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

1 Motivation

2 Proof pattern recognition in ATPs

3 Proof pattern recognition in ITPs

4 Conclusions
Outline

1 Motivation

2 Proof pattern recognition in ATPs

3 Proof pattern recognition in ITPs

4 Conclusions
Motivation

- Automated Theorem Provers (ATPs) and SAT/SMT solvers are
  - ...fast and efficient;
  - ...applied in different contexts: program verification, scheduling, test case generation, etc.
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.
Java Virtual Machine (JVM) is a stack-based abstract machine which can execute Java bytecode.
Motivation

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:
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:

Java code:

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

Java code:

```java
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:

```java
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  .  .  .
```

Goal (Factorial case)

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

Java code:

```java
class Factorial {  
    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
```

Goal (Factorial case)

∀ n ∈ N, running the bytecode associated with the factorial program with n as input produces a state which contains n! on top of the stack.
Running example

Java code:

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

Bytecode:

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>iconst 1</td>
</tr>
<tr>
<td>1</td>
<td>istore 1</td>
</tr>
<tr>
<td>2</td>
<td>iload 0</td>
</tr>
<tr>
<td>3</td>
<td>ifeq 13</td>
</tr>
<tr>
<td>4</td>
<td>iload 1</td>
</tr>
<tr>
<td>5</td>
<td>iload 0</td>
</tr>
<tr>
<td>6</td>
<td>imul</td>
</tr>
<tr>
<td>7</td>
<td>istore 1</td>
</tr>
<tr>
<td>8</td>
<td>iload 0</td>
</tr>
<tr>
<td>9</td>
<td>iconst 1</td>
</tr>
<tr>
<td>10</td>
<td>isub</td>
</tr>
<tr>
<td>11</td>
<td>istore 0</td>
</tr>
<tr>
<td>12</td>
<td>goto 2</td>
</tr>
<tr>
<td>13</td>
<td>iload 1</td>
</tr>
<tr>
<td>14</td>
<td>ireturn</td>
</tr>
</tbody>
</table>

JVM model:

- **counter:** 2
- **stack:** [ ] [ ] [ ] [ ] [...]
- **local variables:** [5] [1] [ ] [ ] [...]

Goal (Factorial case)

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

Java code:

```java
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 \[\cdots\]

local variables: 5 1 \[\cdots\]
## Running example

Java code:

```java
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:

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

Bytecode:

```
0 : icnst 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 : icnst 1
10 : isub
11 : istore 0
12 : goto 2
13 : iload 1
14 : ireturn
```

JVM model:

- **counter:** 5
- **stack:**
  
  
  | 1 |   |   | ... |

- **local variables:**

  | 5 | 1 |   | ... |
Java code:

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

**Bytecode:**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td><code>icnst 1</code></td>
</tr>
<tr>
<td>1</td>
<td><code>istore 1</code></td>
</tr>
<tr>
<td>2</td>
<td><code>iload 0</code></td>
</tr>
<tr>
<td>3</td>
<td><code>ifeq 13</code></td>
</tr>
<tr>
<td>4</td>
<td><code>iload 1</code></td>
</tr>
<tr>
<td>5</td>
<td><code>iload 0</code></td>
</tr>
<tr>
<td>6</td>
<td><code>imul</code></td>
</tr>
<tr>
<td>7</td>
<td><code>istore 1</code></td>
</tr>
<tr>
<td>8</td>
<td><code>iload 0</code></td>
</tr>
<tr>
<td>9</td>
<td><code>icnst 1</code></td>
</tr>
<tr>
<td>10</td>
<td><code>isub</code></td>
</tr>
<tr>
<td>11</td>
<td><code>istore 0</code></td>
</tr>
<tr>
<td>12</td>
<td><code>goto 2</code></td>
</tr>
<tr>
<td>13</td>
<td><code>iload 1</code></td>
</tr>
<tr>
<td>14</td>
<td><code>ireturn</code></td>
</tr>
</tbody>
</table>

**JVM model:**

- **counter:** 6
- **stack:**
  - 5 1 ...
- **local variables:**
  - 5 1 ...
Running example

java code:

```java
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:

```java
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] [ ] ... 
  ```

**Goal (Factorial case)**

∀n ∈ N, running the bytecode associated with the factorial program with n as input produces a state which contains n! on top of the stack.
Running example

Java code:

```java
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  __  ...
```

Goal (Factorial case)

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

Java code:

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

Bytecode:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td><code>iconst 1</code></td>
</tr>
<tr>
<td>1</td>
<td><code>istore 1</code></td>
</tr>
<tr>
<td>2</td>
<td><code>iload 0</code></td>
</tr>
<tr>
<td>3</td>
<td><code>ifeq 13</code></td>
</tr>
<tr>
<td>4</td>
<td><code>iload 1</code></td>
</tr>
<tr>
<td>5</td>
<td><code>iload 0</code></td>
</tr>
<tr>
<td>6</td>
<td><code>imul</code></td>
</tr>
<tr>
<td>7</td>
<td><code>istore 1</code></td>
</tr>
<tr>
<td>8</td>
<td><code>iload 0</code></td>
</tr>
<tr>
<td>9</td>
<td><code>iconst 1</code></td>
</tr>
<tr>
<td>10</td>
<td><code>isub</code></td>
</tr>
<tr>
<td>11</td>
<td><code>istore 0</code></td>
</tr>
<tr>
<td>12</td>
<td><code>goto 2</code></td>
</tr>
<tr>
<td>13</td>
<td><code>iload 1</code></td>
</tr>
<tr>
<td>14</td>
<td><code>ireturn</code></td>
</tr>
</tbody>
</table>

JVM model:

counter: 10

stack: 1 5 ... 

local variables: 5 5 ...
## Running example

Java code:

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

Bytecode:

<table>
<thead>
<tr>
<th>No.</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td><code>iconst 1</code></td>
</tr>
<tr>
<td>1</td>
<td><code>istore 1</code></td>
</tr>
<tr>
<td>2</td>
<td><code>iload 0</code></td>
</tr>
<tr>
<td>3</td>
<td><code>ifeq 13</code></td>
</tr>
<tr>
<td>4</td>
<td><code>iload 1</code></td>
</tr>
<tr>
<td>5</td>
<td><code>iload 0</code></td>
</tr>
<tr>
<td>6</td>
<td><code>imul</code></td>
</tr>
<tr>
<td>7</td>
<td><code>istore 1</code></td>
</tr>
<tr>
<td>8</td>
<td><code>iload 0</code></td>
</tr>
<tr>
<td>9</td>
<td><code>iconst 1</code></td>
</tr>
<tr>
<td>10</td>
<td><code>isub</code></td>
</tr>
<tr>
<td>11</td>
<td><code>istore 0</code></td>
</tr>
<tr>
<td>12</td>
<td><code>goto 2</code></td>
</tr>
<tr>
<td>13</td>
<td><code>iload 1</code></td>
</tr>
<tr>
<td>14</td>
<td><code>ireturn</code></td>
</tr>
</tbody>
</table>

JVM model:

- **counter:** 11
- **stack:**
  ```
  4 [ ] [ ] ...
  ```
- **local variables:**
  ```
  5 5 [ ] ...
  ```
Running example

Java code:

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

Bytecode:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>iconst 1</td>
</tr>
<tr>
<td>1</td>
<td>istore 1</td>
</tr>
<tr>
<td>2</td>
<td>iload 0</td>
</tr>
<tr>
<td>3</td>
<td>ifeq 13</td>
</tr>
<tr>
<td>4</td>
<td>iload 1</td>
</tr>
<tr>
<td>5</td>
<td>iload 0</td>
</tr>
<tr>
<td>6</td>
<td>imul</td>
</tr>
<tr>
<td>7</td>
<td>istore 1</td>
</tr>
<tr>
<td>8</td>
<td>iload 0</td>
</tr>
<tr>
<td>9</td>
<td>iconst 1</td>
</tr>
<tr>
<td>10</td>
<td>isub</td>
</tr>
<tr>
<td>11</td>
<td>istore 0</td>
</tr>
<tr>
<td>12</td>
<td>goto 2</td>
</tr>
<tr>
<td>13</td>
<td>iload 1</td>
</tr>
<tr>
<td>14</td>
<td>ireturn</td>
</tr>
</tbody>
</table>

JVM model:

- **counter:** 12
- **stack:** 
  - 
  - 
  - 
- **local variables:**
  - 4
  - 5
  - 
  - 
  - 

Goal (Factorial case):

∀ n ∈ N, running the bytecode associated with the factorial program with n as input produces a state which contains n! on top of the stack.
Running example

Java code:

```java
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:
```
... 1 2 3 ...
```

local variables:
```
4 5 ...
```
Running example

Java code:

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

Bytecode:

...  

JVM model:

...  

Goal (Factorial case)

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

Java code:

```java
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:

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

Bytecode:

```plaintext
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:

```java
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 \[ \ldots \]
- **local variables:**
  - 0 \[ 120 \[ \ldots \] ]
Motivation

Running example

Java code:

```java
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)**

∀n ∈ ℤ, running the bytecode associated with the factorial program with n as input produces a state which contains n! on top of the stack.
Outline

1 Motivation

2 Proof pattern recognition in ATPs

3 Proof pattern recognition in ITPs

4 Conclusions
Proof pattern recognition in ATPs

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

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

First-order fragments of: Mizar, HOL, etc.

Supervised Learning: SVMs, Naive Bayesian

Automated proof: Vampire, CVC3, etc.
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:

\[ \text{factorial } n = \text{factorial}_{\text{tail}} n \]

\[ C_{mulnA} \]

[0, 1]
Proof pattern recognition in ATPs

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.

First-order fragments of: Mizar, HOL, etc.

Supervised Learning: SVMs, Naive Bayesian

proof reconstruction

Automated proof: Vampire, CVC3, etc.
Proof pattern recognition in ATPs

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.

First-order fragments of:
Mizar, HOL, etc.

Supervised Learning:
SVMs, Naive Bayesian

Automated proof:
Vampire, CVC3, etc.

feature extraction
proof reconstruction

premise hierarchy
Main improvement:

- the number of goals proven automatically increases by up to 20% – 40%
In ITPs, the proof steps are suggested by the user who guides the prover by providing the tactics.
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.

A proof in Coq/SSReflect

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

1 subgoals, subgoal 1 (ID 13)

================================
forall n a : nat, fact_tail_aux n a = a * n'!
Lemma fact_tail_aux_lemma : forall (a n : nat), fact_tail_aux n a = a * n′!.

Proof.
move => n.

1 subgoals, subgoal 1 (ID 14)

n : nat
=================================
forall a : nat, fact_tail_aux n a = a * n′!
A proof in Coq/SSReflect

Proof pattern recognition in ITPs

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 /=].
A proof in Coq/SSReflect

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.

1 subgoals, subgoal 1 (ID 28)

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)′!
ML4PG assists the user providing similar lemmas as proof hints.
Lemma fact_tail_aux_lemma : forall (a n : nat), fact_tail_aux n a = a * n'.
Proof.
Lemma \texttt{fact\_tail\_aux\_lemma} : \texttt{forall} (a n : nat), fact\_tail\_aux n a = a * n'.

Proof.
\texttt{move} => n.
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 /=].

<table>
<thead>
<tr>
<th></th>
<th>tactics</th>
<th>N tactics</th>
<th>arg type</th>
<th>tactic arg is hypothesis?</th>
<th>top symbol</th>
<th>subgoals</th>
</tr>
</thead>
<tbody>
<tr>
<td>g1</td>
<td>move</td>
<td>1</td>
<td>nat</td>
<td>no</td>
<td>forall</td>
<td>1</td>
</tr>
<tr>
<td>g2</td>
<td>elim, move</td>
<td>2</td>
<td>nat, [nat</td>
<td>nat Prop nat]</td>
<td>yes</td>
<td>forall</td>
</tr>
<tr>
<td>g3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>g4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>g5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Lemma \texttt{fact\_tail\_aux\_lemma} : \texttt{forall} (a n : \texttt{nat}), \texttt{fact\_tail\_aux n a = a} * \texttt{n'}!.

Proof.
move => n. elim : n => [a| n IH a /=].
by rewrite /\texttt{theta\_fact} \texttt{fact0 muln1}.
Proof pattern recognition in ITPs

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**

Proof pattern recognition in ITPs

Lemma `fact_tail_aux_lemma`: \( \forall (a \ n : \text{nat}), \text{fact}_\text{tail}_\text{aux} \ n \ a = a * n'! \).

Proof.

\[ \text{move} \Rightarrow n. \ \text{elim} : n \Rightarrow [a| n \ \text{IH} \ a /=]. \]

\[ \text{by rewrite } /\theta_/\text{fact}\ \text{fact0} \ \text{mulp}1. \]

This lemma is similar to lemmas:

- `mult_tail_aux_lemma`
- `power_tail_aux_lemma`
- `expt_tail_aux_lemma`
Outline

1 Motivation

2 Proof pattern recognition in ATPs

3 Proof pattern recognition in ITPs

4 Conclusions
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