Tutorial Series

How to write

F – Logic - Programs

covering OntoBroker® Version 5.x

April 2007

F-Logic Tutorial

Document Version: Tutorial_Flogic_2007_04_27_en

How to Write F-Logic Programs

A Tutorial for the Language F-Logic[1] covers OntoBroker® Version 5.x [2]

---

1 This tutorial refers to the syntactic and semantic capabilities of F-Logic as implemented by ontoprise GmbH (OntoBroker Version 5.x).

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 modeling 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 V5.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 built-ins, 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:

```
/* schema facts */
man::person.
woman:person.
person[hasHasFather=>man].
person[hasHasMother=>woman].
person[hasSon=>>man].
person[hasDaughter=>>woman].

/* facts */
abraham:man.
sarah:woman.
isaac:man[hasHasFather->abraham; hasHasMother->sarah].
ishmael:man[hasHasFather->abraham;
            hasHasMother->hagar:woman].
jacob:man[hasHasFather->isaac; hasMother->rebekah:woman].
esau:man[hasHasFather->isaac; hasMother->rebekah].

/* rules consisting of a rule head and a rule body */
FORALL X,Y X[hasSon->>Y] <- Y:man[hasHasFather->X].
FORALL X,Y X[hasSon->>Y] <- Y:man[hasMother->X].
FORALL X,Y X[hasDaughter->>Y] <- Y:woman[hasHasFather->X].
FORALL X,Y X[hasDaughter->>Y] <- Y:woman[hasMother->X].

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

(Example 2.1)

The first section of this example consists of a set of schema facts to indicate that `man` and `woman` are subclasses of `person` and that every `person` can have one `hasHasFather` and `hasMother`, which are `men` or `women`, respectively, and that they can have `son`s and `daughter`s, again, `men` or `women`, respectively. The second section titled facts, describe that some people belong to the classes `man` and `woman` and give information about the `hasHasFather` and `hasMother` relationships among them. According to the object-oriented paradigm, relationships between objects are modeled by method applications, e.g., applying the method `hasHasFather` 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 third section 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 intentional information.

The final section of the example 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 query could be written as a conjunction of simple sub-goals:

```
FORALL X,Y
<- X:woman AND X[hasSon->>Y] AND Y[hasHasFather->abraham].
```

Methods and classes also are objects, cf. 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 "_" of the ASCII character set. Examples for object names are `abraham`, `man`, `hasDaughter`, 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”. Id-terms can be enclosed in a pair of single quotes in order to use otherwise illegal characters, e.g. `‘Müller’` is a legal term while `Müller` is not.

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 `hasHasFather` and `hasSon` are object names just like `isaac` and `abraham`.

Variables may also 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[hasSon->>{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[hasSon@(leah)->>
            {reuben, simeon, levi, judah, issachar, zebulun};
      hasSon@(rachel)->>{joseph, benjamin};
      hasSon@(zilpah)->>{gad, asher};
      hasSon@(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[hasSon@(leah,1)->>reuben;
      hasSon@(leah,2)->>simeon;
      hasSon@(leah,3)->>levi;
      hasSon@(leah,4)->>judah;
      hasSon@(bilhah,5)->>dan;
      hasSon@(bilhah,6)->>naphtali;
      hasSon@(zilpah,7)->>gad;
      hasSon@(zilpah,8)->>asher;
      hasSon@(leah,9)->>issachar;
      hasSon@(leah,10)->>zebulun;
      hasSon@(rachel,11)->>joseph;
      hasSon@(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[hasSon->>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[hasSon->>{jacob, esau}].
<- isaac[hasSon->>jacob].
<- isaac[hasSon->>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 reflexive) 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

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[hasFather->abraham; hasSon->>{jacob,esau}].
```

(Example 3.4)

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

```
isaac:man.
isaac[hasFather->abraham].
isaac[hasSon->>jacob].
isaac[hasSon->>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 multi-valued, 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[hasSon->>{jacob,esau}].
```

(Example 3.5)

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

(Example 3.6)

```
isaac[hasSon->>jacob].
isaac[hasSon->>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[hasFather=>man].
person[hasDaughter=>>woman].
man[hasSon@(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[hasFather=>{man, person}].
```

(Example 3.9)

```
person[hasFather=>man].
person[hasFather=>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[hasSon@(woman)=>>man].
```

(Example 3.11)

```
man[hasSon@(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[hasFather->abraham;
      hasMother->sarah].
jacob:man[hasFather->isaac;
          hasSon@(rachel)->>{joseph, benjamin}].
man::person[hasSon@(woman)=>>{man, person}].
```

(Example 3.13)

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

```
isaac[hasFather->abraham].
isaac[hasMother->sarah].
jacob:man.
jacob[hasFather->isaac].
jacob[hasSon@(rachel)->>joseph].
jacob[hasSon@(rachel)->>benjamin].
man::person.
man[hasSon@(woman)=>> man].
man[hasSon@(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[hasFather->abraham:man[hasSon@(hagar:woman)->>ishmael];
      hasMother->sarah:woman].
jacob:(man::person).
jacob[(hasFather:method)->isaac].
```

(Example 3.15)

The equivalent set of F-atoms is:

```
isaac[hasFather->abraham].
abraham:man.
abraham[hasSon@(hagar)->>ishmael].
hagar:woman.
isaac[hasMother>>sarah].
sarah:woman.
man::person.
jacob:man.
jacob[hasFather->isaac].
hasFather:method.
```

3.4. F-molecules without any Properties

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

```
thing[].
```

In this example a class thing is considered that does not have any properties (yet).

If we use a similar expression that consists solely of an object name (without the empty pair of brackets, i.e. `thing.`), 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 modeling. For example, the information given in the first three P- atoms in 4.1 could also be expressed as follows:

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

(Example 4.2)

Note that the expressions in the two examples above are alternative representations. They *cannot be used in a mixed manner*, i.e. a query for `married(X,Y)` does not retrieve any results for facts like `isaac[marriedTo -> rebekah]`.

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.

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([Head|Tail]).
```

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, X=b, X=c
```

Merge lists:

```
FORALL X <- concatlists([a,b],[c,d],X).
```

the result will be:

```
X=[a,b,c,d]
```

Add elements to a list:

```
FORALL L q([a|L]) <- p(L).
FORALL X <- q(X).
```

the result will be:

```
X=[a,a,b,c]
```

An extended example calculating a graph using lists is this:

```
// the edges of a graph between two knots
edge(a,b).
edge(b,c).
edge(a,d).
edge(d,e).
edge(e,f).

// add each edge to a path containing two knots
FORALL X,Y path([Y,X]) <- edge(X,Y).

// add every new edge to the appropriate path
FORALL L,H1,H2,T path([H1|L]) <- path(L) and unify(L,[H2,T])
and edge(H2,H1).
```

This query outputs all paths of the graph:

```
FORALL L <- path(L).
```

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.

All available built-in features are listed at Appendix A below. Some selected features will be explained now by short examples.

6.1. 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[hasSon@(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[hasSon@(X,A)->>Z1;
          hasSon@(X,B)->>Z2;
          hasSon@(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 built-in `equal(<term1>,<term2>)` may be used, if both terms are ground. To unify two terms `unify(<term1>,<term2>)` may be used.

6.2. 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 for all 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"

6.3. 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 built-ins 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

6.4. Aggregations

Aggregations are built-ins which have a set of values as a domain. Aggregations must not occur in rule cycles and the tackled values must not occur in the head of rules.

- `count(<key>,<value>,<number>)`

  Counts the values grouped to each key. E.g.:

  ```
  p(a,1).
  p(a,5).
  p(b,3).
  FORALL X,Z <- EXISTS Y p(X,Y) and count(X,Y,Z).
  ```

  Delivers the answers

  ```
  X = a
  Z = 2
  X = b
  Z = 1
  ```

- `list(<key>,<value>,<list>)`

  Creates lists of values grouped to each key.

- `minimum(<key>,<value>,<minimum>)`

  Determines the minimum of a set of numbers.

- `maximum(<key>,<value>,<maximum>)`

  Determines the maximum of a set of numbers.

6.5. Index Server integration

The built-in `msindex(<searchexpression>,<var1>,<var2>,<catalog>)` allows to access Microsoft's 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 is 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

There are two different ways to access external databases. Both of them will be described in the following chapters.

6.6.1. DBAccess

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

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

`<dbtype>` specifies the type of the database. At the moment there exist connectors to MSSQL and ORACLE.

`<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),
         "mssql","db","localhost")
```

returns the first name of "peters" from the table "person" in MSSQL database "db" on localhost.

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

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

6.6.2. SQLExecute

Description for the built-in `SQLExecute`

The functionality of this built-in is to allow a user to directly formulate SQL statements and to use the result for further modeling. This may help to tune performance in projects as special restrictions on the queries can be used that otherwise would be evaluated during the inferencing process.

```
Predicate  = "sqlexecute"
Arity      = 7
Parameters = "SQL statement", "list of variables",
             "db server type", "name of database",
             "host and port", "username", "password"
```

- "SQL statement"
  - the SQL statement has to contain a valid SQL statement and the number of values selected has to be equal to the number of variables given in the list
  - example: `SELECT ID, NAME FROM PERSON`
- list of variables
  - the list of variables has to contain the variables in which the result of the query is expected
  - example: `[X, Y]`
- db server type
  - the server type is one of the supported database types
  - `oracle` or `mssqlserver2000` or `db`
- name of database
  - specifies the name of the database that has to be accessed
  - example: `pubs`
- host and port
  - specifies the host and the port where the database is running
  - example: `localhost:1433`
- username
  - gives a user id to be used for access
  - example: `demo`
- password
  - gives the password for the user that is known to the db
  - example: `demo`

A complete rule would be for example:

```
FORALL X,Y
  X:Person[hasName->>Y]
<-
  sqlexecute("SELECT ID, NAME FROM PERSON", [X,Y],
             "mssqlserver2000", "pubs", "localhost:1433",
             "demo", "demo").
```

Or, using a restriction:

```
FORALL X,Y
  X:Adult[hasName->>Y]
<-
  sqlexecute("SELECT ID, NAME FROM PERSON WHERE AGE > 18",
[X,Y],
             "mssqlserver2000", "pubs", "localhost:1433",
             "demo", "demo").
```

In the second case the execution would be more performant as the selection is done in the database. This is currently not possible with `dbaccessuserdata` as only the `equal` restriction can be taken into account there. A major restriction of this built-in is at the current stage of development that bindings for the variables cannot be taken into account. This will be changed in future releases.

6.7. Other built-ins

There is often the need to query direct sub- or super-concepts of a given concept. Instead of defining this in a logical sense, i.e. give me the maximal subconcept a pseudo built-in `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. Rules and Queries

7.1. 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-Atoms 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-Atoms 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 sub-goals and to the head. Every variable in the head of the rule must also occur in a positive F- or P-Atom in the body of the rule. Assume an object base defining the methods `hasFather` and `hasMother` 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 `hasAncestor`:

```
FORALL X,Y X[hasAncestor ->>Y] <- X[hasFather->Y].
FORALL X,Y X[hasAncestor ->>Y] <- X[hasMother->Y].
FORALL X,Y,Z X[hasAncestor ->>Y] <-
       X[hasFather->Z] AND Z[hasAncestor ->>Y].
FORALL X,Y,Z X[hasAncestor ->>Y] <-
       X[hasMother->Z] AND Z[hasAncestor ->>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[hasAncestor ->>Y] AND
       NOT Y[hasAncestor ->>X].
```

(Example 7.2)

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

```
FORALL X personWithAncestors(X) <-
       EXISTS Y X:person[hasAncestor ->>Y].
```

Rules can also be identified by rule names, e.g. `ancestorFather` in the following rule:

```
RULE ancestorhasFather:
FORALL X,Y  X[hasAncestor ->>Y]  <-  X[hasFather->Y].
```

7.2. Queries

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

```
FORALL Y <- jacob[hasAncestor ->> 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)).
```

(Example 7.4)

The result will be:

```
X = 3.0
```

7.3. Range Restriction

All variables in a rule or a query must be *range restricted*, i.e. for each variable one or more of the following conditions must hold:

1. The variable occurs in a positive (not negated) body literal which is not a built-in-literal (simple built-in, connector built-in, or aggregate).

2. The variable is bound top-down by constants in the query or in connected rules.

3. A variable is bound by the output of a built-in-literal and all input-arguments of the built-in are range-restricted or ground. Which arguments are input and output of a built-in is defined by the signatures of the built-in.

Let us illustrate the above topics in examples. For the following rule the variables all variables are bound and thus the query is range restricted. Variables *X* and *Y* are bound because they occur in the positive literal `p(X,Y)` (condition 1). Built-in *add* has the signature *{number,number,variable}* which means the first two (input) arguments must be bound to numbers and the third can be a variable and is thus an output parameter. Thus variable `Z` is bound because it is the output variable of built-in *add* and all input variables of *add* are bound (condition 3).

```
FORALL X,Y,Z <- p(X,Y) AND add(X,Y,Z).
```

In the next query and rule the variable `Y` of the rule is bound top-down by the constant *5* in the query (condition 2). Variable `X` is again bound by the positive literal `q(X)` (condition 1) and thus `Z` is bound as an output parameter of the *add* built-in (condition 3).

```
FORALL X,Y <- p(X,5).
FORALL X,Y,Z p(X,Y) <- q(X) AND add(X,Y,Z).
```

In the next rule there is a transitive dependency of variable bindings through built-ins given. Thus also `U` is range-restricted (condition 1 and condition 3).

```
FORALL X,Y,Z,U p(X,Y) <-
       q(X,Y) AND add(X,Y,Z) and add(Z,Y,U).
```

The above mentioned conditions have the consequence that a rule or query is not range restricted if a variable occurs in a negated literal only. Rules which have variables occurring in the head only are obscure because these variables must be bound top-down in every case for the rule to be range restricted.

7.4. Quantifier Scoping

The quantifiers `FORALL` and `EXISTS` introduce variables in rules and queries. Syntactically variables, like `X` or `Y`, do not differ to constant symbols in F-Logic, thus, the requirement for explicit declaration with quantifiers. In the unusual situation where there is a conflict between a used variable and an existing constant, it is important to know the scope, i.e. the lifetime of variables. To illustrate the notion of variable scopes we present an example formula where all variables are underlined and all constants are not.

```
FORALL X,Y p(X,Y) <- r(U,Y) AND EXISTS U q(U,Y).
FORALL X,Y p(X,Y) <- EXISTS U q(U,Y) AND r(U,Y).
FORALL X,Y p(X,Y) <- (EXISTS U q(U,Y)) AND r(U,Y).
```

The rule-of-thumb is that each quantifiers binds variables till the end of the complete formula. You can overwrite this pattern only by introducing parenthesis and, thus, explicitly introducing a new scope for the quantifier. Note: the semantics of the first and third formula above is equivalent (the `U` in

the `r` predicate is a constant), whereas formula two is different (here, the `U` in the `r` predicate is bound by the `EXISTS` quantifier).

8. 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. Thus, ontoprise introduced the notion of namespaces to F-Logic, which enabled RDF-like identifiers for objects, classes or properties.

8.1. Declaring Namespaces

The most recent OntoBroker Version 5.0 introduces a new syntax for declaring namespace in F-Logic[3]. The F-Logic file can contain namespace declarations that associate namespace URIs with aliases, that can be used to formulate namespace terms in a more concise way.

```
:- prefix cars="http://www.cars-r-us.tv/".
:- prefix finance="http://www.financeWorld.tv/".
:- prefix xsd="http://www.w3.org/2001/XMLSchema#".
:- prefix ="http://www.myDomain.tv/private#".
```

(Example 8.1)

The code above declares four namespaces. It associates two of them with shortcuts (or aliases) and the third is declared as the default-namespace. Each namespace must represent a valid URI according to RFC 2396 and must end with either “#”, “/” or “:”. This is essential since these characters mark the separator between the namespace and the local part of an identifier. Esp. when exporting to RDF/OWL or reading from these formats, this convention is important.

8.2. Using Namespaces in F-Logic Expressions

In F-Logic expressions every concept, method, object, and function may be qualified by a namespace. To separate the namespace from the name the “#”-sign is used (as conventionally used in the RDF world and in HTML to locate local links inside a web page). The following examples use the name space declaration of 8.1:

```
cars#Car[   cars#driver => cars#Person;
       cars#passenger =>> cars#Person;
       cars#seats => xsd#integer].
```

[3] The former XML-like mechanism is still supported but is deprecated now, so, please abstain from <ns> and use the prefix notation instead. Cf. The appendix in Section 11 for a brief summary of the old namespace declaration.

```
cars#Person[cars#name => xsd#string;
       cars#age => xsd#integer;
       cars#drivingLicenseId => xsd#string].

finance#Bank[finance#customer => finance#Person;
       finance#location =>> finance#City].
finance#Person[cars#name => xsd#string;
       finance#monthlyIncome => xsd#integer].

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

#me:cars#Person[cars#age -> 28].
#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 `ns_("http://www.financeWorld.tv/", Person)`
- `cars#Person` stands for `ns_("http://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.

URIs can also be used directly in namespace terms, i.e. the use of aliases is optional. Because the URI syntax greatly conflicts with the F-Logic grammar, literal namespaces must be quoted, e.g.

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

As described above, the ending character of namespaces is important for compatibility with RDF and OWL. In case where the namespace does not end with one of the characters “/”, “#” or “:” the F-Logic parser automatically adds a “#” at the end of the namespace. This patch is applied to literal namespaces as well as to the namespace declaration.

8.3. 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)

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)

8.4. The Default Namespace

Objects that start with a #-symbol (i.e. do not use a declared namespace alias) refer to objects in the *default namespace*, in our example 8.1. the URI `http://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 `ns_("http://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 no namespace*, i.e. the two terms `#me` and `me` do not unify.

9. Modules

In software engineering modules have been invented to reduce complexity. Closely correlated and interwoven things are packaged in a common module, while only loosely coupled things reside in different modules. The communication between modules should be minimal. These principles have been transferred to knowledge bases. Rules and facts describing a closely related part of the domain reside in one module. Thus, an entire knowledge base can be split up into different modules each containing closely related statements about the domain. In some sense this concept is orthogonal to the concept of namespaces. Identifiers with different namespaces may be addressed in one and the same module. On the other hand identifiers are global over all modules which means that an object with identifier `x` is the same object in all modules. Thus, modules do not separate objects, but statements about objects. Both, ground statements (statements without a variable) like those mentioned in Section 3 as well as rules and queries as mentioned in Section 7 are assigned to modules.

Each F-Logic file must contain ground statements from exactly one module. The (default) module can be defined at the beginning of the file:

```
:- module = module1.
```

The name of a module can be an arbitrary ground term, i.e. a constant, a functional term or a namespace term. In the example above we chose a constant. When using a namespace term and an appropriate alias exists, it can be used for the declaration of the module as well, e.g.

```
:- prefix a="http://www.exmple.org/sample#".
:- module =a#sample.
```

The (default) module is assumed to for all subsequent ground facts, esp. if they do not declare the module explicitly. The notation for assigning a module to a ground fact looks like this.

```
abraham:man@module1.
sarah:woman@module1.
isaac:man[hasFather->abraham; hasMother->sarah]@module1.
```

Since each file can contain only statements from one module the module references can be omitted without changing the semantics.

Module references are more important with rules and queries. As well as ground statements can be assigned to modules, rules can be assigned to modules.

```
RULE ancestorHasFather@module1:
FORALL X,Y
  X[hasAncestor ->>Y] <- X[hasFather->Y].
```

expresses that the rule named `ancestorHasFather` resides in `module1`. This implies that all body and head literals are also assumed to come from this module (unless otherwise specified). The above rule, thus, is equivalent to:

```
RULE ancestorHasFather@module1:
FORALL X,Y
  X[hasAncestor ->>Y]@module1 <- X[hasFather->Y]@module1.
```

Each literal in a rule body and rule head can use its own module. For body literals this means that the reasoner tries to search for the literal in the mentioned module. For head literals this means, that the new fact is asserted to hold true in its module. A complex example could be

```
RULE ancestorHasFather:
FORALL X,Y
  X[hasAncestor->>Y]@module2
<-
  X[hasFather->Y]@module1.
```

This rule expresses that `module2` holds the (derived) fact that `X` has an ancestor `Y` if it is true in `module1` that the same `X` has the same `Y` as father.

Since module names are terms, it is even possible to use variables as module names in rule bodies.

```
RULE ancestorhasFather:
FORALL X,Y,Z
  X[hasAncestor ->>Y] <- X[hasFather->Y]@Z.
```

This rule searches for statements about fathers of `X` in every module and asserts that `Y` is an ancestor of `X` in the default module.

In our previous examples we used only constants for module names. In addition to that complex module names, i.e. module names consisting of functions are allowed too:

```
module(Arg1,...,Argn)
```

If `Arg1, … Argn` contain variables each binding leads to a separate module name, e.g., `module(a,f(b))`.

It is good practice to use namespace terms as module names, and thus creating a universally unique identifier for the modules, e.g. with a declaration such as this:

```
:- prefix a="http://www.exmple.org/sample#".
:- module =a#sample.
```

10. Appendix A: List of Built-ins

Built-in feature	Description	Syntax	Example
+	add two numbers	+(<integer1>,<integer2>)	FORALL X <- X is +(3,2). FORALL X <- X is (3.0+2.0).
-	sub two numbers	-(<integer1>,<integer2>)	FORALL X <- X is (3.0-2.0).
*	multiplies two numbers	*(<integer1>,<integer2>)	FORALL X <- X is (3.0*2.0).
/	divides two numbers	/(<integer1>,<integer2>)	FORALL X <- X is (3.0/2.0).
%	returns the rest of the division of two numbers	%(<integer1>,<integer2>)	FORALL X <- X is (3.0%2.0).
+	multiplies a number with (1)	+(<integer>)	FORALL X <- X is +(1).
-	multiplies a number with (-1)	-(<integer>)	FORALL X <- X is -(1).
Abs	returns the absolute value of a value	abs(<integer>)	FORALL X <- X is abs(-1).
Acos	returns the arc cosine of an angle	acos(<integer>)	FORALL X <- X is acos(0.8).
Add	add two numbers to a third one	add(<integer1>,X,<integer2>) add(<integer1>,<integer2>,<variable>)	FORALL X <- add (1,X,5). FORALL X <- add (1,4,X).
Addvector	add two vectors of numbers		FORALL X <- addvector(v_(2.3,1.7,1.3),v_(3.7,2.1,1.8),X).
Asin	returns the arc sine of an angle	asin(<integer>)	FORALL X <- X is asin(0.8).
Between	returns true, if X is between A and B; for <number> and <string>	between(A,X,B)	FORALL X <- equal(X,"true") and between(3,4,5).
Ceil	returns the smallest value that is not less than the argument and is equal to a mathematical integer	ceil(<integer>)	FORALL X <- X is ceil(2.3).
Chemequal	are two chemical formulas equal?	chemequal(<formula>,<forumula>)	FORALL X <- equal(X,"true") and chemequal("H2(NO)4","H2N4O4").
Chemparser	parses a chemical formula like H2(NO)4	chemparse(<formula>)	FORALL X <- chemparse("H2(NO)4",X).
chemparser1	parses a chemical formula like H2(NO)4	chemparse1(<formula>)	FORALL X <- chemparse1("H2(NO)4",X).
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>)	see predict
Comparevectors	returns indices where vectors are different	comparevectors(<vector>,<vector>,X)	FORALL X <- comparevectors(v_(1,2,0,2),v_(1,0,2,2),X).
Concat	succeeds, if <variable> is the concatenation of <string1> and <string2>	concat(<string1>,<string2>, <variable>)	FORALL X <- concat("a","b",X). FORALL X <- concat("a",X,"ab").
concatlists	merges two lists into a third	concatlists(<list1>,<list2>, <list3>)	FORALL X <- concatlists([a,b,1,"hello"],[3],X).
constant2string	converts a function to a string and vice versa	constant2string(<function>,<string>)	FORALL X <- constant2string(f(3,a),X).
Contains	returns true, if <string2> is contained in <string1>	contains(<string1>, <string2>)	FORALL X <- contains("xyyz","yz") AND equal(X,"true").
Cos	returns the trigonometric	cos(<integer>)	FORALL X <- X is cos(0).

	cosine of an angle		
Count	Counts the values grouped to each key.	count(<key>,<value>, <number>)	needed facts: p(a,1). p(a,5). p(b,3). FORALL X,Z <- EXISTS Y p(X,Y) and count(X,Y,Z).
Cut	returns the <string> n characters shorter	cut(<string>,<n>, <variable>)	FORALL X <- cut("xyz",2,X).
Datecompare	computes the difference in dates between two dates	datecompare(<date>,<date>,<variable>)	FORALL X <- datecompare("23.08.2003","21.08.2003",X).
Dbaccess	Specification to a database access	dbaccess(<tablename>, <access>, <dbtype>, <dbname>, <dbhost>, <login>, <passwd>)	FORALL VAR1 <- (dbaccess("projects",F(project,VAR1),"mssqlserver2000","ontoproject","bunsen:1433").
dbaccessuser	In addition to dbaccess, dbaccessuser allows to specify login name and password.	dbaccessuser(<tablename>, <access>, <dbtype>, <dbname>, <dbhost>, <login>, <passwd>)	FORALL VAR1 <- (dbaccessuser("projects",F(project,VAR1),"mssqlserver2000","ontoproject","bunsen:1433","zope","zope")).
Deriveacid	substracts a H atom of an acid formula	deriveacid(<acid formula>,X)	FORALL X <- deriveacid("H2SO4",X).
Diffvector	subtracts vector2 from vector1 if vector2-vector1 is positive	diffvector(<vector1>,<vector2>,<variable>)	FORALL X <- diffvector(v_(2,2,2),v_(1,1,1),X).
directisa_		directisa_(X,Y)	FORALL X,Y<-directisa_(X,Y).
directsetatttype_		directsetatttype_(X,Y)	FORALL X,Y <- directsetatttype_(X,Y).
directsub_	shows direct sub- or superconcept-relations	directsub_(X,Y).	FORALL X,Y <-directsub_(X,Y).
Documentvector	creates a wordvector of a document	docvector(<url>,<variable>).	FORALL X <- docvector("http://www.ontoprise.de",X).
Dotproduct	calculates the scalar/dot procuct of 2 vectors	dotprod(<vector1>,<vector2>,<variable>)	FORALL X <- dotprod(v_(3,2,1),v_(1,2,3),X).
Durationcompare	computes the overlap of two time periods	overlaps(<date>,<date>,<date>,<date>,X)	FORALL X <- overlaps("13.08.2003","15.08.2003","14.08.2003","16.08.2003",X).
Equal	returns true, if X = Y	equal(X,Y)	FORALL X <- equal(3,3) AND(X is 1).
Escape	escapes characters in a string	escape(<string>,<variable>)	FORALL X <- escape("hello",X).
Exp	returns euler´s number raised to the power of a value	exp(<integer>)	FORALL X <- X is exp(1).
Fill	fills a string at the front with given letters to the given length	fill(<string>,<letter>,<length>,<variable>)	FORALL X <- fill("xy","a",4,X).
Floor	returns the largest value that is not greater than the argument and is equal to a mathematical integer	floor(<integer>)	FORALL X <- X is floor(2.5).
Getvalue	value of ith component of a vector	getvalue(<vector>,<index>,<variable>)	FORALL X <- getvalue(v_(1,0,2),2,X).
ggt	largest common divisor	ggt(<integer1>,<integer2>,<variable>)	FORALL X <- ggt(10,15,X).
Greater	returns true, if X > Y	greater(X,Y)	FORALL X <- greater(3,2) AND (X is 1).
Greaterorequal	returns true, if X >= Y	greaterorequal(X,Y)	FORALL X <- greaterorequal(3,3) AND equal(X,"true"). FORALL X <- greaterorequal(3,2) AND

			equal(X,"true").
Indexinlist	returns true, if index I of element X is in list L	indexinlist(X,L,I)	FORALL X <- indexinlist(X,[3,1,"ontoprise",2],2).
Inlist	returns true, if X is in List L	inlist(X,L)	FORALL X <- inlist(X,[2,1,"ontoprise",0]).
isa_			FORALL X,Y <- isa_(X,Y).
Isconstant	returns true, if <arg> is a constant	isconstant(<arg>)	FORALL X <- equal(X,"true") and isconstant(a).
Isint	returns true, if <arg> is an integer	isint(<arg>)	FORALL X <- equal(X,"true") and isint(6).
Isnumber	returns true, if <arg> is a number	isnumber(<arg>)	FORALL X <- equal(X,"true") and isnumber(6.1).
Isstring	returns true, if <arg> is a string	isstring(<arg>)	FORALL X <- equal(X,"true") and isstring("hello").
Length	determines the length of a string or vector	length(<vector>,<variable>)	FORALL X <- length(v_(1,2,3),X).
Less	returns true, if X < Y	less(X,Y)	FORALL X <- equal(X,"true") and less(2,3).
Lessolve	solve linear equation system	lessolve(<matrix>,<variable>)	FORALL X <- lessolve(m_(v_(1,2),v_(2,4)),X).
Lessorequal	returns true, if X <= Y	lessorequal(X,Y)	FORALL X <- equal(X,"true") and lessorequal(2,3). FORALL X <- equal(X,"true") and lessorequal(3,3).
List	Creates lists of values grouped to each key.	list(<key>,<value>,<list>)	needed facts: see count FORALL X,Z <- EXISTS Y p(X,Y) and list(X,Y,Z).
Listsortbyindex	sorts a given list by indexes given by another list	listsort(<list1>,<list2>,<variable>)	FORALL X <- listsort([3,5,4],[36,112,45],X).
Log	returns the natural logarithm of a value	log(<integer>)	FORALL X <- X is log(1).
Matrixbycolumns	creates a matrix of a list of columns	matrixbycols(<list of columns>,<variable>)	FORALL X <- matrixbycols([v_(1,2,3),v_(2,1,3)],X).
matrixbyrows	creates a matrix out of a list of rows	matrixbyrows(<list of rows>,<variable>)	FORALL X <- matrixbyrows([v_(1,2,3),v_(2,1,3)],X).
Matrixt	transpones a matrix	matrixt(<matrix>,<variable>)	FORALL X <- matrixt(m_(v_(1,2),v_(3,4)),X).
Max	returns the greater of two values	max(<integer>,<integer>)	FORALL X <- X is max(-1,2).
Maximum	Determines the maximum of a set of numbers.	maximum(<key>,<value>,<maximum>)	needed facts : see count FORALL X,Z <- EXISTS Y p(X,Y) and maximum(X,Y,Z).
Min	returns the smaller of two values	min(<integer>,<integer>)	FORALL X <- X is min(-1,2).
Minimum	Determines the minimum of a set of numbers.	minimum(<key>,<value>,<minimum>)	needed facts :see count FORALL X,Z <- EXISTS Y p(X,Y) and minimum(X,Y,Z).
Multiply		multiply(<number>,<number>,<variable>)	FORALL X <- multiply(2,3,X).
multvector	multiplies 2 vectors	multvector(<vector1>,<vector2>,<variable>)	FORALL X <- multvector(v_(3,2,1),v_(1,2,3),X).
Multvectormatrix	multiplies a vector with a matrix	multvm(<vector>,<matrix>,<variable>)	FORALL X <- multvm(v_(1,2),m_(v_(1,3),v_(2,4)),X).
Near	is number1 near number2 with a	near(<integer>,<variable>)	FORALL X <- near(3,3.01) and equal(X,"true").

	tolerance from 1% of number2?		
`Nearest`	computes maximum of a set of values	nearest(<grouping id>,<value>,<nearest_to>,<variable>)	needed facts: see `count` FORALL X,Z <- EXISTS Y p(X,Y) and nearest(X,Y,2,Z).
`Newvector`	creates a vector v with n values x	newvector(n,<value>,<variable>)	FORALL X <- newvector(3,1,X).
`number2roman`	converts numbers to roman numbers	number2roman(<number>,<variable>)	FORALL X <- number2roman(2.3,X).
`partof_`	returns all attribute of an instance	partof_(<variable1>,<variable2>)	FORALL X,Y <- partof_(X,Y).
`Pow`	returns of value of the first argument raised to the power of the second argument	pow(<integer>,<integer>)	FORALL X <- X is pow(-1,2).
`Power`	$x^y=z$	power(X,Y,Z)	FORALL X <- power(2,3,X). FORALL X <- power(2,X,8). FORALL X <- power(X,3,8).
`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>)	needed facts: p([f(a,0),f(b,1)],A). p([f(a,1),f(b,0)],B). FORALL X,Z,Y classifier(X) <- p(Y,Z) and classify(a,Y,Z,X). query: FORALL X <- EXISTS C classifier(C) and predict([f(a,0),f(b,1)],C,X).
`Quadratic`	solves the quadratic equation X = -b +- sqrt(b*b-4ac))/2a	quadratic(<variable>,a,b,c)	FORALL X <- quadratic(X,1,-5,4).
`quadratic1`	solves the quadratic equation X = -b + sqrt(b*b-4ac))/2a	quadratic(<variable>,a,b,c)	FORALL X <- quadratic1(X,1,-5,4).
`Readurl`	reads a document and creates a word vector	readurl(<url>,<variable>)	FORALL X <- readurl("http://www.ontoprise.de",X).
`regexp*`	regular expressions may be used to search in strings, <string2> is the result of the operation with <string1>	regexp("<regular expression>",<string1>, <string2>)	FORALL X,Y <- X is "xyz" and regexp("yz",X,Y).
`regexp1`	regular expressions may be used to search in strings	regexp1("<regular expression>",<string>)	FORALL X <- X is "xyz" and regexp1("xy",X).
`Replace`	replaces characters in a string	replace(<input string>,<regexp>,<replacedby>, <variable>)	FORALL X <- replace("Eddie","[d]","m",X).
`Rint`	returns the value that is closest to the argument and is equal to a mathematical integer	rint(<integer>)	FORALL X <- X is rint(2.5).
`Round`	returns the closest to the argument	round(<integer>)	FORALL X <- X is round(2.3).
`Rounddigit`	rounds a double to the i-th digit after the point	dround(<number>,<number of digits>,<variable>)	FORALL X <- dround(2.5275,3,X).
`setatt_`		setatt_(<variable1>,<variable2>, <variable3>)	FORALL X,Y,Z <-setatt_(X,Y,Z).
`setatttype_`		setatttype_(<variable1>,<variable2>,<variable3>)	FORALL X,Y,Z <- setatttype_(X,Y,Z).
`setdirect_`		setdirect_(<variable1>,variable2>)	FORALL X,Y<-setdirect_(X,Y).

Setvalue	sets value at index I in vector v to new value x	setvalue(<vector>,<value>,<index>,<variable>)	FORALL X <- setvalue(v_(1,1,1),2,0,X).
Signvector	determines whether all components are positive or negative	signvector(<vector>,<variable>)	FORALL X <- signvector(v_(1,1,1),X).
Sin	returns the trigonometric sine of an angle	sin(<integer>)	FORALL X <- X is sin(0).
Smultvector	multiplies vector with a number	smultvector(<vector>,<number>, X)	FORALL X <- smultvector(v_(1,2,3),2,X).
Split	splits a string into tokens based on a given regular expression	split(<string to split>,<tokens>)	FORALL SS <- split("Can you please split this sentence?Into its tokens?",SS).
Splitgen	splits a string into tokens based on a regular expression to be formulated.	splitgen(<string to split>,<regular expression for splitting>,<tokens>)	FORALL SS <- splitgen("Can you please split this sentence?Into its tokens?", "[\\s|!|?|.]",SS).
Splitlocal	splits namespace and concept	splitlocal(<namespace#local>,<variable>)	FORALL X <- splitlocal("http://www.test.de#Person",X).
Sqlexecute		sql(<table>,F(<column1>(X),…,<column>(Y)),dbtype,dbname,dbhost, user, pwd)	FORALL X <- sql("SELECT projects.project FROM projects", [X],"mssqlserver2000","ontoproject","bunsen:1433","zope","zope").
Sqrt	returns the correctly rounded positive square root of a value	sqrt(<integer>)	FORALL X <- X is sqrt(4).
Square	computes the square of a number	square(<number>,<variable>)	FORALL X <- square(2.3,X).
string2number	converts a string to a number and vice versa	string2number(<string>, <number>)	FORALL X,Y <- string2number("1",Y) and X is Y+1.0.
Sum	summarizes values	sum(<grouping id>,<value>,<variable>)	FORALL X,Z <- EXISTS Y p(X,Y) and sum(X,Y,Z).
Sumvector	summarizes vectors	sum(<grouping id>,<vector>,<variable>)	needed facts: p(v_(1,2,3,4)). p(v_(3,4,5,6)). FORALL X,Y <- p(Y) and sumvector(a,Y,X).
Tan	returns the trigonometric tangent of an angle	tan(<integer>)	FORALL X <- X is tan(0).
Tokenize	breaks string into tokens at the delimiters	tokenize(<string>, <delimiters>,<variable>)	FORALL X <- tokenize("xyz","y",X).
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>)	FORALL X <- tolower("I Love Ontologies!",X).
Toupper	transforms all characters into upper characters	toupper(<string>, <variable>)	FORALL X <- toupper("I Love Ontologies",X).
Unify	X unified with Y	unify(X,Y)	
unitconversion	converts value1 with unit1 into value2 with unit2, where value1, unit1 and unit2 are given values	convert(<value1>,<unit1>,<variable>,<unit2>)	FORALL X <- convert(100,"ml",X,"l").
urlCode	encodes and decodes an url represented as string	urlCode (<string>,<encoded string>)	FORALL EURL <- urlCode("http://www.ontoprise.de/facts & things",EURL).

`whichbuiltin`	returns the class names, the flogic names and the arities of all builtins	whichbuiltins	FORALL A,B,C,D,F <- whichbuiltins(A,B,C,D,F).
`Wholenumbered`	takes the first vector and expands the elements until all are whole numbered	wholenumbered(<vector>,<variable>)	FORALL X <- wholenumbered(v_(1.2,1,3),X).
`write1, write2, ..., write6`	prints the parameters	write1(X1), write2(X1, X2), …, write6(X1, X2, X3, X4, X5, X6).	FORALL X,Y <- write(X,Y) and X::Y.

11. Appendix B: Namespace Declaration Syntax (deprecated)

Note: this section is kept only for reference. Please use the current way to declare namespaces (cf. Section 8).

The namespace mechanism of F-Logic is similar to that of XML. 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>
```

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.

12. Imprint

Editor

ontoprise GmbH
Amalienbadstraße 36
(Raumfabrik 29)
76227 Karlsruhe (Germany)
Telefon +49 (0) 721 / 509 809 10
Telefax +49 (0) 721 / 509 809 11
Email support@ontoprise.de
Internet http://www.ontoprise.de

Karlsruhe, April 2007

13. 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.