Efficiently Encoding Meta-Interpretive Learning by Answer Set Programming (Work in Progress)

ILP 2018, Ferrara, Italy

Tobias Kaminski, Thomas Eiter and Katsumi Inoue

{kaminski,eiter}@kr.tuwien.ac.at, inoue@nii.ac.jp

September 3, 2018
Meta-Interpretive Learning [Muggleton et al., 2015]

MIL is an elegant yet powerful approach

- Efficient realization of *Predicate Invention*
  - *Essential feature* for explicitly learning “hidden” concepts
  - Hard problem due to *high combinatorial complexity*

- Enables learning of *recursive rules and programs*

- Meta-rules *constrain search space* effectively
Setting of MIL

Example (MIL-Problem)

Meta-rules $\mathcal{R}$:

$\triangleright \; P(x, y) \leftarrow Q(x, z), R(z, y)$ (chain-rule)

$\triangleright \; P(x, y) \leftarrow Q(x, y), R(y)$ (postcon-rule)

Background knowledge $B$:

$\triangleright \; \text{remove}([X|R], R) \leftarrow \cdot$ (drops the first element from a list)

$\triangleright \; \text{empty}([[]]) \leftarrow \cdot$ (checks if a list is empty)

Positive and negative examples:

$\triangleright \; E^+ = \{p([a, a], []), p([b, b], [])\}$

$\triangleright \; E^- = \{p([a, a, a], [a]), p([b, b, b], [])\}$

Possible solution:

$p(x, y) \leftarrow \text{remove}(x, z), p1(z, y)$.

$p1(x, y) \leftarrow \text{remove}(x, y), \text{empty}(y)$. 
Metagol [Cropper and Muggleton, 2016]

MIL-solver based on classical Prolog meta-interpreter

Example (Induction Based on Positive Examples)

\[
\text{prove}([\text{PExample}|R], BK, BK_H) :- \\
\quad \text{metarule(Name,MetaSub,(PExample :- Body))}, \\
\quad \text{save_subst(metasub(Name,MetaSub),BK,BK_Mod)}, \\
\quad \text{prove(Body,BK_Mod,BK_H)}. 
\]

- Efficient by exploiting \textit{query-driven} mechanism
- Negative examples trigger backtracking \textit{only in the end}
Motivation: Using Answer Set Programming for MIL

ASP well-suited as MIL is essentially *combinatorial search problem*

1. **Purely declarative formalism**  
   - High *flexibility* and modularity  
   - Non-termination is *not an issue*

2. **Performance gains**  
   - Leverage *efficient* ASP solvers  
   - *Conflict propagation* and *learning*

3. **Optimization capabilities**  
   - Finding *minimal* solutions  
   - Handling *noisy* data
Our Approach

Challenge 1: Grounding explodes due to background knowledge!

- HEX extends ASP by external atoms $\& g[p](c)$ [Eiter et al., 2008]
- Only import relevant background knowledge (assuming forward-chained meta-rules)

Challenge 2: Straightforward encoding yields huge search space!
Our Approach

**MIL-Problem**

**ASP Program**

**ASP Solver**

**Hypothesis**

**Challenge 1:** Grounding explodes due to background knowledge!

- HEX extends ASP by *external atoms &g[\vec{p}](\vec{c})* [Eiter et al., 2008]
- Only import *relevant* background knowledge (assuming forward-chained meta-rules)

**Challenge 2:** Straightforward encoding yields huge search space!
Our Approach

**Challenge 1**: Grounding explodes due to background knowledge!

- HEX extends ASP by *external atoms* $\&g[p](c)$ [Eiter et al., 2008]
- Only import *relevant* background knowledge (assuming forward-chained meta-rules)

**Challenge 2**: Straightforward encoding yields huge search space!
Our Approach

**Challenge 1**: Grounding explodes due to background knowledge!

- HEX extends ASP by *external atoms* $\&g[p](c)$ [Eiter *et al.*, 2008]
- Only import *relevant* background knowledge (assuming forward-chained meta-rules)

**Challenge 2**: Straightforward encoding yields huge search space!
Example (Guess-and-check encoding)

% import
unary_bg(empty,X) :- empty[ded](X).
binary_bg(remove,X,Y) :- remove[ded](X,Y).

% guess
{ meta(post,P1,P2,P3) } :- sig(P1), sig(P2), sig(P3).
{ meta(chain,P1,P2,P3) } :- sig(P1), sig(P2), sig(P3).

% derive
ded(P,X,Y) :- binary_bg(P,X,Y).
ded(P1,X,Y) :- meta(post,P1,P2,P3), ded(P2,X,Y), unary_bg(P3,Y).
ded(P1,X,Y) :- meta(chain,P1,P2,P3), ded(P2,X,Z), ded(P3,Z,Y).

% check
:- pos_ex(P,X,Y), not ded(P,X,Y).
:- neg_ex(P,X,Y), ded(P,X,Y).

Extremely many possible combinations of meta-substitutions!
Basic Encoding

Example (Guess-and-check encoding)

% import
unary_bg(empty,X) :- &empty[ded](X).
binary_bg(remove,X,Y) :- &remove[ded](X,Y).

% guess
{ meta(post,P1,P2,P3) } :- sig(P1), sig(P2), sig(P3).
{ meta(chain,P1,P2,P3) } :- sig(P1), sig(P2), sig(P3).

% derive
ded(P,X,Y) :- binary_bg(P,X,Y).
ded(P1,X,Y) :- meta(post,P1,P2,P3), ded(P2,X,Y), unary_bg(P3,Y).
ded(P1,X,Y) :- meta(chain,P1,P2,P3), ded(P2,X,Z), ded(P3,Z,Y).

% check
:- pos_ex(P,X,Y), not ded(P,X,Y).
:- neg_ex(P,X,Y), ded(P,X,Y).

Extremely many possible combinations of meta-substitutions!
Interleave *guessing* and *derive* part ⇒ avoid many redundant rules

### Example (Restricting Guesses)

```prolog
% import
unary_bg(empty, X) :- &empty[ded](X).
binary_bg(remove, X, Y) :- &remove[ded](X,Y).

% guess
{ meta(post, P1, P2, P3) } :- sig(P1), ded(P2, X, Y), unary_bg(P3, Y).
{ meta(chain, P1, P2, P3) } :- sig(P1), ded(P2, X, Z), ded(P3, Z, Y).

% derive
ded(P, X, Y) :- binary_bg(P, X, Y).
ded(P1, X, Y) :- meta(post, P1, P2, P3), ded(P2, X, Y), unary_bg(P3, Y).
ded(P1, X, Y) :- meta(chain, P1, P2, P3), ded(P2, X, Z), ded(P3, Z, Y).

% check
:- pos_ex(P, X, Y), not ded(P, X, Y).
:- neg_ex(P, X, Y), ded(P, X, Y).
```

Still guesses rules not needed for deriving any positive example!
Interleave **guessing** and **derive** part ⇒ avoid many redundant rules

Example (Restricting Guesses)

```
% import
unary_bg(empty,X) :- &empty[ded](X).
binary_bg(remove,X,Y) :- &remove[ded](X,Y).

% guess
{ meta(post,P1,P2,P3) } :- sig(P1), ded(P2,X,Y), unary_bg(P3,Y).
{ meta(chain,P1,P2,P3) } :- sig(P1), ded(P2,X,Z), ded(P3,Z,Y).

% derive
ded(P,X,Y) :- binary_bg(P,X,Y).
ded(P1,X,Y) :- meta(post,P1,P2,P3), ded(P2,X,Y), unary_bg(P3,Y).
ded(P1,X,Y) :- meta(chain,P1,P2,P3), ded(P2,X,Z), ded(P3,Z,Y).

% check
:- pos_ex(P,X,Y), not ded(P,X,Y).
:- neg_ex(P,X,Y), ded(P,X,Y).
```

Still guesses rules not needed for deriving any positive example!
Novel Top-Down Encoding

- **Goal-driven** guessing of rules, starting with positive examples

### Example (Restricting guesses to relevant meta-rules)

% ... import, derive and check part as before

% remove guess for meta-substitutions!

% new sub-goals
  goal(P,X,Y) :- pos_ex(P1,X,Y).
  goal(Q,X,Y) :- ded_u(post,P,Q,R,X,Y,\_).
  goal(Q,X,Z) :- ded_u(chain,P,Q,R,X,Y,Z).
  goal(R,Z,Y) :- ded_u(chain,P,Q,R,X,Y,Z).

% one rule for each sub-goal
  \{ded_u(post,P,Q,R,X,Y,\_) : ord(P,Q), unaryBg(R,Y);
   ded_u(chain,P,Q,R,X,Y,Z) : ord(P,Q), ord(P,R), state(Z)\}=1 :- goal(P,X,Y).

% collect meta-substitutions
  meta(M,P,Q,R) :- ded_u(M,P,Q,R,X,Y,Z), M != bg.
Novel Top-Down Encoding

- *Goal-driven* guessing of rules, starting with positive examples

Example (Restricting guesses to relevant meta-rules)

```prolog
% all possible rule instances
  ded_a(bg,P,n,n,X,Y,n) :- binary_bg(P,X,Y).
  ded_a(post,P,Q,R,X,Y,n) :- ord(P,Q),ded_a(_,Q,_,_,X,Y,_),unary_bg(R,Y).
  ded_a(chain,P,Q,R,X,Y,Z) :- ord(P,Q),ord(P,R),ded_a(_,Q,_,_,X,Z,_),
                             ded_a(_,R,_,_,Z,Y,_).
% new sub-goals
  goal(P,X,Y) :- pos_ex(P1,X,Y).
  goal(Q,X,Y) :- ded_u(post,P,Q,R,X,Y,_).
  goal(Q,X,Z) :- ded_u(chain,P,Q,R,X,Y,Z).
  goal(R,Z,Y) :- ded_u(chain,P,Q,R,X,Y,Z).
% one rule for each sub-goal
% collect meta-substitutions
  meta(M,P,Q,R) :- ded_u(M,P,Q,R,X,Y,Z), M != bg.
```
First Experiments

- **Average runtimes** in sec. over 20/10 instances, timeout 600 s
- Use *hexlite* solver with *clingo* as backend

---

**String Transformation (ST)**

- Metagol
- Bottom-Up
- Top-Down

**East-West Trains (EW)**

- Metagol
- Bottom-Up
- Top-Down
Conclusion

- Novel **HEX-encodings** of Meta-Interpretive Learning
  - Speedup due to efficient propagation of negative examples
  - Grounding feasible due to limited import of BK
- **State abstraction** further reduces grounding [ICLP’18]
- Techniques possibly **applicable** to other approaches (e.g. *ILASP*)?
- Future work:
  - **Formalize** top-down encoding, combine with state abstraction
  - **Solver-implementation** based on **HEX-encodings**
Thank You for Your Attention!
References I


