ObjectLogic Tutorial

© 2010 ontoprise GmbH

Table of Contents

Foreword 4
Example 6
Syntax Basic Statements 9
Schema level statements 9
Subclass-of statements 9
Classes without any methods 10
Signature statements 10
Instance Level Statements 11
Instance-of statements 11
Signature statements on instance level 11
F-Molecules 11
Predicates 12
Basic Syntax Elements 15
Terms 15
Lists 16
Examples 16
Namespaces in ObjectLogic 17
Declaring Namespaces 17
Default Namespace 18
Using Namespaces in ObjectLogic Expressions 18
Querying for Namespaces 19
Built-in Features 21
Numbers, Comparisons and Arithmetic 21
String handling 21
Aggregations 22
Rules and Queries 27
Rules 27
Queries 28
Range Restriction 30
Quantifier Scoping 32
Modules 34

© 2010 ontoprise GmbH

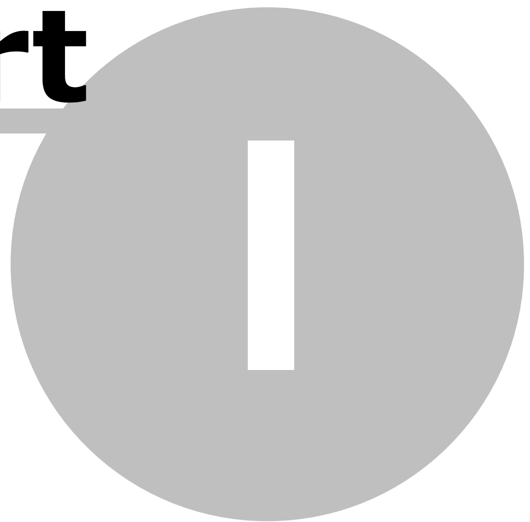
© 2010 ontoprise GmbH

Foreword

Ontoprise is the leading provider of industry-proven SemanticWeb infrastructure technologies and products used to support dynamic semantic information integration and information management processes at the enterprise level. With its mature and standards-based products ontoprise is delivering a key portion for the upcoming SemanticWeb. The company was founded in 1999 as a spin-off from Karlsruhe University, Germany to commercialize the technology that had by then already secured a leading position in the growing field of semantic technologies. Ontoprise has developed a comprehensive product suite designed to support the deployment of semantic technologies in the enterprise.

© 2010 ontoprise GmbH

Part



Example

1 Example

Before explaining the syntax and semantics in detail, we will give a first impression of ObjectLogic by presenting an ObjectLogic-program using ObjectLogic 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 {1:*} *=> xsd#string,
              age {1:1} *=> xsd#integer,
              friend {0:*} *=> Person].
       Vehicle[
              owner {1:1} *=>  Person,
              admissibleDriver {1:*} *=>  Person].
       Car[
              passenger {1:*} *=>  Person,
              seats {1:*} *=>  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 */
       ?X[friend->?Y] :- ?Y:Person[friend->?X].
       ?X[admissibleDriver->?Y] :- ?X:Vehicle[owner->?Y].
       ?X[admissibleDriver->?Z] :- ?X:Vehicle[owner->?Y] AND ?Y:Person[friend->?Z].

       /* query */
       ?- ?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 multiple **admissible drivers**, which are **persons**. The schema also defines that each **person** has a **name** and an **age** of type, **string** and **integer**, 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 ObjectLogic 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

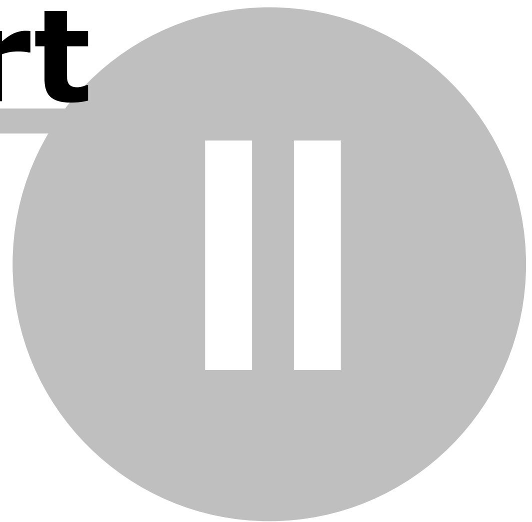
© 2010 ontoprise GmbH

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**.

© 2010 ontoprise GmbH

Part



Syntax Basic Statements

2 Syntax Basic Statements

According to the logic-programming paradigm, ObjectLogic also provides the notion of predicates which represent the atomic pieces of knowledge (statements), which can be true or false. Since ObjectLogic 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 modelling 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 ObjectLogic.

- 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 ObjectLogic comprise F-Atoms and F-Molecules and are syntactically distinguished from plain predicates.

2.1 Schema level statements

The schema defines the vocabulary of an ontology, defining a concept hierarchy as well as applicable methods for the concepts. In the following section we will introduce the ObjectLogic syntax for defining the schema.

2.1.1 Subclass-of statements

In order to define the class hierarchy in ObjectLogic 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.

© 2010 ontoprise GmbH

2.1.2 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).

2.1.3 Signature statements

In ObjectLogic *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. Also each method may have a cardinality restriction that provides a minimum value and a maximum value of how many entries have to be provided for each method at least and how many entries may be provided at most. These restrictions may be viewed as typing constraints. Signature-F-atoms together with the class hierarchy form the schema of an ObjectLogic database. In general the syntax may be described as follows:

```
<domain>[<methodname> {<mincard>:<maxcard>} *=>  <range>].
```

Here are some examples for signature-F-atoms:

```
Person[name {1:*} *=>  xsd#string].
Person[friend {0:*} *=>  Person].
Vehicle[owner(xsd#integer) {1:1} *=>  Person].
```

The first one states that the method **name** is defined for members of the class **person** and the corresponding result object has to belong to the datatype **string**. The cardinality is defined to be minimum 1 and maximum unlimited (*). The second one defines the method **friend** for members of the class **person** restricting the result objects to the class **person**. The cardinality is defined with a minimum of 0 and again the maximum is unlimited. Finally, the third signature-F-atom allows the application of the 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**. The cardinality is minimum 1 and maximum 1 as well which means it has to be exactly 1.

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 {1:*} *=>  {Person, Adult}].
```

is equivalent to

```
Vehicle[owner {1:*} *=> Person].
Vehicle[owner {1:*} *=> Adult].
```

ObjectLogic 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 {1:*} *=> Person].
Vehicle[owner(xsd#integer) {1:1} *=>  Person].
```

© 2010 ontoprise GmbH

2.2 Instance Level Statements

On the instance level it is possible to express information for individuals according to the schema defined for the ontology. There exist syntax elements in ObjectLogic to define instances of classes and their concrete method applications. The following section introduces the ObjectLogic syntax for these statements

2.2.1 Instance-of statements

To assert that an object is an instance of a certain class ObjectLogic 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, ObjectLogic permits that an object is an instance of several classes that are not necessarily linked via the subclass relationship.

2.2.2 Signature statements on instance level

In ObjectLogic, 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.

```
peter[friend -> mary].
```

If more values are given for 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 by separating the single values with a comma.

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

```
?- paul[?X->?Y].
```

2.3 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].
```

© 2010 ontoprise GmbH

```
car74[admissibleDriver->peter].
car74[admissibleDriver->mary].
```

For F-molecules containing multi-valued methods (methods with a cardinality that allows more than only one value), the set of result objects can be divided into singleton sets (recall that the ObjectLogic 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 commas, may be given. If a signature contains more than one class, those can be collected in parentheses, separated by commas:

```
Car::Vehicle[
       passenger {1:*} *=>  Person,
       seats {1:*} *=> xsd#integer].
```

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

```
Car::Vehicle.
Car[passenger {1:*} *=>  Person].
Car[seats {1:*} *=>  xsd#integer].
```

More complex nesting is also possible in ObjectLogic 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:

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

2.4 Predicates

In ObjectLogic, predicates are used in the same way as in predicate logic, e.g. in Datalog. Thus, preserving upward-compatibility from Datalog to ObjectLogic. 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.
```

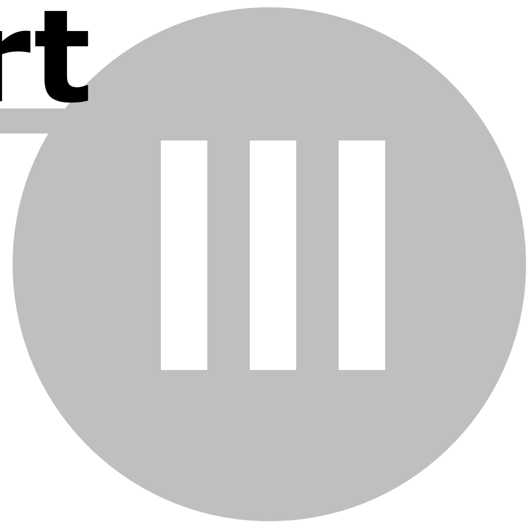
Note that the expressions in the two examples above are alternative but disjoint

© 2010 ontoprise GmbH

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.

© 2010 ontoprise GmbH

Part



Basic Syntax Elements

3 Basic Syntax Elements

The ObjectLogic 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 modelled that represent implicit, intensional knowledge. The knowledge representation is based on the notion of terms and predicates as known from the logic-programming world.

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

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

3.1 Terms

In ObjectLogic 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 by a leading questionmark. Variables are only used in the context of rules and queries.

Data types

Constants represent a value of one the following data types: strings, various number types (integer, decimal, double), strings, IRIs, date, time, dateTime, boolean and some more.

- String constants are enclosed by "quotation marks" and may contain any legal printable character.

© 2010 ontoprise GmbH

- ObjectLogic has five number data types, but normally you only need two of them: integers and double (double precision floating numbers). Integers are e.g. **0**,**1**,**17**,**-17**, double numbers are e.g **0.0**, **1.0**, **-2.7**, **3.12e-12**
- IRI constants are symbols with a namespace. IRIs are an internationalized variant of URIs, i.e. they support international characters. IRIs are typically used as identifiers for concepts and instances. ObjectLogic supports multiple syntax variants for this import data type:
  Person This identifier is already an IRI, as ObjectLogic uses the default namespace for it
  a#Person Here the namespace is specified by an prefix alias
  <http://mycompany.com/hr#Person> This is a full qualified IRI
- Other data types follow the XML schema specification, e.g.
  "2010-06-25"^^_date The date June, 25th 2010
  "16:37:45"^^_time The time 16:37:45
  "2010-06-25T16:37:45"^^_date A dateTime term
  true Boolean true value

- In addition to the basic terms, ObjectLogic also supports lists, which is 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).

3.2 Lists

A special kind of terms are lists. In ObjectLogic 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]
```

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.

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

3.3 Examples

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

Define a fact with a new list:

© 2010 ontoprise GmbH

```
p([a,b,c]).
```

Separate a list:

```
?- p([?Head | ?Tail]).
```

The result will be:

```
?Head=a, ?Tail=[b,c]
```

All elements of the list:

```
?- _member([a,b,c],?X).
```

The result will be:

```
?X=a, ?X=b, ?X=c
```

Merge lists:

```
?- _concat([a,b],[c,d],?X).
```

The result will be:

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

An extended example calculating a graph using lists is this:

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

// add each edge to a path containing two nodes
path([?Y,?X]) :- edge(?X,?Y).

// add every new edge to the appropriate path
path([?H1|?L]) :- path(?L) and _unify(?L,[?H2,?T]) and edge(?H2,?H1).
```

This query outputs all paths of the graph:

```
 ?- path(?L).
```

3.4 Namespaces in ObjectLogic

Namespaces are used to distinguish same names for different intentions in different ontologies. For instance, a concept named "person" in ontology "car" is the same concept as the concept "person" in ontology "finance". Handling more than one definition of a concept "person" in different ontologies thus needs a mechanism to distinguish these concepts. Thus, ontoprise uses the notion of namespaces in ObjectLogic, which enables RDF-like identifiers for objects, classes or properties.

3.4.1 Declaring Namespaces

An ObjectLogic 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#".
:- default 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

© 2010 ontoprise GmbH

the local part of an identifier. Esp. when exporting to RDF/OWL or reading from these formats, this convention is important.

3.4.2 Default Namespace

Objects that do not use a declared namespace alias refer to objects in the default namespace, in the example below the URI **http://www.myDomain.tv/private#**.

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

me:cars#Person[cars#age -> 28].
```

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 **<http://www.myDomain.tv/private#me>**

3.4.3 Using Namespaces in ObjectLogic Expressions

In ObjectLogic expressions every concept, method, object, predicate 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 namespace declaration from above:

```
cars#Car[
       cars#driver {1:*} *=>  cars#Person,
       cars#passenger {1:*} *=>  cars#Person,
       cars#seats {1:*} *=>  xsd#integer].
cars#Person[
       cars#name {1:*} *=>  xsd#string,
       cars#age {1:1} *=>  xsd#integer,
       cars#drivingLicenseId {1:1} *=>  xsd#string].

finance#Bank[
       finance#customer {1:*} *=>  finance#Person,
       finance#location {1:*} *=>  finance#City].
finance#Person[
       cars#name {1:*} *=>  xsd#string,
       finance#monthlyIncome {1:*} *=>  xsd#integer].

?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 ObjectLogic program the aliases are resolved, such that the following pairs are constructed.

- **finance#Person** stands for **<http://www.financeWorld.tv/Person>**
- **cars#Person** stands for **<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 ObjectLogic object. URIs can also be used directly in namespace terms, i.e. the use of aliases is optional.

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 ObjectLogic parser automatically adds a "#" at the end of the namespace. This patch is applied to literal namespaces as well as to the namespace declaration.

3.4.4 Querying for Namespaces

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

```
?- ?X [_localName->Person, _namespace->?N].
```

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

```
person(?Name)[?Attr -> ?Value] :-
                (EXIST ?X
                        ?X:finance#Person[?Attr -> ?Value, finance#name -> ?Name] OR
                        ?X:cars#Person[?Attr -> ?Value, cars#name -> ?Name]).
```

Part

Built-in Features

4 Built-in Features

The ontoprise implementation of ObjectLogic provides some built-in features which greatly extends the expressivity and versatility of the language. OntoBroker supports procedural attachments that can be used to do operations that are not really suitable for a logics-based mechanism, such as arithmetic or string-operations. Additionally, this mechanism 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.

4.1 Numbers, Comparisons and Arithmetic

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 expressions. For example, the following query asks for all car owners younger than 22:

```
?- ?X:Car[owner->?P] AND ?P[age->?A] AND ?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**.

```
?-
 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
```

4.2 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.

```
_isTypeOf(_string, <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.,

```
?- _concat("a","b",?X).
```

returns the binding ?X = "ab" whereas

© 2010 ontoprise GmbH

```
?- _concat("a",?Y,"ab").
```

leads to ?Y = "b"

```
_cut(<string>,<n>,<variable>)
```

cuts the last n characters from <string>

```
_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":

```
?- married(?X) and _regexp("[pt]",?X,?Y).
```

delivers

```
?X = "peter", ?Y = "p"
?X = "peter", ?Y = "t"
?X = "tom", ?Y="t"
```

4.3 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.

The syntax looks generally like this:

```
?- ?RESULT = sum{<input value>[<groupingkey>] | <query for values>}.
```

`<input value>` These are the input values for the aggregation.

`<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
```

`<query for values>` Query defining on how to get the values for the input for the aggregation.

© 2010 ontoprise GmbH

`?RESULT` Variable containing the result of the aggregation.

Calculating salaries

Here we have data about the employees of some company. Bill and Marc work in sales, Joe, Jack, Susan and Valerie develop software and Jane and Steve work in the research department:

```
Bill[hasSalary->60].
Bill[worksIn->Sales].
Marc[hasSalary->60].
Marc[worksIn->Sales].

Joe[hasSalary->60].
Joe[worksIn->Development].
Jack[hasSalary->40].
Jack[worksIn->Development].
Susan[hasSalary->100].
Susan[worksIn->Development].
Valerie[hasSalary->20].
Valerie[worksIn->Development].

Jane[hasSalary->70].
Jane[worksIn->Research].
Steve[hasSalary->30].
Steve[worksIn->Research].
```

The CEO wants to find out the department with the highest personnel costs. So she executes the following query:

```
?- ?SUM = sum{?S [?D] | ?X[worksIn->?D] AND ?X[hasSalary->?S]}.
```

The [?C] is the grouping variable (we want to group by the department). As we want to sum up the salaries, we specify the sum (?S) as the aggregation input. The result is

```
?D             ?SUM
------------------
Sales           120
Development     220
Research        100
```

So the software department has the highest personnel costs. But as the number of employees differs we want to calculate the average salaries in each department:

```
?- ?AVG = avg{?S [?C] | ?X[worksIn->?C] AND ?X[hasSalary->?S]}.
```

The result is

```
?D             ?AVG
------------------
Sales           60
Development     55
Research        50
```

So the employees of the sales department have the highest average salary.

© 2010 ontoprise GmbH

Simple counting example

The ontology consists of the following facts:

```
p(gid1,key1,1).
p(gid1,key2,1).
p(gid1,key3,1).
p(gid1,key4,1).
p(gid2,key1,1).
p(gid2,key2,1).
p(gid2,key3,1).
```

The following query counts for every `gid` (`gid1` and `gid2`) all existing values.

```
?- ?C = count{?Z [?X] | p(?X,?Y,?Z)}.
```

Identical keys are eliminated, so the result is

```
gid1,4
gid2,3
```

ObjectLogic counting example

```
A[ref {0:*} *=> A].
A[ref1 {0:*} *=> A].
a:A.
b:A.
c:A.
d:A.
a[ref->{a,b}].
a[ref->d].
a[ref1->d].
c[ref1->d].
```

To see, which instance `X` has how many values `ATTVAL`, use the following query:

```
?- ?C = count{?ATTVAL [?X] | ?X[?ATT->?ATTVAL]}.
```

The result is

© 2010 ontoprise GmbH

Query

Module: _defaultModule

Default namespace: http://kaon2.semanticweb.org/internal#

Queries:

UserRulesOff | Profile | InferOff | ProfileAll | IgnoreImports | Trace | FillNull

Prefix	Namespace
kaon2	http://kaon2.semanticweb.org/internal#
owl	http://www.w3.org/2002/07/owl#
owlx	http://www.w3.org/2003/05/owl-xml#

```
?- ?C = count{?ATTVAL [?X] | ?X[?ATT->?ATTVAL]}.
```

Execute | Stop

Result

Abbreviate namespaces

X	C
a	4
c	1

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

```
?- ?C = count{?ATTVAL [?X,?ATT] | ?X[?ATT->?ATTVAL]}.
```

This query will deliver the results

```
X       ATT    C
---------------
a        ref     3
a        ref1    1
c        ref1    1
```

© 2010 ontoprise GmbH

Part

Rules and Queries

5 Rules and Queries

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

Queries are similar to SQL-queries and can be used to retrieve facts from the ObjectLogic 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.

5.1 Rules

Based on a given object base (which can be considered as a set of 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 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 <--> 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.

```
?X[friend->?Y] :- ?Y:Person[friend->?X].
?X[admissibleDriver->?Y] :- ?X:Vehicle[owner->?Y].
?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**:

```
?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**:

```
personWithFriends(?X) :-
        ?X:Person AND
        (EXIST ?Y ?X[friend -> ?Y]).
```

© 2010 ontoprise GmbH

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

```
@{MutualFriendship} ?X[friend -> ?Y] :- ?Y:Person[friend -> ?X].
```

The rule name can be any arbitrary ground term.

5.2 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**:

```
?- 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).
?- p(?X) AND (FORALL ?Y (p(?Y) --> ?Y <= ?X)).
```

The result will be:

```
?X = 3.0
```

© 2010 ontoprise GmbH

Part

Range Restriction

6 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/3 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).

```
?- 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).

```
p(?X,?Y) :- q(?X) AND _add(?X,?Y,?Z).
?- p(?X,5).
```

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).

```
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.

© 2010 ontoprise GmbH

Part

Quantifier Scoping

7 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 ObjectLogic, 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.

```
p(?X,?Y) :- r(?X,?Y) AND (EXIST ?U q(?U,?Y)).
p(?X,?Y) :- (EXIST ?U q(?U,?Y) AND r(?U,?Y)).
p(?X,?Y) :- (EXIST ?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 **EXIST** quantifier).

© 2010 ontoprise GmbH

Part

Modules

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 ObjectLogic 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 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.
```

Important note: 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 ground statements, rules are always stored in the defining module. But they can infer statements in any module.

```
@{ancestorHasFather}
       ?X[hasAncestor ->?Y] :- ?X:Person, ?X[hasFather->?Y]@module2.
```

expresses that the rule named **ancestorHasFather** resides in **sampleModule**. All head and body formulas are assuming the same module if the module is not explicitly specified. The above rule, thus, is equivalent to:

```
@{ancestorHasFather}
       ?X[hasAncestor ->?Y]@a#sampleModule :- ?X:Person@a#sampleModule, ?X[hasFather->?Y]
@a#sampleModule.
```

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:

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

© 2010 ontoprise GmbH

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.

```
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 IRI 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.
```

© 2010 ontoprise GmbH

Ontoprise GmbH
An der 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

© Copyright 2010 ontoprise GmbH

All rights reserved.

No part of this publication may be reproduced or transmitted in any form or for any purpose without the express permission of ontoprise GmbH. The information contained herein may be changed without prior notice.
These materials are subject to change without notice. These materials are provided by ontoprise GmbH for informational purposes only, without representation or warranty of any kind, and ontoprise GmbH shall not be liable for errors or omissions with respect to the materials. The only warranties for ontoprise GmbH products and services are those that are set forth in the express warranty statements accompanying such products and services, if any. Nothing herein should be construed as constituting an additional warranty.
OntoEdit®, OntoBroker®, OntoAnnotate®, OntoCollect®, OntoMap®, Ontovision®, Ontoware®, Ontoprise®, OntoStudio®, SemanticMiner®, SemanticGuide®, SemanticIntegrator®, The RDF company® und The OWL Company® are registered trademarks of Ontoprise GmbH. Parts of the technology in the products OntoStudio®, OntoBroker®, OntoEdit®, OntoAnnotate®, OntoCollect®, OntoMap®, SemanticGuide® and SemanticMiner® are patented oder patent-registered.

Karlsruhe, July 2010