Inductive Object-Oriented Logic Programming

Erivan Alves de Andrade and Jacques Robin

Centro de Informática, Universidade Federal de Pernambuco, Recife, Brazil
{eaa,jr}@cin.ufpe.br

Abstract. In many of its practical applications, such as natural language processing, automatic programming, expert systems, semantic web ontologies and knowledge discovery in databases, Inductive Logic Programming (ILP) is not used to substitute but rather to complement manual knowledge acquisition.This manual acquisition is increasingly done using hybrid languages integrating objects with rules or relations. Since using a common representation language for both manually encoded and ILP learned knowledge is key to their seamless integration, this raises the issue of using such hybrid languages for induction. In this paper, we present Cigolf, an ILP system that uses the object-oriented logic language Flora for knowledge representation. Cigolf takes as input a background knowledge base, a set of examples, and a learning bias specification, all represented in Flora. It translates this object-oriented input into a relational input specification for the ILP system Aleph. It then uses a tabled Prolog version of Aleph to induce new knowledge and translates back this learned knowledge into Flora. We describe the issues raised by this bi-directional translation process and the solution we adopted. We also compare the respective performance of Cigolf and Aleph on a few ILP benchmarks to assess the overhead associated with using an object-oriented logic representation language instead of a purely logic one for learning tasks.

1 Introduction

Inductive Logic Programming (ILP) stands out among machine learning techniques by virtue of the following distinctive characteristics:

1. Ability to learn *generic relations* among *several* domain entity classes, each one represented by a distinct universally quantified variable, overcoming the limitation of attribute-based learning to the properties of *a single* domain entity class;
2. Ability to learn from both examples *and intentional knowledge*, as opposed to only from examples;
3. Ability to learn *recursive definitions*.

These three unique features have made ILP a leading contender for the following machine learning applications:

1. Automatic programming in CASE [6];

2. Knowledge acquisition for Natural Language Processing (NLP) systems, expert systems in deterministic yet structurally complex domains and semantic web ontologies [5];
3. Mining multiple table databases, as opposed to simplistic single table transaction databases [4].

For all these applications, while complete automation remains often unrealistic, consolidated manual knowledge acquisition techniques, and in some cases partially reusable knowledge bases are available. Thus, the ability of ILP to complete, from data, prior intentional knowledge that was manually modeled is crucial. In addition, much of the knowledge needed by these applications is relational in nature, reinforcing the special adequacy of ILP. Finally, both automatic programming and NLP knowledge acquisition require learning recursive rules.

Recently, these key application niches of ILP are one by one making the switch to object-oriented or hybrid object-rule representation languages for the manually encoded part of their knowledge base. Object-Orientation (OO) has become the leading paradigm for modeling and programming languages in modern software engineering. Similarly, the most widely used languages in symbolic NLP are based on typed feature structures [12] that incorporate key OO concepts such as complex composite structures with type signatures and inheritance. Modern expert systems in deterministic[1] domains usually represent knowledge in a formalism that integrates rules with an OO language such as Java, C++, or a description logic. Semantic web ontology languages are also evolving towards a hybrid, classes plus rules language. Finally, the SQL standard and the main DBMS vendors have recently switched from the purely relational data model to the object-relational one.

An impedance mismatch is thus slowly creeping between, on the one hand, the OO and hybrid languages used for manual knowledge acquisition in the niche applications of ILP, and on the other hand, the purely rule and relation oriented language used by available ILP systems. This mismatch hinders ILP potential for seamlessly integrating machine learning with manual modeling in comprehensive knowledge, data and software engineering workbenches.

In this paper, we suggest that to maintain its edge in its niche applications, ILP should consider to follow the language paradigm shift that occurred within these applications. We show the feasibility and practicality of Inductive Object-Oriented Logic Programming (IOOLP) by presenting Cigolf[2], an implemented system able to learn a first-order theory from examples, background knowledge and a bias specification, all four represented in the hybrid OO Horn logic language Flora [16], a Frame Logic [8] dialect.

---

[1] In non-deterministic domains, Bayesian and decision networks have become the dominant representation scheme.

[2] F-Logic in reverse

2 Outline of the approach

When we started working on IOOLP, our aim was to be able to empirically evaluate its feasibility and practicality as early as possible. Our key idea for fast prototyping Cigolf was to reuse an existing OOLP language together with an existing ILP engine and focus on bi-directional translation between the OOLP language and the standard LP language used by the ILP engine.

Within this framework, our next design step was to choose both the ILP engine and the OOLP language to be reused. For the engine choice, we used the following criteria: (1) versatility of the ILP task classes that it can carry out, (2) proven effectiveness in practical, real world domains and (3) declarative implementation that facilitates modifications. For the language choice, we used following criteria: (1) covering of OO concepts, (2) expressive power, (3) well-studied formal semantics and associated deductive inference mechanism properties, (4) proven effectiveness in diverse practical applications, and (5) proximity from the chosen ILP engine language.

These criteria led us to base Cigolf on the ILP engine Aleph[15] and on the OOLP language Flora. We review their main characteristics in turn in what follows, emphasizing those that make them fit our criteria.

2.1 Aleph

Among the freely available ILP systems, Aleph was the only one to meet our three criteria to serve as the inductive engine of Cigolf. First, Aleph is probably the most versatile ILP engine for the following reasons:

- In contrast to most ILP systems that implement a single induction algorithm, Aleph implements a variety of them, which turns it into a comprehensive workbench able to partially emulate[3] the functionalities of six other systems: Progol, FOIL, FORS, MIDOS, Tilde and WARMR;
- It consequently supports a variety of supervised and unsupervised relational learning tasks including classification, numerical regression, outlier analysis, clustering and association pattern discovery;
- It supports learning from positive examples only in addition to standard learning from both positive and negative examples;
- It can learn single or multiple predicate definitions;
- It can learn autonomously or interactively;
- Its knowledge representation language is full pure Prolog, including recursive rules and function symbols, whereas most ILP systems ban one or the other;
- It is easily extensible and customizable by allowing user defined hypothesis refinement operators, evaluation functions and other search heuristics;
- It provides over 50 parameters for setting and tuning the learning task.

[3] Naturally, these Aleph emulations tend to be less efficient and scalable than the corresponding specialized engines.

Second, it has been successfully used in various applications, including molecular biology, drug design and natural language grammar learning. Third, it is implemented mostly declaratively in the largely standard conformant YAP Prolog [3]

2.2 Flora

Flora (Frame LOgic tRAnslator) is a highly declarative, object-oriented, dynamic, high-order, tabled extension of Prolog with well-founded negation as failure.

Syntactically and semantically, the Flora language integrates three languages that extend Prolog in orthogonal yet synergetic ways: Frame Logic (F-Logic), Transaction Logic (TR) [1] and HiLog [2]. F-Logic is an object-oriented logic programming language that overcomes the limitations of purely relational languages such as Prolog, to elegantly model taxonomic knowledge as well as complex and semi-structured data. TR is a dynamic logic programming language that provides backtrackable, declarative update predicates to overcome the inability of Prolog to correctly represent, execute and reason about database updates, transactions and procedural knowledge, *within* its logical framework. HiLog extends Prolog with high-order syntactic sugar while semantically remaining first-order. It thus overcomes the inability of Prolog to express meta-level rules and queries declaratively, *within* its logical framework, and it does so without incurring the inference complexity blow up of semantically high-order languages. The Flora deductive engine currently available runs on top of the tabled Prolog engine XSB[14]. It consists of two main components: the Flora *compiler* that transforms a Flora program into a semantically equivalent XSB program, and the Flora *shell* that provides an interactive command and query run time layer on top of XSB.

Let us now evaluate Flora in terms of the criteria we defined above for the OOLP language to reuse in Cigolf. In terms of OO concepts, the Flora language inherits from F-Logic complex objects, object identity, encapsulation, type and class hierarchies with multiple inheritance, overriding, overloading and late binding. The Flora engine currently implements all these concepts except encapsulation. It also lacks automatic type checking. Integrating TR and HiLog with F-Logic, allows the Flora engine to implement two other key OO concepts declaratively, *within* its logical framework: state changing and reflection methods. While some other OOLP languages implement encapsulation and type checking, all of them rely on procedural extra-logical Prolog predicates to implement state change and reflection, thus not truly bringing these last two concepts within the OOLP paradigm.

In terms of expressive power, Flora stands at the high end of the scale among OOLP languages thanks to its integration of F-Logic, with TR, HiLog and the well-founded negation provided by the underlying XSB engine. The same is true with respect to its theoretical foundations: all three components of the Flora language, F-Logic, TR and HiLog possess a correct and refutation complete proof theory. In terms of practical applicability, Flora has been successfully used for disparate data integration, semantic web ontology engineering and security policy management.

To assess proximity to standard Prolog, one must first note that while many different approaches have been taken to integrate logic with objects, it is possible to

identify two main broad classes: (1) embedding logic inside objects, such as Pluto[10], and (2) embedding objects inside logic, such as Flora. Flora's embedding of objects inside logic makes it closer to standard Prolog. Overall, Flora thus fitted our five conceptual criteria much better than other OOLP languages. On the practical side, the fact the Flora engine is implemented by compiling Flora programs into Prolog programs allowed a large part of the bi-directional translation between Cigolf and Aleph learning task specifications and results, to be implemented in Cigolf by reusing and modifying the Flora compiler.

To give a feel for the structure and syntax of a Flora program and to illustrate how it can be used as an ILP language, we give in Prog.1 a partial Flora representation of the Bongard classification problem shown in Fig. 1. Every example is a set of geometric figures. The problem is to induce a set of generic spatial relations between figure classes that together differentiate the positive examples in lower board, from the negative ones in the upper board.

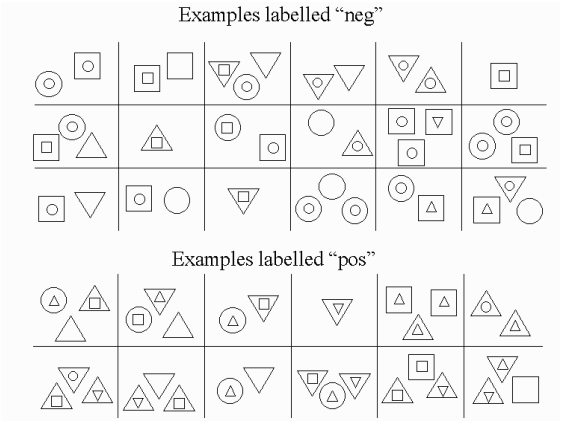

Fig. 1. A Bongard classification problem

```
% Background knowledge: class hierarchy together with type signatures
 (1)  circle::shape.
 (2)  square::shape.
.(3)  triangle::shape[direction*=>dir].
 (4)  shape[in*=>>shape,leftOf*=>>shape].
 (5)  bgEx[shapes*=>>shape].
% Example base: object creation together with attribute value assignments
% Only the positive example on line 1, column 4 is shown
 (6)  ta22:triangle[direction->down].
 (7)  tb22:triangle[in->>ta22,direction->down].
 (8)  ex22:bgEx[shapes->>{ta22,tb22}].
% Background knowledge: deductive rules
(9)   X:shape[in->>B]:-X[in->>C],C[in->>B].
(10)  X:shape[leftOf->>B]:-X[leftOf->>C],B[in->>C].
% Queries on backrgound knowledge and example base
```

```
(11) ?-X:shape[leftOf->>Y].
(12) ?-X[M->>Y].
```

Prog. 1 An example Flora program representing a simple Bongard problem.

A Flora program is made of four main types of clauses: two corresponding to its OO part, the class hierarchy definition facts and object creation facts corresponding, and two to its corresponding logical part.

A Flora class hierarchy fact follows the syntactic pattern: `class::superclass[`$\texttt{attr}_1$ $\texttt{typOp}_1$ $\texttt{type}_1$`,...,`$\texttt{attr}_N$ $\texttt{typOp}_N$ $\texttt{type}_N$`]` to specify the superclass of a class together with its proper attribute filler and method return type constraints. There are four typing operators in Flora: `*=>`, `*=>>`, `=>` and `=>>`. The presence or absence of the `*` prefix distinguishes between inheritable and non-inheritable type constraints, whereas the `>` and `>>` suffixes indicates whether the attribute is single valued or set valued.

Object creation facts follow the syntactic pattern `object:class[`$\texttt{attr}_1$ $\texttt{assignOp}_1$ $\texttt{value}_1$`,...,`$\texttt{attr}_N$ $\texttt{assignOp}_N$ $\texttt{value}_N$`]` to create a new instance of a class while assigning its proper attribute and method return values. There are four value assignment operators `*->`, `*->>`, `->` and `->>`, that follow the same prefix and suffix conventions than the typing operators.

Together, Flora class definition and object creation facts are called *F-Molecules*. Flora deductive rules and queries are essentially Prolog rules and queries in which logical terms may be substituted by F-molecules. Logical variables can appear in any position inside these molecules: as object name, class name, attribute name, method name, attribute value, method return value or method input parameter. This freedom provides Flora with a high-order syntax that is very powerful for concise meta-programming, as illustrated by the query on line 18 which asks for all the triplets `X, M, Y` such that `X` and `Y` are objects of any class and `M` is a set valued attribute of `X` that includes `Y` in its value set.

2.3 The architecture of Cigolf

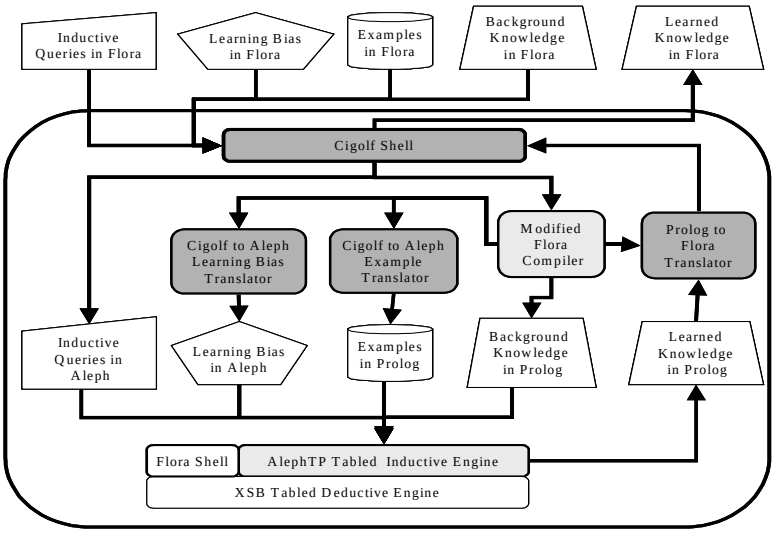

Fig. 2. The Cigolf architecture.

The architecture of Cigolf is given in Fig. 2. It consists of four entirely new components (shown in dark gray boxes), two modified reused components (shown in light gray boxes) and two components reused “as they are”(shown in a white box), namely the XSB tabled deductive engine and the Flora shell. The *Cigolf shell* accepts the Flora compiler commands together with a set inductive commands and queries equivalent to those of Aleph, but that work with learning problems encoded in OO syntax. The *Cigolf to Aleph learning bias translator* translates the object-oriented learning bias specification that we defined for Cigolf into an equivalent relational bias specification accepted as input by Aleph. The *Cigolf to Aleph example translator* translates the types contained in Flora objects that codify examples in Cigolf into equivalent Aleph typing predicates. The *Prolog to Flora translator* translates the best hypotheses returned by Aleph into Flora syntax. Due to Flora's high expressive flexibility, there are many different syntactical alternatives for this translation. In order to allow Cigolf to choose the alternative that conforms to the user specified input OO bias, this translator consists of a Prolog rewrite rule base that is generated at run time by our modified version of the Flora compiler. The *Flora compiler* was modified to (a) recognize Flora objects that are Cigolf learning bias specification and examples and dispatch them to the new components that handle them, and (b) generate the learned knowledge back translation rules. The *Aleph engine* was modified to run on top of the tabled deductive engine XSB instead of YAP Prolog, resulting in AlephTP (Aleph in Tabled Prolog). This port was necessary because, one the one hand, the Flora compiler generates code that requires tabling to execute properly, especially to terminate and implement the well-founded semantics for

negation as failure, and, on the other hand, Aleph relies on some non-standard features of YAP Prolog that are not available in XSB. To execute the Flora compiler generated code passed to XSB through AlephTP, the run-time predicates of the Flora shell must be loaded in XSB.

3 Object-oriented representation of ILP bias and examples

In Aleph, learning bias can be expressed in a variety of ways that include: induction parameter setting directives, mode facts, integrity constraints, and pruning directives. We now review how Cigolf codifies such bias in OO syntax.
The 50 induction parameters provided by AlephTP are encoded in Cigolf as attributes of the special class `settingILP`. They are set by Flora facts instantiating one or several objects of that class. For example, the fact below specifies that Cigolf must learn from positive only examples clauses with at most than five literals.

```
(13) _:settingILP[evalfn->posonly, clauselength->5].
```

Cigolf modes are specified using facts of the form `modeh(Recall,Ft)` or `modeb(Recall,Ft)` that indicate what terms can respectively appear in hypothetic rule heads and bodies. `Ft` is a Flora term with variables labeled with unification direction constraints. The labels are: `++` and `--` for input and output variables, and `##` for constants. `Recall` is an integer that indicates the maximum number of alternative ways that `Ft` can be instantiated when applying deduction on the hypothesized rule in the context of prior knowledge. For unlimited number of alternatives `Recall = '*'`. For example, the mode definitions on lines (15a-16a) below instruct Cigolf to only consider rules defining Bongard figures in terms of (a) the inclusion relations between the shapes that it contains, (b) the classes of these shapes, and (c) their directions. The alternative encoding on lines (15b-16b), takes advantage of Flora high-order and object embedding syntax to specify search for flexible rule patterns.

(14) modeh(1, ++A::bgEx).
(15a) modeb(10, --Y:##C[belong -> ++A:bgEx, in ->> --Z:##C2]).
(16a) modeb(*, ++Z[belong -> ++A:bgEx, direction -> --D:##E]).
(15b) modeb(*, ++A:bgEx[##S ->> --F:##C[##M->> --F2:##C1]]).
(16b) modeb(10, ++A:bgEx[##S->> --F:##C[##M->> --F2: ##C1, ##N-> --D]]).

Integrity constraints are rules of the form `false :- Body`, where `Body` is a conjunction of Flora terms that must not be entailed from valid hypotheses and background knowledge. For example, the constraints below express that an object cannot be contained in another object that it contains.

```
(26) false :- X:shape[in ->> Y[in ->> X]].
```

Pruning directives have the form prune((CHead:-CBody)):-Body, where CHead:-CBody are Flora rule patterns to discard from the hypothesis space when the condition Body follows from the background knowledge. For example, the directive below excludes rules where there are one triangle is left of square.

```
(27) prune((E::bgEx :- T:triangle[belong -> E:bgEx,
                                  leftOf ->> S:square])).
```

Examples are Flora facts that instantiate `modeh` patterns. The positive ones are prefixed by `|-`, while the negative ones are prefixed by `|~`, as for example in lines 28-29 below. The attributes of these two examples are specified as background knowledge as shown in Prog. 1 fror ex22.

```
(28) |~ ex1::bgEx.
(29) |- ex22::bgEx.
```

4 Extending the Flora compiler

The Flora compiler consists of two main parts: *wrapping rules* and *trailer rules*. The wrapping rules rewrite domain knowledge represented as OO F-Molecules by first flattening them into a conjunction of binary relations between two classes, objects, attributes or method names, called *F-Atoms*. These F-Atoms are then substituted by Prolog terms using wrapper predicates. The correspondence between some F-Atoms patterns and their corresponding wrapper predicates is given in table 1.

Table 1. Main Flora F-Atoms patterns and their corresponding wrapper predicates.

F-Atom	Wrapper predicate	F-Atom	Wrapper predicate
`C1::C2`	`sub(C1,C2)`	`C[A=>>T]`	`mvdsig(C,A,T)`
`A:B`	`isa(A,B)`	`O[A*->V]`	`ifd(O,A,V)`
`O[A->V]`	`fd(O,A,V)`	`O[A*=>V]`	`ifdsig(O,A,V)`
`O[A->>V]`	`mvd(O,A,V)`	`O[A*->>V]`	`imvd(O,A,V)`
`C[A=>T]`	`fdsig(C,A,T)`	`O[A*=>>V]`	`imvdsig(O,A,V)`

For example, the wrapping rules would rewrite the Flora rule on line 30 below into the Prolog rule on line `a`.

```
(30)B:bgEx:-M:triangle[belong->B:bgEx,in->>P:triangle].
(a)derived_isa(B,bgEx):-isa(M,triangle),fd(M,belong,B),
              isa(B,bgEx),mvd(M,in,P),isa(P,triangle).
```

This example illustrates that F-Atoms in different context may get rewritten into different wrapper predicates, such as `derived_isa` in a rule head and `isa` in a rule body. Wrapping rules alone rewrite a Flora program into a Prolog program only *syntactically*. *Semantic* translation further requires concatenating trailer rules at the end of the rewritten program. The trailer rule base is a domain independent Prolog axiomatization, in terms of wrapper predicates, of the complex OO semantics of Flora, inincluding inheritance, overwriting and object identity equality.

The Flora compiler designed to translate an OOLP into an equivalent relational LP for deduction with XSB, needed to be extended to perform such translation for induction with AlephTP. To that effect, we created two additional wrapper predicate classes: mode wrappers `map_modeN/m+1` and complex type wrappers $\texttt{tcg\_V}_i\texttt{/1}$. For each Cigolf mode declaration of the form `M(Recall,Ft)`, where `M` is `modeh` or `modeb`, our induction-oriented version of the Flora compiler:

1. Creates a new wrapper predicate of the form
   `map_modeN(#tmo,D1tcg_V1,...,Dmtcg_Vm)`
   where `N` records that it wraps the N^{th} mode declaration, $V_1, \ldots, V_m$ are the variables occurring in the Flora term `Ft`, and $D_1, \ldots, D_m$ are their respective unification direction constraints;
2. Adds the following clauses to the AlephTP translation of the Cigolf input:
   – An Aleph mode declaration of the form
     `:- M(Recall,map_modeN(#tmo,D1tcg_V1,...,Dmtcg_Vm)`
   – A fact `tmo(N)`;
   – For each $V_i \in \{V_1, \ldots, V_m\}$, a wrapper rule of the form:
     $tcg\_V_i(V)$ `:-` Cv_i, where Cv_i is the Flora wrapper predicate conjunction of all the sub-terms of `Cft` in which V_i occurs;
   – If `M = modeb`, a mode wrapper rule of the form:
     $map\_modeN(N,V_1,\ldots,V_m)$ `:- Cft,` where `Cft` is the Flora wrapper predicate conjunction that results from calling the Flora compiler onto `Ft.`

Finally, for each example `Eft`, whose OO type unifies with the `Ft` occurring in the N^{th} Cigolf mode declaration `modeh(Recall,Ft)`, our modified Flora compiler adds an example wrapper fact of the form: $map\_modeN(N,A_1,..,A_m)$ to the AlephTP input bias specification. $A_1,..,A_m$ are constants of `Eft` that unify with the labeled variables of `Ft`. An example of such translation is given below. Lines 31-32 contain the Cigolf mode declarations and lines 33-34 the Cigolf examples. Lines *b-c* contain the Aleph mode declarations with embedded Cigolf mode wrapper predicates. These predicates are defined in line *j-m* in terms of Flora compiler wrapper predicates and Cigolf example wrapper predicates. Lines *d-e* contains the Cigolf `tmo` (Type Mode Order) wrapper predicates and lines *f-i* defines the Cigolf typing wrapper predicates in terms of Flora wrapper predicates.

(31) modeh(1, ++A::bgEx).
(32) modeb(*, --Y:##C[belong -> ++A:bgEx, in ->> --Z:##C]).
(33) |- ex22::bgEx.
(34) |~ ex1::bgEx.
(b) :- modeh(1, map_mode1(#tmo,+tcg_A)).
(c) :- modeb(*,map_mode2(#tmo,-tcg_Y,#tcg_C, +tcg_A,-tcg_Z).
(d) tmo(1).
(e) tmo(2).
(f) tcg_Y(Y):- isa(Y,C), fd(Y,belong,A), mvd(Y,in,Z).
(g) tcg_C(C):- isa(Y,C), isa(Z,C).
(h) tcg_A(A):- fd(Y,belong,A).
(i) tcg_Z(Z):- mvd(Y,in,Z), isa(Z,C).
(j) map_mode2(2,Y,C,A,Z):- isa(Y,C), fd(Y,belong,A), isa(A,bgEx), mvd(Y,in,Z), isa(Z,C).
(l) map_mode1(1,ex22). % in positive example file
(m) map_mode1(1,ex17). % in negative example file

5 Testing CIGOLF on ILP Benchmarks

We tested Cigolf on two standard relational classification benchmarks: a 30 example Bongard problem and a 10 example train problem. The second problem consists of learning rules that predict the direction of a train given the properties of its wagons. For each problem, we compared a relational representation tested with AlephTP and two OO representations tested with Cigolf. The first uses flat F-molecules with variables only in object and class positions. The second uses embedded F-molecules with variables also in attribute positions. These experiments provide a first assessment of the run time overhead resulting from using a hybrid object-rule representation for machine learning. Their goal is to check whether such overhead is an acceptable price to pay for the sizable data preparation and interpretation speed-up achieved by avoiding time-consuming and tedious manual translation to the purely relational format of available ILP engines from the examples, background knowledge and induced knowledge often available and usable only in an hybrid object-rule or object-relational format.

Table 2. Cigolf and AlephTP test runs on the Bongard and Train problem.

	Bongard				Train			
Model	Learned Rules	Time	Bottom Clause Size	Bottom Clause Generalizations	Learned Rules	Time	Bottom Clause Size	Bottom Clause Generalizations
AlephTP	2	3.14s	20	216	2	9.1s	25	882
Cigolf1	2	13.94s	60	227	2	5.7s	22	374
Cigolf2	2	8.62s	21	42	2	6.9s	29	404

The results given in Table 3 are encouraging in the sense that the efficiency overhead between of IOOLP with Cigolf as compared to relational ILP with AlephTP is generally small. Interestingly, the second OO representation that more fully exploits Flora's flexible high-order syntax even outperforms AlephTP in some cases. A more systematic analysis of various bias specifications and their interaction with the use of tabling for learning, should bring interesting insights on the interactions between object-oriented modelling, high-order syntax and induction search space size.

6 Related works

Learning knowledge in a hybrid language that integrates first-order Horn rules with object-centered descriptions with inheritance was implemented in two main previous systems: WiM-D[13] and CILGG [7]. These two proposals share with ours the main idea: reusing an existing ILP engine to perform induction and developing a bi-directional translation between the hybrid language and the ILP engine's purely relational language.

In terms of language, WiM-D differs from Cigolf in that it implements learning in a *subset* of F-Logic that excludes methods, set valued attributes and class attributes,

while Cigolf implements learning in Flora, a *superset* of F-Logic that extends it with HiLog high-order syntax and Transaction Logic declarative backtrackable updates predicates. In terms of learning capabilities, WiM-D is less versatile than Cigolf for it cannot, for example, learn from noisy data.

In terms of language, CILGG differs from Cigolf in that it implements learning in CARIN-ALN [9], an extension of Datalog that allows terms in rule bodies and queries to be substituted by concepts defined in the description logic ALN. CARIN-ALN differs from Flora in four ways: with respect to description-rule integration, to relational terms, to description terms, and to semantics. With respect to description-rule integration, Flora is more expressive than CARIN-ALN for allowing description terms in rule heads. With respect to relational terms, Flora is more expressive than CARIN-ALN for allowing function symbols. With respect to description terms, Flora also seems to be more expressive than CARIN-ALN. First, cyclic and disjunctive definitions are allowed in Flora class signatures while excluded from ALN descriptions. Semantically, CARIN-ALN differs from Flora in that it works under the open-world assumption while Flora works under the closed-world assumption. However, for learning purposes, CILGG changes the semantics of CARIN-ALN to a hybrid semi-closed world assumption.

7 Conclusions

A key role of ILP in machine learning research is to explore the issues involved in learning representations that are even more intuitive, flexible, abstract and closer to the knowledge level. In this paper, we proposed to go further in that direction by learning in Flora, a language that extends the purely relational first-order Horn logic of traditional ILP with object-orientation, high-order syntax, declarative database updates predicates and well-founded negation. We demonstrated the conceptual and computational feasibility of inducing knowledge represented in Flora by reusing and integrating components from a relational ILP system and a deductive OOLP system. This integration resulted in Cigolf, the first comprehensive IOOLP system. Cigolf is sufficiently versatile and expressive to serve as a test bed to assess the strength and applicability of the IOOLP paradigm for many different learning tasks. In particular, it opens the door for the seamless integration of relational machine learning with mainstream object-oriented data, knowledge and software engineering.

In future work, we intend to test the scalability of Cigolf for larger classification tasks, as well as its versatility to perform other classes of learning tasks. The research presented here is part of a larger project that aims to develop a multi-agent software engineering methodology and environment that supports two complementary paths for automated agent code generation: one from manually built UML models, and the other from input/output training examples. In this project, Flora is used as the common target language integrating these two paths. Potentially incomplete Flora code is first generated from the UML model. Cigolf then uses this code as background knowledge to inductively complete it, based on the examples.

References

1. Bonner, A.J., Kifer, M. "An Overview of Transaction Logic," Theoretical Computer Science, vol. 133, pp.205-265,October 1994. http://citeseer.nj.nec.com/bonner94overview.html
2. Chen, W., Kifer, M., Warren, D.S. HiLog: A Foundation for Higher-Order Logic Programming. Journal of Logic Programming, number 3, volume 15 (1989)
3. Costa, V., Damas, L., Reis, R., Azevedo, R. YAP User's Manual. http://www.ncc.up.pt/~vsc/Yap/ (2000)
4. Dzeroski, S., Lavrac, N., (Eds.) Relational Data Mining. Springer-Verlag. (2001).
5. Fensel, D. Ontologies: Silver Bullet for Knowledge Management and ElectronicCommerce, Springer-Verlag, Berlin,(2001).
6. Flener, P., Partridge, D. Inductive Programming.Automated Software Engineering,Special Issue on Inductive Programming 8(2001).
7. Kietz, J-U. A Data-Preprocessing Method Enabling ILP-Systems to Learn CARIN-ALN Rules. http://www.afia.polytechnique.fr/CAFE/ILP01_Archi/Archi/kietz.pdf(2001).
8. Kifer, M., Lausen, G., Wu, J. Logical Foundations of Object-Oriented and Frame-Based Languages. Technical Report 90/14, Department of Computer Science, State University of New York at Stony Brook- SUNY(1990).
9. Levy, A.Y., Rousset, M.-C. CARIN: A representation language combining Horn rules and description logics. In Proc. of the 12th European Conf. on Artificial Intelligence (ECAI-96), pages 323—327(1996).
10. Liu, M. . Pluto: An Object-Oriented Logic Programming Language. Proceedings of the 39th International Conference and Exhibition on Technology of Object-Oriented Languages and Systems(TOOLS39), Santa Barbara, California (2001).
12. Nebel, B., Smolka, G. Attributive Description Formalisms. In C. Rollinger O. Herzog, editor, Text Understanding in LILOG, LNAI 546. Springer-Verlag, Berlin, Germany(1991).
13. Popelinsky, L. Object-oriented data modelling and rules: ILP meets databases.Proceedings of Knowledge Level Modelling Workshop,ECML'95 Heraklion, Crete(1995).
14. Sagonas, K., Swift, T., Warren, D. S. XSB as an efficient deductive database engine. In R. T. Snodgrass and M. Winslett, editors, Proc. of the 1994 ACM SIGMOD Int. Conf. on Management of Data (SIGMOD'94), pages 442—453(1994).
15. Srinivasan, A. The Aleph Manual. Oxford University, Machine Learning Group at the Computing Laboratory.http://web.comlab.ox.ac.uk/oucl/research/areas/machlearn/Aleph/ (2003).
16. Yang, G., Kifer,M. FLORA: Implementing an Efficient DOOD System Using a Tabling Logic Engine. In 6th International Conference on Rules and Objects in Databases (DOOD), http://citeseer.nj.nec.com/309887.html(2000).