Higher-order Logic Learning

Niels Pahlavi and Stephen Muggleton

Department of Computing, Imperial College London,
180 Queen's Gate, London SW7 2BZ, UK
{namdp05,shm}@doc.ic.ac.uk

Abstract. This paper presents Higher-order Logic Learning (HOLL), which consists of generalizing logic-based Machine Learning, and more particularly Inductive Logic Programming (ILP), from the first-order to the higher-order context. The power of expressivity of Higher-order Logic (HOL) is used to improve significantly the learning capacity and efficiency of logic-based Machine Learning – by both allowing to learn new tasks but also to perform better than first-order Machine Learning systems on some already learnable problems. We describe a Higher-order ILP system, called λProgol, adapting the ILP system Progol and based on the HOL formalism λProlog, along with its first implementation and promising results on motivating worked examples. We intend to extend the implementation, tests and evaluation of λProgol further, and to develop a theory of HOLL.

1 Introduction

Much of logic-based Machine Learning research is based on First-order Logic (FOL) and Prolog, including Inductive Logic Programming (ILP). Yet, Higher-order Logic (HOL), which allows for quantification over predicates and functions, is intrinsically more expressive than FOL and has been seldom used. According to [6], "the logic programming community needs to make greater use of the power of higher-order features and the related type systems. Furthermore, HOL has generally been under-exploited as a knowledge representation language". In [6], the use of HOL in Computational Logic, which has been "advocated for at least the last 30 years" is illustrated: functional languages, like Haskell98 [5]; Higher-order programming introduced with λProlog [8]; integrated functional logic programming languages like Curry or Escher; or the higher-order logic interactive theorem proving environment "HOL".

As we were interested in discovering learning problems for ILP, we decided to try to adapt ILP within a HOL framework, to develop Higher-order Logic Learning (HOLL). ILP seems to be rather intuitively adaptable to a FOL formalism. It is natural when considering HOLL to both develop a theory and to implement a higher-order ILP system and to test and evaluate it. We decided to choose Higher-order Horn Clauses (HOHC) [10] as a HOL formalism, since it is one of the logical foundations of λProlog. As a ILP system, we chose to adapt Progol [9], which is a popular and efficient implementation.

Presented at the 19th International Conference on Inductive Logic Programming (ILP'09).

H0LL will be tested and assessed on new problems and applications not learnable by ILP (which includes the learning of higher-order predicates), but also on how well it performs on problems already handled by ILP to compare it with existing ILP systems. It would be therefore of interest to look at learning problems not handled well by ILP. One of these is learning tasks involving recursion. According to [7], "learning first-order recursive theories is a difficult learning task" in a normal ILP setting. However, we can expect a higher-order system to learn better than a first-order system on such problems, because we could use higher-order predicates as background knowledge to learn recursive theories (a similar approach is already used for some recursive functions in the Haskell prelude [5]); and it will be sounder, more natural and intuitive, hence probably more efficient than meta-logical features which come from functional languages. More generally, the expressivity of HOL would make it possible to represent mathematical properties like transitivity or reflexivity, which would allow to handle equational reasoning and functions within a logic-based framework.

2 λ-Prolog and Higher-order Horn Clauses

λProlog is a higher-order logic programming language handling polymorphic typing, scoping over names and procedures, modular programming, abstract data types, the use of lambda terms as data structures and, more importantly for this paper, higher-order programming illustrated by the predicates *mappred*, *trans* and $foreach$ in Exs. 1 and 2.

It is based on HOHC, which are "a generalization of Horn clauses to a higher-order logic" and a "basis for logic programming". According to [10], HOHC can be "characterized as those obtained from first-order goal formulas and definite sentences by supplanting first-order terms with the terms of a typed λ-calculus and by permitting quantification over function and predicate symbols".

In [10], a theorem proving procedure for HOHC is outlined and its soundness is proved; this result is essential for λProgol allowing to implement a HOL Prolog interpreter (see Sect. 3). This is based on Huet's semi-decision algorithm to search for unification in typed λ-calculus [4].

3 λProgol: a Higher-order ILP System

In this section, λProgol, a higher-order ILP formalism is presented. It is based upon Progol and Mode-Directed Inverse Entailment as defined in [9]. However, it generalizes this approach on HOHC and λProlog. Since our implementation is in Prolog, a λProlog interpreter in Prolog is needed. The main differences in the λProgol algorithm, compared to Progol, come from this interpreter and from the fact that it requires background knowledge and examples to be not Horn clauses but λProlog clauses.

Definition 1. λProlog Interpreter. *A λProlog clause is of the form* $(HeadAtom \Leftarrow [BodyAtom_1, \ldots, BodyAtom_n])$*, where HeadAtom has to be rigid. A λProlog formula is : (1) A variable or a constant, (2) (X/F), where F is a formula, (3)*

(F1@F2) where F1 and F2 are formulae, (4) (sigma F), where F is a formula, (5) (pi F), where F is a formula. sigma and pi represent respectively the existential and universal quantifiers. / represents abstraction and @ represents function in λ-calculus. A list is of the form $cons@el_1@\ldots@el_m@nil$*. nil is the empty list.*

Ex. 1 shows a λProgol input file to learn the higher-order predicate mappred, defined in [8], which "given a predicate of two arguments and two lists, checks that corresponding elements of these two lists are related by the given predicate", along with its bottom clause and clause learned by λProgol.

Example 1. Mappred.
modeh(*,mappred@+pred@+list@+list). modeb(*,+list=cons@-any@-list).
modeb(*,+pred@+any@+any). modeb(*,#pred@+pred@+list@+list).
Type declarations:
list(nil). list(cons@X@Y) :- list(Y).
any(X) :- person(X). any(X) :- integer(X).
pred(mappred). pred(age). person(bob). person(sue). person(ned).
Background Knowledge:
age@bob@23 <= []. age@sue@24 <= []. age@ned@23 <= [].
mappred@P@nil@nil <= [].
Positive Example:
mappred@age@(cons@ned@nil)@(cons@23@nil) <= [].
Result: Bottom Clause generated:
mappred@A@B@C<=[B=cons@D@E,C=cons@F@E,A@D@F,mappred@A@E@E].
Clause to be learned
mappred@A@B@C<=[B=cons@D@E,C=cons@F@G,A@D@F,mappred@A@E@G].

The following algorithms (Algs. 1, 2 and 3) which constitute the λProgol algorithm are very similar to the Progol algorithms. The mode declarations and mode language are identical to Progol (Definitions 20, 21, 22 in [9]) except that the mode atoms can be different because λProlog atoms are different from FOL atoms, as it can be seen in Ex. 1. The construction of $\bot_i$, which is the least general element of the bounded sub-lattice for each example e is described in Alg. 1. i represents the maximum variable depth determining how many times step 5 is executed; *Recall* determines how many times the λProlog interpreter is called for each instantiation of the clause in step 4. The line 5.a.i in the algorithm is specific to λProgol, it is to prevent the call of flexible atoms by the λProlog interpreter. Indeed, the type *pred* is set to correspond to higher-order predicate, which can be uninstantiated (i.e. still variable) when called by the λProlog interpreter. The call to $pred(u)$ instantiates these variables.

Algorithm 1. Construction of $\bot_i$.

1. Given natural numbers i, λProlog clauses B, λProlog clause e and set of mode declarations M.
2. Let $k = 0$, $hash : Terms \rightarrow N$ be a hash function which uniquely maps terms to natural numbers, $\overline{e}$ be $\overline{a} \wedge b_1 \wedge \ldots \wedge b_n$, $\bot_i = \langle\rangle$ and $InTerms = \emptyset$.
3. If there is no modeh in M such that $a(m) \preceq a$ then return the empty clause $\square$. Otherwise let m be the first modeh declaration in M such that m subsumes a with substitution θ_h. For each v/t in θ_h

(a) if v corresponds to a $\#type$ then replace v in m by t
(b) otherwise replace v in m by v_k where $k = hash(t)$ and
(c) add t to $InTerms$ if v corresponds to $+type$.

Add m to $\bot_i$.

4. If $k = i$ return $\bot_i$ else $k = k + 1$.
5. For each modeb m in M, let $\{v_1, \ldots, v_n\}$ be the variables of $+type$ in m and $T(m) = T_1 \times \ldots \times T_n$ be a set of n-tuples of terms such that each T_i corresponds to the set of all terms from $InTerms$ of the type associated with v_i in m (t is tested to be of a particular type by calling $type(t)$ with the λProlog interpreter).
   (a) For each $\langle t_1, \ldots, t_n \rangle$ in $T(m)$ and $\theta = \{v_1/t_1, \ldots, v_n/t_n\}$. Repeat recall times:
      i. for every variable u in $m\theta$ of type $pred$, add the call $pred(u)$ to the λProlog interpreter
      ii. if the λProlog interpreter succeeds on goal $m\theta$ with answer substitution θ'then for each v/t in θ and θ' if v corresponds to a $\#type$ then replace v in m by t otherwise replace v in m by v_k where $k = hash(t)$ and add t to $InTerms$ if v corresponds to $-type$. Add $\overline{m}$ to $\bot_i$.
6. Goto step 4.

The search for a single clause in the subsumption lattice is described in Alg. 2. $best(s)$, $prunes(s)$, $terminated(s)$,$\rho(s)$ are defined like in Progol to find a clause with maximal compression but other types of searches can be used like the search used in Aleph which is simpler.

Algorithm 2. Algorithm for searching $\square \preceq C \preceq \bot_i$.

1. Given λProlog clauses B, λProlog clause e, and $\bot_i$ obtained in Alg. 1.
2. Let $Open = \{\langle \square, \emptyset, 1 \rangle\}$ and $Closed = \emptyset$.
3. Let $s = best(Open)$ and $Open = Open - \{s\}$.
4. Let $Closed = Closed \cup \{s\}$.
5. If $prune(s)$ goto 7.
6. Let $Open = (Open \cup \rho(s)) - Closed$.
7. If $terminated(Closed, Open)$ then return $best(Closed)$.
8. If $Open = \emptyset$ then print "no compression" and return $\langle e, \emptyset, 1 \rangle$.
9. Goto 3.

Alg. 3 is a simple cover set algorithm similar to Alg. 44 in [9].

4 Results and Implementation

An implementation of λProgol has been made and is available. It has been tested successfully for Alg. 1 on several cases of learning higher-order predicates [8] as well as cases of learning first-order recursive theories with higher-order predicates as background knowledge, including Exs. 1,2,3,4 and 5. Table 1 gives a table of runtimes for learning the bottom clause for these examples.

Our first choice of implementation was based on λProlog but revealed to be too inconvenient and inefficient to use; instead the current implementation is in Prolog, which is more convenient and more efficient; a requirement is the use of a λProlog interpreter, which was implemented using a depth-first approach.

Ex. 2 details a practical example [7] showing the advantage of using HOL background knowledge in a simple learning problem involving recursion. It consists of learning the predicate *ancestor* given the predicate *parent* and the higher-order predicate *trans*, which "given a predicate of two arguments, constructs its transitive closure".

Example 2. Ancestor.
modeh(*,ancestor@+person@+person).
modeb(*,#@#pred@+person@+person).
Type declarations:
pred(trans). pred(parent). pred(married). pred(brother).
person(john). person(jim). person(jane). person(bob). person(james). person(bill).
Background Knowledge:
trans@R@X@Y <= [R@X@Y]. trans@R@X@Z <= [R@X@Y,trans@R@Y@Z].
parent@john@jim <= []. parent@john@jane <= []. parent@jim@bob <= [].
parent@bob@james <= []. married@john@jane <= []. brother@bob@bill <= [].
Positive Examples:
ancestor@john@bob <= []. ancestor@jim@james <= [].

In this example, both the bottom clause and the clause to be learned are:
ancestor@X@Y ⇐ [trans@parent@X@Y].
By closed world assumption, we can infer the equivalence and by unfolding.
ancestor@X@Y ⇐ [parent@X@Y].
ancestor@X@Y ⇐ [parent@X@Z, trans@parent@Z@Y].
Which gives the following first-order theory (with Prolog notations).
ancestor(X,Y) :- parent(X,Y).
ancestor(X,Y) :- parent(X,Z), ancestor(Z,Y).
This shows how natural and efficient it is to use HOL as background knowledge to handle the learning of a first-order recursive theory.

Table 1. Bottom Clause Results

Example	Predicate	Number of Mode Declarations	Number of Clauses of Background	Length of Bottom Clause	Time (sec)
1	mappred	4	13	5	0.004
2	ancestor	2	18	2	0.016
3	less_than	2	18	2	0.024
4	foreach	4	18	4	0.008
5	trans	3	21	3	0.004

Ex. 3 also uses *trans* and *successor*, which given an integer N returns $N+1$ as background knowledge to learn *less_than*, which given two integers determines if the first one is smaller than the second one. Ex. 4 learns the higher-order predicate $foreach$, [8], which "given a predicate of one argument and a list, checks that every element of that list satisfies that predicate". In Ex. 5, *trans* is not used as background knowledge but is the predicate to be learned.

5 Conclusion

There have been attempts to use HOL for logic-based Machine Learning such as by Harao starting in [3] or by Feng and Muggleton [1]. They provide different higher-order extensions of least general generalization in order to handle higher-order terms in a normal ILP setting, whereas we use λProlog, a HOL framework, as a logical foundation to extend first-order ILP to a higher-order context. John Lloyd deals with related issues [6]. It details a learning system, called $ALKEMY$. A main difference is that Lloyd's approach is not based on Logic Programming and therefore on ILP. According to Flach's review of the book [2], "it is almost a rational reconstruction of what ILP could have been, had it used Escher-style higher-order logic rather than Prolog"; whereas we intend, through the use of HOHC to keep the Horn clauses foundations of LP and ILP and to extend them.

We intend to present theoretical results for HOLL. ILP theory seems to be rather intuitively adaptable within a HOL framework. For λProgol, we will have to prove that higher-order inverse entailment is possible and to generalize correctness and complexity results for the Progol Bottom Clause and Search algorithms. In [11], a model-theoretic semantics for HOHC is provided. We will also extend the implementation of λProgol and test and evaluate it further.

We also aim to compare λProgol with already existing ILP systems, for example by considering learning tasks where it could perform better than Progol. Then, we intend to investigate tasks and discoveries not learnable by first-order ILP. It could be of interest to look at recursion, of course, but also at HOL theorem provers, or integrated functional logic programming languages. Further objectives may be to investigate abduction, introduce Probability, generalize Probabilistic Logic Learning, look at applications such as Atomic Theory, Synthetic Biology or Bioinformatics, where ILP has been successfully applied and consider other logics within λProlog.

References

1. C. Feng and S.H. Muggleton. Towards inductive generalisation in higher order logic. In *Proc. Ninth Int. Work. on Machine Learning*, pages 154–162, 1992.
2. P. Flach. Book review: Logic for Learning. Available at http://www.cs.kuleuven.ac.be/ dtai/projects/alp/tplp/reviews/files/, 2003.
3. M. Harao. Analogical reasoning based on higher-order unification. In *ALT*, pages 151–163, 1990.
4. G. Huet. A unification algorithm for typed λcalculus. *Theor. Comp. Sci.*, 1975.
5. S. Peyton Jones and J. Hughes. Haskell98: A non-strict purely functional language. Available at http://haskell.org/.
6. J.W. Lloyd. *Logic for Learning.* Springer, Berlin, 2003.
7. D. Malerba. Learning recursive theories in the normal ILP setting. *Fundam. Inform.*, 57(1):39–77, 2003.
8. Dale Miller. *λProlog: An Introduction to the Language and its Logic.* 1998.
9. S.H. Muggleton. Inverse entailment and Progol. *New Generation Computing*, 13:245–286, 1995.
10. G. Nadathur and D. Miller. Higher-order Horn Clauses. *Journal of the ACM*, 1990.
11. D. A. Wolfram. A semantics for λProlog. *Theor. Comp. Sci.*, pages 277–289, 1994.