Florid – A Prototype for F-logic

Paul-Th. Kandzia Christian Schlepphorst*

Institut für Informatik, Albert-Ludwigs-Universität
D-79110 Freiburg, Germany
e-mail: {kandzia,schlepph}@informatik.uni-freiburg.de

Abstract

Florid is a deductive object–oriented database system relying on F-logic, which as a distinctive feature provides rich modeling facilities. In this paper we first give an overview about the language and its implementation. Furthermore, some examples illustrate advanced data modelling, particularly the solution of some nonmonotonic problems using inheritance is pointed out.

1 Introduction

Florid – **F-Lo**gic **R**easoning **I**n **D**atabases, provides an implementation of all essential features of F-logic [8], which is a powerful language combining the advantages of deductive databases with the rich modeling capabilities of object oriented concepts (objects, methods, class hierarchies, nonmonotonic inheritance). The language is one of the most advanced in the deductive object oriented database (DOOD) area, providing a uniform handling of data and schema (class hierarchy etc.) which allows among others data–dependent schema definition by rules.

Recently, several other prototypes for DOOD have been developed. ConceptBase [5] is built upon a flat relational data structure integrating the object oriented features by axioms. In contrast to F-logic it is not possible to create new objects within rule heads. ROL [10] is very close to F-logic but much more restrictive. Schema and data are strictly separated; methods have to be declared statically and variables can not range over them. Finally, approaches like LOLA [2], $\mathcal{LDL}$ [12], CORAL [13] extend deductive databases by complex datatypes like sets and lists but can not be considered as object oriented.

In this paper first a short overview over F-logic and Florid is given. The second part shows how distinctive features of the system may be used for advanced data modeling. In particular, the semantics of multiple inheritance and its usage for nonmonotonic reasoning is discussed.

2 Florid Basics

2.1 Syntax of F-logic

For a short glance, the main constructs can be described as follows (the full definition is found in [4, 8]):

Definition 1

- The alphabet of an F-Logic language consists of a set of *object constructors*, playing the role of function symbols, a set of variables, several auxiliary symbols like), (,], [, $\rightarrow$, $\bullet\!\!\rightarrow$, $\twoheadrightarrow$, $\bullet\!\!\twoheadrightarrow$, :, and the usual first-order logic connectives.

*The work of this author is supported by the Deutsche Forschungsgemeinschaft, La 598/3-2.

For convention, object constructors start with lowercase letters whereas variables start with uppercase ones.

- *ID-terms* serve as names of objects, classes and methods. They are composed from object constructors and variables.

In the sequel let $o, c, c_1, c_2, m, p_1, \ldots, p_n$ and r be ID-terms for $n \geq 0$.

- An *is-a atom* is an expression of the form $o : c$ (o is a member of class c) or $c_1 :: c_2$ (c_1 is a subclass of c_2).
- The following expressions are *data atoms:*
  - $o[m@(p_1, \ldots, p_n) \rightarrow r]$[1]: applying the scalar method m with parameters $p_1, \ldots, p_n$ to o – as an object – results in r,
  - $o[m@(p_1, \ldots, p_n) \bullet\!\!\rightarrow r]$[1]: o – as a class – provides the inheritable scalar method m to its members, which, if called with parameters $p_1, \ldots, p_n$, results in r,
  - $o[m@(p_1, \ldots, p_n) \twoheadrightarrow r]$[1] and $o[m@(p_1, \ldots, p_n) \bullet\!\!\twoheadrightarrow r]$[1]: as $\rightarrow$ and $\bullet\!\!\rightarrow$ above for a multivalued method m.
- An *eq–atom* is an expression of the form $p_1 = p_2$ with the meaning that p_1 and p_2 denote the same object.
- A *rule* $h :- b_1, \ldots, b_k.$ with $k \geq 1$, is a logic rule over atoms $h, b_1, \ldots, b_k$. A *fact* is a formula $h.$ for an atom h.
- A *query* is a formula $?- b_1, \ldots, b_k.$ for atoms $b_1, \ldots, b_k$, $k \geq 1$.
- A *program* is a set of facts and rules.

□

As a short notation several atoms can be combined to a *molecule*, so the expression

jacob : man[father→isaac; mother→rebekah : woman[son↠jacob]]

stands for the conjunction of the following atoms:

jacob : man, jacob[father→isaac], jacob[mother→rebekah], rebekah : woman, and rebekah[son↠jacob]

We omit the exact definition.
For illustration we give a simple example, namely a family tree. The method son is defined intensionally by rules; the query searches for all women X with a son Y whose father is abraham.

```
man :: human.
woman :: human.

abraham : man.
sarah : woman.
isaac : man[father→abraham; mother→sarah].
ishmael : man[father→abraham; mother→hagar : woman].
jacob : man[father→isaac; mother→rebekah : woman].                (2.1)
esau : man[father→isaac; mother→rebekah].

X[son↠Y]  :- Y : man[father→X].
X[son↠Y]  :- Y : man[mother→X].

?- sys.eval[].       % This is a system command.
?- X : woman[son↠Y[father→abraham]].
```

Note that F-logic does not distinguish between classes, methods, and objects which uniformly are denoted by ID-terms; also variables can occur at arbitrary positions of an atom.

[1]If $n = 0$ the parameter information @() is omitted.

2.2 Semantics of F-logic

The semantics of F-logic extends the semantics of first–order predicate logic. Formulas are interpreted over a semantic structure. We restrict our discussion to Herbrand–interpretations where the universe consists of all ground ID-terms. An H-structure is a set of ground atoms describing an object world, thus it has to satisfy several *closure axioms* related to general properties of Object Orientation.

Definition 2 Let H be a (possibly infinite) set of ground atoms. H is an *H-structure* if the following conditions hold for arbitrary ground ID-terms u, $u_0, \ldots, u_n$, u_r, u'_r and u_m:

- $u :: u \in H$ (subclass reflexivity),
- if $u_1 :: u_2 \in H$ and $u_2 :: u_3 \in H$ then $u_1 :: u_3 \in H$ (subclass transitivity),
- if $u_1 :: u_2 \in H$ and $u_2 :: u_1 \in H$ then $u_1 = u_2 \in H$ (subclass acyclicity),
- if $u_1 : u_2 \in H$ and $u_2 :: u_3 \in H$ then $u_1 : u_3 \in H$ (instance–subclass dependency),
- if $u_0[u_m @(u_1, \ldots, u_n) \rightsquigarrow u_r] \in H$ and $u_0[u_m @(u_1, \ldots, u_n) \rightsquigarrow u'_r] \in H$ then $u_r = u'_r \in H$, where $\rightsquigarrow$ stands for $\rightarrow$ or $\bullet\!\!\rightarrow$ (uniqueness of scalar methods).

Furthermore, the well known free equality axioms for $=$ have to hold. □

With respect to an H-structure the meaning of atoms and formulas is given in the usual way, moreover, minimal models can be defined as standard meaning of a program.

2.3 Program evaluation

After introducing the language we now will have a look at the implementation. FLORID is written in C++ and at present working in main memory only.
Programs are evaluated in a set–oriented bottom–up manner well–suited for databases (a feasible top down strategy for full F-logic has not been found yet, anyway). Thus, with every program a Datalog–like immediate consequence operator over H-structures is associated. In addition, the deduced logical consequences have to be completed by the atoms derivable due to the closure axioms to obtain an H-structure as result of the operator. Formally, a program could be augmented by the axioms. However, in FLORID this completion is not accomplished by the bottom–up evaluation. Instead, to improve efficiency, an appropriate data structure representing an H-structure, the *Object Manager* (OM), takes care of the closure axioms. To enable equating of objects during evaluation (due to eq–atoms in rule heads or closure axioms) a (non–injective) mapping from ID-terms to internal object identies (OID) is needed. This mapping also is stored in the OM. All information about one object is comprised in an *object frame* accessible by its OID. Finally, access components "hard–wiring" the closure axioms encapsulate the OM.
The least fixpoint of the consequence operator gives the minimal model of a program. For an efficient computation in FLORID a modified *seminaive evaluation* [14] is used. The crucial problem here is that the closure axioms have to be respected though they are not explicit in the program. To handle negation in F-logic, a syntactical stratification of the rules, as found in Datalog, turns out to be not very useful, since too many potential dependencies occur caused by the flexibility of the language (e.g. it is possible to define data–dependent equations by rules). Therefore, FLORID uses the *inflationary semantics* [9] for negated subgoals together with a user defined stratification of the programs.

2.4 The User Interface

FLORID's basic interface is a Prolog–like text shell, where the user carries out all interaction in F-logic syntax, too; however, to this aim special interface objects are used. E.g. in example (2.1) the system query `?– sys.eval[].` starts the evaluation of the rules. Output from the system is processed by a *PrettyPrinter* to find a readable form for the internal OID's. This is not a trivial task since an object may be denoted by several ID-terms or path expressions (see section 3.1, too). Besides the text shell there is a Tcl/Tk GUI for FLORID. Furthermore, FLORID is integrated into the emacs/Xemacs environment [3]. FLORID can be used on-line via the WWW, too[1].

3 Advanced Data modeling in Florid

After introducing the basics, in this section the distinctive F-logic programming style will be highlighted using examples from various areas.

3.1 Path Expressions and Object Creation

Usually, objects are accessed immediately by their object names, but it is also possible to use *path expressions* [4] to denote an object that is the result of a method application. In example (2.1) the object described by the object name `abraham` may also be denoted by applying the method `father` to the object `isaac`. The corresponding path expression is `isaac.father`. A path expression may occur at any place instead of an ID-term, furthermore a complex nesting of molecules and path expressions (called *references*) is allowed. Again we omit the details.
Besides enabling efficient object navigation, path expressions also facilitate the creation of objects which have no ID-term as name at all. In the next example, extending example (2.1), we define a generic method inverse on methods of the class baseMethod and apply it to mother to find the children of rebekah.

$$\begin{array}{l} \mathsf{mother : baseMethod.} \\ \mathsf{X[(M.inverse) \twoheadrightarrow Y]\ \ :\!-\ Y[(M{:}baseMethod) \rightarrow X].} \\ \mathsf{?\!-\ rebekah[(mother.inverse) \twoheadrightarrow X].} \end{array} \tag{3.1}$$

Since the method invocation mother.inverse has not been defined by a fact or rule anywhere else, there is nothing known about the result. FLORID creates therefore a new object and the given object world is extended by the fact mother[inverse→mother.inverse]; since there is no direct object name for this new object, it can exclusively be accessed by the path expression mother.inverse.
Note how the above program takes advantage of the meta reasoning facilities of F-logic: reasoning about methods is possible in the same way as it is about objects.

3.2 Synonymous Object Names

In a DOOD scalar methods may cause a problem if different results are deduced, e.g. as in the following example:

$$\mathsf{mary[mother \rightarrow gillian].} \qquad \mathsf{mary[mother \rightarrow gill].} \tag{3.2}$$

In contrast to other approaches F-logic resolves such inconsistencies automatically (the system will give a warning, though) by deducing the fact gill = gillian (due to the closure axioms, see Definition 2). Hence, in F-logic the equality predicate may be used in rule heads to make object names synonymous. However, as a drawback of the rule–defined equality and some other features, syntactical analysis of a program (stratification, magic set, etc.) becomes much more difficult than in other languages.

3.3 Miscellaneous

Besides the elements described in this paper F-logic supports Prolog–like predicates, signatures with structural inheritance, and the usual arithmetical builtins.

[1] URL: `http://www.informatik.uni-freiburg.de/~dbis/flogic-project.html`

4 Inheritance and Nonmonotonic Reasoning

Inheritance is one of the classical interesting but difficult problems of knowledge representation. A well known example is *Nixon's Diamond*:

$$\begin{array}{ll} \mathsf{quaker}[\mathsf{policy}\bullet\!\!\rightarrow\mathsf{pacifist}]. & \mathsf{nixon} : \mathsf{quaker}. \\ \mathsf{republican}[\mathsf{policy}\bullet\!\!\rightarrow\mathsf{hawk}]. & \mathsf{nixon} : \mathsf{republican}. \end{array} \tag{4.1}$$

In this program the policy of Nixon is ambigous: Once a fact from one class is inherited, the other is blocked by overriding, thus requiring nonmonotonic reasoning. Other DOOD–languages will therefore reject the above schema as an inconsistent one. However, since in F-logic the class hierarchy and method definitions are not statically given, such a situation may not be detected before runtime. Hence, the language must be able to handle ambiguous inheritance.

4.1 The Trigger Semantics for multiple Inheritance

The F-logic solution for the problems of nonmonotonic inheritance are *inheritance–canonic models* [8]. In a nutshell the idea is, to first derive a minimal model without considering inheritance at all, and then to decompose each inheritance step into a pair of sub–steps: 1) a "pure" inheritance step, which may introduce new facts but whose output may no longer be a model of the program; and 2) a new fixpoint derivation which extends the result of the previous step to a model. These operations are repeated until no more inheritance can be done.
The notion of an inheritance trigger formalizes when a method result can be inherited from a class to a certain instance: when 1) it is defined in the H-structure and 2) the H-structure contains no contradicting atom (due to overriding).
"Firing" a trigger, that is transferring the result of the method to the instance or subclass, leads to derivation of new facts by inheritance that cannot be derived using classical deduction alone. Once fired, a trigger is "deactivated" and no longer a trigger in the resulting H-strucure. When a trigger is fired, previously inactive program rules may become applicable. The combination of firing a trigger and subsequent fixpoint computation of the consequence operator is called *one–step inheritance*. Since in general an H-structure has more than one trigger a one–step inheritance transformation is not unique. An H-structure obtained by applying the one–step inheritance as often as possible is called an *inheritance–canonic* (or *i-canonic*) model of the program. It is clear that such models are not unique, too; following a credulous approach we accept an arbitrary i-canonic model as semantics for the program.
In the case of Nixon's Diamond (4.1) we have two triggers, $< \mathsf{policy}, \bullet\!\!\rightarrow, \mathsf{nixon} : \mathsf{quaker} >$ and $< \mathsf{policy}, \bullet\!\!\rightarrow, \mathsf{nixon} : \mathsf{pacifist} >$. Firing one of them will deactivate the other. Hence, this program has two i-canonic models and the result is indeterminate (in fact, FLORID will choose the first trigger found).
The FLORID implementation improves the concept of triggers by considering only the *relevant* ones, which actually are used for further deduction or for answering a query. The relevant triggers can be determined by syntactical analysis of the program rules and queries and inquiring the class information given by the computed model in the OM. The answers to queries are not altered by restricting the evaluation to relevant triggers.
Surprisingly, the trigger semantics can be helpful even in applications normally not using inheritance, as we will see in the following (see also [6]).

4.2 Disjunctive Information

As the example of "Nixon's Diamond" shows, multiple inheritance can lead to inherently disjunctive information ("Nixon is a pacifist or he is a hawk"). Vice versa, inheritance can be used to model disjunctive statements in FLORID. Suppose, a database should express the following (see [1]): "Mary visits Europe or Australia. People visiting one of those destinations are happy and those travelling to both are bankrupt afterwards." Since FLORID provides no direct operator for disjunction we introduce two auxiliary classes, eFriends and aFriends. By default, members of those classes visit Europe and Australia, respectively. The method visit is multivalued, since a person may go to several places. The object mary belongs to both classes, hence will inherit one of the destinations. The rest of the program is straightforward.

$$
\begin{array}{ll}
\mathsf{eFriends[visit{\bullet\!\!\twoheadrightarrow}europe].} & \mathsf{mary : eFriends.} \\
\mathsf{aFriends[visit{\bullet\!\!\twoheadrightarrow}aus].} & \mathsf{mary : aFriends.} \\
 & \\
\multicolumn{2}{l}{\mathsf{X : bankrupt :- X[visit{\twoheadrightarrow}europe], X[visit{\twoheadrightarrow}aus].}} \\
\multicolumn{2}{l}{\mathsf{X : happy :- X[visit{\twoheadrightarrow}europe].}} \\
\multicolumn{2}{l}{\mathsf{X : happy :- X[visit{\twoheadrightarrow}aus].}}
\end{array}
\tag{4.2}
$$

FLORID gives a reasonable semantics based on the credulous approach: In any case Mary is happy and never bankrupt, but the destination of her visit is not determined.

4.3 Temporal Modeling

The characteristic of the trigger semantics is a consecutive sequence of logical fixpoint computation and one–step inheritance. Interestingly, due to this operational semantics inheritance in FLORID enables the programming of sequentially computed deductive fixpoints. Opposed to FLORID's explicit program stratification, when using inheritance triggers the number of fixpoints is not fixed but can be controlled by the program. The following example illustrates the principle.

Consider the simulation of an electronic part. This part – modeled as an object – can be active (in this case the method active is defined with result 1); active and inactive states alternate continuously, e.g. driven by a clock signal. States are denoted simply by naturals. The electronic part can be described as follows:

$$
\begin{array}{l}
\mathsf{part[active@(N){\rightarrow}1] :-} \\
\quad \mathsf{S : state,\ N : state,\ N = S + 1,\ not\ part[active@(S){\rightarrow}1].}
\end{array}
\tag{4.3}
$$

Note that the parameter enables the versioning of the method active over various states. Now the temporal behaviour is implemented. The first fixpoint computation corresponds to state 0. If a state (fixpoint) is entirely computed, an appropriate trigger given by the fact state[done•→true] becomes active and fires.[1] This inheritance step logically entails the creation of the next state and so on. To terminate the state sequence, a break condition (here: at most 6 states) is added.

$$
\begin{array}{l}
\mathsf{0 : state.} \\
\mathsf{state[done{\bullet\!\!\rightarrow}true].} \\
\mathsf{N : state :- S : state[done{\rightarrow}true],\ N = S + 1,\ N < 6.}
\end{array}
\tag{4.4}
$$

After running the program we state the query

?– part[active@(S)→V].

and get the following answer:

S/1 V/1   S/3 V/1   S/5 V/1

It should be emphasized that the trigger mechanism guarantees the correct semantics, since it is essential that a new state is deduced not before all information of its predecessor is computed. Thus, correct evaluation of the negated subgoal by the inflationary semantics is ensured (which corresponds to a stratification according to the temporal states).

A deeper discussion of temporal modeling with FLORID is found in [11].

[1] In presence of other triggers not related to the state modeling a state may consist of multiple fixpoint computations. However, the state transition is defined by the related trigger.

5 Conclusion

As we have seen, FLORID not only provides the rich modeling facilities of F-logic; in addition various important nonmonotonic problems can be tackled using the inheritance mechanism. Hence, FLORID represents a powerful system for rapid prototyping even if the user does not know in advance, which features are needed for her application (see also [7]). A more efficient direct implementation of the problem may be obtained from the F-logic version afterwards.

Acknowledgements: Our thanks go to all colleagues and students, particularly to those involved in the daily nonspectacular work which is essential for such a project.

References

[1] S. Brass, J. Dix, and T. C. Przymusinski. Super Logic Programs. In *5. International Conference on Principles of Knowledge Representation and Reasoning*, 1996.

[2] B. Freitag, H. Schütz, and G. Specht. LOLA - a logic language for deductive databases and its implementation. In *Proc. 2nd International Symposium on Database Systems for Advanced Applications (DASFAA)*, 1991.

[3] J. Frohn, R. Himmeröder, P.-T. Kandzia, and C. Schlepphorst. FLORID, Version 1.0, User Manual, 1996. Available from ftp://ftp.informatik.uni-freiburg.de/pub/florid/manual.ps.gz.

[4] J. Frohn, G. Lausen, and H. Uphoff. Access to objects by path expressions and rules. 1994.

[5] M. Jarke, R. Gallersdörfer, M. A. Jeusfeld, and M. Staudt. ConceptBase - a deductive object base for meta data management. *Journal of Intelligent Information Systems*, 4(2):167–192, 1995.

[6] P.-T. Kandzia. Nonmonotonic Reasoning in Florid. In *Proc. Intl. Conference on Logic Programming and Nonmonotonic Reasoning*, number 1265 in LNAI. Springer, 1997.

[7] P.-T. Kandzia and C. Schlepphorst. DOOD and DL – Do we Need an Integration? In *Proc. Workshop Knowledge Representation Meets Databases (KRDB)*, 1997.

[8] M. Kifer, G. Lausen, and J. Wu. Logical foundations of object-oriented and frame-based languages. *Journal of the ACM*, 42(4):741–843, July 1995.

[9] P. Kolaitis and C. Papadimitriou. Why not negation by fixpoint? In *Proc. ACM Symposium on Principles of Database Systems*, pages 231–239, 1988.

[10] M. Liu. ROL: A typed deductive object base language. *Proc. Intl. Conference on Database and Expert Systems Applications (DEXA)*, 1996.

[11] W. May, C. Schlepphorst, and G. Lausen. Integrating dynamic aspects into deductive object-oriented databases. 1997.

[12] S. Naqvi and S. Tsur. *A logical Language for Data and Knowledge Bases.* Computer Science Press, New York, 1989.

[13] R. Ramakrishnan, D. Srivastava, S. Sudarshan, and P. Seshadri. Implementation of the CORAL deductive database system. In *Proc. ACM SIGMOD Intl. Conference on Management of Data*, 1993.

[14] J. D. Ullman. *Principles of Database and Knowledge-Base Systems*, volume 1. Computer Science Press, New York, 1988.