
5	:	*iload* 0
6	:	*imul*
7	:	*istore* 1
8	:	*iload* 0
9	:	*iconst* 1
10	:	*isub*
11	:	*istore* 0
12	:	*goto* 2
13	:	*iload* 1
14	:	*ireturn*

JVM model:

counter:

13

stack:

0				...

local variables:

0	120			...

Running example

Java code:

```
static int factorial(int n)
{
  int a = 1;
  while (n != 0){
    a = a * n;
    n = n-1;
    }
  return a;
}
```

Bytecode:

0	:	*iconst* 1
1	:	*istore* 1
2	:	*iload* 0
3	:	*ifeq* 13
4	:	*iload* 1
5	:	*iload* 0
6	:	*imul*
7	:	*istore* 1
8	:	*iload* 0
9	:	*iconst* 1
10	:	*isub*
11	:	*istore* 0
12	:	*goto* 2
13	:	*iload* 1
14	:	*ireturn*

JVM model:

counter:

14

stack:

120				...

local variables:

0	120			...

Running example

Java code:

```
static int factorial(int n)
{
  int a = 1;
  while (n != 0){
    a = a * n;
    n = n-1;
    }
  return a;
}
```

Bytecode:

0	:	*iconst* 1
1	:	*istore* 1
2	:	*iload* 0
3	:	*ifeq* 13
4	:	*iload* 1
5	:	*iload* 0
6	:	*imul*
7	:	*istore* 1
8	:	*iload* 0
9	:	*iconst* 1
10	:	*isub*
11	:	*istore* 0
12	:	*goto* 2
13	:	*iload* 1
14	:	*ireturn*

JVM model:

counter:

15

stack:

120				...

local variables:

0	120			...

Running example

Java code:

```
static int factorial(int n)
{
  int a = 1;
  while (n != 0){
    a = a * n;
    n = n-1;
    }
  return a;
}
```

Bytecode:

0	:	*iconst* 1
1	:	*istore* 1
2	:	*iload* 0
3	:	*ifeq* 13
4	:	*iload* 1
5	:	*iload* 0
6	:	*imul*
7	:	*istore* 1
8	:	*iload* 0
9	:	*iconst* 1
10	:	*isub*
11	:	*istore* 0
12	:	*goto* 2
13	:	*iload* 1
14	:	*ireturn*

JVM model:

counter:

15

stack:

120				...

local variables:

0	120			...

Goal (Factorial case)

$\forall n \in \mathbb{N}$, running the bytecode associated with the factorial program with n as input produces a state which contains $n!$ on top of the stack.

Outline

Proof pattern recognition in ATPs

Given a proof goal, ATPs apply various lemmas to rewrite or simplify the goal until it is proven.

Proof pattern recognition in ATPs

Given a proof goal, ATPs apply various lemmas to rewrite or simplify the goal until it is proven.

Goal

Apply machine-learning techniques to improve the premise selection procedure on the basis of previous experience.

Proof pattern recognition in ATPs

Given a proof goal, ATPs apply various lemmas to rewrite or simplify the goal until it is proven.

Goal

Apply machine-learning techniques to improve the premise selection procedure on the basis of previous experience.

References:

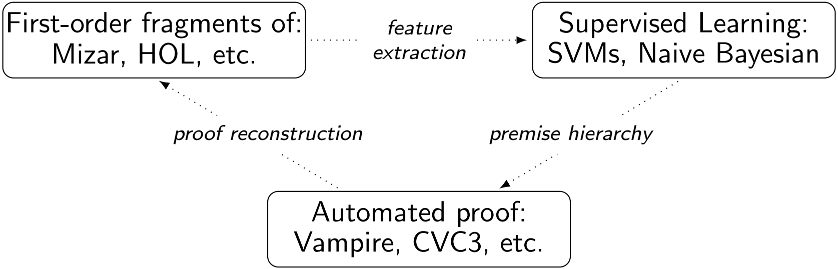
- D. Kühlwein et al. MaSh: Machine Learning for Sledgehammer. In ITP'13, 2013
- C. Kaliszyk and J. Urban. Learning-assisted Automated Reasoning with Flyspeck. 2012
- D. Kühlwein et al. Overview and evaluation of premise selection techniques for large theory mathematics. In IJCAR12, LNCS 7364, pages 378–392, 2012.
- E. Tsivtsivadze et al. Semantic graph kernels for automated reasoning. In SDM11, pages 795–803, 2011.

Application to ITPs

Several ITPs use ATPs to discharge proof obligations. Then, the ATP approach can be used to speed up those proofs.

Application to ITPs

Several ITPs use ATPs to discharge proof obligations. Then, the ATP approach can be used to speed up those proofs.



Intuitive idea

Goal

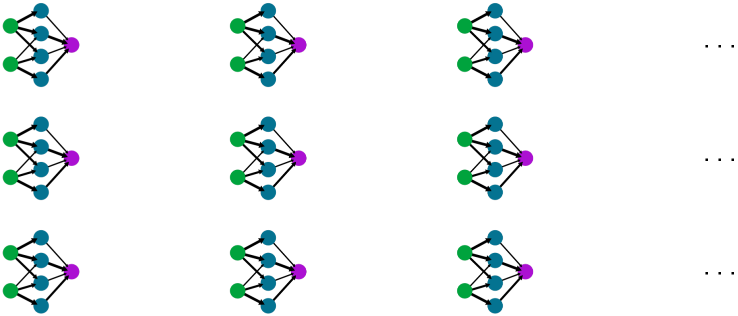
Determine the lemmas that can be useful to prove the equivalence between the recursive and tail-recursive versions of factorial.

Intuitive idea

Goal

Determine the lemmas that can be useful to prove the equivalence between the recursive and tail-recursive versions of factorial.

A classifier for each lemma in the library.



Intuitive idea

Goal

Determine the lemmas that can be useful to prove the equivalence between the recursive and tail-recursive versions of factorial.

Training phase:

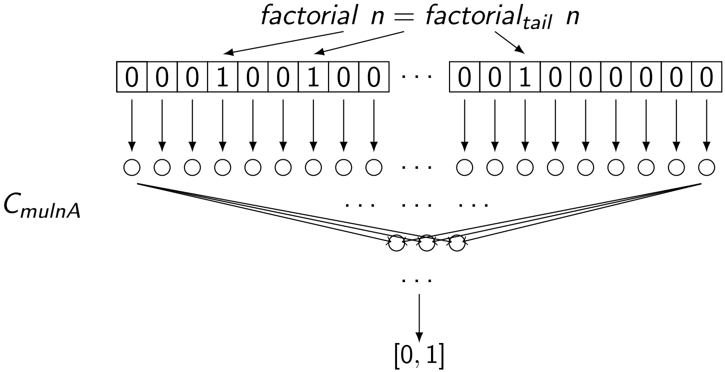
- lemma A is used in the proof of lemma $B \implies < A > (B) = 1$;
- otherwise $\implies < A > (B) = 0$;

Intuitive idea

Goal

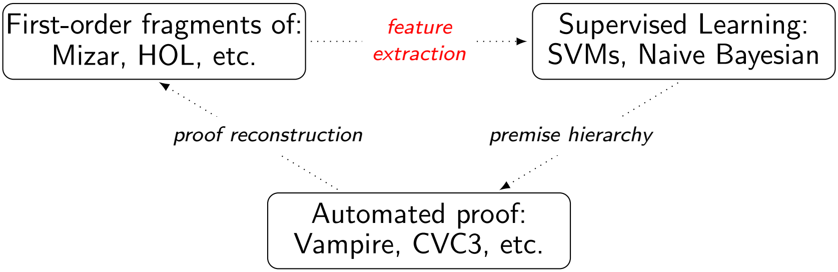
Determine the lemmas that can be useful to prove the equivalence between the recursive and tail-recursive versions of factorial.

Testing phase:



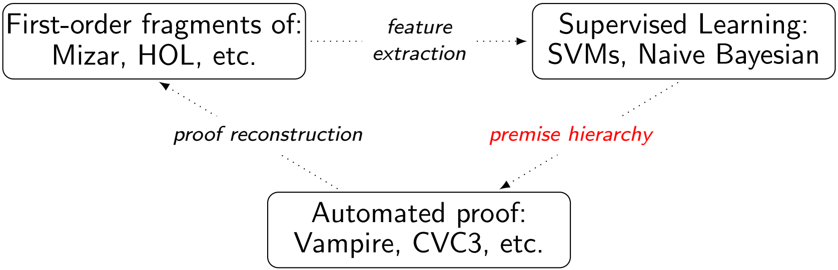
Features of this approach

1. Feature extraction:
   - features are extracted from first-order formulas;
   - sparse feature vectors (10^6 features);
   - classifier for every lemma of the library.



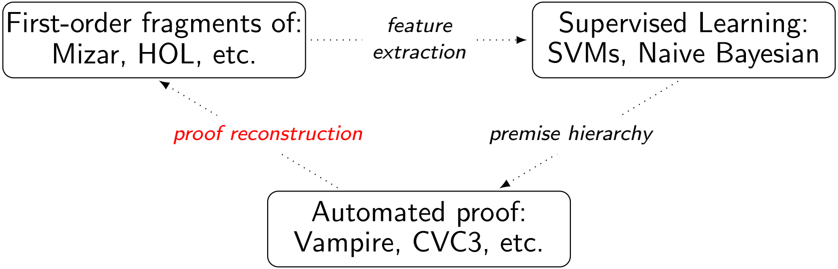
Features of this approach

2. Machine-learning tools:
   - work with supervised learning;
   - algorithms range from SVMs to Naive Bayes learning;
   - sparse methods; using software such as SNoW.



Features of this approach

3. Main improvement:
   - the number of goals proven automatically increases by up to 20% – 40%



Outline

Proof pattern recognition in ITPs

In ITPs, the proof steps are suggested by the user who guides the prover by providing the tactics.

Proof pattern recognition in ITPs

In ITPs, the proof steps are suggested by the user who guides the prover by providing the tactics.

Goal

Apply machine-learning methods to:

- find common proof-patterns in proofs across various scripts, libraries, users and notations;
- and provide proof-hints.

Proof pattern recognition in ITPs

In ITPs, the proof steps are suggested by the user who guides the prover by providing the tactics.

Goal

Apply machine-learning methods to:

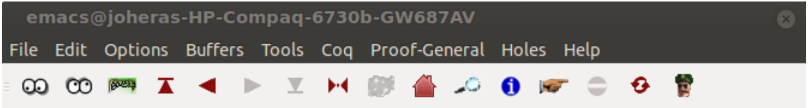
- find common proof-patterns in proofs across various scripts, libraries, users and notations;
- and provide proof-hints.

ML4PG:

- Proof General extension which applies machine learning methods to Coq/SSReflect proofs.

E. Komendantskaya, J. Heras and G. Grov. Machine learning in Proof General: interfacing interfaces. EPTCS Post-proceedings of User Interfaces for Theorem Provers. 2013.

A proof in Coq/SSReflect




```
Lemma fact_tail_aux_lemma : forall (a n : nat), fact_tail_aux n a = a
      * n`!.
Proof.
```

```
-U:**-  lists.v        All L1     (Coq Script(0) Holes)---------------------------
```

```
1 subgoals, subgoal 1 (ID 13)

============================
 forall n a : nat, fact_tail_aux n a = a * n`!
```

```
-U:%%-  *response*     All L1     (Coq Response)-----------------------------------
```

A proof in Coq/SSReflect

emacs@joheras-HP-Compaq-6730b-GW687AV

File Edit Options Buffers Tools Coq Proof-General Holes Help

```
Lemma fact_tail_aux_lemma : forall (a n : nat), fact_tail_aux n a = a
      * n`!.
Proof.
move => n.
```

```
-U:**-  lists.v       All L1     (Coq Script(0) Holes)----------------------------
```

```
1 subgoals, subgoal 1 (ID 14)

n : nat
============================
 forall a : nat, fact_tail_aux n a = a * n`!
```

```
-U:%%-  *response*    All L1     (Coq Response)-----------------------------------
```

A proof in Coq/SSReflect

emacs@joheras-HP-Compaq-6730b-GW687AV

File Edit Options Buffers Tools Coq Proof-General Holes Help

```
Lemma fact_tail_aux_lemma : forall (a n : nat), fact_tail_aux n a = a
      * n`!.
Proof.
move => n. elim : n => [a| n IH a /=].
```

```
-U:**-  lists.v        All L1     (Coq Script(0) Holes)---------------------------
```

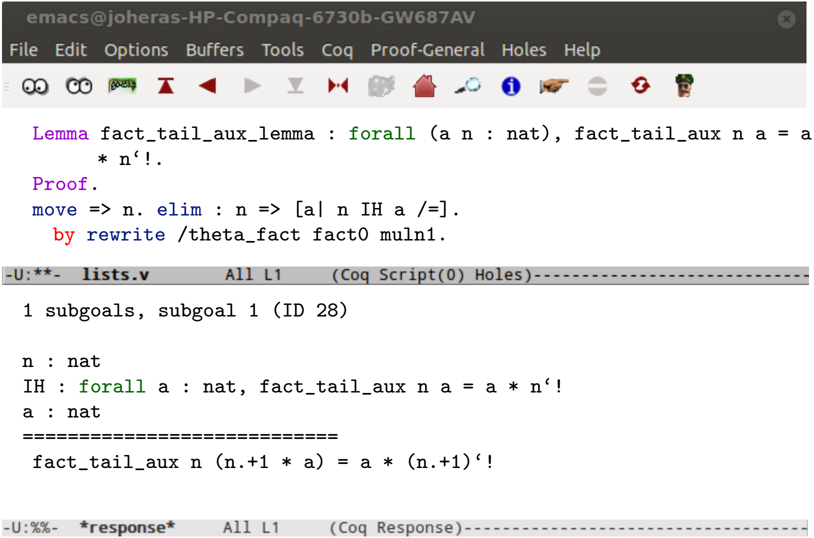
```
 2 subgoals, subgoal 1 (ID 24)

 a : nat
 ============================
  fact_tail_aux 0 a = a * 0`!

subgoal 2 (ID 28) is:
 fact_tail_aux n (n.+1 * a) = a * (n.+1)`!
```

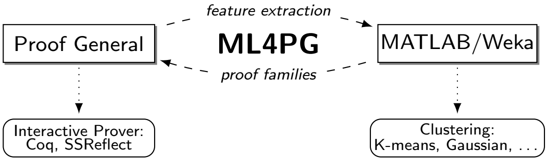
```
-U:%%-  *response*     All L1     (Coq Response)------------------------------------
```

A proof in Coq/SSReflect



ML4PG

ML4PG assists the user providing similar lemmas as proof hints.



Feature extraction mechanism

```
Lemma fact_tail_aux_lemma : forall (a n : nat), fact_tail_aux n a = a
     * n`!.
Proof.
```

	tactics	*N tactics*	*arg type*	*tactic arg is hypothesis?*	*top symbol*	*subgoals*
g1						
g2						
g3						
g4						
g5						

Feature extraction mechanism

```
Lemma fact_tail_aux_lemma : forall (a n : nat), fact_tail_aux n a = a
      * n`!.
Proof.
move => n.
```

	tactics	*N tactics*	*arg type*	*tactic arg is hypothesis?*	*top symbol*	*subgoals*
g1	move	1	nat	no	forall	1
g2						
g3						
g4						
g5						

Feature extraction mechanism

```
Lemma fact_tail_aux_lemma : forall (a n : nat), fact_tail_aux n a = a
      * n`!.
Proof.
move => n. elim : n => [a| n IH a /=].
```

	tactics	*N tactics*	*arg type*	*tactic arg is hypothesis?*	*top symbol*	*subgoals*
g1	move	1	nat	no	forall	1
g2	elim, move	2	nat, [nat | nat Prop nat]	yes	forall	2
g3						
g4						
g5						

Feature extraction mechanism

```
Lemma fact_tail_aux_lemma : forall (a n : nat), fact_tail_aux n a = a
      * n‘!.
Proof.
move => n. elim : n => [a| n IH a /=].
  by rewrite /theta_fact fact0 muln1.
```

	tactics	*N tactics*	*arg type*	*tactic arg is hypothesis?*	*top symbol*	*subgoals*
g1	move	1	nat	no	forall	1
g2	elim, move	2	nat, [nat | nat Prop nat]	yes	forall	2
g3	rewrite	1	Prop, Prop, Prop	EL, EL, EL	equal	1
g4						
g5						

Features of this approach

1. Feature extraction:
   - features are extracted from higher-order propositions and proofs;
   - feature extraction is built on the method of proof-traces;
   - the feature vectors are fixed at the size of 30;
   - longer proofs are analysed by means of the proof-patch method.

Features of this approach

2. Machine-learning tools:
   - work with unsupervised learning (clustering) algorithms implemented in MATLAB and Weka;
   - use algorithms such as Gaussian, K-means, and farthest-first.

A proof in Coq/SSReflect with ML4PG help

emacs@joheras-HP-Compaq-6730b-GW687AV

File Edit Options Buffers Tools Coq Proof-General Holes Help

```
Lemma fact_tail_aux_lemma : forall (a n : nat), fact_tail_aux n a = a
      * n`!.
Proof.
move => n. elim : n => [a| n IH a /=].
  by rewrite /theta_fact fact0 muln1.
```

-U:**- **lists.v** All L1 (Coq Script(0) Holes)

```
n : nat
IH : forall a : nat,
       fact_tail_aux n a =
             a * n`!
a : nat
============================

 fact_tail_aux n (n.+1 * a)
      =
 a * (n.+1)`!
```

This lemma is similar to lemmas:
- mult_tail_aux_lemma
- power_tail_aux_lemma
- expt_tail_aux_lemma

-U:%%- ***response*** All L1 (Coq Response)

Outline

Conclusions

Different Machine Learning methods are suitable for ATPs and ITPs.

Statistical Machine Learning for Theorem Proving: Automated or Interactive?*

Katya Komendantskaya and Jónathan Heras

University of Dundee

11 April 2013

*Funded by EPSRC First Grant Scheme