How to Write F-Logic Programs

A Tutorial for the Language F-Logic[1]

covers OntoBroker Version 3.5

Contents

[1] This tutorial refers to the syntactic and semantic capabilities of F-Logic as implemented by ontoprise GmbH (F-Logic Parser V 2.4, Ontobroker version V 3.5).

1. Introduction

F-Logic [KLW95] is a deductive, object oriented database language which combines the declarative semantics and expressiveness of deductive database languages with the rich data modelling capabilities supported by the object oriented data model.

The theoretical foundations of F-Logic have been described in the F-Logic report [KLW95]. For this tutorial parts of the F-Logic tutorial of the Florid project at the university of Freiburg have been used (http://www.informatik.uni-freiburg.de/~dbis/florid). The present tutorial describes how to apply F-Logic in the Ontobroker system. Therefore, this tutorial explains the various features of F-Logic by example and shows how to use them for typical problems. Section 2 gives a taste of how F-Logic programs look like. The same simple model world taken from the Old Testament also serves as a background database throughout the tutorial. The following Sections 3 to 7 focus on data modeling and present the language concepts of F-Logic. In section 8 it is described how F-Logic is used within the Ontobroker system.

We assume that the reader of this tutorial is familiar with the basic concepts of deductive databases, e.g., Datalog [AHV95, CGT90, Ull89], and the principles of object oriented database systems [ABD + 89].

This covers the features of the Ontoprise Ontobroker version V 3.X. The F-Logic variant of Ontoprise differs from the versions in [KLW95] and [FHK] in using a slightly different syntax (e.g. <- is used instead of :-) and in providing a lot of extensions (like builtins, name spaces etc.). In this version additionally any logical formula may occur in the bodies of rules.

2. A First Example

Before explaining the syntax and semantics in detail, we give a first impression of F-Logic. The following F-Logic program models biblical persons and their relationships:

```
/* facts */
abraham:man.
sarah:woman.
isaac:man[father->abraham; mother->sarah].
ishmael:man[father->abraham; mother->hagar:woman].
jacob:man[father->isaac; mother->rebekah:woman].
esau:man[father->isaac; mother->rebekah].

/* rules consisting of a rule head and a rule body */
FORALL X,Y X[son->>Y] <- Y:man[father->X].
FORALL X,Y X[son->>Y] <- Y:man[mother->X].
FORALL X,Y X[daughter->>Y] <- Y:woman[father->X].
FORALL X,Y X[daughter->>Y] <- Y:woman[mother->X].

/* query */
FORALL X,Y <- X:woman[son->>Y[father->abraham]].
```

(Example 2.1)

The first part of this example consists of a set of facts to indicate that some people belong to the classes man and woman, respectively, and give information about the father and mother relationships among them. According to the object-oriented paradigm, relationships between objects are modeled by method applications, e.g., applying the method father to the object isaac yields the result object abraham. All these facts may be considered as the extensional database of the F-Logic program. Hence, they form the framework of an object base which is completed by some closure properties.

The rules in the second part of Example 2.1 derive new information from the given object base. Evaluating these rules in a bottom-up way, new relationships between the objects, denoted by the methods son and daughter, are added to the object base as intensional information.

The third part of Example 2.1 contains a query to the object base. The query shows the ability of F-Logic to nest method applications. It asks about all women and their sons, whose father is Abraham. The same question could be written as a conjunction of simple subgoals:

```
FORALL X,Y <- X:woman AND X[son->>Y] AND Y[father->abraham].
```

Methods and classes also are objects, see Sections 3.1.1 and 3.1.2.

3. Objects and their Properties

As we have already seen in Example 2.1 objects are the basic constructs of F-Logic. Objects model real world entities and are internally represented by object identifiers which are independent of their properties. According to the principles of object oriented systems these object identifiers are invisible to the user. To access an object directly the user has to know its object name. Every object name refers to exactly one object. Following the object oriented paradigm, objects may be organized in classes. Furthermore, methods represent relationships between objects. Such information about objects is expressed by F-atoms.

3.1. Object Names and Variable Names

Object names and variable names are also called id-terms and are the basic syntactical elements of F-Logic. To distinguish object names from variable names, the later are always declared using logical quantifiers FORALL and EXISTS.

After the first letter, object names and variable names may both contain uppercase letters, lowercase letters, numerals or the underscore symbol "_" . Examples for object names are abraham, man, daughter, for variable names are X, Method. There are two special types of object names that carry additional information: integers and strings.

Every positive or negative integer may be used as an object name, e.g., +3, 3, -3, and also every string enclosed by quotation marks "".

Complex id-terms may be created by function symbols where other id-terms may be used as arguments, e.g., couple(abraham, sarah), f(X). An id-term that contains no variable is called a ground id-term.

3.1.1. Methods

In F-Logic, the application of a method to an object is expressed by data-F-atoms which consist of a host object, a method and a result object, denoted by id-terms. Any object may appear in any location: host object, result position, or method position. Thus, in our Example 2.1 the method names father and son are object names just like isaac and abraham.

Variables may also be used be used at all positions of a data-F-atom, which allows queries about method names like

```
FORALL X,Y <- isaac[X->>Y].
```

Methods may either be single-valued (->), i.e. can have one value only or they may be multi-valued (->>), i.e. can have more values. If more values are given for multi-valued attributes the values must be enclosed in curly brackets:

```
jacob[son->>{reuben, simeon, levi, judah, issachar, zebulun}].
```

Methods with Parameters. Sometimes the result of the invocation of a method on a host object depends on other objects, too. For example, Jacob's sons are born by different women. To express this, the method son is extended by a parameter denoting the corresponding mother of each of Jacob's sons. Like methods, parameters are objects as well, denoted by id-terms. Syntactically a parameter list is always included in parentheses and separated by "@" from the method object.

```
jacob[son@(leah)->>
        {reuben, simeon, levi, judah, issachar, zebulun};
   son@(rachel)->>{joseph, benjamin};
   son@(zilpah)->>{gad, asher};
   son@(bilhah)->>{dan, naphtali}].
```

(Example 3.1)

The syntax extends straightforwardly to methods with more than one parameter. If we additionally want to specify the order in which the sons of Jacob were born, we need two parameters which are separated by commas:

```
jacob[son@(leah,1)->>reuben; son@(leah,2)->>simeon;
   son@(leah,3)->>levi; son@(leah,4)->>judah;
   son@(bilhah,5)->>dan; son@(bilhah,6)->>naphtali;
   son@(zilpah,7)->>gad; son@(zilpah,8)->>asher;
   son@(leah,9)->>issachar; son@(leah,10)->>zebulun;
   son@(rachel,11)->>joseph; son@(rachel,12)->>benjamin].
```

(Example 3.2)

In Examples 3.1 and 3.2 the method son is used with a different number of parameters. This so-called overloading (see also Section 3.3) is supported by F-Logic. Given the object base described in Example 2.1, questioning the sons of Isaac

```
FORALL X <- isaac[son->>X].
```

yields all his known sons:

```
X = jacob
X = esau
```

Note that variables in a query may only be bound to individual objects, never to sets of objects, i.e., the above query does not return X = {jacob,esau}.

In case of a query with a set of ground id-terms at the result position, however, it is only checked whether all these results are true in the corresponding object base; there may be additional result objects in the database. With the object base above, all the following queries yield the answer true.

```
<-isaac[son->>{jacob, esau}].
<-isaac[son->>jacob].
<-isaac[son->>esau].
```

If we want to know if a set of objects is the exact result of a multi-valued method applied to a certain object, we have to use negation, see Example 7.2.

3.1.2. Class Membership and Subclass Relationship

Isa-F-atoms state that an object belongs to a class, subclass-F-atoms express the subclass relationship between two classes. Class membership and the subclass relation are denoted by a single colon and a double colon, respectively. In the following example the first three isa-F-atoms express that Abraham and Isaac are members of the class man, whereas Sarah is a member of the class woman. Furthermore, two subclass-F-atoms state that both classes man and woman are subclasses of the class person:

```
abraham:man.
isaac:man.
sarah:woman.
woman::person.
man::person.
```

(Example 3.3)

In isa-F-atoms and subclass-F-atoms, the objects and the classes are also denoted by id- terms because classes are objects as well as methods are objects. Hence, classes may have methods defined on them and may be instances of other classes which serve as a kind of metaclass. Furthermore, variables are permitted at all positions in an isa- or subclass-F- atom. In contrast to other object oriented languages where every object is instance of exactly one most specific class (e.g., ROL [Liu96]), F-Logic permits that an object is an instance of several classes that are incomparable by the subclass relationship. Analogously, a class may have several incomparable direct superclasses.

Thus, the subclass relationship specifies a partial order on the set of classes, so that the class hierarchy may be considered as a directed acyclic (but not reflexiv) graph with the classes as its nodes.

Note that in analogy to HiLog [CKW93] a class name does not denote the set of objects that are instances of that class.

3.2. Expressing Information about an Object: F-Molecules

Instead of giving several individual atoms, information about an object can be collected in F-molecules. For example, the following F-molecule denotes that Isaac is a man whose father is Abraham and whose sons are Jacob and Esau.

```
isaac:man[father->abraham; son->>{jacob,esau}].
```

(Example 3.4)

This F-molecule may be split into several F-atoms:

```
isaac:man.
isaac[father->abraham].
isaac[son->>jacob].
isaac[son->>esau].
```

For F-molecules containing a multi-valued method, the set of result objects can be divided into singleton sets (recall that our semantics is multivalued, not set-valued). For singleton sets, it is allowed to omit the curly braces enclosing the result set, so that the three given in 3.4, 3.5 and 3.6 are equivalent, which means that they yield the same object base:

```
isaac[son->>{jacob,esau}].
```

(Example 3.5)

```
isaac[son->>{jacob}].
isaac[son->>{esau}].
```

(Example 3.6)

```
isaac[son->>jacob].
isaac[son->>esau].
```

(Example 3.7)

3.3. Signatures

Signature-F-atoms define which methods are applicable for instances of certain classes. In particular, a signature-F-atom declares a method on a class and gives type restrictions for parameters and results. These restrictions may be viewed as typing constraints. Signature- F-atoms together with the class hierarchy form the schema of an F-Logic database. To distinguish signature-F-atoms from data- F-atoms, the arrow body consists of a double line instead of a single line. Here are some examples for signature-F-atoms:

```
person[father=>man].
person[daughter=>>woman].
man[son@(woman)=>>man].
```

The first one states that the single-valued method father is defined for members of the class person and the corresponding result object has to belong to the class man. The second one defines the multi-valued method daughter for members of the class person restricting the result objects to the class woman. Finally, the third signature-F-atom allows the application of the multi-valued method son to objects belonging to the class man with parameter objects that are members of the class woman. The result objects of such method applications have to be instances of the class man. By using a list of result classes enclosed by parentheses, several signature-F-atoms may be combined in an F-molecule. This is equivalent to the conjunction of the atoms: the result of the method is required to be in all of those classes:

```
person[father=>{man, person}].
```

(Example 3.9)

```
person[father=>man].
person[father=>person].
```

(Example 3.10)

Both expressions in the Examples 3.9 and 3.10 are equivalent and express that the result objects of the method father if applied to an instance of the class person have to belong to both classes man and person.

Overloading F-Logic supports method overloading. This means that methods denoted by the same object name may be applied to instances of different classes. Methods may even be overloaded according to their arity, i.e., number of parameters. For example, the method son applicable to instances of the class man is used as a method with one parameter in Example 3.11 and as a method with two parameters in Example 3.12. The corresponding signature-F-atoms look like this:

```
man[son@(woman)=>>man].
```

(Example 3.11)

```
man[son@(woman,integer)=>>man].
```

(Example 3.12)

Of course, the result of a signature may be enclosed in parentheses as well, if it consists of just one object.

As already shown in Example 3.4, properties of an object may be expressed in a single, complex F-molecule instead of several F-atoms. For that purpose, a class membership or subclass relationship may follow after the host object. Then, a specification list, a list of method applications (with or without parameters) separated by semicolons, may be given. If a method yields more than one result, those can be collected in curly braces, separated by commas; if a signature contains more than one class, those can be collected in parentheses, also separated by commas:

```
isaac[father->abraham; mother->sarah].
jacob:man[father->isaac; son@(rachel)->>{joseph, benjamin}].
```

```
man::person[son@(woman)=>>{man, person}].
```

(Example 3.13)

The following set of F-atoms is equivalent to the F-molecules in 3.13:

```
isaac[father->abraham].
isaac[mother->sarah].
jacob:man.
jacob[father->isaac].
jacob[son@(rachel)->>joseph].
jacob[son@(rachel)->>benjamin].
man::person.
man[son@(woman)=>> man].
man[son@(woman)=>>person].
```

(Example 3.14)

Besides collecting the properties of the host object, the properties of other objects appearing in an F-molecule, e.g., method objects or result objects, may be inserted, too. Thus, a molecule may not only represent the properties of one single object but can also include nested information about different objects, even recursively:

```
isaac[father->abraham:man[son@(hagar:woman)->>ishmael];
        mother->sarah:woman].
jacob:(man::person).
jacob[(father:method)->isaac].
```

(Example 3.15)

The equivalent set of F-atoms is:

```
isaac[father->abraham].
abraham:man.
abraham[son@(hagar)->>ishmael].
hagar:woman.
isaac[mother>>sarah].
sarah:woman.
man::person.
jacob:man.
jacob[father->isaac].
father:method.
```

F-Logic molecules are evaluated from left to right. Thus, nested properties have to be included in parentheses if those properties belong to a method object (cf. Section 7), class object or superclass object. Note the difference between the following two F-molecules. The first one states that Isaac is a man and Isaac believes in god, whereas the second one says that Isaac is a man and that the object man believes in god (which is probably not the intended meaning).

```
isaac:man[believesin->>god].
isaac:(man[believesin->>god]).
```

Moreover, omitting parentheses at method or result position can lead to syntactically incorrect molecules, e.g.,

```
isaac[(father::ancestor)->abraham]
```

is correct, whereas

```
isaac[father::ancestor->abraham]
```

results in a parsing error, and

```
isaac[father->(abraham:man)]
```

is correct, whereas

```
isaac[father->abraham:man]
```

results in a parsing error.

3.4. F-molecules without any properties

If we want to represent an object without giving any properties, we have to attach an empty specification list to the object name, e.g.

```
thing[].
```

If we use an expression like this that consists solely of an object name as a molecule, it is treated as a 0-ary predicate symbol (see next section).

4. Predicate Symbols

In F-Logic, predicate symbols are used in the same way as in predicate logic, e.g., in Datalog, thus preserving upward-compatibility from Datalog to F-Logic. A predicate symbol followed by one or more id-terms separated by commas and included in parentheses is called a P-atom to distinguish it from F-atoms. Example 4.1 shows some P-atoms. The last P-atom consists solely of a 0-ary predicate symbol. Those are always used without parentheses.

```
married(isaac,rebekah).
male(jacob).
sonof(isaac,rebekah,jacob).
true.
```

(Example 4.1)

Information expressed by P-atoms can usually also be represented by F-atoms, thus obtaining a more natural style of modelling. For example, the information given in the first three P- atoms in 4.1 can also be expressed as follows:

```
isaac[marriedto->>rebekah].
jacob:man.
isaac[son@(rebekah)->>jacob].
```

(Example 4.2)

Similar to F-molecules, P-molecules may be built by nesting F-atoms or F-molecules into P-atoms. The P-molecule

```
married(isaac[father->abraham], rebekah:woman).
```

is equivalent to the following set of P-atoms and F-atoms:

```
married(isaac,rebekah).
isaac[father->abraham].
rebekah:woman.
```

Note, that only F-atoms and F-molecules may be nested into P-atoms, but not vice versa.

5. Lists

A special kind or terms are lists. In F-Logic lists of terms can be represented as in Prolog. A list containing the constants `a` to `e` looks like this:

```
[a, b, c, d, e]
```

Internally a list is represented by recursively nesting the binary function symbol `l_()`. Its first argument represents the first element of the list and its second argument represents the rest of the list (i.e. *head* and *tail* in Prolog-speak, or *car* and *cdr* in Lisp-speak). The example list presented above looks like this in its functional representation.

```
l_(a, l_(b, l_(c, l_(d, l_(e, nil_)))))
```

Note the 0-ary function symbol `nil_` to represent the end of the list. This symbol can be used to represent an empty list outside of `l_()` terms as well. Due to the canonical mapping even open lists with no fixed length can be represented, e.g.

```
[a, b, c, d | Tail]
```

The variable `Tail` represents the currently not bound list, following the fourth element of this list. Note the "`|`"-symbol after `d`. This symbol separates the remainder of the list of the lists firsts element. When replacing "`|`" by "`,`" (yielding) represents a list of exactly five elements, whose first elements are fixed and whose fifth element is not yet bound.

```
l_(a, l_(b, l_(c, l_(d, Tail))))
```

In this case `Tail` may even also represent a list, but then the two example lists would still be different, since in this case the list `Tail` is the fifth element not the *cdr*. Assume `Tail` to be `[X, Y]`. Then the two lists would be

```
[a, b, c, d| Tail] = l_(a,l_(b,l_(c,l_(d, Tail))))
                   = l_(a,l_(b,l_(c,l_(d, l_(X,l_(Y,nil_))))))
[a, b, c, d, Tail] = l_(a,l_(b,l_(c,l_(d, l_(Tail, nil_)))))
                   = l_(a,l_(b,l_(c,l_(d, l_(l_(X,l_(Y,nil_)), nil_)))))
```

In particular, these two lists do not unify.

5.1. Examples

For list operations you may use the built-in features *concat* and *inlist* (see chapter "Built-in Features").

Define a new list: `p([a,b,c])`

Separate a list: `FORALL Head,Tail <- p([H |T]).`
the result will be: `Head=[a], Tail=[b,c]`

All elements of the list: `FORALL X <- inlist(X,[a,b,c]).`
the result will be: `X=[a,b,c]`

Merge lists: `FORALL X <- concat([a,b],[c,d],X).`
the result will be: `X=[a,b,c,d]`

Add elements to a list: `FORALL q([`**`a`**` |L]) <- p(L).`
`FORALL X <- q(X).`
the result will be: `X=[`**`a`**`,a,b,c]`

6. Built-in Features

Our implementation of F-Logic provides some built-in features like the built-in class number, several comparison predicates, the basic arithmetic operators, predicates for string handling, and aggregate functions.

6.1. List of Built-in Features

All available built-in features are listed below. Some selected features (*) will be explained further in the next chapters by short examples.

Built-in feature	Description	
altavista	searches for documents with altavista	altavista(<search string>, <beginning index>, <number of hits>, <language>, <url>, <title>, <description>, <number of found hits>)
between	returns true, if X is between A and B; for <number> and <string>	between(A,X,B)
classify -> set	learns a classifier for vectors: first argument is an id, second a feature list, third the class, fourth the resulting classifier	classify(<grouping id> <vector>, <class>, <classifier>)
concat*	succeeds, if <string3> is the concatenation of <string1> and <string2>	concat(<string1>,<string2>, <string3>)
concatlists	merges two lists into a third	concatlists(<list1>,<list2>, <list3>)
constant2string*	converts a function to a string and vice versa	constant2string(<function>, <string>)
contains	returns true, if <string2> is contained in <string1>	contains(<string1>, <string2>)
count*	Counts the values grouped to each key.	count(<key>,<value>, <number>)
cut*	returns the <string> n characters shorter	cut(<string>,<n>, <variable>)
dbaccess*	accesses a database	dbaccess(<tablename>, <access>, <dbtype>, <dbname>, <dbhost>)
dbaccessuser*	In addition to dbaccess, dbaccessuser allows to specify login name and password.	dbaccessuser(<tablename>, <access>, <dbtype>, <dbname>, <dbhost>, <login>, <passwd>)
directsub_	shows direct sub- or superconcept-relations	*directsub_(A,B).*
equal	returns true, if X = Y	equal(X,Y)
evaluable_	math functions: +, -, *, /, sin, cos, tan, asin, acos, ceil, floor, exp, rint, sqrt, round, max, min, power	i.e. (X + Y) * Z

greater	returns true, if X > Y	greater(X,Y)
greaterorequal	returns true, if X >= Y	greaterorequal(X,Y)
indexinlist	returns true, if index I of element X is in list L	indexinlist(X,L,I)
inlist	returns true, if X is in List L	inlist(X,L)
isconstant	returns true, if <arg> is a constant	isconstant(<arg>)
isint	returns true, if <arg> is an integer	isint(<arg>)
isnumber	returns true, if <arg> is a number	isnumber(<arg>)
isstring*	returns true, if <arg> is a string	isstring(<arg>)
less	returns true, if X < Y	less(X,Y)
lessorequal	returns true, if X <= Y	lessorequal(X,Y)
list*	Creates lists of values grouped to each key.	list(<key>,<value>,<list>)
maximum*	Determines the maximum of a set of numbers.	maximum(<key>,<value>, <maximum>)
minimum*	Determines the minimum of a set of numbers.	minimum(<key>,<value>, <minimum>)
multiply		
msindex*	accesses Microsoft's index server	msindex(<searchexpression>, <var1>,<var2>,<catalog>)
power	$x^y=z$	power(X,Y,Z)
predict	predicts a class for given feature vectors; first argument is the feature list, second the classifier, third the predicted class	predict(<feature vector>, <classifier>, <class>)
regexp*	regular expressions may be used to search in strings, <string2> is the result of the operation with <string1>	regexp("<regular expression>",<string1>, <string2>)
regexp1	regular expressions may be used to search in strings	regexp1("<regular expression>",<string>)
string2number*	converts a string to a number and vice versa	string2number(<string>, <number>)
tokenize*	breaks string into tokens at the delimiters	tokenize(<string>, <delimiters>,<variable>)
tokenizen*	breaks string into maximal n tokens at the delimiter	tokenizen(<string>,<n>, <delimiters>,<variable>)
tolower*	transforms all characters into lower characters	tolower(<string>, <variable>)
toupper*	transforms all characters into upper characters	toupper(<string>, <variable>)
unify	X unified with Y	unify(X,Y)
write, write2, ..., write6	prints the parameters	write(X1), write2(X1, X2), …, write6(X1, X2, X3, X4, X5, X6)

6.2. Numbers, Comparisons and Arithmetics

Objects denoting numbers or strings are different from other objects because the usual comparison operators are defined for them, as well as several arithmetic functions. Within a query or a rule body, relations between numbers or strings may be tested with the comparison predicates less, lessorequal, greater, greaterorequal. For example, the following query asks for the first three sons of Jacob:

```
FORALL X,Y,Z <- jacob[son@(X,Y)->>Z] AND less(Y,4).
```

(Example 6.1)

Comparison predicates are not allowed in rule heads.

The arithmetic operations addition +, subtraction -, multiplication * and division / are also implemented. Arithmetic expressions may be constructed in the usual way, even complex expressions, e.g., 3 + 5 + 2 or 3 + 2 * 3 are possible. By default, multiplication and division are prior to addition and subtraction. As usual, the evaluation order may be changed by using parentheses, e.g., (3 + 2) * 3.

The following example contains the query whether Jacob has three sons born consecutively by the same woman.

```
FORALL X,A,B,C,Z1,Z2,Z3 <-
  jacob[son@(X,A)->>Z1; son@(X,B)->>Z2; son@(X,C)->>Z3] AND
  (B is A+1) AND
  (C is A+2).
```

Additionally the following mathematical functions are implemented:

```
sin,cos,tan,asin,acos,ceil,floor,exp,rint,sqrt,round,max,min,pow
```

To test the equality of two terms the builtin equal(<term1>,<term2>) may be used, if both terms are ground. To unify two terms unify(<term1>,<term2>) may be used.

6.3. String handling

Analogously to numbers, there are several predefined operations for strings. These are provided by the built-in predicates which all have a fixed arity. Furthermore these predicates can only be used in rule bodies:

- `isString(<arg>)`

  is true, if <arg> is a string.

- `concat(<string 1 > , <string 2 > , <string 3 >)`

  succeeds if < string 3 > is the concatenation of < string 1 > and < string 2 >, e.g.,

  ```
  FORALL X <- concat("a","b",X).
  ```

  returns the binding X = "ab" whereas

  ```
  FORALL X <- concat("a",Y,"ab").
  ```

  leads to Y = "b".

- `cut(<string>,<n>,<variable>)`

  returns the <string> n characters shorter

- `tokenize(<string>,<delimiters>,<variable>)`

  breaks string into tokens at the delimiters

- `tokenizen(<string>,<n>,<delimiters>,<variable>)`

  breaks string into maximal n tokens at the delimiter

- `tolower(<string>,<variable>)`

  transforms all characters into lower characters

- `toupper(<string>,<variable>)`

  transforms all characters into upper characters

- Regular Expressions

  Regular expressions may be used to search in strings. For that purpose a regular expression predicate is available:

  ```
  regexp("<regular expression>",<string1>,<string2>)
  ```

  The first parameter defines the search string as regular expression. Regular expressions are defined as PERL regular expressions. The second parameter defines the string to search in, and the last parameter defines the resulting string, i.e. the region that matched the pattern, e.g.

  ```
  married("peter").
  married("tom").
  married("mary").
  ```

  The query "search forall married people with a "p" or "t" in their name:

  ```
  FORALL X <- married(X) and regexp("[pt]",X,Y).
  ```

  delivers X = "peter", Y = "p", X = "peter", Y = "t" and X = "tom", Y="t"

6.4. Type conversion

There are three different basic object types: numbers, strings and functional expressions. Numbers are denoted by any kind of number, integers and floating point numbers are not distinguished. Strings are enclosed in quotation marks. There exist two builtins to convert these basic types:

```
constant2string(<function>,<string>)
```

converts a function to a string and vice versa

```
string2number(<string>,<number>)
```

converts a string to a number and vice versa, e.g.,

```
FORALL X <- constant2string(f(3,a),X).
```

delivers "f(3,a)" as result

6.5. Index Server integration

The builtin msindex(<searchexpression>,<var1>,<var2>,<catalog>) allows to access Microsofts index server. *Searchexpression* specifies a search expression (look at the index server documentation) and the paths of documents satisfying the search expression are bound to the variable *var1*. Additionally a description of the contents of the files are delivered (if this feature is switched on for MS index server). *Catalog* specifies the index server catalog:

```
FORALL X,Y <-
   msindex("@Contents \"car\" and "motor\"",X,Y,System)
```

delivers the paths of all files containing "car" and "motor".

6.6. Access to databases

Ontobroker is able to access a lot of relational databases. This access may be used in F-Logic via the builtin:

```
dbaccess(<tablename>, <access>, <dbtype>, <dbname>, <dbhost>)
```

<dbtype> specifies the type of the database. At the moment there exist connectors to MSSQL, ORACLE, DB2, MYSQL.

<access> specifies the tables and the columns to access. It has the form

```
F(columnname,<variable>|<string,…, columnname, <variable>|<string)
```

If a string is given it is used for selection, a variable is instantiated with the corresponding value, e.g.

```
dbaccess(person,F(lastname, "peters", firstname,
          X),"mysql","db","localhost")
```

returns the firstname of “peters” from the table “person” in MYSQL database “db” on localhost.

```
dbaccessuser(<tablename>, <access>, <dbtype>, <dbname>, <dbhost>,
        <login>, <passwd>)
```

In addition to dbaccess, dbaccessuser allows to specify login name and password.

6.7. Other builtins

There is often the need to query direct sub- or superconcepts of a given concept. Instead of defining this in a logical sense, i.e. give me the maximal subconcepts a pseudo builtin *directsub_* is available. Thus a fact *A::B* leads to an instance *directsub_(A,B).* No facts should be added to *directsub_* because this has no influence on the is-a relation of concepts.

7. Path Expressions

Objects may be accessed directly by their object names. On the other hand it is also possible to navigate to them by applying a method to another object using path expressions. For example, the object described by the object name abraham may also be accessed by calling the method father on the object isaac. The corresponding constructs are called path expressions and look like this: isaac.father.[2] Example 6.1 shows that path expressions may also contain multi-valued methods and methods with parameters and that it is possible to chain up path expressions by successively applying methods to the result object of the preceding method call. At the end of each line you find the object name of the result object that is denoted by the path expression. The underlying object base is taken from the Examples 2.1 and 3.2:

```
isaac..son                             {jacob, esau}
jacob..son@(rachel,11)                 joseph
benjamin.father.father.mother          sarah
```

(Example 6.1)

Some path expressions may even denote objects in the object world which have no id-term as object name

Note, that the symbol to delimit components of path-expressions is the dot. The same symbol terminates rules, facts and queries. To be able to unambiguously parse F-logic programs, these symbols must become distinguishable. The heuristic used for this purpose demands the use of line terminators directly following the dot at the end of a rule, a fact or a query.

7.1. Nesting of Path Expressions and F-Molecules

As mentioned before, every (ground) path expression corresponds to an object. This object is called the object value of a path expression. Thus, it is possible to nest path expressions in F-molecules as well as in P-molecules in any position where id-terms are allowed in bodies of queries:

```
jacob..son@(rachel,11)[mother->rachel; father->jacob].
abraham[son->>{jacob.father}].
jacob[son@(joseph.mother)->>{benjamin}].
male(jacob.father).
```

(Example 6.2)

How parentheses affect the meaning of path expressions will become clear when looking at the next two examples:

```
jacob.(father.twice):person.
jacob.father.twice:person.
(jacob.father).twice:person.
```

(Example 6.3)

As path expressions are evaluated left to right, the second and third F-molecule are equivalent. In our context, however, they are not meaningful (evaluating to false)

[2] Note, that in earlier versions of F-Logic the symbol “#” was used instead of “.” to delimit constituents of parts.

because jacob.father is a person (isaac) and not a method, so that twice cannot be applied to this object.

Assume the object base defined by Example 2.1 is given.

```
jacob:(god.people).
jacob:god.people.
(jacob:god).people.
```

(Example 6.4)

In Example 6.4 the first F-molecule states that applying the method people to the object called god yields a class jacob belongs to. The second expression, which is equivalent to the third one, states that the object jacob is a member of the class god and denotes the application of the method people to the object jacob. However, the last two expressions are path expressions -not F-molecules- as they do not end with a specification list or an isa/subclass relationship (see Section 3.4).

Besides using path expressions instead of simple id-terms in F-molecules, it is also possible to nest path expressions and F-molecules the other way round: Intermediate objects in a path expression may have specification lists, turning them into F-molecules. As an example the path expression jacob.mother may be extended by specifying some properties for Jacob:

```
jacob:man[father->isaac].mother
```

In a rule body, this feature is useful to restrict the set of objects matching a path expression by selecting those with a certain property. For a formal analysis of such terms, see the reference semantics and object semantics of F-Logic expressions, e.g., in [LHL + 98].

7.2. Path Expressions in Queries

Path expressions in a rule body or query help the user to describe the information in question more concisely, avoiding auxiliary variables for intermediate results. If for example the grandfather of Isaac is requested, this query can be written as

```
FORALL X <- isaac.father[father->X].
```

instead of

```
FORALL X,Y <- isaac[father->Y] AND Y[father->X].
```

Path expressions may be eliminated from F-molecules in rule bodies or queries by decomposing the molecules into a set of F-atoms using new variables for the result values.

8. Rules and Queries

8.1. Rules

Based upon a given object base (which can be considered as a set a facts), rules offer the possibility to derive new information, i.e., to extend the object base intensionally. Rules encode generic information of the form: Whenever the precondition is satisfied, the conclusion also is. The precondition is called rule body and is formed by an arbitrary logical formula consisting of P- or F-molecules, which are combined by OR, NOT, AND, <-, -> and <->. A -> B in the body is an abbreviation for NOT A OR B, A <- B is an abbreviation for NOT B OR A and <-> is an abbreviation for (A->B) AND (B<- A). Variables in the rule body may be quantified either existentially or universally. The conclusion, the rule head, is a conjunction of P- and F-molecules. Syntactically the rule head is separated from the rule body by the symbol <- and every rule ends with a dot. Non-ground rules use variables for passing information between subgoals and to the head. Every variable in the head of the rule must also occur in a positive F-Atom in the body of the rule. Assume an object base defining the methods father and mother for some persons, e.g., the set of facts given in Example 2.1. The rules in Example 7.1 compute the transitive closure of these methods and define a new method ancestor:

```
FORALL X,Y X[ancestor->>Y] <- X[father->Y].
FORALL X,Y X[ancestor->>Y] <- X[mother->Y].
FORALL X,Y,Z X[ancestor->>Y] <- X[father->Z] AND Z[ancestor->>Y].
FORALL X,Y,Z X[ancestor->>Y] <- X[mother->Z] AND Z[ancestor->>Y].
man::person.
woman::person.
```

(Example 7.1)

Partial logical formulae in the rule body may be negated. E.g. the following rule computes for every person X all persons Y not related to X :

```
FORALL X,Y
  X[notrelated->>Y] <-
       X:person AND
       Y:person AND
       NOT X[ancestor->>Y] AND
       NOT Y[ancestor->>X].
```

(Example 7.2)

The following rule computes all persons X for whom an ancestor is known:

```
  FORALL X X[] <- EXISTS Y X:person[ancestor->>Y].
```

8.2. Queries

A query can be considered as a special kind of rule with empty head. The following query asks about all female ancestors of Jacob:

```
FORALL Y <- jacob[ancestor->>Y:woman].
```

(Example 7.3)

The answer to a query consists of all variable bindings such that the corresponding ground instance of the rule body is true in the object base. Considering the object base described by the facts of Example 2.1 and the rules in 7.1, the query 7.3 yields the following variable bindings:

```
Y = rebekah
Y = sarah
```

The following query computes the maximum value X for which p(X) holds. The rule body expresses that all Y for which p(Y) holds must be less or equal to the searched X.

```
p(1).
p(2).
p(3).
FORALL X <- p(X) AND FORALL Y (p(Y) -> lessorequal(Y,X)).
```

9. Namespaces in F-Logic

Without namespaces in F-Logic the names in different ontologies can not be distinguished from each other. For instance, a concept named "person" in ontology "car" is the same concept as the concept "person" in ontology "finance". Handling more than one ontology thus needs a mechanism to distinguish these concepts. This is the reason for introducing namespaces in F-Logic.

9.1. Declaring Namespaces

The namespace mechanism of F-Logic is similar to that of XML[3]. If you are familiar with XML-namespaces you will find namespaces in F-Logic easy to understand and use. You can introduce namespaces and associate aliases for them anywhere where a rule or query is allowed. This namespace declaration contains the XML-Element `<ns>` with a number of XML-attributes with the prefix "`ontons`". The scope of declared namespaces ends when the corresponding end-element `</ns>` is reached in the program, e.g.

```
<ns     ontons:cars="www.cars-r-us.tv"
        ontons:finance="www.financeWorld.tv"
        ontons="www.myDomain.tv/private">

  //Here the aliases "cars" and "finance" can be used.
  <ns   . . .>
        // Here inner aliases can be used.
        // Outer aliases are also visible if not redefined.
  </ns>

</ns>
```

(Example 8.1)

In our example three namespaces are declared. Each namespace must represent a valid URI according to RFC 2396 and can optionally be associated with an alias. The namespaces *www.cars-r-us.tv* and *www.financeWorld.tv* are associated with the aliases "`cars`" and "`finance`", respectively. The third namespace is not associated with an alias and thus, represents the default namespace.

As in XML these namespace declarations can be arbitrarily nested and aliases may be temporarily associated with other URIs by inner namespace declarations.

9.2. Using Namespaces in F-Logic Expressions

In F-Logic expressions every concept, method, object, function, and predicate may be qualified by a namespace. To separate the namespace from the name the "`#`"-sign is

[3] Note: Since version 2.2 of the F-Logic parser, the namespace mechanism has slightly changed. The new means for declaring namespaces is more powerful and closer to the spirit of XML.

ontoprise
- SEMANTICS FOR THE WEB -

used (as conventionally used in the RDF world and in HTML to locate local links inside a web page)[4]. The following examples use the name space declaration of 8.1:

```
cars#Car[  cars#driver => cars#Person;
           cars#passenger =>> cars#Person;
           cars#seats => NUMBER].
cars#Person[cars#name => STRING;
           cars#age => NUMBER;
           cars#drivingLicenseId => STRING].

finance#Bank[finance#customer => finance#Person;
           finance#location =>> finance#City].
finance#Person[cars#name => STRING;
           finance#monthlyIncome => NUMBER].

FORALL X,Y Y[finance#hasBank ->> X] <-
     Y:finance#Person AND
     X:finance#Bank[finance#customer ->> Y].

#me:cars#Person[cars#age -> 32].
#myBank:finance#Bank[finance#location ->> karlsruhe].
```

(Example 8.2)

The semantics of a namespace-qualified object is always a pair of strings, i.e. each object is represented by a URI (its namespace) and a local name. Thus `finance#Person` and `cars#Person` become clearly distinguishable. During parsing of the F-Logic program the aliases are resolved, such that the following pairs are constructed.

- `finance#Person` stands for `("www.financeWorld.tv", Person)`
- `cars#Person` stands for `("www.cars-r-us.tv", Person)`

In case no declared namespace URI is found for a used alias, the alias itself is assumed to represent the namespace of an F-Logic object. Because pure URIs conflict with the F-Logic grammar, literal namespaces, i.e. URIs, must be quoted, e.g.

- `"www.cars-r-us.tv"#Person` is equivalent to `cars#Person`

Note, that declared namespace URIs are taken literally, i.e. two URIs are equivalent only if they syntactically do not differ, e.g. *www.cars-r-us.tv* is ***not*** equivalent to *http://www.cars-r-us.tv*.

9.3. Querying for Namespaces

This mechanism enables users even to query for namespaces (URIs not aliases) and to provide variables in namespaces. For instance, the following query asks for all namespaces X that contain a concept “Person”.

```
FORALL X <- X#Person[].
```

(Example 8.3)

[4] Note: In earlier versions of this parser, the $-symbol was used instead. This must be changed.

The following inference rules integrate knowledge from different ontologies using the namespace mechanism (and a so called Skolem-function).

```
FORALL Name,Attr,Value
  person(Name)[Attr -> Value] <- EXISTS X
       X:finance#Person[Attr -> Value; finance#name -> Name] OR
       X:cars#Person[Attr -> Value; cars#name -> Name].
FORALL Name,Attr,Value
  person(Name)[Attr ->> Value] <- EXISTS X
       X:finance#Person[Attr ->> Value; finance#name -> Name] OR
       X:cars#Person[Attr ->> Value; cars#name -> Name].
```

(Example 8.4)

Predicate symbols are somehow special in F-Logic. Although they can contain namespaces they must not contain variables, i.e. all predicate symbol names must be ground.

Note that simple types like STRING or NUMBER must not be qualified by a namespace. Namespace qualifying is forbidden for builtins and arithmetic functions.

9.4. Default Namespaces

Objects that start with a #-symbol (i.e. have no declared namespace alias) refer to objects in the default namespace, in our example 8.1. the URI *www.myDomain.tv/private*. The default mechanism is used when a large number of objects, concepts, or methods from the same namespace are used, e.g.

- `#me` stands for `("www.myDomain.tv/private", me)`

Objects with an explicit reference to the current default namespace (i.e. starting with a #) must be clearly distinguished from objects without the leading #. The latter explicitly are defined to belong to the null namespace (or, if you like, to no namespace at all), e.g. in contrast to the defaulted line above

- `me` stands for `("", me)`

To state it clearly: The default namespace is not equivalent to the null namespace.

10. Compiler Switches

For internal purposes another markup element has been added to F-Logic. So-called compiler switches may influence the (not necessarily functional) behaviour of rules, facts, or queries. They look similar to namespace declarations since they too use XML-syntax.

```
forall X0 t(X0) <- tt(X0).
<compilerSwitch   materialize="on"
                  funkyFeature="off"
                  otherSwith=”27B”>
  forall X1 q(X1) <- qq(X1).
  <compilerSwitch materialize="off">
      forall X2 p(X2) <- pp(X2).
  </compilerSwitch>
  forall X3 r(X3) <- rr(X3).
</compilerSwitch>
forall X4 s(X4) <- ss(X4).
```

Example (9.1)

A compilerSwitch declaration contains of an XML-element with tag name `compilerSwitch` and a list of attribute=value pairs. The attributes and value must conform to XML conventions, esp. each value must be enclosed in single or double quotes. Arbitrary keywords can be used as attribute names, but only special keywords[5] will be recognized by the compiler or the inference engine when executing the rules or queries, that were marked by these switches. The same that has been said for namespace declarations about nesting and visibility holds for these compilerSwitch declarations. For example, the rule with `X2` sees `materialize=”off”`, while the rules with `X1` and `X3` see `materialize=”on”`.

[5] A list of reserved keywords for attributes or values is not yet available.

11. Imprint

Editor
ontoprise GmbH
Haid-und-Neu-Strasse 7
76131 Karlsruhe
Germany

Telefon	+49 (0) 721 / 66 57 910
Telefax	+49 (0) 721 / 66 57 911
Email	*info@ontoprise.de; support@ontoprise.de*
Internet	*http://www.ontoprise.de*

Karlsruhe, December 2002

12. References

[ABD+ 89] Malcolm Atkinson, Francois Bancilhon, David DeWitt, Klaus Dittrich, David Maier, and Stanley Zdonik. The object-oriented database system manifesto. In Intl. Conference on Deductive and Object-Oriented Databases (DOOD), pages 40-57. North-Holland/Elsevier Science Publishers, 1989.

[AHV 95] Serge Abiteboul, Richard Hull, and Victor Vianu. Foundations of Databases. Addison Wesley, 1995.

[CGT 90] S. Ceri, G. Gottlob, and L. Tanca. Logic Programming and Databases. Springer, 1990.

[CKW 93] W. Chen, M. Kifer, and D.S. Warren. HiLog: a foundation for higher-order logic programming. Journal of Logic Programming, 15(3):187-230, 1993.

[FLU 94] Jürgen Frohn, Georg Lausen, and Heinz Upho . Access to objects by path expressions and rules. In Intl. Conference on Very Large Data Bases (VLDB), pages 273-284, 1994.

[FHK] J. Frohn, R. Himmeröder, P. Kandzia, C. Schlepphorst. How to Write F-logic Programs in FLORID - A Tutorial for the Database Language F-logic. http://www.informatik.uni-freiburg.de/~dbis/florid/

[KLW 95] Michael Kifer, Georg Lausen, and James Wu. Logical foundations of object-oriented and frame-based languages. Journal of the ACM, 42(4):741-843, 1995.

[LHL+ 98] Bertram Ludäscher, Rainer Himmeroder, Georg Lausen, Wolfgang May, and Christian Schlepphorst. Managing semistructured data with orid: A deductive object-oriented perspective. Information Systems, 23(8):589-612, 1998.

[Liu 96] M. Liu. ROL: A typed deductive object base language. In Intl. Conference on Database and Expert Systems Applications (DEXA), 1996.

[Ull 89] Jeffrey D. Ullman. Principles of Database and Knowledge-Base Systems, volume 2. Computer Science Press, New York, 1989.