Tutorial Series

How to write

F–Logic - Programs

covering OntoBroker® Version 5.2

September 2008

F-Logic 5.2 Manual

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].

The present manual describes how to apply F-Logic in the OntoBroker system. Therefore, this document explains the various features of F-Logic by example and shows how to use them for typical problems. It covers the features of the OntoBroker version V5.x. The F-Logic variant of ontoprise slightly 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 the OntoBroker variant of F-Logic additionally arbitrary logical formulas can be used in the bodies of rules.

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].

2. Example

Before explaining the syntax and semantics in detail, we will give a first impression of F-Logic by presenting an F-Logic-program using F-Logic syntax. We will refer to the contents of this model in later sections of the documentation.

```
        /* schema facts */
        Car:Vehicle.
        Boat:Vehicle.
        Bike:Vehicle.

        Person[
                name => xsd#string;
                age => xsd#integer;
                friend =>> Person].
        Vehicle[
                owner => Person;
                admissibleDriver =>> Person].
        Car[
                passenger =>> Person;
                seats => xsd#integer].

        /* facts */
        peter:Person[
                name -> "Peter";
                age -> 17].
        paul:Person[
                name -> "Paul";
                age -> 21;
                friend->peter].
        mary:Person[
                name -> "Mary";
                age -> 17].
        bike26:Bike[
                owner -> paul].
        car74:Car[
                owner -> paul].

        /* rules consisting of a rule head and a rule body */
        FORALL X,Y X[friend->>Y] <- Y:Person[friend->>X].
        FORALL X,Y X[admissibleDriver->>Y] <- X:Vehicle[owner->Y].
        FORALL X,Y,Z X[admissibleDriver->>Z] <- X:Vehicle[owner->Y] AND
Y:Person[friend->>Z].

        /* query */
        FORALL X,Y <- X[admissibleDriver->>Y] AND X:Vehicle[owner->paul].
```

The first section of this example consists of a set of schema facts. The schema represents in an object-oriented way the classes and their relationships, e.g. to indicate that **car** and **bike** are subclasses of **vehicle**. It also describes that every **vehicle** has an **owner** and potentially multiply **admissible drivers**, which are **persons**. The schema also defines that each **person** has a **name** and an **age** of type, **string** and **int**, respectively.

The second section titled "facts", describes that some people belong to the class **person** and gives information about them, such as their **name** and **age**. Also it defines a relationship between the objects namely that **peter** is the **friend** of **paul**. According to the object-oriented paradigm, relationships between objects are represented as methods, e.g. applying the method **friend** to the object **paul** yields the result object **peter**. 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 the example 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 **friend** and **admissibleDriver**, are added to the object base as intentional information.

The final section of the example contains a query to the object base. It is asking for all **vehicles** that are **owned** by **paul**. For each such **vehicle** it also retrieves the **admissible drivers.**

3. Basic Syntax

The F-Logic language allows formulating logic programs that represent knowledge about objects, about their relationships and also about the classes they belong to. In addition to this factual knowledge rules and queries can be modeled that represent implicit, intensional knowledge. The basic knowledge representation is based on the notion of terms and predicates as known from the logic-programming world.

Terms represent all the differente entities of a F-Logic program, i.e. objects, classes, methods and method values. Because all these "first-class citizens" have names, we can query for them, which gives F-Logic the appeal and partially also the power of a second-order language.

Of course, a logic program must make assertians about the objects. These assertions are made with logical predicate. Refer to the "Statements" section below.

3.1. Terms

In F-Logic all objects have names. This includes classes and instances, values but also methods. The names of objects are formed by logical terms, known from datalog or prolog. Essentially, there are three types of terms:

1. constants, like **Person**, **car74** or **admissibleDriver**

   Each constant starts with a letter followed by (uppercase or lowercase) letters, digits or the underscore symbol "_" of the ASCII character set.

2. functions, like **f(X)**, **maximumSpeed(germany, autobahn)**

   Functions are complex terms that consist of a function symbols (which follows the same grammar as the constants above) and a list of one or more terms (enclosed in parenthesis) representing the arguments.

3. variables, like **X** or **Y**

   Variables follow the same grammar as the constants above. To distinguish constants from variables, the latter are always declared using logical quantifiers **FORALL** and **EXISTS**. Variables are only used in the context of rules and queries.

Additionally, there are three special types of constants: numbers and strings and symbols.

- Every positive or negative number may be used as a term, e.g., 17, -2.7, or 1E100.
- String constants are enclosed by "quotation marks" and may contain any legal printable character.
- Alternatively, strings 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. Note: the terms **'Müller'** and **"Müller"** are different and do not unify (in logical sense).

In addition to the basic function introduced above two special functions can be used in F-Logic.

- Lists
- namespace terms

These special cases deserve more attention and are described in more detail below.

Following the object oriented paradigm, objects may be organized in classes. Furthermore, methods represent relationships between objects. This information about objects is expressed by F-atoms (cf. the Statements-section below).

3.2. 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)
```

4. Basic Syntax -- Statements

According to the logic-programming paradigm, F-Logic also provides the notion of predicates which represent the atomic pieces of knowledge (statements), which can be true or false. Since F-Logic is also based on the object-oriented paradigm it not only provides plain predicates (as e.g. known from Prolog) but also offers epistemological primitives for modeling in an object-oriented way, i.e. subclass, and instance-of relations but also specification of the signatures for methods or the definition of the values for method applications.

In this section we will present the different kinds of statements available in F-Logic.

- Schema level statements
- Instance level statements
- Plain predicates

Later in this documentation we will also discuss rules, which from a logical point of view also represent statements.

The object-oriented statements of F-Logic comprise F-Atoms and F-Molecules and are syntactically distinguished from plain predicates.

4.1. Schema level statements

Subclass-of statements

In order to define the class hierarchy in F-Logic the language provides the so-called subclass-F-atoms. The subclass relationship between two classes is denoted by a double colon. In the following example we present two subclass-F-atoms that state that the classes **car** and **bike** are subclasses of the class **vehicle**:

```
        Car:Vehicle.
        Bike:Vehicle.
```

In subclass-F-atoms, the classes are denoted by id-terms. 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 also permitted in all positions of subclass-F-atoms.

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.

Signature statements

In F-Logic *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. Syntactically signature-F-atoms use an equals character and one or two greater-than characters. Here are some examples for signature-F-atoms:

```
Person[name => xsd#string].
Person[friend =>> Person].
Vehicle[owner@(xsd#integer) => Person].
```

The first one states that the single-valued method **name** is defined for members of the class **person** and the corresponding result object has to belong to the datatype **string**. The second one defines the multi-valued method **friend** for members of the class **person** restricting the result objects to the class **person**. Finally, the third signature-F-atom allows the application of the single-valued method **owner** to objects belonging to the class **vehicle** with parameter objects that are values of the datatype **integer**. The result objects of such method applications will be instances of the class **person**. 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, i.e. the result of the method is required to be in all of those classes:

```
Vehicle[owner =>> {Person, Adult}].
```

is equivalent to

```
Vehicle[owner =>> Person].
Vehicle[owner =>> Adult].
```

F-Logic also 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 **owner** applicable to instances of the class **vehicle** can be used as a method without parameter or as a method with one parameter. The corresponding signature-F-atoms look like this:

```
Vehicle[owner => Person].
Vehicle[owner@(xsd#integer) => Person].
```

Classes without any methods

As a special case, 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 "created" 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 the section below).

4.2. Instance Level Statements

To assert that an object is an instance of a certain class F-Logic provides so-called isa-F-atoms. The class membership is denoted by a single colon separating two id-terms, representing the instance and the class. The following example lists three isa-F-atoms express that **peter** and **paul** are members of the class **person**, whereas **car74**is a member of the class **car**.

```
peter:Person.
paul:Person.
car74:Car.
```

In contrast to other object-oriented languages, where every object instantiates exactly one class, F-Logic permits that an object is an instance of several classes that are not necessarily linked via the subclass relationship.

In F-Logic, the application of a method on an object is expressed by data-F-atoms which consist of a host object, a method and a result object, denoted by id-terms.

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

```
FORALL X,Y <- paul[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:

```
peter[friend->>{paul, mary}].
```

Sometimes the result of the invocation of a method on a host object depends on other objects, too, i.e. methods can also have parameters. For example, the **paul** might sell the **car74** to **peter**, which means that for different dates the car has different **owners**.

```
car74[owner@(2007)-> paul].
car74[owner@(2008)-> peter].
```

The syntax extends straightforwardly to methods with more than one parameter.

4.3. F-Molecules

Instead of giving several individual atoms, information about an object can be collected in F-molecules, which combine multiple F-atom statements in a concise way. For example, the following F-molecule denotes that **car74** is a **car** whose **owner** is **paul** and whose **admissible drivers** are **peter** and **mary**.

```
car74:Car[owner->paul; admissibleDriver->>{peter, mary}].
```

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

```
car74:Car.
car74[owner->paul].
car74[admissibleDriver->>peter].
car74[admissibleDriver->>mary].
```

For F-molecules containing multi-valued methods, the set of result objects can be divided into singleton sets (recall that the F-Logic semantics is multi-valued, not set-valued). For singleton sets, it is allowed to omit the curly bracket enclosing the result set, so that the two variants above are equivalent, which means that they yield the same object base.

The same can be done for schema-level statements such as subclass-F-atoms or signature-F-atoms. For that purpose, a subclass relationship may follow after the host object. Then, a specification list of signatures separated by semicolons, may be given. If a signature contains more than one class, those can be collected in parentheses, separated by commas:

```
Car::Vehicle[
        passenger =>> Person;
        seats => xsd#integer].
```

The following set of F-atoms is equivalent to the above F-molecule:

```
Car::Vehicle.
Car[passenger =>> Person].
Car[seats => xsd#integer].
```

More complex nesting is also possible in F-Logic f-molecules. 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:

```
car74:Car[owner->paul:Person[friend->peter:Person[age->17]].
```

This complex f-molecule is equivalent to the following set of f-atoms.

The equivalent set of F-atoms is:

```
peter:Person.
peter[age -> 17].
paul:Person.
paul[friend->peter].
car74:Car.
car74[owner -> paul].
```

4.4. Predicates

In F-Logic, predicates 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 terms separated by commas and included in parentheses is called a P-atom to distinguish it from F-atoms. The example below shows some P-atoms. The last P-atom consists solely of a 0-ary predicate symbol. Those are always used without parentheses.

```
owner(car74, paul).
adult(paul).
true.
```

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 two P-atoms could also be expressed as follows:

```
car74[owner->paul].
paul:adult.
```

Note that the expressions in the two examples above are alternative but disjoint representations. They cannot be used in a mixed manner, i.e. a query for **owner(X,Y)** does not retrieve any results for facts represented in the object-oriented way with F-Atoms.

5. 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.

5.2. Declaring Namespaces

Since OntoBroker Version 5.0 a new syntax for declaring namespace in F-Logic was introducd. 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#".
```

The code above declares four namespaces. It associates three of them with shortcuts (or aliases) and the last 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.

5.2. 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 from above:

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

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.

5.3. 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[].
```

The following inference rule integrates 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].
```

5.4. 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 the example above 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.

6. Built-in Features

The ontoprise implementation of F-Logic provides some built-in features which greatly extends the expressivitiy and versatility of the language. OntoBroker supports procedural attachements that can be used to do operations that are not really suitable for a logics-based mechanism, such as arithmetics or string-operations. Additionally, this mechanisms allows to access external data sources at run time and to integrate data external to the knowledge base into the reasoning and query-answering process.

The procedural attachments are integrated into the logic framework in the shape of built-in predicates. These predicates cannot be syntactically distinguished from ordinary predicates. The inference engine calls some external Java code to compute the extension of the predicates instead of executing its normal logics-based reasoning.

An overview of all built-ins provided by OntoBroker is given in the OntoBroker documentation. Here, we only describe a few built-ins briefly.

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 all car owners younger than 22:

```
FORALL X,P,A <- X:Car[owner->P] AND P[age->A] AND less(A,22).
```

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 supported. By default, multiplication and division have a higher precedence than 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 that computes the average age of **peter** and **paul**.

```
FORALL A,P1,P2<-
        peter[age->P1] AND
        paul[age->P2] AND
        (A is (P1+P2)/2.0) .
```

Additionally the following mathematical functions are implemented:

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

6.2. String handling

Analogously to numbers, there are several predefined operations for strings. These built-in predicates all have a fixed arity and (as all built-in predicate) must not be used in the head of rule.

- `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

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

  Regular expressions may be used to search in strings with this predicate. 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"
X = "tom", Y="t"
```

6.3. 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.

```
xsum(<groupingkey>,<key>,<input value>,<result>)
```

<groupingkey> the grouping key groups the results to a group. Given the following example:

Given the values:
```
g1, k1, 5
g2, k2, 10
g1, k3, 2
```
results in the following values:
```
g1, 7
g2, 10
```

<key> The key is a UNIQUE key for each input value which has to be considered. If two input values with the same key are given one of them is filtered out. In some cases every DIFFERENT value has to be considered only, in other cases every value (independent whether they are unique have to be considered. Let's have a look at an example. We want to sum up the revenue values from different countries grouped by the country:

```
FORALL C,V,R <- C:Country[revenue->V] and xsum(C,f(C,V),V,R).
```

The unique key is given by the value and the country. The reason for this is that during the inference process sometimes values are generated more than once and with the key it is determined what are duplicates and what are values which have to be considered.

<input value> These are the input values.

<result> This is either the result variable or it may be a constant to check the truth.

For a list of available aggregations see builtindescript.html.

Some executable examples:
Part 1:

```
p(gid1,key1,1.0).
p(gid1,key2,1.0).
p(gid1,key3,1.0).
p(gid1,key4,1.0).
p(gid2,key1,1.0).
p(gid2,key2,1.0).
p(gid2,key3,1.0).
p(gid2,key4,1.0).
```

```
QUERY q1: FORALL X,Y,Z,C
<- p(X,Y,Z)
and xcount(X,Y,Z,C). orderedby X,C
```

q1 counts for every gid (gid1 and gid2) all existing values, eleminating identical keys:
"gid1,4.0","gid2,4.0"

```
QUERY q2: FORALL X,Y,Z,C
<- p(X,Y,Z)
and xcount(X,Z,Z,C). orderedby X,C
```

counts for every gid (gid1 and gid2) all existing values that differ from each other:
"gid1,1.0","gid2,1.0"

Caution: If you use

```
p(gid1,key1,1.0).
p(gid1,key1,2.0).
```

with query q1, you have not consistent data. You cannot know which value will be used and which one will be ignored. This is irrelevant for xcount, but for other aggregations like xsum it will make a difference.

Part 2:

```
A[ref=>>A]@m1.
A[ref1=>A]@m1.
a:A@m1.
b:A@m1.
c:A@m1.
d:A@m1.
a[ref->{a,b}]@m1.
a[ref->d]@m1.
a[ref1->d]@m1.
c[ref1->d]@m1.
```

To see, which instance X has how many values ATTVAL, use:

```
QUERY q1: FORALL X,ATT,ATTVAL,C
<- X[ATT->ATTVAL]@m1
and xcount(X,f(ATT,ATTVAL),ATTVAL,C). orderedby X,C
results: "a,4.0", "c,1.0"
```

Within this query the key contains ATT, too. xcount(X,ATTVAL,ATTVAL,C) would just count the different values, seeing a[ref->d]@m1. as a duplicate of a[ref1->d]@m1. This is not necessary if ATT is a part of the grouping id.

To see, which instance X has how many values ATTVAL for which attribute ATT, use f(X,ATT) as gid:

```
QUERY q2: FORALL X,ATT,ATTVAL,C
<- X[ATT->ATTVAL]@m1
and xcount(f(X,ATT),ATTVAL,ATTVAL,C). orderedby X,ATT,C
results: "a,ref,3.0", "a,ref1,1.0", "c,ref1,1.0"
```

Within this query the grouping id contains ATT, too. For this example it is not necessary to use X or ATT as a key. ATTVAL, f(X,ATTVAL), f(ATT,ATTVAL) and f(X,ATT,ATTVAL) as key will work to filter duplicates, as X and ATT both are used for the grouping id.

7. Rules and Queries

An F-Logic knowledge base consists of a number of (extensional) ground facts. In order to formulate more complex knowledge F-Logic provides the notion of rules, which allows to specify dependencies between known facts and the creation of new, additional facts based the existing ones.

Queries are similar to SQL-queries and can be used to retrieve facts from the F-Logic knowledge base. Since we usually are interested in the entailment of applied rules to the basic facts, queries actually return facts from the derived model, which is built from the closure of all facts and rules.

7.1. Rules

Based on 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 of a rule is satisfied, the conclusion of the rule is also true.

The precondition is called rule body and is formed by an arbitrary logical formula consisting of P-Atoms (predicates) 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
- **A %lt;-> B** 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 **friend** and **owner** for some persons. The rules below compute the reflexive closure of **friend** and define a new method **admissibleDriver** based on **friend**- and **owner**-facts.

```
        FORALL X,Y X[friend->>Y] <- Y:Person[friend->>X].
        FORALL X,Y X[admissibleDriver->>Y] <- X:Vehicle[owner->Y].
        FORALL X,Y,Z X[admissibleDriver->>Z] <- X:Vehicle[owner->Y] AND
Y:Person[friend->>Z].
```

Partial logical formulae in the rule body may be negated. E.g. the following rule computes for every **car X** all persons **Y** that are **prohibited as drivers** for **X**:

```
        FORALL X,Y
                X[prohibitedDriver->>Y] <-
                        X:Car AND
                        Y:Person AND
                        NOT X[admissibleDriver ->> Y].
```

The following rule computes all persons **X** that do have (at least one) **friend**:

```
        FORALL X
                personWithFriends(X) <-
                        X:Person AND
                        EXISTS Y X[friend ->> Y].
```

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

```
        RULE MutualFriendship:
                FORALL X,Y
                        X[friend ->> Y] <-
                                Y:Person[friend ->> X].
```

The rule name can be an arbitrary ground term.

7.2. Queries

A query can be considered as a special kind of rule with an empty head. The following query asks about all **admissible drivers** of **car74**:

```
FORALL Y <-
        car74[admissibleDriver ->> Y].
```

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 and rules of the example from the beginning of this manual the above query yields the following variable bindings:

```
Y = paul
Y = peter
```

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 = {paul, peter}**.

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 given object base, all the following queries yield the answer true.

```
<- car74[admissibleDriver ->> {peter, paul}].
<- car74[admissibleDriver ->> {paul, peter}].
<- car74[admissibleDriver ->> peter].
<- car74[admissibleDriver ->> paul].
```

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 would need to use negation.

More complex queries can be formulated that also contain arbitrary first-order formulas in the (rule) body: The following query computes the maximum value **X** for which **p(X)** holds. The rule body expresses that all **Y** for which **p(Y)** (also) 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)).
```

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. Modules

In software engineering modules have been invented to reduce complexity. Closely related and interwoven things are packaged in a common module, while 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 variables) as well as rules and queries 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#sampleModule.
```

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

```
peter:Person@module1.
paul:Person@module1.
bike26:Bike[
        owner -> paul]@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 within 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 MutualFriendship@module1:
        FORALL X,Y
                X[friend ->> Y] <-
                        Y:Person[friend ->> X].
```

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 fact in the mentioned module. For head literals this means, that the new fact is asserted to hold true in its module. A complex example looks like this:

```
FORALL X,Y
        friend(X,Y)@module2 <-
                X:Person[friend ->> Y:Person]@module1.
```

This rule expresses that **module2** holds the (derived) fact **friend(X,Y)** if it is true in **module1** that the **X** and **Y** are **persons** and related via the **friend** method.

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

```
FORALL X,Y,M
        friend(X,Y) <-
                X:Person[friend ->> Y:Person]@M.
```

This rule searches for statements about **friends** in every module **M** and asserts a new fact 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, e.g.

```
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.
```

9. 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]** Juergen 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. Himmeroeder, 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 Ludaescher, 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.

Imprint

Editor

ontoprise GmbH

Amalienbadstraße 36

(Raumfabrik 29)

76227 Karlsruhe (Germany)

Telefon +49 (0) 721 / 509 809 0

Telefax +49 (0) 721 / 509 809 11

Email support@ontoprise.de

Internet http://www.ontoprise.de

Karlsruhe, September 2008