OO-logic: a Successor of F-logic

Jürgen Angele[1] and Kevin Angele[23]

[1] adesso, Competence Center Artificial Intelligence
juergen.angele@adesso.de
http://adesso.de
[2] Semantic Technology Institute Innsbruck
Department of Computer Science, University of Innsbruck, Innsbruck, Austria
kevin.angele@sti2.at
http://sti2.at
[3] Onlim GmbH, Telfs, Austria
http://onlim.com

Abstract. Object oriented logic (abbreviated: OO-logic) combines the advantages of conceptual modelling that comes from object-oriented frame-based languages with the declarative style, compact and simple syntax, and the well defined semantics of a logic-based language. OO-logic supports typing, meta-reasoning, complex objects, properties, classes, inheritance, rules, queries, modularization, and scoped inference. In this paper we describe the capabilities of knowledge representation systems based on OO-logic and illustrate the use of this logic for ontology specification. OO-logic is a successor of F-logic. It incorporates the experience of a decade of using F-logic in real life applications. OO-logic simplifies the syntax of F-logic and has its focus on rule based reasoning with large sets of data (billions of triples).

Keywords: Rule language · Conceptual Modelling · F-logic · Ontology Language.

1 Introduction

A conceptual model (or an ontology) is an abstract, declarative description of the information for an application domain. It includes the relevant vocabulary, constraints on the valid states of the information, and the ways to draw inferences from that information. As applications grew in sophistication, computing power increased and our knowledge of algorithms for query processing enriched, the use of rule-based languages for processing information has become more and more attractive. A further push came from the Semantic Web, which increased the awareness of the need for logic-based languages for processing ontologies and other distributed knowledge on the Web. This awareness led W3C to create a working group that was chartered with the creation of a Rule Interchange Format

(RIF) [9] for Web-enabled applications written in rule-based languages. Datalog [1] is the grandfather of all database rule languages. It has a model-theoretic semantics, can be efficiently implemented, and is reasonably expressive. However, it does not support function symbols, which are important for representing objects, and it is a poor choice as a modelling language, as it does not provide different syntaxes for different semantic concepts. F-logic (or Frame Logic) [3,11] has emerged as a popular extension that provides negation, function symbols, high-level modelling constructs, and frame-based syntax. OO-logic simplifies the syntax of F-logic and has its focus on rule based reasoning with large sets of data (billions of triples). The simplification of the language reflects the experience in various industrial projects. This simplified subset is what users really understood and what they really used in those past projects. This paper takes a view of OO-logic as an ontology modelling language as well as a language for building applications that use these ontologies. The ability to span both sides of the engineering process, ontologies and applications, is a big advantage of OO-logic.

In the second section (section 2) we will give a short introduction to OO-logic including the semantics and the implementation of the language. Afterwards in section 3 we compare it to other rule languages and finally in section 4 a first industrial use case is presented.

2 A brief introduction to OO-logic

OO-logic is an object-oriented language and ontologies are modelled in this language in an object-oriented style. One starts with class hierarchies, proceeds with type specifications, defines the relationships among classes and objects using rules, and finally populates the classes with concrete objects. The first part of this example is a small ontology.

The last line of Listing 1 contains a query (1) to the object base. It inquires the sons of Abraham and will return the answer `?Y = Isaac`.

Object names and variable names Object names and variable names, also called id-terms, are the basic syntactical elements of OO-logic. To distinguish object names from variable names, the later are prefixed with a `?`-sign. Examples of object names are `Abraham`, `Man`, `father`, and of variable names are `?X`, `?Y`. Complex id-terms are created from function symbols and other id-terms as usual in predicate logic: `couple(Abraham, Sarah)`, `f(?X)`.

Class hierarchies Class hierarchies are defined with the help of isa-OO-atoms and subclass-OO-atoms. An isa-OO-atom of the form `o:c` states that an object `o` is a member of class `c`. A subclass-OO-atom of the form `sc::cl` says that the class `sc` is a subclass of the class `cl`. In our example it is stated that `Man` is a sub-class of class `Person` and `Abraham` is an instance of class `Man`. In contrast to other object-oriented languages where an object can be an instance of exactly

```
/* class hierarchy */
Woman::Person.                                  // every woman is a person
Man::Person.                                    // every man is a person
Person[properties: {father,son}].               // a person has properties father and son
father[range: Man].                             // property father has values which are men
father[minCardinality:1, maxCaridnality:1].     // everybody has exactly one father
son[range: Man].                                // property son has values which are men

/* rules consisting of a rule head and a rule body */
rule1: ?X[son: ?Y]                              // if ?Y is a man and ?Y has father ?X
    :- ?Y:Man, ?Y[father: ?X].                  // then ?X has son ?Y


/* facts */
Abraham:Man.                                    // Abraham is a man
Isaac:Man.                                      // Isaac is a man
Isaac[father: Abraham].                         // Isaacs father is Abraham

/* query */
?-  Abraham[son: ?Y].                           // give me all sons of Abraham (1)
```

Listing 1: A small ontology in OO-logic

one most specific class (e.g. [12]), OO-logic permits to be an instance in several, possibly incomparable, most specific classes. Likewise, a class can have several incomparable most specific super-classes. Thus, the class hierarchy is a directed acyclic graph.

Properties Application of a property to an object is specified using data-OO-atoms. A remarkable feature of OO-logic is that properties are also denoted by objects and can be handled like regular objects without any special language support. For instance, in Listing 1 the property names `father` and `son` are object names just like `Isaac` and `Abraham`. A property can have multiple values. Properties are inherited by sub-classes and instances.

Variables Variables are permitted at all positions in isa-, subclass-, and data-OO-atoms, so objects, properties, and classes are represented and queried uniformly using the same language facilities. In this way, OO-logic naturally supports the meta-information facility.

Signatures In contrast to F-logic which provides a sophisticated schema modelling language in OO-logic meta modelling is used. This means that classes and properties itself are seen as objects and we can have properties for these objects as well. This is the main difference to F-logic. In our previous example we expressed that class `Person` has properties `son` and `father`. For property `father` we stated that the range is `Man`. Property `father` must have at least one value (minCardinality = 1). Meta modelling allows to add various information to such properties.

Predicate Symbols Experience shows that it is convenient to be able to use predicates alongside objects. In OO-logic, predicate symbols are used in the same way as in other deductive languages, e.g., in Datalog. A predicate formula is constructed out of a predicate symbol followed by one or more id-terms separated by commas and included in parentheses. Such a formula is called a P-atom. In the following some P-atoms are shown.

```
married(Isaac,Rebekah).
male(Jacob).
```

Built-ins OO-logic supports built-ins which define procedural attachments that are seamlessly embedded into OO-Logic. Usually a built-in represents an infinite relation. Built-ins are always preceded by an "_" (underscore). E.g. adding two numbers to a third can be considered as an infinite relation between 3 numbers:

```
_plus(?X,?Y,?Z)
```

Dependent on two of the three arguments an algorithm behind the built-in computes the third argument. OO-Logic provides among others all methods of the Java packages Math and String as built-ins.

Aggregations Aggregations are special built-ins. In contrast to ordinary built-ins they have a whole set of values as input and generate output values dependent on the input. In newer logic like F-logic a special syntax has been introduced for such aggregations:

```
?N := count{?X | ?X:Person}
```

This expression means: count all instances of class `Person` and assign this number to variable `?N`.

Such aggregation expressions can also have a grouping variable. This means that the results are grouped according to the values of the grouping variable `?G`:

```
r: numberOfPersons(?G,?N) :-
        ?N := count{?X [?G] | ?X:Person, ?X[hasAge: ?G]}.
```

This rule determines for all occurring ages the number of persons with that age. As an extension to F-logic whole tuples are allowed here in the variable and grouping positions.

Path expressions Path expressions are a standard feature in most object-oriented languages. In OO-logic, a path expression of the form `obj.expr` denotes the set of objects `{a1,a2,...}`, such that `obj[expr: {a1,a2,...}]` is true. Here are some path expressions and the sets of objects they refer to in our example (Listing 1).

```
Abraham.son                     {Isaac}
Isaac.father.son                {Isaac}
```

Rules Rules are one of the best known technologies for building applications around ontologies. We already have seen examples of rules in Listing 1. In general, a rule is an expression of the form `head :- body`, where the head of the rule is a Boolean combination of OO-molecules and the body is a Boolean combination of OO-molecules or negated OO-molecules. Conjunctions are represented using commas. Disjunction in the body is represented by semicolons. Molecules may contain variables, and all variables are implicitly ∀-quantified outside of the rule. Variables can be anonym like `?_`. Using an anonymous variable in a rule means that the value which instantiates the variable does not matter. Rules in OO-logic have a logical semantics based on the principles developed in the context of logic programming and deductive databases. Consider the following rule from Listing 1:

```
rule1:  ?X[son: ?Y] :- ?Y:Man, ?Y[father: ?X].
```

Scoped Inference: Modularization and Integration The concept of scoped inference [8,10] is central to modularization and integration of knowledge. It was first proposed in TRIPLE [13] and FLORA-2 [15]. The concept of a module is well known in software engineering, and it is equally important in knowledge engineering. It is especially important for representing distributed knowledge, such as ontologies scattered over the Web, since rules and concepts that belong to different ontologies may interact in subtle and unintended ways. The basic idea is that a knowledge base is a collection of scopes of inference or modules. Each module is a collection of rules and facts. The notion of a rule is extended as follows. As before, it is a statement of the form

```
Head :- Body.
```

Predicates and molecules in a rule can optionally be labeled with module references like this: `pred-or-molecule@module-name`. A subformula of the form `L@N` inside body or a rule is a query to module `N`, which asks whether `L` is implied by the knowledge base that resides in module `N`. For instance, some data source, `gendata`, may provide information about parents of various individuals. One may not be able to (or may not want to) insert new rules into that data source in order to preserve the integrity of that data. However, it is possible to create a different module, say `mygenealogy`, which reference the information in the aforesaid data source. In the following we see queries for information in those two different modules:

```
?-  ?X[parent: ?Y]@gendata.
?-  ?X[parent: ?Y]@mygenealogy.
```

Module names can be arbitrary identifiers. As an extension to F-logic variable names are allowed in module positions as well. Besides modularization, the concept of a module is a potent vehicle for integration of and reasoning about ontologies that reside at different sources. If one just unions the rules and the facts found at the sources of interest, as implied by the import mechanism of

the OWL language, the rules may contradict each other or have subtle and unintended interactions. In contrast, if different sources are treated as separate modules, one can distinguish the information defined at the different sources and then specify the appropriate integration rules. These rules may give preference to some sources, partially or completely disregard information supplied by others, or clearly flag conflicting information.

Special syntax For convenience reasons OO-logic provides a special syntax for comparisons and for mathematical expressions:

```
r0: ?X:\todo{blank?} YoungPeople :- ?X[age: ?Z], ?Z < 20.
r1: ?X: Teenager :- ?X[age: ?Z], 13 <= ?Z < 20.
```

An assignment of the value a mathematical expression is given in the following way:

```
r: ?X[a: ?Y] :- ?X[b: ?Z], ?Y := (?Z+5)*3.
```

These mathematical expressions allow besides the basic arithmetical operations also functions like sin, cos, sqrt, etc.

Implementation of OO-logic semantics OO-logic is implemented in the system sem.reasoner, which is a direct successor of OntoBroker [2]. Very similar algorithms as in OntoBroker are used for reasoning. sem.reasoner uses its own relational deductive engine. sem.reasoner's main inference mode is bottom-up, but it includes several enhancements inspired by top-down inference, such as optimized, embedded Magic Sets [5].

OO-atom	Predicate
A::B	subclass(A,B)
o:C	member(o,C)
o[A: b]	value(o,A, b)
p(b1,.,bn)	p(b1,.,bn)
A::B@M	subclass(A,B,M)
o:C@M	member(o,C,M)
o[A: b]@M	value(o,A,b,M)

Table 1. Transformation of OO-logic atoms into predicate notation

The main ideas of the actual translation from OO-logic into the relational syntax are as follows: First, molecular expressions are replaced by equivalent conjunctions of atomic molecules. Next, these atomic expressions are represented by first-order predicates. The resulting set of rules augmented with additional closure rules to capture the specific semantics of OO-logic. Some rules are needed

to express statements such as the transitivity of the subclass relationship. Others implement inheritance, and so on. Table 1 shows the second stage in the above process. Whenever an OO-logic specification is split into modules, the predicates type, subclass, member, value, etc., in table are disambiguated for different modules by adding an additional attribute.

The following are examples of some of the closure rules added in stage 3 of the above process:

```
// closure rules for ?X :: ?Y
a1: subclass(?X, ?Z) :-  subclass(?X, ?Y), subclass(?Y, ?Z).
// closure rules for ?X : ?C
a2: member(?O, ?C) :-  subclass(?C1, ?C), member(?O, ?C1).
// structural inheritance of signatures
a3: value(?C1, properties, ?A) :-
        subclass(?C1, ?C2), value(?C2, properties,?A).
```

The resulting set of rules and facts is a set of Horn clauses with negation. This set of logic clauses is then processed using stratified semantics (minimal model semantics) for rule-based languages [14].

Experience with OntoBroker has shown that implementing a rule based system on top of a database or a triple store heavily restricts the performance of the system. Therefore the reasoning engine sem.reasoner has its own proprietary persistency layer. This persistency layer is based on B+ trees [4]. This allows to efficiently integrate large data sets into rule based reasoning. Several billions of triples have already been loaded into this persistency layer.

3 Comparison to other rule languages

In the following we compare OO-logic with F-logic, SWRL, and SPIN - languages used in the semantic web field.

Comparison to F-logic OO-logic is a successor to F-logic. Nearly a decade of using F-logic has shown that especially signatures in F-logic have a too complex syntax. Therefore we chose meta-modelling for defining properties and properties of those properties. This seems to be much easier to understand for subject matter experts. In additon we allow variables at all positions of OO-atoms and OO-molecules. So even at the property and module positions variables are allowed. Our special syntax for expressing comparisons, mathematical expressions was very well adopted from SMEs. Finally to have a full set of built-ins from Java Math and String library was very important for the applications. In the same way as it was with OntoBroker built-ins are easily extendible.

Comparison to SWRL SWRL (Semantic Web Rule Language) [7] is a member submission to W3C. SWRLs syntax is based on predicates. Thus it does not syntactically distinguish different concepts of ontologies, like membership to

classes, sub-class relationships and properties like it is done in OO-logic. Thus the SWRL syntax is not very well memorable by humans. A rule language is Turing-complete as soon as it provides function symbols. SWRL is based on OWL and thus does not provide function symbols. This heavily restricts expressiveness of SWRL. In contrast to OO-logic SWRL allows to express equality between different terms. SWRL allows to express existentially quantified variables. This can be done in OO-logic by using negation only.

Comparison to SPIN SPIN [6] is also a member submission to W3C. It is also called SPARQL rules. SPINs syntax is closely related to SPARQL syntax which in turn is related to SQL syntax. The syntax is dominated by a SQL like structure and there is no syntactical differentiation between ontology concepts like instance memberships, sub-classes, and properties. SPARQL is a query language for RDF and OWL and does not provide function symbols. So expressiveness of SPIN is also very limited.

4 A first Industrial Use Case

In the following we describe an industrial application "Routing and Reconciliation of SWIFT transactions in a bank" where OO-logic is already applied in.

OO-logic is currently used in a core banking system of a large European bank for reconciliation and routing of SWIFT messages. SWIFT (Society for Worldwide Interbank Telecommunication) is a telecommunication infrastructure for handling international money transactions. SWIFT also defines messages for those transactions. OO-logic rules are used for describing routing and reconciliation of such messages. Routing means that incoming messages are routed to different target systems. Reconciliation means that related messages are checked against each other. For instance a company has two bank accounts A,B in different banks. All incomes go to A, all outcomes go from B. At the end of each month a large amount of money has to be transferred from A to B in order to cover all the salaries. Now it has to be checked whether the money transferred from A to B covers all the outgoing salary transactions. For this purpose a small ontology describing the classes and properties of those SWIFT messages has been created. The incoming SWIFT messages are translated to OO-logic objects and are validated against the ontology. OO-logic rules are used to describe where a SWIFT message has to be routed to and rules do the check for the different reconciliation scenarios according to the values of the properties of the SWIFT messages. The standard routing and matching scenarios have been directly described using OO-logic rules. For derived scenarios inductive learning (rule learning) has been applied to automatically create appropriate rules. This application processes up to 100000 transactions a day. A transaction corresponds to one money transfer. This is the whole volume of transactions of the bank for that domain. The application is required to have a maximum down time of 4 hours a year. An example for such a rule is the following:

```
Match3: match(?MT910,?MT103,Match3) :-
    ?MT103:MT103, ?MT910:MT910,
    ?MT103[sendersReference:?R, currency:?C],
    ?MT910[referenceToRelatedTransaction:?R, currency:?C, value:?V],
    ?S:= sum{?V [?MT910] | ?MT103:MT103, ?MT910:MT910,
        ?MT103[sendersReference:?R, currency:?C, value:?V],
        ?MT910[referenceToRelatedTransaction:?R, currency:?C]},
    ?S = ?V.
```

It states that a message of type MT910 matches with several messages of type MT103 if the sum of the values in MT103 equals to the value in MT910.

5 Conclusion and future work

We have presented OO-logic which is a successor of F-logic. Both languages differentiate conceptual concepts by its syntaxes. OO-logic heavily simplifies the syntax of F-logic, adds additional useful features and adds special syntax for convenience reasons. Among others OO-logic includes all Math and String functionalities of Java as built-ins. OO-logic is implemented by the reasoner sem.reasoner. Its architecture allows to efficiently do reasoning in large data sets. OO-logic is a logical consequence of a decade using F-logic in real world applications. With OO-logic we think we have a mature ontology and rule language. adesso (`https://adesso.de`) has OO-logic adopted in several of its projects. In future we will focus on the further development of the reasoning engine sem.reasoner. So topics like clustering, data safety, distributed reasoning are on the agenda.

References

1. Abiteboul, S., Hull, R., Vianu, V.: Foundations of databases: the logical level. Addison-Wesley Longman Publishing Co., Inc. (1995)
2. Angele, J.: Ontobroker. Semantic Web **5**(3), 221–235 (2014)
3. Angele, J., Kifer, M., Lausen, G.: Ontologies in f-logic. In: Handbook on Ontologies, pp. 45–70. Springer (2009)
4. Bayer, R., McCreight, E.: Organization and maintenance of large ordered indexes. In: Software pioneers, pp. 245–262. Springer (2002)
5. Beeri, C., Ramakrishnan, R.: On the power of magic. The journal of logic programming **10**(3-4), 255–299 (1991)
6. Fürber, C., Hepp, M.: Using sparql and spin for data quality management on the semantic web. In: International Conference on Business Information Systems. pp. 35–46. Springer (2010)
7. Horrocks, I., Patel-Schneider, P.F., Boley, H., Tabet, S., Grosof, B., Dean, M., et al.: Swrl: A semantic web rule language combining owl and ruleml. W3C Member submission **21**(79), 1–31 (2004)
8. Kifer, M.: Nonmonotonic reasoning in flora-2. In: International Conference on Logic Programming and Nonmonotonic Reasoning. pp. 1–12. Springer (2005)
9. Kifer, M., Boley, H.: Rif overview. W3C working draft, W3C,(October 2009). http://www. w3. org/TR/rif-overview (2013)

10. Kifer, M., De Bruijn, J., Boley, H., Fensel, D.: A realistic architecture for the semantic web. In: International Workshop on Rules and Rule Markup Languages for the Semantic Web. pp. 17–29. Springer (2005)
11. Kifer, M., Lausen, G., Wu, J.: Logical foundations of object-oriented and frame-based languages. Journal of the ACM (JACM) **42**(4), 741–843 (1995)
12. Liu, M.: Deductive database languages: problems and solutions. ACM Computing Surveys (CSUR) **31**(1), 27–62 (1999)
13. Sintek, M., Decker, S.: Triplea query, inference, and transformation language for the semantic web. In: International Semantic Web Conference. pp. 364–378. Springer (2002)
14. Van Gelder, A., Ross, K.A., Schlipf, J.S.: The well-founded semantics for general logic programs. Journal of the ACM (JACM) **38**(3), 619–649 (1991)
15. Yang, G., Kifer, M., Zhao, C.: Flora-2: A rule-based knowledge representation and inference infrastructure for the semantic web. In: OTM Confederated International Conferences" On the Move to Meaningful Internet Systems". pp. 671–688. Springer (2003)