FLORiD

How to Write F-Logic Programs in Florid

A Tutorial for the Database Language F-Logic
Version 3.0 (FloXML)

Wolfgang May

may@informatik.uni-freiburg.de

Institut für Informatik, Universität Freiburg
Germany

October 2000

This manual is based on the former versions by Jürgen Frohn, Rainer Himmeröder, Paul-Th. Kandzia, Christian Schlepphorst, and Heinz Uphoff.

Contents

1 Introduction

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

In version 1.0 (1996) FLORID implemented the basic features of F-Logic. The theoretical foundations of F-Logic have been described in the F-Logic report [KLW95] and [FLU94]. The present tutorial describes how to *apply* F-Logic in the FLORID system.

Therefore, this tutorial explains the various features of F-Logic by example and shows how to use them for typical database problems. Section 2 gives a first impression of how F-Logic programs look like. The same simple model world taken from the Old Testament also serves as a background database throughout the tutorial. The following Sections 3 to 8 focus on data modeling and present the language concepts of F-Logic relevant for FLORID.

In order to write programs in FLORID, the reader also has to be aware of how FLORID implements some more difficult points. Questions regarding safety of rules, treatment of negation, non-monotonic inheritance and type checking are covered in Sections 10 and 12. In Section 14, a number of examples illustrate special features of FLORID and offer more material to start with.

As a major new feature, FLORID 2.0 (1998) provided access to Web documents. Its usage is described in Section 13. FLORID 2.1 and 2.2 (1999) incorporated several modifications and improvements concerning compilation. In FLORID 2.5 (April 2000), first XML handling features have been added, and the memory managent has been improved significantly. Until the current version 2.82, XML support has been completed.

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]. We refer the reader to the user manual [MM00] for a description of FLORID system commands.
Please send enquiries, comments, suggestions, and bug reports to

```
florid@informatik.uni.freiburg.de
```

2 A First Example

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

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

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

// query
?- sys.eval.  // This is a system command.
?- X:woman[son->>{Y[father->abraham]}].
```

(2.1)

The first part of this example consists of a set of facts which express that some persons belong to the classes man and woman, respectively, and give information about the father and mother relationships between some of these persons. According to the object-oriented paradigm, relationships between objects are modeled by method applications, e.g., applying the method father on the object isaac yields the result object abraham. All these facts may be considered as the extensional database of the F-Logic program. Hence, they form the framework of an object base which is completed by some closure properties. For more details about an object base see Section 8.

The rules in the second part of Example (2.1) derive new information from the given object base. Evaluating these rules in a bottom-up way, new relationships between the objects, denoted by the methods son and daughter, are added to the object base as intensional information. Note that the methods son and daughter are *multi-valued*, which is indicated by the double-headed arrow "->>" and the curly braces enclosing the result, whereas the methods father and mother are *functional*, indicated by the simple arrow "->". As known from other logical database languages, object names always begin with a lowercase letter, e.g., sarah, mother, man[1] and variable names in general begin with an uppercase letter, e.g., X.

The third part of Example (2.1) contains a query to the object base. The expression "?- sys.eval." is a system command to start the evaluation of the program (see also the user manual [MM00]). The query shows the ability of F-Logic to nest method applications. It asks about all women and their sons, whose father is Abraham. The same question could be written as a conjunction of simple subgoals:

```
?- X:woman, X[son->>{Y}], Y[father->abraham].
```

Also, the above program shows the different comment formats available in F-Logic: As in C and C++, text between "/*" and "*/" is ignored, even if it is ranging over several lines. The symbols "//" (C++-Style) or "%" (Prolog and LaTeX style) mark the rest of a line as a comment.

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

3 Objects and their Properties

As we have already seen in Example (2.1) *objects* are the basic constructs of F-Logic. Objects model real world entities and are internally represented by *object identifiers* which are independent of their properties. According to the principles of object oriented systems these object identifiers are invisible to the user. To access an object directly the user has to know its *object name*. Every object name refers to exactly one object. However, an object may be denoted by more than one object name as we will see in Section 7.1.

Following the object oriented paradigm, objects may be organized in *classes*. Furthermore, *methods* represent relationships between objects. Such information about objects is expressed by *F-atoms*.

3.1 Object Names and Variable Names

Object names and variable names are also called *id-terms* and are the basic syntactical elements of F-Logic. To distinguish object names from variable names, the first always begin with a lowercase letter, whereas the latter always begin with an uppercase letter or an underscore (cf. Prolog). After the first letter, object names and variable names both may contain uppercase letters, lowercase letters, numerals or the underscore symbol "_" . Examples for object names are abraham, man, daughter, for variable names are X, Method, _42. There are two special types of object names that carry additional information: integers and strings. Every positive or negative integer may be used as an object name, e.g., +3, 3, -3, and also every string enclosed by quotation marks "".

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

3.1.1 Methods

In F-Logic, the application of a method to an object is expressed by *data-F-atoms* which consist of a host object, a method and a result object, denoted by id-terms. All objects may occur in all places: host object, result position, or method position. Thus, in our Example 3.2 the method names father and son are object names just like isaac and abraham. Variables may also be used be used at all positions of a data-F-atom, which allows queries about method names like

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

As every person has at most one father, the method father is defined as a *functional* (or *scalar*) method, represented by the single headed arrow "->". In contrast, the method son may result in more than one object; this is indicated by the double headed arrow "->>". Such methods are called *multi-valued*. Note that the *set* consisting of the result objects is **not** an object, thus we preserve first-order semantics.

Empty Sets as Result of Multi-valued Methods. The result position of a multi-valued data-F-atom may also consist of the empty set, e.g., "isaac[son->>{}].". This does not necessarily mean that isaac has indeed zero sons. Since the object base may contain other information about Isaac having sons, e.g., "isaac[son->>{jacob}].", the former data-F-atom

expresses that Isaac has at least zero sons. This can be viewed as a kind of declaration of the method son for the object isaac.

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

```
jacob[son@(leah)->>{reuben, simeon, levi, judah, issachar, zebulun};
      son@(rachel)->>{joseph, benjamin};
      son@(zilpah)->>{gad, asher};
      son@(bilhah)->>{dan, naphtali}].
```
(3.1)

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

```
jacob[son@(leah,1)->reuben; son@(leah,2)->simeon;
      son@(leah,3)->levi; son@(leah,4)->judah;
      son@(bilhah,5)->dan; son@(bilhah,6)->naphtali;
      son@(zilpah,7)->gad; son@(zilpah,8)->asher;
      son@(leah,9)->issachar; son@(leah,10)->zebulun;
      son@(rachel,11)->joseph; son@(rachel,12)->benjamin].
```
(3.2)

In Examples (3.1) and (3.2) the method son is used with a different number of parameters. This so-called *overloading* (see also Section 3.4) is supported by F-Logic.

Queries with Multi-Valued Methods. If a query contains a multi-valued method with a variable at the result position, each result object of this method application in the object base is a possible binding for this variable. Given the object base described in Example 3.2, questioning the sons of Isaac yields all his known sons:

```
?- isaac[son->>X].

Answer to query : ?- isaac[son->>X].

X/jacob
X/esau
```

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

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

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

If we want to know if a set of objects is the *exact* result of a multi-valued method applied to a certain object, we have to use negation, see Example 14.6.

3.1.2 Class Membership and Subclass Relationship

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

```
abraham:man.
isaac:man.
sarah:woman.                                    (3.3)
woman::person.
man::person.
```

In isa-F-atoms and subclass-F-atoms, the objects and the classes are also denoted by id-terms because **classes are objects** – as well as methods are objects. Hence, classes may have methods defined on them and may be instances of other classes which serve as a kind of metaclasses. Furthermore, variables are permitted at all positions in an isa- or subclass-F-atom.

In contrast to other object oriented languages where every object is instance of exactly one most specific class (e.g., ROL [Liu96]), F-Logic permits that an object is an instance of several classes that are incomparable by the subclass relationship. Analogously, a class may have several incomparable direct superclasses.

Thus, the subclass relationship specifies a partial order on the set of classes, so that the *class hierarchy* may be considered as a directed acyclic, but reflexive graph with the classes as its nodes.

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

3.2 Expressing Information about an Object: F-Molecules

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

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

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

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

For F-molecules containing a multi-valued method, the set of result objects can be divided into singleton sets (recall that our semantics is *multivalued, not set-valued*). For singleton sets, it is allowed to omit the curly braces enclosing the result set, so that the three expressions

given in (3.4), (3.5) and (3.6) are equivalent, which means that they yield the same object base:

```
isaac[son->>{jacob,esau}].                                    (3.4)

isaac[son->>{jacob}].
isaac[son->>{esau}].                                          (3.5)

isaac[son->>jacob].
isaac[son->>esau].                                            (3.6)
```

3.3 Behavioral Inheritance

In object oriented systems, an instance (resp. subclass) may inherit properties of its class (resp. superclass). We distinguish *structural inheritance*, i.e., propagation of a type restriction for a method from a superclass to its subclasses (see Section 3.4 and 8.3), and *behavioral inheritance*, i.e., propagation of results of a method application from a class to its instances and subclasses.

To express behavioral inheritance in F-Logic we use *inheritable methods*, indicated by a special arrow type: "*->" for inheritable functional methods and "*->>" for inheritable multi-valued methods. If an inheritable method is defined for a class, this method application and the corresponding result is propagated to every instance and subclass of that class unless it is overridden.

The following inheritable data-F-atom denotes that in our object base every person, i.e., every object that is an instance of the class person, believes in god.

```
person[believes_in*->god].                                    (3.7)
```

Given this inheritable data-F-atom together with the subclass relationships and class memberships from Example (3.3) we can derive the following information:

```
abraham[believes_in->god].
isaac[believes_in->god].
sarah[believes_in->god].                                      (3.8)
woman[believes_in*->god].
man[believes_in*->god].
```

Example (3.8) shows that inheritable methods remain inheritable when they are passed to subclasses, but they become non-inheritable when passed to instances.

The following example demonstrates overriding of a method application (non-monotonic inheritance). Even though Ahab is a person, he does not believe in god but in Baal. Thus, the method believes_in is explicitly defined for the object ahab:

```
ahab:person[believes_in->baal].
```

This explicit information overrides inheritance from the class person.
In F-Logic, even multiple inheritance is supported. For a more detailed discussion about inheritance see Section 11.

3.4 Signatures

Signature-F-atoms define which methods are applicable for instances of certain classes. In particular, a signature-F-atom declares a method on a class and gives type restrictions for

parameters and results. These restrictions may be viewed as typing constraints. Signature-F-atoms together with the class hierarchy form the *schema* of an F-Logic database. As in data-F-atoms, the arrow head indicates whether a signature F-atom describes a functional method "=>" or a multi-valued method "=>>". To distinguish signature-F-atoms from data-F-atoms, the arrow body consists of a double line instead of a single line. Here are some examples for signature-F-atoms:

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

The first one states that the functional method father is defined for members of the class person and the corresponding result object has to belong to the class man. The second one defines the multi-valued method daughter for members of the class person restricting the result objects to the class woman. Finally, the third signature-F-atom allows the application of the multi-valued method son to objects belonging to the class man with parameter objects that are members of the class woman. The result objects of such method applications have to be instances of the class man.

By using a list of result classes enclosed by parentheses, several signature-F-atoms may be combined in an F-molecule. This is equivalent to the *conjunction* of the atoms: the result of the method is required to be in *all* of those classes:

```
person[father=>(man, person)].                    (3.9)
```

```
person[father=>(man)].
person[father=>(person)].                         (3.10)
```

Both expressions in the Examples (3.9) and (3.10)[2] are equivalent and express that the result objects of the method father if applied to an instance of the class person have to belong to both classes man and person.

As a special case, empty parentheses are allowed at the result position of a signature-F-atom. This means that the result objects are not restricted to certain classes. The following atom defines believes_in as a functional method which may be applied to instances of the class person without any requirements for the class membership of the result objects.

```
person[believes_in=>()].
```

More information about type checking—i.e., how to ensure that every method application in the object base is covered by a corresponding signature—may be found in Section 12.

Overloading F-Logic supports overloading of methods. 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 scalarity (functional or multi-valued), their arity, i.e., number of parameters, or their inheritability. For example, the method son applicable to instances of the class man is used as a multi-valued method with one parameter in Example (3.1) and as a functional method with two parameters in Example (3.2). The corresponding signature-F-atoms look like this:

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

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

3.5 Exploring the Database

Above we have shown how various kinds of information are represented by properties using different kinds of *arrow types* and parameters. For obtaining an overview which properties are stored in the database, the describe predicate can be used:

describe(*object*,*method*,*arity*,*arrowtype*) and
describe(*object*,*method*,*arity*,*arrowtype*,*result*)

yield all bindings such that an atom of the form

object[*method*@(o_1,...,o_{arity})⋯≫ *result*]

is defined in the database, e.g.,

```
?- describe(H,M,A,T)
H/man    M/son    A/0    T/"=>>"
H/man    M/son    A/1    T/"=>>"
H/man    M/son    A/2    T/"=>"
H/isaac  M/son    A/0    T/"->>"
H/isaac  M/son    A/1    T/"->>"
H/isaac  M/son    A/2    T/"->"
⋮
```

The arguments *object* and *method* are bound to objects, *arity* is bound to an integer, and *arrowtype* to a string representing the method type. This can, e.g., be used for generically exploiting the database, or for specialized queries which methods are in fact defined for some object, e.g.,

```
?- describe(H,M,A,"->"), not describe(H,M,A,"=>").
```

gives for all scalar properties for which no signature atom is present.

4 Nesting Object Properties

As already shown in Example 3.2, properties of an object may be expressed in a single, complex F-molecule instead of several F-atoms. For that purpose, a class membership or subclass relationship may follow after the host object. Then, a *specification list*, a list of method applications (with or without parameters) separated by semicolons, may be given. If a multi-valued method yields more than one result, those can be collected in curly braces, separated by commas; if a signature contains more than one class, those can be collected in parentheses, also separated by commas:

```
isaac[father->abraham; mother->sarah].
jacob:man[father->isaac; son@(rachel)->>{joseph, benjamin}].        (4.1)
man::person[son@(woman)=>>(man, person)].
```

The following set of F-atoms is equivalent to the F-molecules in (4.1):

```
isaac[father->abraham]
isaac[mother->sarah].

jacob:man.
jacob[father->isaac].
jacob[son@(rachel)->>{joseph}].
jacob[son@(rachel)->>{benjamin}].

man::person.
man[son@(woman)=>>(man)].
man[son@(woman)=>>(person)].
```
(4.2)

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

```
isaac[father->abraham:man[son@(hagar:woman)->>ishmael];
      mother->sarah:woman].
jacob:(man::person).
jacob[(father:method)->isaac].
```
(4.3)

The equivalent set of F-atoms is:

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

man::person.
jacob:man.

jacob[father->isaac].
father:method.
```

F-Logic molecules are evaluated from left to right. Thus, nested properties have to be included in parentheses if those properties belong to a method object (cf. Section 6), class object or superclass object. Note the difference between the following two F-molecules. The first one states that Isaac is a man and Isaac believes in god, whereas the second one says that Isaac is a man and that the *object* man believes in god (which is probably not the intended meaning).

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

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

```
isaac[(father::ancestor)->abraham]    is correct, whereas
isaac[father::ancestor->abraham]    results in a parsing error, and

isaac[father->(abraham:man)]    is correct, whereas
isaac[father->abraham:man]    results in a parsing error.
```

4.1 F-molecules without any properties

If we want to represent an object without giving any properties, we have to attach an empty specification list to the object name, e.g., "abraham.". If we use an expression like this that consists solely of an object name as a molecule, it is treated as a 0-ary predicate symbol (see next section).

5 Predicate Symbols

In F-Logic, predicate symbols are used in the same way as in predicate logic, e.g., in Datalog, thus preserving upward-compatibility from Datalog to F-Logic. A predicate symbol followed by one or more id-terms separated by commas and included in parentheses is called a *P-atom* to distinguish it from F-atoms. Example (5.1) shows some P-atoms. The last P-atom consists solely of a 0-ary predicate symbol. Those are always used without parentheses.

```
married(isaac,rebekah).
male(jacob).
son_of(isaac,rebekah,jacob).
true.
```
(5.1)

Information expressed by P-atoms can usually also be represented by F-atoms, thus obtaining a more natural style of modelling. For example, the information given in the first three P-atoms in (5.1) can also be expressed as follows:

```
isaac[married_to->>rebekah].
jacob:man.
isaac[son@(rebekah)->>jacob].
```
(5.2)

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

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

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

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

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

6 Path Expressions

Objects may be accessed directly by their object names. On the other side it is also possible to navigate to them by applying a method to another object using *path expressions*. For example, the object described by the object name abraham may also be accessed by calling the method father on the object isaac. The corresponding path expression is "isaac.father". Example (6.1) shows that path expressions may also contain methods with parameters and that it is possible to chain up path expressions by successively applying methods to the result object of the preceding method call. At the end of each line you find the object name of the result object that is denoted by the path expression. The underlying object base is taken from the Examples (2.1) and (3.2):

```
jacob.son@(rachel,11)            joseph
benjamin.father.father.mother    rebekah                    (6.1)
god.people                       ?
```

Some path expressions may even denote objects in the object world which have no id-term as object name. The last path expression in Example (6.1) defines a new object by applying the method people to the object god which is intended to be a class collecting the people of Israel. However, there is no direct object name denoting this object. The concept of object creation by path expressions is described in detail in Section 6.2.

6.1 Nesting of Path Expressions and F-Molecules

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

```
jacob.son@(rachel,11)[mother->rachel; father->jacob].
abraham[son->>{jacob.father}].
jacob[son@(joseph.mother)->>{benjamin}].                    (6.2)
male(jacob.father).
jacob.father.father = abraham.
```

F-Logic expressions are evaluated from left to right. Thus, if a path expression should occur at the method position or at the class position, resp. superclass position, it has to be enclosed by parentheses. To illustrate this we define a new "meta"-method twice by the following rule:

```
X[(M.twice)->Z] :- X[M->Y[M->Z]].
```

This new method may be invoked on other method names meaning that the original method is applied twice, e.g., the method denoted by the path expression father.twice would specify the "grandfather on the father's side" method. Hence, applying father.twice to Jacob yields Abraham as result. The other F-molecule denotes that Jacob belongs to god's people.

```
jacob[(father.twice)->abraham].
jacob:(god.people).                                         (6.3)
```

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

```
jacob.(father.twice):person.
jacob.father.twice:person.                                  (6.4)
(jacob.father).twice:person.
```

The path expression in the first F-molecule denotes abraham as in (6.3). As path expressions are evaluated left to right, the second and third F-molecule are equivalent. In our context[3], however, they are not meaningful (evaluating to false) because jacob.father is a person (isaac) and not a method, so that twice cannot be applied to this object.

```
jacob:(god.people).
jacob:god.people.                                           (6.5)
(jacob:god).people.
```

In Example (6.5) the first F-molecule states that applying the method people to the object called god yields a class jacob belongs to. The second expression, which is equivalent to

[3] Assume the object base defined by Example (2.1) is given.

the third one, states that the object jacob is a member of the class god and denotes the application of the method people to the object jacob. However, the last two expressions are path expressions—not F-molecules—as they do not end with a specification list or an isa/subclass relationship (see Section 4.1).
Besides using path expressions instead of simple id-terms in F-molecules, it is also possible to nest path expressions and F-molecules the other way round: Intermediate objects in a path expression may have specification lists, turning them into F-molecules. As an example the path expression jacob.mother may be extended by specifying some properties for Jacob:

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

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

6.2 Object Creation with Scalar Path Expressions

Scalar path expressions are allowed in rule heads to induce the creation of new objects. This effect occurs whenever a path expression consists of a host object with a method application that has not been defined otherwise. When applying the method father to the object abraham, we create a new object representing Abraham's father:

```
abraham.father:man.
```

Here, the method father is defined for the object abraham and yields the new object abraham.father. Analogously,

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

creates a new object god.people by defining the method people for the object god. Both objects, abraham.father and god.people, have no ground id-term as object name; the user has to access these objects with their path expressions.

On the other hand, if the method application is defined somewhere else for the host object, the path expression evaluates to the corresponding result object. Assume the object base described by Example (2.1). The, the path expressions in

```
isaac.father:person.
isaac[married_to->>jacob.mother].
```

do not create objects because the methods are already defined for the corresponding host objects. Instead, the facts abraham:person and isaac[married_to->>rebekah] are added.
Object creation with path expressions reveals its full power in connection with variables. See Section 9 for the use of variables in rules.

Object Names and Path Expressions. Because of the possibility to create new objects using path expressions we have to redefine the set of object names: now there are objects which are not associated to a ground id-term. For that purpose, the set of object names is defined by the union of the set of all ground id-terms and the set of all ground, unnested, functional path expressions[4]. These are path expressions containing neither variables nor any F-molecules, therefore consisting just of a host object followed by one or more functional method applications (possibly with parameters). Examples for such path expressions are:

[4]Id-terms may be considered as a trivial case of path expressions (without any method application)

```
benjamin.father.father.mother
jacob.son@(rachel,1)                                    (6.6)
abraham.father
```

6.3 Path Expressions in Queries

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

```
?- isaac.father[father->X].   instead of
?- isaac[father->Y], Y[father->X].
```

Path expressions may be eliminated from F-molecules in rule bodies or queries by decomposing the molecules into a set of F-atoms using new variables for the result values (in the above example, also a *don't-care* variable could be used which does not occur in the result set, see Section 9).

6.4 Multi-valued Path Expressions

Up to now only functional methods in path expressions have been considered. It is also possible to build path expressions with mult-valued methods, resulting in *multi-valued path expressions*. The use of a multi-valued method in a path expression is indicated by a double dot, e.g., "isaac..son". Such a path expression matches not just a single object (like a functional path expression does) but each object from a set of answers.

Every path expression is either functional or multi-valued. The *scalarity* of a path expression can be determined syntactically by considering the corresponding *unnested path expression* which is built by removing all isa and method specifications. Example (6.7) shows some path expressions and their corresponding unnested path expressions:

```
jacob[son->>{joseph}].(father.double):person
jacob.(father.double)

jacob[father->abraham..son].mother
jacob.mother                                            (6.7)

isaac:man..son[mother->rebekah]..son
isaac..son..son

jacob..son@(laban..daughter:woman)
jacob..son@(laban..daughter)
```

A path expression is multi-valued if its corresponding unnested path expression contains at least one multi-valued method application. Hence, in Example (6.7) the first two path expressions are functional and the last two are multi-valued. The second path expression is functional, although it contains a multi-valued path expression abraham..son. However, this multi-valued path expression appears at the result position of a specification and thus has no influence on the unnested path expression. The last path expression jacob..son@(laban..daughter) is multi-valued not only because of the host object but also because of the parameter that is denoted by a multi-valued path expression. Even if the method son were used as a functional method in this example, the path expression would be multi-valued due to the multi-valued parameter.

Semantics of Multi-valued Path Expressions As already mentioned, a multi-valued path expression describes a set of objects. Note that the set itself cannot be referenced in F-Logic. Instead, we consider each object in the set as one possible result of the method application. The query in (6.8) does not ask whether *all* of Abraham sons are sons of Sarah, too, but whether *one* of Abraham's sons is a son of Sarah[5] and therefore yields the answer true.

```
?- sarah[son->>abraham..son].                                    (6.8)
```

Considering sets as a whole as in languages like LDL [NT89] requires stratification. The comparison of sets is not a built-in feature in F-Logic but can easily be done by an F-Logic program (see Section 14.6).

Since multi-valued path expressions do not denote sets, they may be used at any syntactical position in an F-molecule[6]. The following queries, for example, ask for those persons whose father is one of Abraham's sons, and for the sons of Jacob born by a daughter of Laban (Laban is the father of Leah and Rachel).

```
?- X:person[father->abraham..son].
?- jacob[son@(laban..daughter)->X].                              (6.9)
```

In contrast to scalar path expressions, multi-valued path expressions *are only allowed in queries or rule bodies*. If we allowed multi-valued path expressions in facts or rule heads the following problem would occur: Since the multi-valued path expression abraham..son denotes the two objects isaac and ishmael the fact "abraham..son[lives->israel]." would be satisfied if Isaac, Ishmael or both lived in Israel. The last alternative would violate the minimality of the object base (see Section 10.1). If only one of Abraham's sons lives in Israel, it is not clear which one. This situation is similar to disjunctive expressions in a rule head, hence, we would have to handle a non-deterministic program. To avoid such problems, multi-valued path expressions are disallowed in rule heads.

6.5 Path Expressions with Inheritable Methods

Path expressions may contain inheritable methods, too. Such methods are indicated by one or two exclamation marks instead of dots. The path expression in example (6.10) denotes the object god if we consider the object base defined in example (3.7).

```
person!believes_in                                               (6.10)
```

6.6 F-Molecules vs. Path Expressions

We have seen that F-molecules and path expressions may be combined and nested in several ways to obtain complex expressions. This is possible because every ground F-molecule and every ground path expression has an *object value* and a *truth value* according to a given object base.

An object base corresponds to an F-structure as introduced in [KLW95]. If an atom t is true in a given object base or F-structure $\mathcal{I}$, we write $\mathcal{I} \models t$.

Although we define object values and truth value for both of them, the set of F-molecules and the set of path expressions are strictly disjunct: An F-molecule always ends with a

[5] The semantics of this example differs from the semantics introduced in [FLU94].

[6] In this point the semantics of FLORID does not agree with [FLU94].

specification, i.e., an isa specification denoting a class membership or subclass relationship, or a list of method specifications expressing the results of method applications to an object. A path expression always ends with a dot followed by a method application.

The object value and truth value is recursively defined for arbitrary complex expressions in an intuitive way corresponding to the evaluation of F-molecules:

Definition 1 (Semantics [LHL+98]) *For an F-Logic database $\mathcal{I}$, the object values of ground expressions are given by the following mapping obj from ground expressions to sets of ground references:*

$$\begin{array}{lcl} obj(t) & := & t \text{ for a ground id-term } t \\ obj(r[spec]) & := & \{o \in obj(r) \mid \mathcal{I} \models o[spec]\} \\ obj(r:c) & := & \{o \in obj(r) \mid \mathcal{I} \models o:c\} \\ obj(c :: d) & := & \{c' \in obj(c) \mid \mathcal{I} \models c' :: d\} \\ obj(r.m) & := & \{r' \in obj(v) \mid \mathcal{I} \models r[m\text{->}v]\} \\ obj(r..m) & := & \{r' \in obj(v) \mid \mathcal{I} \models r[m\text{->>}\{v\}]\} \end{array}$$

analogously for $r!m$ and $r!!m$.

The $\models$ relation extends from atoms to pure references and nested expressions as follows:

$$\begin{array}{lcl} \mathcal{I} \models r & \Leftrightarrow & obj(r) \neq \emptyset \quad \text{for a ground pure reference } r, \\ \mathcal{I} \models r_1[r_2@(p_1,...,p_n)\text{->}r_3] & \Leftrightarrow & \mathcal{I} \models o_1[o_2@(q_1,...,q_n)\text{->}o_3] \text{ for any } o_i \in obj(r_i),\ q_j \in obj(p_j) \\ & & \text{analogously for -}\text{>>, =}\text{>>, and =}\text{>}. \\ \mathcal{I} \models r_1:r_2 & \Leftrightarrow & \mathcal{I} \models o_2:o_2 \text{ for any } o_1 \in obj(r_1),\ o_2 \in obj(r_2) \\ & & \text{analogously for } ::. \end{array}$$

An F-Logic expression (i.e., an F-molecule or a path expression) is true if the corresponding object value contains at least one object, i.e., if one object has the appropriate properties. It is false if the object value is empty. Thus, also a path expressions may be used as subgoals in a rule – the are true if their object value is nontrivial.

Some examples showing F-molecules and path expressions as well as their object and truth values are presented in Figure 1. The underlying object world is the one defined in Example (2.1). An empty object value is written as $\emptyset$.

7 Built-in Features

The FLORID implementation of F-Logic provides some built-in features, namely the *equality predicate*, the built-in class *integer*, several *comparison predicates*, the basic *arithmetic operators*, predicates for *string handling*, and *aggregate functions*.

7.1 Equality

Objects in F-Logic are uniquely determined by their *object identifiers* which are only used internally and are invisible to the user, who has to access objects by their *object names*. Every object name references exactly one object. However, there may be several object names denoting the same object. In such a case the object names are said to be **equal**, indicated by the equality predicate "=", e.g., abram = abraham.[7] The equality predicate induces a

[7] Note that the equality predicate is used in infix notation in contrast to other predicate symbols.

expression	object value	truth value
isaac	isaac	true
isaac	isaac	true
isaac:man	isaac	true
isaac:woman	∅	false
isaac[son->>{jacob}]	isaac	true
isaac[son->>{abraham}]	∅	false
isaac.father	abraham	true
isaac.father	abraham	true
isaac..son	jacob, esau	true
abraham..son	isaac, ishmael	true
abraham..son[mother->sarah]	isaac	true
abraham..son:woman	∅	false

Figure 1: Object values and truth values

congruence relation on the set of object names and may be used in facts or rule heads to equate two object names, as well as in queries or rule bodies to ask about the equivalence of object names. Objects may also be equated *implicitly* by redefining a scalar method (since the functionality constraint forces both result objects to be equal) or by defining a circular subclass relationship. The equality predicate may also be used in connection with variables. If an equality predicate in a rule body has variables on both sides, safety of the rule is a critical issue (see Section 9).

7.2 Integers, Comparisons and Arithmetics

Objects denoting *integer* numbers are different from other objects because the usual comparison operators are defined for them, as well as several arithmetic functions. We have not implemented a built-in *class* integer because it would contain an infinite number of instances, making static safety checking impossible[8]. Instead there is a built-in *predicate* integer(<argument> in FLORID.

Within a query or a rule body, relations between integer numbers may be tested with the *comparison predicates*[9] "<",">", "<=" or ">=". For example, the following query asks for the first three sons of Jacob:

```
?- jacob[son@(X,Y)->Z], Y <= 3.
```
(7.1)

To guarantee safety (see Section 9), variables that appear in a comparison P-atom have to be bound by another atom or molecule in the rule body. Comparison predicates are not allowed in rule heads.

The basic arithmetic operations addition "+", subtraction "-", multiplication "*" and integer division "/" are also implemented in FLORID. Arithmetic expressions may be constructed in the usual way, even complex expressions, e.g., "3 + 5 + 2" or "3 + 2 * 3" are possible. Note that the blanks between an arithmetic operator and its operands are mandatory, 2+2

[8] A subgoal like X:Y with variable Y bound yields an infinite answer set, if Y is bound to the object name integer.

[9] Note that comparison predicates are used in infix notation in contrast to other predicate symbols.

leads to a parser error message. By default, multiplication and division are prior to addition and subtraction. As usual, the evaluation order may be changed by using parentheses, e.g., "(3 + 2) * 3".

Arithmetic expressions are only allowed in (in-)equality-P-atoms in a rule body, e.g., "X = Y + 2", "3 * X = Y + 4". Furthermore, every variable in an arithmetic expression has to be bound by another subgoal in the rule body (see Section 9). The following example contains the query whether Jacob has three sons born consecutively by the same woman.

```
?- jacob[son@(X,A)->Z1; son@(X,B)->Z2; son@(X,C)->Z3],
    B = A + 1, C = A + 2.
```

Objects denoting integers must not be equated to other integer or string objects because their object identity is not independent from their object name. Unfortunately, avoiding this by a static check is impossible since integer objects may be bound to variables in a rule head. If an equation of two different integer objects is derived during the evaluation of an F-Logic program, an error is reported.

7.3 String handling

Analogously to integers, there are several predefined operations for strings. These are provided by the built-in predicates which all have a fixed arity. Using them with a wrong arity causes a parser error message. Furthermore these predicates can only be used in rule bodies:

string(<**arg**>**)** is true, if <arg> is a string.

strlen(<**string**>**,** <**value**>**)** holds if <value> (which can be given a constant or a variable) represents the length of <string>. For practical reasons, variables at position of <string> have to be bound by other molecules in the rule body (otherwise <strlen> fails), because a query like "*show all strings with a certain arity*" e.g., "?- strlen(X,40)." may lead to an answer set that is not infinite but too huge to be handled. Normally strlen is used in the following way to return the length of a given string:
"?- strlen("logic",X)."

strcat(<**string**$_1$>**,** <**string**$_2$>**,** <**string**$_3$>**)** succeeds if <string$_3$> is the concatenation of <string$_1$> and <string$_2$>. E.g., "?- strcat("a","b",X)." returns the binding X/"ab" whereas "?- strcat("a",Y,"ab")." leads to Y/"b". If the arguments contain more than one variable, e.g., "?- strcat("a",Y,X).", either Y or X have to be bound by other molecules in rule body, otherwise strcat fails. The reason for this limitation is the same as in the case of strlen.

substr(<**string**$_1$>**,** <**string**$_2$>**)** holds if <string$_1$> is a substring of <string$_2$>. Comparison between the two strings is case insensitive, for example "?- substr("DaTA","database")." returns true. If the arguments contain any variable, it has to be bound by other molecules in the rule body, otherwise substr fails. For "?- substr("logic",X)." the corresponding answer set would be infinite. Besides strings, Web documents (see Section 13.1) may also appear as second argument. Thus, substr can also be used to check if a certain string occurs in a Web document.

match(<**string**>**,** <**pattern**>**,** <**fmt-list**>**,** <**variable-list**>**),**

pmatch(<**string**>**,** <**pattern**>**,** <**fmt-list**>**,** <**variable-list**>**)** finds all strings contained in <string> (which may be a string or a Web document) which match the pattern given by a regular expression in <pattern>. <fmt-list> is a format string describing how the matched strings should be returned in <variable-list>. This feature is useful when using groups (expressions enclosed in \(...\)) in <pattern>. In the format string <fmt-list>, groups are referred

to by their number: `$n`,where `n` ranges from 1 to 9. If <fmt-list> is the empty string ("") all groups are returned without formatting. With the exception of <variable-list>, all arguments must be bound. pmatch does the same, using Perl regular expressions; we recommend using pmatch. A full explanation of the syntax and use of regular expressions is given in Appendix B.

For example,

```
?- match("linux98","\([0-9]\)\([0-9]\)","$2swap$1",X).
```

returns X/"8swap9" (the first and second match are swapped).

Example 1 (Wrapping by regular expressions) *Assume that the author list of a paper is given on a Web page (see also Section 13.1) as*

auth$_1$, *auth*$_2$, ... , and *auth*$_n$: *title*. number n in
Volume v of *series*, pages $p_1 - p_2$, *year*.

Then, the following predicate assigns the relevant substrings to the corresponding variables:

```
pmatch(STRING,
       "/\A ([:]*): (.*)\.\s
         Number ([0-9]*) in Volume ([0-9]*) of ([a-Z]*),
         pages ([0-9-]*),([0-9]*)/",
       "[$1,$2,"$4($3)", $5, $6, $7]",
       "[AuthList, Title, Num, Series, Pages, Year]")
```

Then, the string bound to the variable Num is of the form "volume(number)".

7.4 Data Conversion

FLORIDinternally distinguishes objects (e.g., john from literals (e.g., 42, 3.14, or "John"). The literal types are further distinguished: strings are always given in quotes ("john"), integer objects are given as-is, and floats have to be distinguished by #, from literals (#3.14).

Example 2 *Note that 3.14 and #3.14* actually *are different things: In the first example, #3.14 is interpreted as a float:*

```
b[m->#3.14].
?- sys.strat.doIt.
?- b[m->C].
?- b[m->C], D = C + 1.
% Answer to query : ?- b[m -> C].
C/#3.14
% Answer to query : ?- b[m -> C], D = C + 1.
C/#3.14   D/#4.14
```

In the second example, 3.14 is not a float*:*

```
a[m->#3.14].
?- sys.strat.doIt.
?- a[m->C].
?- a[m->C], D = C + 1.
% Answer to query : ?- a[m -> C].
C/3.14
```

```
% Answer to query : ?- a[m -> C], D = C + 1.
false
```

But, what happens there? Obviously, the program is syntactically correct. So, lets ask what the database looks like then:

```
?- sys.theOM.dump.
3 [14 -> 3.14].
a [m -> 3.14].
```

... which is correct: 3.14 is interpreted as the result (an anonymous object) of applying the method 14 to the object 3 (similar to what happens for john.father.

Often, a conversion between datatypes is needed. In FLORID, data conversion is provided by built-in predicates:

string2integer(A,B) is true, if B is the integer obtained when reading the string A from left to right as far as possible. e.g., the following holds:

```
string2integer(''42'',42), string2integer(''3D'',3), string2integer(''3.14'',3).
```
(7.2)

string2float(A,B) is true, if B is the float obtained when reading the string A from left to right as far as possible (a leading # is ignored, so both "normal" floats and the F-Logic representation of floats can be read). E.g., the following holds:

```
string2float(''42'',42), string2float(''3D'',3), string2float(''3.14'',#3.14), strin
```
(7.3)

Note that integers are seamlessly integrated with floats: FLORID never prints 3 as #3, but always understands #3 as 3.

string2object(A,B) is true, if B is the object whose id is A (converted to lowercase letters).

```
?- string2object("john",O).
% Answer to query : ?- string2object("john",O).
O/john

?- string2object("John",O).
% Answer to query : ?- string2object("John",O).
O/john

?- string2object(S,john).
% Answer to query : ?- string2object(S,john).
S/"john"
```

If the given string denotes an integer or a float, B is the corresponding integer or float object.

Note that the above predicates are "bidirectional", but at least one of the arguments has to be bound:

```
?- string2float("3.14",B).
% Answer to query : ?- string2float("3.14",B).
B/#3.14

?- string2float(X,#3.14).
```

```
% Answer to query : ?- string2float(X,#3.14).
X/"3.14"
```

7.5 Aggregation

Since version 2.0, aggregation has been implemented in FLORID. An aggregation term has the form

agg{X[G_1,...,G_n]; body}

where *agg* is one of the usual aggregation operators min, max, count, and sum and b is the aggregation body (that is, a conjunction of literals).

agg{X[G_1,...,G_n]; body}

returns one value for every vector of values for [G_1,...,G_n]: All variable bindings satisfying body are calculated (intentionally yielding bindings for X,G_1,...,G_n). Then, the X's are grouped by [G_1,...,G_n] and for every group, agg is calculated and returned.
The list of grouping variables [G_1,...,G_n] is optional and may be omitted, e.g.,

?- count{C; C:person}.

If the aggregation body contains other variables than the grouping variables, these are local to the aggregation. The grouping variables may occur anywhere in the rule body (respectively the higher level aggregation body).

Like arithmetic expressions, aggregation terms may only occur in the built-in predicates "=", "<", ">", "<=", ">=". However, the aggregation body may contain built-in predicates with other aggregation terms. Thus, aggregation can be nested:

```
?- Z = max{X; john[salary@(Year)->X]}.
?- Z = max{X; john[salary@(Year)->X], Year < 1990}.
?- Z = count{Year; john.salary@(Year) <
                    max{X; john[salary@(Y2)->X], Y2 < Year}}.
```

The first query asks for the highest salary John ever got. The second query contains an additional aggregation goal restricting the aggregation to the years before 1990. The last query yields the number of years in which John earned less than in the best year before.

The semantics of an aggregation term in a comparison predicate, e.g., the inequality $V < agg\{X[G_1,\ldots,G_n]; body\}$ is defined as the conjunction $V < Z, Z = agg\{X[G_1,\ldots,G_n]; body\}$.

Syntactic Restrictions For obvious reasons the following syntactic restrictions apply:

- The aggregation variable and all grouping variables must occur in the aggregation body.
- The variables X,G_1,...,G_n are pairwise distinct.
- The aggregation body has to obey the safety restrictions, i.e., no variable may represent an infinite answer set.

Violating these restriction will cause an error.

The operator count gives the total number of variable bindings to the aggregation variable. Note that, different from count, the operators min, max and sum ignore objects other than integer without producing any error message or warning.

```
myset[items->>{10,40,apple,27,cheese}].
?- Z = count{X; myset[items->> X]}.
?- Z = sum{X; myset[items->> X]}.
```

will yield 5 and 77.

Aggregates and Stratification As in the case of negation, the set of objects to aggregate has to be completely established before the actual aggregation. In doubt, the user has to ensure this by explicitely specifying a strata separation (see Section 10.2).

In the following simple example we want to compute the shortest path in a given graph. Of course, cyclic graphs may lead to nontermination problems. See Example 14.3 for a possible solution.

```
%% EDB graph
edge[from=>node; to=>node; dist=>int].
edge::path.
shpath::path. %% shortest path

%% Rule to calculate transitive closure
p(E,P):path[from->X; to->Z; dist->D] :-
    E:edge[from->X; to->Y], P:path[form->Y; to->Z],
    D = E.dist + P.dist.

?- sys.strat.dolt. %% separate the program strata
P:shpath :- P:path[from->X; to->Z; dist->M,
    M = min{D[X,Z]; P:path[from->X; to->Z; dist->D]}.

?- sys.eval.
?- X:path.
?- X:shpath[from->M; to->N; dist->D].
```

8 The Object Base

An object base is specified by a set of facts which is completed by additional information maintained by the system. These so-called closure properties express some basic features of the object-oriented paradigm and are discussed in detail in this section.

In contrast to [KLW95], in FLORID some closure properties are not defined for all possible object names but only for the set of *active object names*. This ensures that (if we disregard integers) the object model remains finite, enabling easier static safety checking. Remember that the set of object names includes the set of ground unnested functional path expressions (see Section 6.2). A ground id-term is an active object name if it appears at any syntactical position of one of the facts specifying the object base. A ground unnested functional path expression is an active object name if the appropriate method is defined in the object base. We write $o \in \mathit{active}(\mathcal{I})$ if an object name o is active in $\mathcal{I}$.

8.1 Closure Properties of the Equality Predicate

The equality predicate determines a congruence relation over the set of object names which leads to the following implications: Let o, p, q be object names, t, t' atoms and $\mathcal{I}$ an arbitrary object base.

- If $\mathsf{o} \in \mathit{active}(\mathcal{I})$, then $\mathcal{I} \models \mathsf{o}{=}\mathsf{o}$ (reflexivity).
- If $\mathcal{I} \models \mathsf{o}{=}\mathsf{p}$, then $\mathcal{I} \models \mathsf{p}{=}\mathsf{o}$ (symmetry).
- If $\mathcal{I} \models \mathsf{o}{=}\mathsf{p}$ and $\mathcal{I} \models \mathsf{p}{=}\mathsf{q}$, then $\mathcal{I} \models \mathsf{o}{=}\mathsf{q}$ (transitivity).
- If $\mathcal{I} \models \mathsf{o}{=}\mathsf{p}$ and $\mathcal{I} \models t$, then $\mathcal{I} \models t'$ where t' is obtained from t by replacing o with p (substitution).

8.2 Closure Properties of Subclass Relationships

As already mentioned in Section 3.1.2, the subclass relationship specifies a partial order on the set of object names. Besides that, every object belonging to a class is always an instance of every superclass of this class. These properties are stated by the following implications: Let o, p, q be object names and $\mathcal{I}$ an arbitrary object base.

- If $o \in active(\mathcal{I})$, then $\mathcal{I} \models$ o::o (subclass reflexivity).
- If $\mathcal{I} \models$ o::p and $\mathcal{I} \models$ p::q, then $\mathcal{I} \models$ o::q (subclass transitivity).
- If $\mathcal{I} \models$ o::p and $\mathcal{I} \models$ p::o, then $\mathcal{I} \models$ o=p (subclass acyclicity).
- If $\mathcal{I} \models$ o:p and $\mathcal{I} \models$ p::q, then $\mathcal{I} \models$ o:q (subclass inclusion).

8.3 Closure Properties of Signatures

In combination with a given class hierarchy, a signature-F-atom implies other signature-F-atoms (e.g., the definition of a method for a class is valid for every subclass of that class, too). Other implications concern the replacement of a parameter class by a subclass or the result class by a superclass. This leads to the following implications (analogously for multi-valued signatures =>>): Let o, m, $a_1,\ldots,a_n$, r, x be object names ($n \geq 0$) and $\mathcal{I}$ an arbitrary object base.

- If $\mathcal{I} \models$ o[m@($a_1,\ldots,a_n$)=>r] and $\mathcal{I} \models$ x::o, then $\mathcal{I} \models$ x[m@($a_1,\ldots,a_n$)=>r] (type inheritance).
- If $\mathcal{I} \models$ o[m@($a_1,\ldots,a_i,\ldots,a_n$)=>r] and $\mathcal{I} \models$ x::a_i, then $\mathcal{I} \models$ o[m@($a_1,\ldots$,x,$\ldots,a_n$)=>r] (input-type restriction).
- If $\mathcal{I} \models$ o[m@($a_1,\ldots,a_n$)=>r] and $\mathcal{I} \models$ r::x, then $\mathcal{I} \models$ o[m@($a_1,\ldots,a_n$)=>x] (output-type relaxation).

8.4 Miscellaneous Properties

Some more closure properties hold for every object base in F-Logic. The first one states that a functional method may not yield different result objects if it is applied to the same host object with the same parameter objects. Furthermore, every functional path expression is equated to the corresponding result object. The third statement ensures that active object names without any properties are true in every object base. The fourth one says that if a path expression is active, it is a result of the corresponding method application. Finally the empty set, resp. empty signature, is implied as a result by any data-F-atom with a multi-valued method, resp. any signature-F-atom.
Let o, m, $a_1,\ldots,a_n$, r, s be object names ($n \geq 0$) and $\mathcal{I}$ an arbitrary object base.

- If $\mathcal{I} \models$ o[m@($a_1,\ldots,a_n$)->r] and $\mathcal{I} \models$ o[m@($a_1,\ldots,a_n$)->s], then $\mathcal{I} \models$ r=s (analogously for inheritable functional methods).
- If $\mathcal{I} \models$ o[m@($a_1,\ldots,a_n$)->r], then $\mathcal{I} \models$ o.m@($a_1,\ldots,a_n$)=r (analogously for inheritable functional methods).
- If $o \in active(\mathcal{I})$, then $\mathcal{I} \models$ o.
- If o.m@($a_1,\ldots,a_n$)$\in active(\mathcal{I})$, then $\mathcal{I} \models$o[m@($a_1,\ldots,a_n$)->o.m@($a_1,\ldots,a_n$)] (analogously for other types of path expressions).
- If $\mathcal{I} \models$ o[m@($a_1,\ldots,a_n$)->>{r}], then $\mathcal{I} \models$ o[m@($a_1,\ldots,a_n$)->>{}] (analogously for inheritable multi-valued methods).

- If $\mathcal{I} \models$ o[m@(a_1,...,a_n)=>(r)], then $\mathcal{I} \models$ o[m@(a_1,...,a_n)=>()] (analogously for multi-valued signatures).

9 Rules and Queries

Rules. Based upon a given object base (which can be considered as a set a facts), rules offer the possibility to derive new information, i.e., to extend the object base *intensionally*. Rules encode generic information of the form: "Whenever the precondition is satisfied, the conclusion also is". The precondition is called *rule body* and is formed by a conjunction of subgoals, i.e., possibly negated P- or F-molecules. The conclusion, the *rule head*, is a conjunction of P- and F-molecules. Syntactically the rule head is separated from the rule body by the symbol ":-" and every rule ends with a dot followed by a whitespace (blank, tab or return).[10]

Non-ground rules use *variables* for passing information between subgoals and to the head. Variables can be considered as implicitly universally quantified and range over one rule at a time. Thus, using the same variable in different subgoals in fact defines a join condition.

Assume an object base defining the methods father and mother for some persons, e.g., the set of facts given in Example (2.1). The rules in (9.1) compute the transitive closure of these methods and define a new method ancestor:

```
X[ancestor->>Y]  :-  X[father->Y].
X[ancestor->>Y]  :-  X[mother->Y].
X[ancestor->>Y]  :-  X[father->Z], Z[ancestor->>Y].
X[ancestor->>Y]  :-  X[mother->Z], Z[ancestor->>Y].
```
(9.1)

Queries. A query can be considered as a special kind of rule with empty head. For syntactical distinction, a query begins with the query symbol "?-" followed by a rule body and ends with a dot followed by a whitespace. The following query asks about all female ancestors of Jacob:

```
?- jacob[ancestor->>Y:woman].
```
(9.2)

The answer to a query consists of all variable bindings such that the corresponding ground instance of the rule body is true in the object base. Considering the object base described by the facts of Example (2.1) and the rules in (9.1), the query (9.2) yields the following variable bindings:

```
Y/rebekah
Y/sarah
```

[10] The condition that every rule or query has to end with a dot followed by a whitespace is necessary to avoid ambiguities because the dot is also used in path expressions:

```
X[grandma->sarah] :- X:man.father[mother->sarah].
```

Note that if the dot and father in that rule would be separated by a whitespace (typing error!), this would be equal to the following rule and an additional fact.

```
X[grandma->sarah] :- X:man.
father[mother->sarah].
```

Don't Care Variables. When asking queries, it is often useful to have variables for intermediate results that do not show up in the answer set. This can be achieved with *don't care variables*, that is, variable names starting with the underscore symbol, e.g., _1,_Y,_little. In rules, such a variable is not propagated to the head (using don't care variables in rule heads will cause an error message). The grandchildren of sarah could be queried ad hoc by the equivalent queries

```
?- X:man.father[mother->sarah].        and
?- X:man[father->_Y], _Y[mother->sarah].
```
(9.3)

The variable "_" (a single underscore) plays a special role: every occurrence is considered as a *distinct* don't care variable (internally, they are transformed into distinct variable names like _#1,_#2).

Don't care variables can often be avoided by using path expressions, but not in every case. Don't care variables can even spare additional rules when occurring in negated subgoals.

Negated Subgoals and Safety. Similar to Datalog, a rule has to satisfy some constraints to guarantee that the number of newly derived facts remains finite. A rule is called *safe* if all its variables are limited. A variable in a rule is *limited* if it is limited by at least one subgoal. A subgoal containing a variable X limits this variable if it is positive and none of the following cases applies:

- The subgoal is an equality P-molecule where the variable X or the molecule X is one of the arguments and the other argument is
  - either an arithmetic expression containing a variable that is not limited by another subgoal in the same rule, e.g., X = Y + 3
  - or it is just a variable, say Y, or a molecule of the form Y, where Y is not limited by another subgoal in the same rule, e.g., X = Y, X = Y, X = Y.
- The variable X or the molecule X appears as a an operand in an arithmetic expression[11], e.g., Z = X + 2, Z = 3 * X.
- The subgoal is a subclass F-molecule where both arguments are exactly a variable or a variable followed by empty brackets and none of them is limited by another subgoal in the same rule, e.g., X::Y, X::(Y).
- The subgoal is an F-molecule of the form X::Y.
- The subgoal is just the variable X followed by an empty specification, i.e., X.
- The subgoal consists of the built-in predicate integer with the variable X or the molecule X as its argument, i.e., integer(X), integer(X).
- The variable X or the molecule X is one of the arguments of a comparison P-molecule, e.g., X > 5, X > 5, X > Y.

Additionally, a don't-care variable in a negated subgoal is considered as limited, if this is the only occurrence of this variable in the rule.

Note that the concept of limited variables is slightly different from Datalog since equality as a reflexive relation is defined only for active object names (see Section 8). Hence, in F-Logic the variables in the subgoal f(X) = Y are limited, while in Datalog, they are not. Moreover, a variable appearing in a path expression or an F-molecule is always limited, e.g., the variable X is limited by each of the following subgoals:

[11] Remember that arithmetic expressions are only allowed in equality P-molecules

```
X[father->Y] = Z
X.father = Z
X.age > 5
integer(X.age)
```
(9.4)

However, the restriction to active object names does not solve all problems because the number of active object names is infinite due to the existence of the integer objects. For this reason a subgoal like X = Y does not limit the variables X or Y. But if just one of the two variables is limited by another subgoal, the other one is limited, too. This analogously holds for subgoals of the form X::Y.
The following Examples (9.5) and (9.6) illustrate the concept of limited variables and safe rules:

```
X[father->Y] :- X:person.
X:bachelor :- X:person, not X[spouse->Y].
X:object :- X.
isaac[age->X] :- rebekah[age->Y], Y = X - 10.
```
(9.5)

None of these rules is safe. In the first rule, the variable Y only appears in the rule head and therefore it cannot be limited by a subgoal. The second rule contains a negated subgoal with the variable Y which is not limited by any other subgoal. Nevertheless, replacing Y by a don't care variable makes it safe:

```
X:bachelor :- X:person, not X[spouse->_Y].
```

In the third rule, the variable X is not limited because it is used as a single variable with an empty specification and due to the integer objects the number of variable bindings for X is infinite. However, by replacing the subgoal by f(X) the rule would become safe because there is only a finite number of active object names matching the term f(X). Finally, the last rule contains an equality P-atom with an arithmetic expression as argument. Since the variable X is not limited by another subgoal the rule is not safe. Actually in the last case the problem is not the derivation of an infinite number of new facts. However, the evaluation strategy for arithmetic expressions requires that all their variables are bound. The rule can be made safe by rewriting the equation s.t. Y (bound) occurs in the arithmetic part instead of X:

```
isaac[age->X] :- rebekah[age->Y], X = Y + 10.
```

These following rules are safe:

```
X:adult :- X[age->Y], Y = Z, Z > 18.
X[older->>Y] :- X.age > Y.age.
X:person :- X.father.
```
(9.6)

In the first one, the variables X and Y are limited by the first subgoal. The equality P-atom limits the variable Z because Y is already limited. In the second rule the arguments of the comparison predicate are not single variables, so they are limited. This becomes even more obvious by rewriting the rule body as X[age->A], Y[age->B], A > B. In the third rule the variable X is limited because of the method father since in any object base this method is only defined for a finite number of objects.

Infinite number of object names. Another problem is the creation of an infinite number of new object names by the use of function symbols or path expressions. Consider the following rule:

```
X.father:person :- X:person.
```

If our object base contains at least one object belonging to the class person, an infinite number of new objects is created by evaluating this rule again and again. Every time another father object is created and the evaluation will never stop. As this kind of recursion may also occur indirectly involving several rules, it cannot be fixed by a static check.

10 Programs and Evaluation

An F-Logic program is a collection of facts and rules in arbitrary order. Evaluating these facts and rules bottom-up, an object base is computed which may then be queried. Note, however, that **queries are not part of a program**.

10.1 Fixpoint Semantics

The evaluation strategy for F-Logic programs (without inheritance) is basically the same as for Datalog programs. The bottom-up evaluation of an F-Logic program starts with a given object base. Initially, this is the empty object base. Facts are rules with an empty body, therefore always considered as true. The rules and facts of a program are evaluated iteratively in the usual way. If there are variable bindings such that the rule body is valid in the actual object base, these bindings are propagated into the rule head. New information corresponding to the ground instantiations of the rule head or deduced due to the closure properties is inserted into the object base[12]. This evaluation of rules is continued as long as new information is obtained. As in the case of Datalog, the evaluation of a negation-free F-Logic program reaches a fixpoint which coincides with the unique minimal model of that program[13]. The minimal object base of an F-Logic program is defined as the smallest set of P- and F-atoms such that all closure properties and all facts and rules of the program are satisfied.

10.2 Negation and Stratification

Negation in FLORID is handled according to the *inflationary semantics* [KP88]. Remember that in a safe rule, every variable in a negated subgoal has to be limited by other subgoals. Thus, only ground instantiated negated subgoals have to be considered during the evaluation. Such an instantiation of a negated subgoal is evaluated as true if and only if the intermediate object base given in the moment of the evaluation of the rule does not contain the corresponding information.
Consider the following program:

```
isaac[father->abraham].
isaac:orphan :- not isaac[father->abraham].
?- sys.eval.
?- isaac:orphan.
```
(10.1)

You may be surprised to get the answer true. However, this is correct under the inflationary semantics because in the first iteration of the evaluation the object base still is empty, so the negated subgoal evaluates as true and the information that Isaac is an orphan is inserted into

[12] To avoid redundancy, in FLORID most of the information generated by the closure properties is not inserted into the object manager explicitely, but deduced when retrieving information.
[13] Note that this fixpoint is not necessarily finite, cf. Example 9.

the object base as well as the fact that Abraham is Isaac's father. In the second iteration, the rule does not fire any more, but information inserted in the object base is never removed from there. Hence, the answer to the query is true.

As in the case of Example (10.1) inflationary semantics often yields unintended results. Hence, there are other concepts to handle negation in logic programs. One of the most general solutions is the three-valued *Well-Founded Semantics* [VGRS91]. A three-valued model is not supported by FLORID but can easily be simulated [MLL97] (cf. Example 14.10). A very common approach is to *stratify* logic programs and to compute the perfect model [ABW88, Prz88]. Unfortunately, due to the powerful syntax of F-Logic, the class of stratified programs is very small. Thus, automatic stratification cannot be done by FLORID for a reasonable large class of programs. To enable the programmer to specify an explicit stratification, FLORID provides a system command "?- sys.strat.doIt." which divides a program into several strata. Information queried by a negated subgoal has always to be derived in lower strata than the stratum of the rule containing the negated subgoal[14].
The stratification command causes the evaluation of the rules in the higher stratum to be deferred until the fixpoint of the lower stratum is computed. Moreover, the rules of the lower stratum are not considered any more during the further evaluation. To illustrate this we extend Example (10.1) by the stratification command:

```
isaac[father->abraham].
?- sys.strat.doIt.
isaac:orphan :- not isaac[father->abraham].                    (10.2)
?- sys.eval.
?- isaac:orphan.
```

The first stratum of the program consists only of the fact that Abraham is Isaac's father. Hence, this is the only information inserted into the object base during the evaluation of the first stratum. Next, the rule belonging to the second stratum is evaluated. Since it is already derived that Abraham is the father of Isaac the rule does not fire anymore and Isaac is not said to be an orphan.

Negation of Complex Molecules. Negated subgoals may consist not only of atoms but also of molecules as the following example shows. This rule states that two persons are not related to each other if they have no common ancestor[15]:

```
X[notrelated->>Y] :- X:person, Y:person,
                     not X[ancestor->>Y..ancestor].            (10.3)
```

Although FLORID can handle negated molecules, their decomposition is not as easy as it is for positive molecules. It is not possible to transform this rule into a single one consisting of F-atoms only without changing the semantics. On the one hand, negated conjunctions become disjunctions of negated atoms what cannot be expressed in F-Logic. Path expressions on the other hand have to be decomposed by introducing existentionally quantified variables what is not possible in F-Logic, either[16].

[14] There are other possibilities to ensure that the negated subgoal becomes active only after the information to be negated has been inferred, cf. Example 14.10.

[15] To achieve the desired result the ancestor relation has to be reflexive, i.e., X[ancestor->>X] :- X:person.

[16] In version 2.0, don't-care variables have been introduced that can be viewed as existentionally quantified if they occur in negated subgoals. However, for safety reasons, negated don't-care variables cannot be propagated to another subgoal, so that this is no solution for flattening the rule (10.3). See also Section 9.

Instead, decomposing negated molecules requires an additional rule with a new predicate symbol in its head. The arguments of this predicate are all the variables appearing in the negated molecule and it is defined by a new rule whose body consists of the decomposition of the molecule (see Sections 4 and 6.3). Then, the negated molecule in the original rule is replaced by the negation of this new redicate. The result of decomposing the negated molecule in Example (10.3) is:

```
common_anc(X,Y)   :-  X[ancestor->>Z], Y[ancestor->>Z].
?- sys.strat.doIt.
X[notrelated->>Y]   :-  X:person, Y:person, not common_anc(X,Y).
```

Stratification to Enhance Efficiency. The system command "?- sys.strat.doIt." may also be used in negation-free F-Logic programs to speed up the evaluation. Consider the facts denoting the father and mother relationships in Example (2.1) and assume that we want to define the relationship ancestor and its inverse descendant:

```
% facts defining the father and mother relationships
...
X[ancestor->>Y] :- X[father->Y].
X[ancestor->>Y] :- X[mother->Y].
% all direct ancestors are computed so these two rules
% do not have to be considered anymore
?- sys.strat.doIt.                                              (10.4)
X[ancestor->>Y] :- X.father[ancestor->>Y].
X[ancestor->>Y] :- X.mother[ancestor->>Y].
% all ancestors are computed and the derivation of the descendant
% relation is deferred until now
?- sys.strat.doIt.
X[descendant->>Y] :- Y[ancestor->>X].
```

The stratification command may enhance efficiency by dividing a negation-free program into several strata whenever every subgoal in a rule only depends on rules belonging to the same or lower strata. This way we can avoid to evaluate non-recursive rules again and again always deriving the same information.

Another strategy to avoid redundant derivation of information is semi-naive evaluation [Ban86]. An adoption of this strategy is included in FLORID version 2.0. However, it turned out in practice that reasonably subdivide the programs into strata is still advantageous and will keep the overhead of semi-naive evaluation small.

11 Inheritance

FLORID provides *nonmonotonic behavioral inheritance*, i.e., propagation of results of a method application from a class to its instances and subclasses. Example (11.1) illustrates how inheritance works in the case of overriding. This program states that all persons believe in Baal except for Abraham and his descendants, who believe in god.

```
abraham:person[believes_in->god; descendant->>{isaac:person}].
ahab:person.
person[believes_in*->baal].
X[believes_in->god] :- abraham[descendant->>X:person].
?- sys.eval.
?- X[believes_in->god].
```
(11.1)

Evaluating each rule and fact of the program once derives that Abraham, Isaac and Ahab are persons, Isaac is a descendant of Abraham, Abraham believes in god and persons in general believe in Baal. Note that the method believes_in cannot be inherited to Abraham because it is already defined. However, this method could – at first sight – be inherited to Isaac. But this inheritance would lead to a wrong answer. Instead, inheritance is deferred until a fixpoint of the program is reached. Computing this fixpoint, it is derived that Isaac believes in god because he is a descendant of Abraham. Since the method believes_in is now defined for Isaac, inheritance is blocked then. Only inheritance of believes_in to Ahab is still possible and, since the evaluation has reached a fixpoint, is applied.

To improve efficiency, new information inherited from a class to an instance or a subclass is only derived when this information is needed to evaluate a rule body or to answer a query. For example, evaluating the facts in Example (11.2) without any other rule or query does not yet infer that Abraham believes in god:

```
person[believes_in*->god].
abraham:person.
```
(11.2)

Only when a query is stated

```
?- abraham[believes_in->X].
```

inheritance is evaluated and the answer is `X/god`.

To implement this concept FLORID uses *inheritance triggers* [KLW95, MK98]: An inheritance trigger is a tuple (c, o, m, v), meaning "class c can inherit the result v of the method application m to the object o". Given an inheritable method, two conditions have to be satisfied to activate an inheritance trigger:

1. The inheritable method is not yet defined for the object which would inherit the method.
2. There has to be an appropriate subgoal in a rule body or query which matches the inheritable method. With regard to Example (11.2) the following query meets this condition. The corresponding inheritance trigger consists of the method believes_in applied to the object abraham.

```
?- X[believes_in->Y].
```
(11.3)

However, *firing* an inheritance trigger is deferred until the evaluation of the program has reached a fixpoint. The reason is that by evaluating the rules of a program new information could be derived that overrides the inherited information and deactivates the inheritance trigger. Thus, logic rules are given priority over inheritance.

Multiple Inheritance. Dealing with multiple inheritance is a common problem in object oriented systems. Multiple inheritance means that an inheritable method is defined for two different classes which have a common instance. We have to distinguish two cases:
If there is a subclass relationship between these two classes, the method result of the superclass is overridden by the one of the subclass. This means that the method defined by the subclass is

inherited to the object. In Example (11.4) the object potiphar inherits the method believes_in from the class egyptian, hence, Potiphar believes in the Pharaoh.

```
person[believes_in*->god].
egyptian::person[believes_in*->pharaoh].
potiphar:egyptian.                                    (11.4)
?- sys.eval.
?- potiphar[believes_in->X].
```

In the other case there is no subclass relationship, i.e., there are two classes, incomparable in the partial order induced by the class hierarchy, which may inherit information to the same object.

Inheriting a functional method from both classes would cause the result objects to be equated. Again, this problem is solved by firing only one inheritance trigger at a time. However, the choice which one will fire is non-deterministic. In Example (11.5) Paul either serves Jesus or he serves Caesar but not both of them because inheriting the method serves once deactivates the trigger to inherit it from the other class, too.

```
paul:roman.
paul:christian.
roman[serves*->caesar].                               (11.5)
christian[serves*->jesus].
?- sys.eval.
?- paul[serves->X].
```

The question is which of the two inheritance triggers fires and which one is deactivated afterwards. This problem is similar to dealing with a disjunction in a rule head of a logic program. Hence, it is *non-deterministic* which of the two instances inherits the method from the class (see also [Kan97]). So, there are two minimal models satisfying Example (11.5).

Let us summarize the concept of inheritance in FLORID. Having reached a fixpoint we determine which inheritance triggers are active and fire exactly one of them. Next, we check if the inherited information may be used to derive other information by evaluating all rules again until a fixpoint is reached. So, fixpoint computation and firing a single inheritance trigger at a time alternate until a fixpoint is reached where no inheritance triggers are active anymore.

This strategy avoids the inheritance of lots of information which is not needed for the evaluation of a given F-Logic program. Nevertheless, the strategy is correct since every inherited method appearing in a rule body or query activates a trigger and other inherited methods not occurring in any rule body or query have no influence on the evaluation of the program.

On the other hand, the performance of evaluation may become very low if a lot of inheritance triggers are active because they are fired successively, only one at a time. Example 14.9 illustrates how evaluation may be improved in this case.

For a theoretical investigation of inheritance and a comparison with Default Logic, see [MK98]. In [MSL97] it is shown how interitance can also be used to model dynamic processes.

12 Type checking

Although signature-F-atoms are supported, automatic type checking is not implemented in FLORID because static type checking is not possible for general F-Logic programs. This is due

to the possibility to generate new subclass relationships or equalities dynamically by rules. However, given an object base (e.g., the fixpoint of an F-Logic program), type checking may be realized by some additional rules.

To verify whether every method in an object base is typed correctly we have to examine *type safety* and *type correctness*. Type safety means that there is no method application without a corresponding signature in the object base. Type correctness implies that every signature covering a given method application is satisfied by the latter, i.e., the result object of a method application has to be an instance of *all* result classes of the appropriate signatures.

The following rules check if every functional method without parameters is type safe and type correct. Other kind of methods may be handled analogously. To defer the evaluation of the type checking rules until the fixpoint of the original program is computed we use the stratification command "?- sys.strat.dolt." (see Section 10.2).

```
?- sys.strat.dolt.
type_safe(O,M,R,funct) :- O[M->R], O:C, C[M=>()].
?- sys.strat.dolt.
type_unsafe(O,M,R,funct) :- O[M->R], not type_safe(O,M,R,funct).
type_incorrect(O,M,R,funct) :- O[M->R], O:C, C[M=>D], not R:D.

?- sys.eval.
?- type_unsafe(Object,Method,Result,KindOfMethod).
?- type_incorrect(Object,Method,Result,KindOfMethod).
```

13 Querying the World Wide Web with Florid

Since version 2.0, FLORID allows access of the World Wide Web via a generic Web interface. In this section, an integration of Web access into F-Logic is presented which allows to explore, query, and restructure Web information, combining Web data and the local database in a uniform way. More detailed presentations and case-studies can be found in the following papers:

- [LHL+98] Using FLORID for handling semistructured data, especially on the World-Wide Web.
- [May99a] A case study in all details.
- [May99b] The data model and internal representation of the Web in FLORID.
- [MHLL99] Using generic rules for typical wrapping tasks.
- [May00b] Architecture and data-driven Web exploration.

13.1 Modeling the Web in F-Logic

Every resource available in the Web has a unique address, called *Uniform Resource Locator* (URL). URLs can be used to initiate Web access. When a Web document is fetched, FLORID associates it with an object name derived from its URL by exploiting the object creation feature of scalar path expressions (see Section 6). There is a predefined class url containing Web address strings and a predefined built-in active method get that, when applied to a member of url, accesses the corresponding Web document and integrates it into the database. The schema for Web access is:

```
url::string[get => webdoc].
```

Whenever for an instance u of class url the method u.get is called by a rule head, the document at the address u is automatically loaded and analyzed (u.get in a rule body simply addresses the object u.get).
By evaluating rules of the form

u.get :- <body> ,

the internal database is extended by a new Web document according to the Web model:

- the Web document which is accessible via the url u is accessed,
- it is assigned to the newly created object u.get (conceived as a large string),
- it is made an instance of class webdoc, and
- several properties are automatically filled in (cf. Figure 2):
  - The references to other urls are stored in the built-in attribute hrefs:

    u.get[hrefs@(ℓ) ->> u'] $\Leftrightarrow$ u.get contains " ℓ " .
  - If the Web access fails when a document is accessed via its url, u.get[error ->> <error_msg>] holds for the resulting error message.

```
url::string[get=>webdoc].                    % signature of url and
webdoc[url =>url; author =>string;                    % ... webdoc
       type =>string; hrefs@(string) =>>url ;
       modif =>string; error =>>string].
-------------------------------------------------------------------
wd1:webdoc[url->"url1"; hrefs@(" label")->>{"url2"}; ...]
wd2:webdoc[url->"url2"; type->"html"; ...]
```

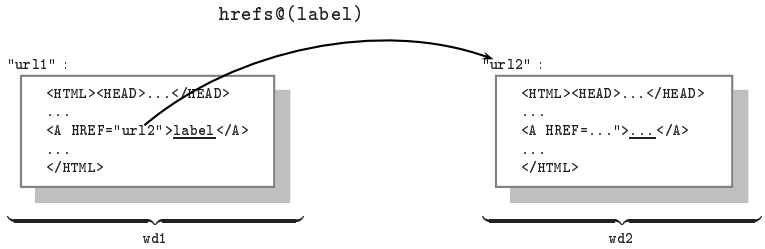

Figure 2: F-Logic Web model: signature and example data

Thus, loading Web documents is completely data-driven. New documents are fetched depending on information and links (i.e., URLs) found in already known documents.
Web documents can be seen from two points of view:

- Large strings: then, substr and pmatch are used for analyzing them (cf. Example 1).
- Trees of HTML tags: then, navigation on their parse-trees is used for analyzing them (see Section 13.3).

13.2 Traversing the Web: A First Example

We present some examples to illustrate the practical usage of these concepts. The first example deals with graph traversal of a limited area of the Web, starting at the homepage of our research group: `"http://www.informatik.uni-freiburg.de/~dbis/"` denotes an instance

of class url. The object representing the corresponding document is created by the fact "http://www.informatik.uni-freiburg.de/~dbis/".get.

```
"http://www.informatik.uni-freiburg.de/~dbis/" = ourUrl.
ourUrl:url.
ourServer = "www.informatik.uni-freiburg.de/".                    (13.1)
X.get:ourPage :- X:ourUrl.
Y.get:ourPage :- X:ourPage[hrefs@(_) ->> Y], substr(ourServer,Y).
```

The facts in the first two lines declare the URL of our homepage to be a member of the class url and its subclass ourUrl. The third fact defines a shorthand for the hostname of our server, which is later used to restrict the graph traversal. Then the call of the method X.get initializes the traversal. It implicitly triggers loading the document under ourUrl and puts it into the class ourPage. The last rule resembles the well-known transitive closure example and fetches all pages from our server which are reachable by following links from our homepage. As our server is not left, the search is limited and will terminate.

After evaluating the program above, the properties of the loaded pages can be examined. In order to detect missing links or documents that are not processed, type

```
?- X:ourPage[error ->> Y].                                        (13.2)
```

If it is not known what properties are defined, you may use variables at method position, e.g:

```
?- X:ourPage[Y -> Z].
?- X:ourPage[Y ->> Z].                                            (13.3)
```

In the context of Example (13.1) it might be interesting to get all mail addresses found on the visited pages:

```
?- X:ourPage[hrefs@(_) ->> Y], substr("mailto:",Y).               (13.4)
```

By the system command "?- sys.prn.style@("html").", a special output style is set that writes all query answers to an html file and calls a Netscape browser to display it. This has the advantage that URLs in the output are automatically tagged and highlighted, so they can be immediately examined with Netscape.

Another way to control graph traversals is to limit the search depth. Using the facts of Example (13.1) again, all pages are visited that can be reached from the start in less than four steps:

```
X.get:ourPage[depth ->> 0] :- X:ourUrl.
Y.get:ourPage[depth ->> N] :- X:ourPage[depth ->> M;hrefs@(_)->>Y],  (13.5)
        N = M + 1, N < 4.
```

Note that the method depth is multi-valued because there are possibly different paths leading to the same document from the start. If these differ in length, depth may have more than one result value. In case of a scalar method different integer values would be equated, which leads to system error messages.

13.3 Parse-Trees of Web documents

Similar to url.get, the active method url.parse generates the F-Logic representation of the parse-tree of an SGML document (for this, FLORID employs the SGML-parser *nsgmls*).

FLORID 2.0 provided only a 0-ary *u*.parse which generated an internal parse-tree representation which can serve as input for wrapper rules; although, the *modeling* does not fully

mirror the SGML or XML ideas. FLORID 2.5 and subsequent versions will support it for compatibility reasons, but we recommend to use the extended versions u.parse@(sgml) and u.parse@(xml) which are described in [May00a].

The parse-tree is transformed into an object-oriented representation and is made an object u.parse : parsetree:

- for every Web document wd, every SGML-tagged group <tag> ... </tag> is made an object. The class $wd.tag$ contains all such objects <tag> ... </tag> on wd, e.g., X:(wd.table) holds for all tables x on wd.
- each tag induces a method for navigation in a parse-tree: Let x : ($wd.tag$) be a node of the parse-tree, then $x.tag$@(0), ..., $x.tag$@(n) address the distinguished segments inside x, e.g.,
- For suitable $k = 0, 1, 2, \ldots$, wd.html@(k) addresses all distinguished segments of wd between <html> and </html>, e.g., wd.html@(0) is the head, providing wd.html@(0).head@(i) ($i = 1, 2, \ldots$) and wd.html@(1) is the body, providing wd.html@(1).body@(j) ($j = 1, 2, \ldots$).

Example 3 (Tables) *Tables are represented in HTML by the tags <table>, <tr> (table row), <td> (column elements containing data), and <th> (column elements containing header entries), e.g.,*

```
<table>
 <tr><th>Name</th><th>Birthday</th><th>Affiliation 1998</th></tr>
 <tr><td>D.E.Knuth</td>  <td>10-01-1938</td>Stanford Univ.<td></td></tr>
 <tr><td>...</td>  <td>...</td>  <td>...</td></tr>
 ...
</table>
```

- *all tables whose header contains '1998' in any header row/column are identified by*

  T:(wd.table), T.table@(R).tr@(C)[th@(0)->S], substr(S,''1998''). (13.6)

- *the third column of the 17th row of a given table tab is addressed by tab.table@(17).tr@(3).*

Example 4 (Parsetree in F-Logic representation) *Figure 4 shows the parsetree of the* Information Systems, Vol. 1 *page of the DBLP server [DBL98] as given Figure 3.*
Note the difference between tag instances, tag classes, tag attributes, *and* tag contents*:*

element	class	attribute	contents
"journals/is/is1".parse	html	–	html@(0),html@(1)
"journals/is/is1".parse.html@(1)	body	–	body@(0),...,body@(4)
"...".parse.html@(1).body@(0)	h1	–	h1@(0)
"...".parse.html@(1).body@(0).h1@(0)	a	href	a@(0)
"...".parse.html@(1).body@(0).h1@(0).href	url	–	"is/index.html"
"...".parse.html@(1).body@(0).h1@(0).a@(0)	string	–	"Information Systems"
"...".parse.html@(1).body@(2)	ul	–	ul@(1),...
"...".parse.html@(1).body@(2).ul@(0)	li	–	li@(0),li@(1)
"...".parse.html@(1).body@(2).ul@(0).li@(0)	a	href	a@(0)
"...".parse.html@(1).body@(2).ul@(0).li@(0).href	url	–	"a-tree/s/Senko.html"
"...".parse.html@(1).body@(2).ul@(0).li@(0).a@(0)	string	–	"M.E. Senko"

Then, rules containing path expressions can be used for extracting information from the parse-trees.

```
<html>
<head>
<title>Information Systems, Volume 1 </title>
</head>
<body>
<h1>
  <a href="index.html"> Information Systems </a>, Volume 1, 1975
</h1>
<h2> Volume 1, Number 1 </h2>
<ul>
<li><a href="../../indices/a-tree/s/Senko:Michael_E=.html">
      Michael E. Senko</a>:
    Information Systems Records, Relations, Sets, Entities, and
    Things. 3-13
<li><a href="../../indices/a-tree/g/Ghosh:Sakti_P=.html">
      Sakti P. Ghosh</a>,
    <a href="../../indices/a-tree/l/Lum:Vincent_Y=.html">
      Vincent Y. Lum</a>:
    Analysis of Collisions when Hashing by Division. 15-22
<li><a href="../../indices/a-tree/s/Schneider:G=_Michael.html">
      G. Michael Schneider</a>,
    <a  href="../../indices/a-tree/d/Desautels:Edouard_J=.html">
      Edouard J. Desautels</a>:
    Creation of a File Translation Language for Networks. 23-31
</ul>
<h2>Volume 1, Number 2 </h2>
<ul>
<li> [...]
<li> [...]
<li> [...]
</ul>
</body>
</html>
```

Figure 3: Sourcecode of the DBLP *Information Systems, Vol. 1* page

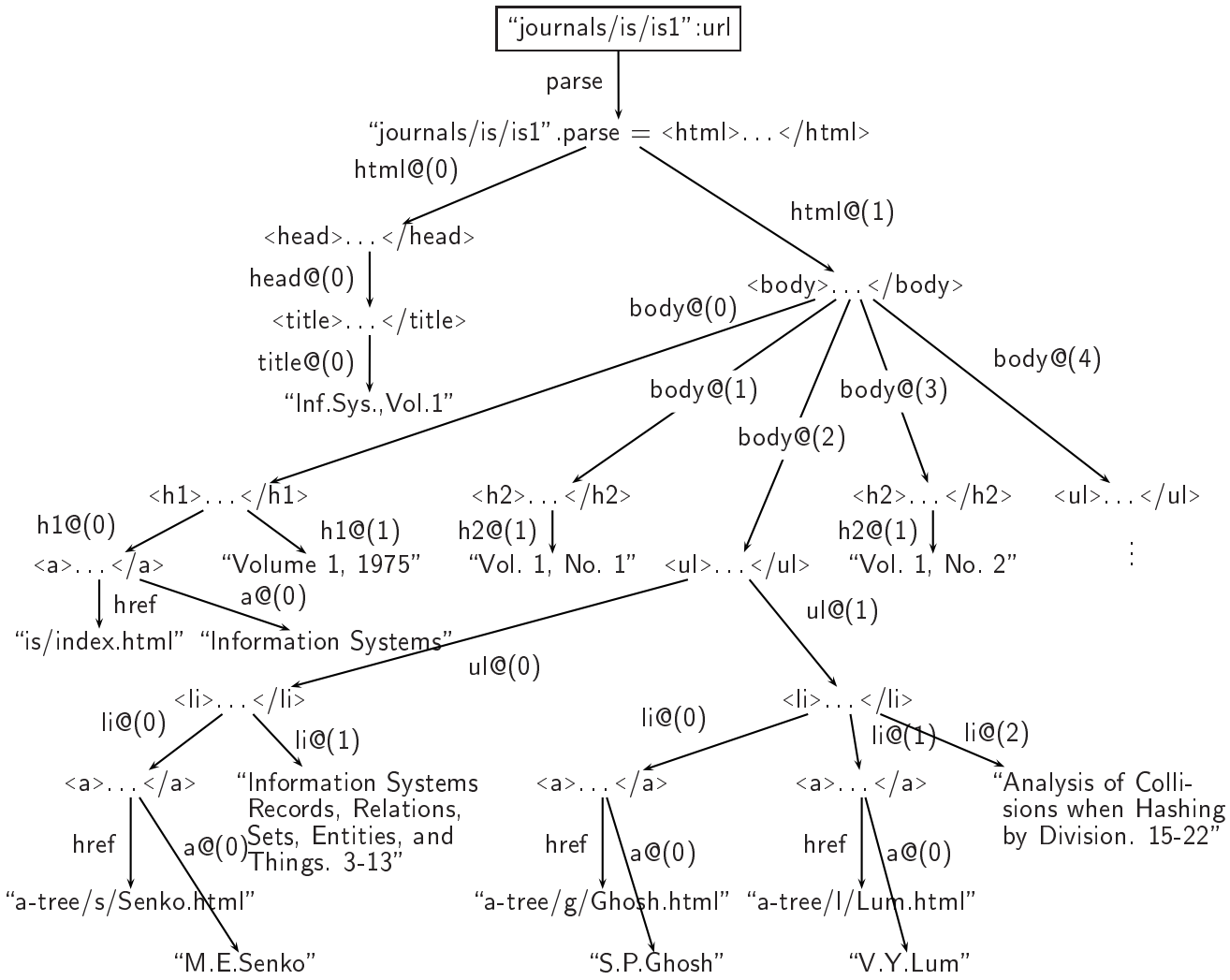

Figure 4: F-Logic representation of the parse-tree of the DBLP *Information Systems, Vol. 1* page

SGML and XML. For the SGML/XML handling functionality of FLORID, see [May00a].

14 Some Example Programs

The programs presented in this section demonstrate data modeling in F-Logic and serve to explain some typical problems that may arise. All these programs are found in the directory florid/examples/tutorial. For additional information about installing and using the system see the user manual.

14.1 Rules and Path Expressions

```
%% family.flp
%% Shows the use of scalar and multivalued methods and path expressions

/*
                  henry
                    |
           ------------------
           |                 |
          tim              theo
          |                 |
     ----------            ----
     |        |               |
   paul     peter            mike
     |        |               |
   ----     ---               ---
   |  |     |                   |
john jack cliff                abe

*/

henry : person.
tim:person[father -> henry].
theo:person[father -> henry].
paul:person[father -> tim].
peter:person[father -> tim].
mike:person[father -> theo].
john:person[father -> paul].
jack:person[father -> paul].
cliff:person[father -> peter].
abe:person[father -> mike].

/* RULES */
X[son ->> Y]      :- Y[father -> X].
X[brother ->> Y]  :- X[father -> Z], Y[father -> Z], not X=Y.
% equivalent rule with path expressions:
% X[brother ->> Y]  :- X[father -> Y.father], not X=Y.

X[uncle ->> Y]    :- X[father -> Z] , Z[brother ->> Y].
```

```
% equivalent rule with path expressions:
% X[uncle ->> Y]    :- X.father[brother ->> Y].

Y[nephew ->> X]   :- X[uncle ->> Y].
X[cousin ->> Z]   :- X[uncle ->> Y] , Z[father -> Y].
X[ancestor ->> Y] :- X[father -> Y].
X[ancestor ->> Y] :- X[ancestor ->> Z[father -> Y]].
% equivalent rule with path expressions:
% X[ancestor ->> Y] :- X..ancestor[father -> Y].

?- sys.eval.

/* QUERY */
?- X[cousin ->> Y].
?- X[ancestor ->> Y].
```

This example models a (somewhat patriarchalic) family tree. The rules define the methods son, brother, uncle, nephew, cousin and ancestor in terms of the given method father. The negation in the brother rule is not critical because no user defined method or predicate, but only a built-in predicate is involved.

Variables only occurring in the rule body may often be avoided by using path expressions, as the alternative rules for brother and uncle show[17]. Replacing a variable in the rule head by a path expression, however, may alter the semantics. For example, if we change the son rule into

```
Y.father[son ->> Y] :- Y:person.
```

a new object henry.father would be created.

14.2 Generic Methods

```
%% generic_methods.flp
%% Demonstrates a more complex use of path expressions

%% calculate transitive closure of a method M

X[(M.tc)  ->> Y] :- X[(M:basicmethod) -> Y].
X[(M.tc)  ->> Y] :- X..(M.tc)[(M:basicmethod) -> Y].

%% define the inverse of a method M

X[(M.inv)  ->> Y] :- Y[(M:basicmethod) -> X].

%% same generation problem for a method M

X[(M.sg)  ->> X] :- X:object, M:basicmethod.
```

[17] In most cases, keeping the number of subgoals small will lead to a faster evaluation because the additional information from the complex molecules can be used in the matching algorithm.

```
X[(M.sg)  ->> Y] :- X.(M:basicmethod)..(M.sg)[(M.inv) ->> Y].

%% apply this to family.flp:

father:basicmethod.
person::object.

?- sys.consult@("family.flp").

?- X[(father.tc)  ->> Y].
?- X[(father.inv)  ->> Y].
?- X[(father.sg)  ->> Y].
```

Here a similiar functionality as in family.flp is implemented in a more involved but general way. The variable M is bound to every instance of the class basicmethod and its transitive closure, inverse and same generation relation is calculated. Here, it is applied to the method father of the preceding example[18]. The new method (father.tc) is just the same as anc, and (father.inv) corresponds to son. However, the relation expressed by (father.sg) denotes a proper superset of the relation defined by cousin since it also contains the brother relation and the identity relation.

Note: The way F-Logic handles variables ranging over methods is similar to HILOG [CKW93]. Note also that path expressions in rule heads may cause the creation of new objects (e.g., father.tc). As mentioned above, the names of such objects are path expressions.

14.3 Using Equality to Ensure Finiteness

```
%% cycles.flp
%% Eliminate cycles from pathes by equating

/* Schema */

edge[start =>node; end =>node].

path[start =>node; end =>node;
     add@(edge) =>path;
     add@(path) =>path].

edge::path.
cycle::path.

/* Facts */

e7:edge[start -> n1; end -> n2]. %% This is a simple 3-cycle
e8:edge[start -> n2; end -> n3].
e9:edge[start -> n3; end -> n1].
```

[18] The consult command assumes that FLORID is started from a directory containing family.flp

```
/* RULES */

%% add edges to paths
P.E:path[start -> X; end -> Z] :-
           P:path[start -> X; end -> Y],
           E:edge[start -> Y; end -> Z].

%% concatenate paths
P1[(P2.E) -> P3]:-
        P1.P2[E -> P3], E:edge.

%% detect cycles
P:cycle :- P:path[start -> P.end].

%% eliminate cycles by equating
P.C = P :- P.(C:cycle).

?- sys.eval.

/* Query */
?- P:path.
```

When scanning a graph that possibly has cycles, it is often necessary to avoid situations like in Example 9 to keep the model finite. In such cases, the use of equating can be helpful. In the example above, the three edges `e7`,`e8`,`e9` form a cycle. The first two rules define the possible paths in this graph as usual. Due to object creation, this rules alone would not yield a finite model because cycling generates arbitrary long paths. The last two rules detect cycles in the EDB graph and equate paths that differ only in the number of cycles. This means that the path `e7.e8.e9.e7` is only a different (and longer) name for the object denoted by `e7` and thus the fixpoint remains finite.

14.4 Unintended Equality

```
%% blasphemy.flp
%% Example for side effects of equality in
%% combination with scalar methods

Z[arrow@(X) -> Y]:- Z[X -> Y].   %% This seems to be independent of the
Z[arrow@(X) -> Y]:- Z[X *-> Y].  %% rest of the program

creation[createdBy *-> god]. %% God created everything
man::creation.
male::man.
female::man.
adam:male.
eve:female.
```

```
%% But the devil sent a snake...

snake::creation[createdBy -> devil].

%% It turned man into a creation of the devil:

?- sys.eval.

?- adam[createdBy -> devil].
```

This example shows that an implicite object equating may have unexpected side effects to the rest of the program. The crucial point is that for the class snake the method createdBy is overloaded with respect to the inheritability. This means that the invocation of the method createdBy to the object snake yields two different results: Since snake is a subclass of creation it inherits the method createdBy which *remains inheritable* and results in god. On the other hand the method createdBy is directly defined as a *non-inheritable* method yielding devil.

The arrow rules are apparently independent of the rest of the program[19] but in fact they derive the equality god = devil. The first rule defining the method arrow derives the fact "snake[arrow@(createdBy)->devil].", the fact "snake[arrow@(createdBy)->god]." is derived by the second rule. Since the method arrow is functional and it is applied to the same object with the same parameter the result objects god and devil are equated. We can eliminate this effect by using a multi-valued method arrow instead.

Note: See also the section about debug modi in the user manual.

14.5 Negation and Stratification

```
%% bachelor.flp
%% User stratification is needed to achieve the
%% intended negation semantics

/*
    john --- mary
          |
      ---------
      |       |
    tina    peter --- sally
                    |
              -----------
              |         |
            tim       jack
*/

john : person[spouse -> mary].
peter : person[father -> john; mother -> mary; spouse -> sally:person].
```

[19] Initially the aim of these rules was to collect all methods defined for an object, whether inheritable or non-inheritable, so that they could be queried in a uniform way. The case that *both* are defined was not considered.

```
mary : person.
tina : person[father -> john; mother -> mary].
tim : person[father -> peter; mother -> sally].
jack : person[father -> peter; mother -> sally].

X[spouse -> Y] :- Y[spouse -> X].
X : married :- X[spouse -> Y].

X[par ->> Y] :- X[father -> Y].
X[par ->> Y] :- X[mother -> Y].

X[anc ->> Y] :- X[par ->> Y].
X[anc ->> Y] :- X..anc[par ->> Y].

married :: person.
bachelor :: person.

?- sys.strat.doIt.

X : bachelor :- X : person, not X : married.

?- sys.eval.

?- X : bachelor.
```

To use negation on user-defined information (i.e., information that is not part of the built-in concepts) you have to specify a stratification of the program. This is done by inserting the system command "`?- sys.strat.doIt.`" between the strata. If all facts that may match the negated subgoal are derived in previous strata, not in the same stratum or higher strata, the program will yield the intended answer. In the example above, the class married is established in the first stratum and used negatively in the second one.

Note: Due to the expressive power of F-Logic, no sufficiently large class of stratified programs exists which is statically decidable. So additional information has to be supplied by the user. It is topic of current research to provide this in a declarative way instead of the nondeclarative one described above.

14.6 Subset relationship

```
%% set_comparison.flp
%% Example for subset relationship of multi-valued methods
abraham[son ->> {ishmael, isaac}].
sarah[son ->> {isaac}].

% Is the set of Abraham's sons a subset of Sarah's sons?
?- sys.strat.doIt.
nosubset(abraham,son,sarah,son) :- abraham[son ->> Z], not sarah[son ->> Z].
?- sys.strat.doIt.
```

```
?- not nosubset(abraham,son,sarah,son).
```

Although variables in F-Logic can only range over objects, not over sets, set operations may be expressed by rules. This example asks for the subset relationship of two sets given as the result of a multi-valued method application. The query whether the set of Abraham's sons is a subset of Sarah's sons (the answer is *false*) is evaluated in two steps. First, it has to be inferred whether there are any sons of Abraham who are not sons of Sarah, then the negation of this answer is taken as result.

Note the difference to the query

```
?- sarah[son ->> abraham..son].
```

from Example (6.8)

```
% Is the set of X's sons a subset of Y's friends?
?- sys.strat.doIt.
nosubset(X,son,Y,friend) :- X[son ->> Z], Y[friend ->> {}], not Y[friend ->> Z].
?- sys.strat.doIt.
?- X[son ->> {}], Y[friend ->> {}], not nosubset(X,son,Y,friend).
```

All F-atoms containing the empty set as result are only needed for safety reasons to limit the variables occurring in the negated subgoal (see Section 9).

14.7 Inheritance and Negation

```
%% jonathan.flp
%% negation together with inheritance triggering causes problems

bird::animal.
bird[can *-> fly].
jonathan:bird.

?- sys.strat.doIt.

X[can -> swim] :- X:animal, not X[can -> fly].

?- sys.eval.

?- jonathan[can -> fly].
?- jonathan[can -> swim].
```

In this example the negation does not work as intended. The outcome of the program is that jonathan can swim but cannot fly. What happens is that in the first stratum the (active) trigger that makes jonathan inherit the method can from the class bird is not fired because there is no subgoal in this stratum activating this trigger. So in the second stratum jonathan[can->swim] is inferred and when asking the query the inheritance trigger is not active anymore. To avoid this we have to enforce inheritance triggering with a dummy rule in the first stratum:

```
bird::animal.
bird[can *-> fly].
jonathan:bird.

dummy:- X[can -> Y].    % dummy rule to enforce triggering

?- sys.strat.doIt.

X[can -> swim] :- X:animal, not X[can -> fly].
```

This version will yield the correct result.

Note: Negation combined with inheritance triggering is a major problem. See also appendix B in [KLW95] for a theoretical approach.

14.8 Negation with Inflationary Semantics

```
%% adulterer.flp
%% Inflationary semantics of negation yields unsupported model
%% Here, the problem is caused by crosswise negation and equating

john[spouse -> jane].
john[kissed -> peter.sister].
john:adulterer:-john[kissed -> Woman], not john[spouse -> Woman].
john[spouse -> Woman]:-john[kissed -> Woman], not john:adulterer.

?- sys.eval.
?- john:adulterer.
?- john[kissed -> X].
?- peter[sister -> Y].
```

The first evaluation round derives the facts that John is married to Jane and kissed Peter's sister. In the next round it is deduced that John is an adulterer because Jane is not known to be Peter's sister. At the same time it follows from the second rule that John is married to Peter's sister because John is not known to be an adulterer. The scalarity of the method spouse forces the equating of the two objects jane and "peter.sister".

In a third round none of the rules fires and the inferred facts are not removed from the object base. So John is an adulterer because he kissed his wife.

Note: This behavior is due to the pure inflationary treatment of negation and only avoidable by stratification. However, this program is not stratified because the corresponding dependency graph is cyclic [Ull89]. Another solution of this problem is be the application of the well-founded semantics [VGRS91]. See Example 14.10.

14.9 Speed up Inheritance

```
%% tones.flp
%% Calculates the intervals between tones within one octave
%% Heavy use of inheritance triggering
```

```
%% base tones

a:tone[number -> -1;chro -> -2].
b:tone[number -> 0;chro -> 0].
c:tone[number -> 1;chro -> 1].
d:tone[number -> 2;chro -> 3].
e:tone[number -> 3;chro -> 5].
f:tone[number -> 4;chro -> 6].
g:tone[number -> 5;chro -> 8].

tone::note.

%% notes are derived from base tones by sharpening or flattening

sharp(X):note[number -> X.number;chro -> Y]:- Y=X:tone.chro + 1.
flat(X):note[number -> X.number;chro -> Y]:- Y=X:tone.chro - 1.

%% base intervals are defined by the distance between base tones

prime::interval[diff *-> 0].
second::interval[diff *-> 1].
third::interval[diff *-> 2].
fourth::interval[diff *-> 3].
fifth::interval[diff *-> 4].
sixth::interval[diff *-> 5].
seventh::interval[diff *-> 6].

%% there are derivates of base intervals with different real distance

plainPrime:prime[chrodiff -> 0].
augPrime:prime[chrodiff -> 1].

dimSecond:second[chrodiff -> 0].
smallSecond:second[chrodiff -> 1].
bigSecond:second[chrodiff -> 2].
augSecond:second[chrodiff -> 3].

dimThird:third[chrodiff -> 2].
smallThird:third[chrodiff -> 3].
bigThird:third[chrodiff -> 4].
augThird:third[chrodiff -> 5].

dimFourth:fourth[chrodiff -> 4].
plainFourth:fourth[chrodiff -> 5].
augFourth:fourth[chrodiff -> 6].

plainFifth:fifth[chrodiff -> 7].
```

```
dimFifth:fifth[chrodiff -> 6].
augFifth:fifth[chrodiff -> 8].

dimSixth:sixth[chrodiff -> 7].
smallSixth:sixth[chrodiff -> 8].
bigSixth:sixth[chrodiff -> 9].
augSixth:sixth[chrodiff -> 10].

dimSeventh:seventh[chrodiff -> 9].
smallSeventh:seventh[chrodiff -> 10].
bigSeventh:seventh[chrodiff -> 11].

%% (insert optimization here)

%% Two notes give an interval
X[to@(Y) -> Z] :- X:note,Y:note,
                  V=Y.chro-X.chro, U=Y.number - X.number,
                  Z:interval[diff -> U;chrodiff -> V].

?- sys.eval.

?- X[to@(Y) -> Z].
```

There is no semantic problem here. The program works fine but is quite slow. The reason is that all intervals inherit the attribute diff from their respective base intervals. The trigger concept of F-Logic requires that these triggers fire one at a time with computing a complete fixpoint afterwards. So 24 fixpoints are needed for this simple program and the time consuming last rule is evaluated over and over.

The first approach to an optimization is to separate the interval definitions from the last rule by stratification (inserting "?- sys.strat.doIt." before the last rule). Unfortunately this does not change anything because FLORID only fires a trigger when it is enabled by a subgoal[20]. However, as there is no appropriate subgoal no inheritance trigger fires in the first stratum. In the second stratum firing one trigger and computing a new fixpoint alternates, just as it was the case without stratification.

The solution is that all interesting triggers are enforced to fire already in the first stratum which can be achieved by a dummy rule:

```
dummy:-X:interval[diff -> Z]. %% dummy rule forcing inheritance triggering
?- sys.strat.doIt.            %% user stratification
```

The evaluation is sped up with this optimization approximately by factor 20.

Note: Version 2.0 of FLORID now contains an adoption of the semi-naive evaluation strategy. This also speeds up inheritance triggering, but it turned out that the effect of the dummy rule is by far greater. However, combining the dummy rule with semi-naive evaluation is even better and yields the best results.

[20] This is sort of a top-down element in the evaluation algorithm used.

14.10 Well-founded Semantics

```
%% wellfounded.flp
%% uses F-Logic triggers to emulate well-founded semantics

%----------  CONTROL RULES FOR WELL-FOUNDED DATALOG -------------

% inherit 'fixpoint'  after fixpoint is reached:
state[fixpoint  *-> t].

% initial state:
0:state.

% after fixpoint of an active state [S] is reached, create [S+1]:
s(S):state :- S:active.fixpoint.

% create all even states (while [S] is active)
0:even.
s(s(S)):even :- (S:state):even.

% [0] is active.
% [S+1] is active if [S] is an active even state.
% [S+2] is active if [S] is even and new tuples have been computed.
0:active.
s(S):active :- (S:active):even.
s(s(S)):active :- S:even, win(s(s(S)),X), not win(S,X).

% the first inactive even state is the final state
S:final :- (S:even).fixpoint, not S:active.

% compute the final outcome:
%       everything in the (even) final state is true
%       everything in the previous but not in the final state is undefined
win_true(X) :- S:final, win(S,X).
win_undef(X) :- s(S):final, win(S,X), not win(s(S),X).


%------------- USER-DEFINED WELL-FOUNDED RULES ------------------------------

% EDB
move(a,b).
move(b,a).
move(b,c).
move(c,d).
move(d,e).
move(e,f).
```

```
% IDB:
% for negated subgoals use previous state!

win(s(S),X) :- s(S):state, move(X,Y), not win(S,Y).

//?- sys.theEval.debugOn@(cout).
?- sys.eval.
?- win_true(X).
```

In this example, the procedural part of the inheritance definition via triggers is used to emulate the well-founded semantics for Datalog. This serves as a wrap for any Datalog program, here the prototypic example for the well-founded semantics, the win-move program is used. The method carries over to all F-Logic programs that do not use inheritance. Programs with inheritance can be handled, too, if the control rules generating the next state are the last ones in the program. FLORID always fires the first active trigger found in the program. Thus, the next state is only generated when there is no more active trigger in the target program for the actual state.

Acknowledgements. First of all, we want to thank Georg Lausen, the head of our group, who made the FLORID project possible. Furthermore, our thanks go to the former team members Jürgen Frohn, Rainer Himmeröder, Paul-Th. Kandzia, Bertram Ludäscher, Christian Schlepphorst, Markus Seilnacht, and Heinz Uphoff who developed FLORID up to version 2.0 together with students from the universities at Mannheim and Freiburg.

A Grammar of F-Logic Syntax in Backus-Naur-Form

A.1 Lexical structure

Before defining how F-Logic expressions are correctly built we describe the basic lexical structure of F-Logic in Figure 5:

RuleDelimiter	=	"." Whitespace
Whitespace	=	*Blank*
	|	*Tabulator*
	|	*Return*
ISASymbol	=	":" | "::"
ImplicationSymbol	=	":-"
QuerySymbol	=	"?-"
MethodArrow1	=	"->" | "*->"
MethodArrow2	=	"->>" | "*->>"
MethodArrow3	=	"=>" | "=>>"
Dot	=	"." | ".." | "!" | "!!"
Predicate	=	*character string starting with a lower case letter*
BuiltInPredicate	=	"integer"
InfixBuiltInPred	=	">" | "<" | "=" | ">=" | "<="
BuiltInOperator	=	"+" | "-" | "*" | "/"
Functor	=	*character string starting with a lower case letter*
Variable	=	*character string starting with an upper case letter or* "_"
String	=	*character string enclosed in " "*
Integer	=	*integer*

Figure 5: Lexical structure

A.2 Grammar of F-Logic syntax

Figure 6 shows the BNF grammar of the F-Logic syntax. Expressions enclosed in curly braces are optional.

```
Program          =  { ListOfRules }
ListOfRules      =  Rule RuleDelimiter { ListOfRules }
Rule             =  Head { ImplicationSymbol Body }
Query            =  QuerySymbol Body RuleDelimiter
Head             =  ListOfMolecules
ListOfMolecules  =  Molecule { "," ListOfMolecules }
Body             =  ListOfLiterals
ListOfLiterals   =  Literal { "," ListOfLiterals }
Literal          =  { "not" } Molecule
Molecule         =  F-Molecule | P-Molecule

P-Molecule       =  Predicate { "(" ListOfExpressions ")" }
                 |  BuiltInPredicate  "(" ListOfExpressions ")"
                 |  ArithExpr InfixBuiltInPred ArithExpr
ListOfExpressions =  Expression { "," ListOfExpressions }
Expression       |  PathExpression
                 |  F-Molecule
                 |  Aggregate
ArithExpr        =  Expression
                 |  ArithExpr BuiltInOperator ArithExpr
                 |  "(" ArithExpr ")"
Aggregate        =  ID-Term "{" Variable { "[" ListOfVars "]" }
                    ";" ListOfLiterals "}"
F-Molecule       =  PathExpression Specification
PathExpression   =  ID-Term
                 |  "(" Expression ")"
                 |  PathExpression Dot MethodApplication
                 |  F-Molecule Dot MethodApplication

Specification    =  ISASpecification "[" { ListOfMethods } "]"
                 |  ISASpecification
                 |  "[" { ListOfMethods } "]"
ISASpecification =  ISASymbol ID-Term
                 |  ISASymbol "(" Expression ")"
MethodApplication =  ID-Term { "@(" ListOfExpressions ")" }
                 |  "(" Expression ")" { "@(" ListOfExpressions ")" }
ListOfMethods    =  MethodApplication MethodResult { ";" ListOfMethods }
MethodResult     =  MethodArrow1 Expression
                 |  MethodArrow2 Expression
                 |  MethodArrow2 "{" { ListOfExpressions } "}"
                 |  MethodArrow3 Expression
                 |  MethodArrow3 "(" { ListOfExpressions } ")"
ID-Term          =  BasicID-Term
                 |  Functor "(" ListOfExpressions ")"
BasicID-Term     =  Functor | Variable | String | Integer
ListOfVars       =  Variable { "," ListOfVariables }
```

Figure 6: Grammar of F-Logic syntax

B Syntax of Regular Expressions

The following description of regular expression was taken from a xemacs[21] info file. Regular expressions have a syntax in which a few characters are special constructs and the rest are "ordinary". An ordinary character is a simple regular expression which matches that character and nothing else. The special characters are '$', '^', '.', '*', '+', '?', '[', ']' and '\'; no new special characters will be defined. Any other character appearing in a regular expression is ordinary, unless a '\' precedes it.

For example, 'f' is not a special character, so it is ordinary, and therefore 'f' is a regular expression that matches the string 'f' and no other string. (It does not match the string 'ff'.) Likewise, 'o' is a regular expression that matches only 'o'.

Any two regular expressions A and B can be concatenated. The result is a regular expression which matches a string if A matches some amount of the beginning of that string and B matches the rest of the string.

As a simple example, you can concatenate the regular expressions 'f' and 'o' to get the regular expression 'fo', which matches only the string 'fo'. To do something nontrivial, you need to use one of the following special characters:

- '. (Period)'
  is a special character that matches any single character except a newline. Using concatenation, you can make regular expressions like 'a.b', which matches any three-character string which begins with 'a' and ends with 'b'.
- '*'
  is not a construct by itself; it is a suffix, which means the preceding regular expression is to be repeated as many times as possible. In 'fo*', the '*' applies to the 'o', so 'fo*' matches one 'f' followed by any number of 'o's. The case of zero 'o's is allowed: 'fo*' does match 'f'.
  '*' always applies to the smallest possible preceding expression. Thus, 'fo*' has a repeating 'o', not a repeating 'fo'.
  The matcher processes a '*' construct by immediately matching as many repetitions as it can find. Then it continues with the rest of the pattern. If that fails, backtracking occurs, discarding some of the matches of the '*'-modified construct in case that makes it possible to match the rest of the pattern. For example, matching 'ca*ar' against the string 'caaar', the 'a*' first tries to match all three 'a's; but the rest of the pattern is 'ar' and there is only 'r' left to match, so this try fails. The next alternative is for 'a*' to match only two 'a's. With this choice, the rest of the regexp matches successfully.
- '+'
  is a suffix character similar to '*' except that it requires that the preceding expression be matched at least once. For example, 'ca+r' will match the strings 'car' and 'caaaar' but not the string 'cr', whereas 'ca*r' would match all three strings.
- '?'
  is a suffix character similar to '*' except that it can match the preceding expression either once or not at all. For example, 'ca?r' will match 'car' or 'cr'; nothing else.
- '[...]'
  '[' begins a "character set", which is terminated by a ']'. In the simplest case, the

[21] xemacs is free software under the GNU General Public License.

characters between the two form the set. Thus, ‘[ad]’ matches either one ‘a’ or one ‘d’, and ‘[ad]*’ matches any string composed of just ‘a’s and ‘d’s (including the empty string), from which it follows that ‘c[ad]*r’ matches ‘cr’, ‘car’, ‘cdr’, ‘caddaar’, etc.

You can include character ranges in a character set by writing two characters with a ‘-’ between them. Thus, ‘[a-z]’ matches any lower-case letter. Ranges may be intermixed freely with individual characters, as in ‘[a-z$%.]’, which matches any lower-case letter or ‘$’, ‘%’, or period.

Note that inside a character set the usual special characters are not special any more. A completely different set of special characters exists inside character sets: ‘]’, ‘-’, and ‘^’.

To include a ‘]’ in a character set, you must make it the first character. For example, ‘[]a]’ matches ‘]’ or ‘a’. To include a ‘-’, write ‘---’, which is a range containing only ‘-’. To include ‘^’, make it other than the first character in the set.

- ‘[^ ...]’
  ‘[^’ begins a "complement character set", which matches any character except the ones specified. Thus, ‘[^a-z0-9A-Z]’ matches all characters except letters and digits.
  ‘^’ is not special in a character set unless it is the first character. The character following the ‘^’ is treated as if it were first (‘-’ and ‘]’ are not special there).
  Note that a complement character set can match a newline, unless newline is mentioned as one of the characters not to match.
- ‘^’
  is a special character that matches the empty string, but only if at the beginning of a line in the text being matched. Otherwise, it fails to match anything. Thus, ‘^foo’ matches a ‘foo’ that occurs at the beginning of a line.
- ‘$’
  is similar to ‘^’ but matches only at the end of a line. Thus, ‘xx*$’ matches a string of one ‘x’ or more at the end of a line.
- ‘\’
  does two things: it quotes the special characters (including ‘\’), and it introduces additional special constructs.
  Because ‘\’ quotes special characters, ‘\$’ is a regular expression that matches only ‘$’, and ‘\[’ is a regular expression that matches only ‘[’, and so on.

Note: for historical compatibility, special characters are treated as ordinary ones if they are in contexts where their special meanings make no sense. For example, ‘*foo’ treats ‘*’ as ordinary since there is no preceding expression on which the ‘*’ can act. It is poor practice to depend on this behavior; better to quote the special character anyway, regardless of where is appears.

Usually, ‘\’ followed by any character matches only that character. However, there are several exceptions: characters which, when preceded by ‘\’, are special constructs. Such characters are always ordinary when encountered on their own. Here is a table of ‘\’ constructs.

- ‘\|’
  specifies an alternative. Two regular expressions A and B with ‘\|’ in between form an expression that matches anything A or B matches.

Thus, ‘foo\|bar’ matches either ‘foo’ or ‘bar’ but no other string.
‘\|’ applies to the largest possible surrounding expressions. Only a surrounding ‘\(... \)’ grouping can limit the grouping power of ‘\|’.
Full backtracking capability exists to handle multiple uses of ‘\|’.

- ‘\(... \)’
is a grouping construct that serves three purposes:
1. To enclose a set of ‘\|’ alternatives for other operations. Thus, ‘\(foo\|bar\)x’ matches either ‘foox’ or ‘barx’.
2. To enclose a complicated expression for the postfix ‘*’ to operate on. Thus, ‘ba\(na\)*’ matches ‘bananana’, etc., with any (zero or more) number of ‘na’ strings.
3. To mark a matched substring for future reference.
This last application is not a consequence of the idea of a parenthetical grouping; it is a separate feature which happens to be assigned as a second meaning to the same ‘\(... \)’ construct because in practice there is no conflict between the two meanings. Here is an explanation:
- ‘\DIGIT’
after the end of a ‘\(... \)’ construct, the matcher remembers the beginning and end of the text matched by that construct. Then, later on in the regular expression, you can use ‘\’ followed by DIGIT to mean ”match the same text matched the DIGIT’th time by the ‘\(... \)’ construct.”
The strings matching the first nine ‘\(... \)’ constructs appearing in a regular expression are assigned numbers 1 through 9 in order that the open-parentheses appear in the regular expression. ‘\1’ through ‘\9’ may be used to refer to the text matched by the corresponding ‘\(... \)’ construct.
For example, ‘\(.*\)\1’ matches any newline-free string that is composed of two identical halves. The ‘\(.*\)’ matches the first half, which may be anything, but the ‘\1’ that follows must match the same exact text.
- ‘\‘’
matches the empty string, provided it is at the beginning of the buffer.
- ‘\’’
matches the empty string, provided it is at the end of the buffer.
- ‘\b’
matches the empty string, provided it is at the beginning or end of a word. Thus, ‘\bfoo\b’ matches any occurrence of ‘foo’ as a separate word. ‘\bballs?\b’ matches ‘ball’ or ‘balls’ as a separate word.
- ‘\B’
matches the empty string, provided it is not at the beginning or end of a word.
- ‘\<’
matches the empty string, provided it is at the beginning of a word.
- ‘\>’
matches the empty string, provided it is at the end of a word.
- ‘\w’
matches any word-constituent character. The editor syntax table determines which characters these are.
- ‘\W’

matches any character that is not a word-constituent.

- ‘\sCODE’
  matches any character whose syntax is CODE. CODE is a character which represents a syntax code: thus, ‘w’ for word constituent, ‘-’ for whitespace, ‘(’ for open-parenthesis, etc.
- ‘\SCODE’
  matches any character whose syntax is not CODE.

Here is a complicated regexp used by Emacs to recognize the end of a sentence together with any whitespace that follows. It is given in Lisp syntax to enable you to distinguish the spaces from the tab characters. In Lisp syntax, the string constant begins and ends with a double-quote. ‘\"’ stands for a double-quote as part of the regexp, ‘\\’ for a backslash as part of the regexp, ‘\t’ for a tab and‘\n’ for a newline.

```
"[.?!][]\"')]*\\($\\|\t\\|  \\)[ \t\n]*"
```

This regexp contains four parts: a character set matching period, ‘?’ or ‘!’; a character set matching close-brackets, quotes or parentheses, repeated any number of times; an alternative in backslash-parentheses that matches end-of-line, a tab or two spaces; and a character set matching whitespace characters, repeated any number of times.

References

[ABD+89] Malcolm Atkinson, François 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.

[ABW88] Krzysztof R. Apt, H. Blair, and A. Walker. Towards a theory of declarative knowledge. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 89–148. Morgan Kaufmann, 1988.

[AHV95] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases.* Addison Wesley, 1995.

[Ban86] F. Bancilhon. Naive evaluation of recursively defined relations. In L. Brodie and J. Mylopoulos, editors, *On Knowledge Base Management Systems*, pages 165–178. Springer, 1986.

[CGT90] S. Ceri, G. Gottlob, and L. Tanca. *Logic Programming and Databases.* Springer, 1990.

[CKW93] 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.

[DBL98] DBLP computer science bibliography. `http://www.dblp.uni-trier.de`, 1998.

[FLU94] Jürgen Frohn, Georg Lausen, and Heinz Uphoff. Access to objects by path expressions and rules. In *Intl. Conference on Very Large Data Bases (VLDB)*, pages 273–284, 1994.

[Kan97] Paul-Th. Kandzia. Nonmonotonic Reasoning in Florid. In *Intl. Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR)*, number 1265 in LNAI. Springer, 1997.

[KLW95] 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.

[KP88] P.G. Kolaitis and C.H. Papadimitriou. Why not negation by fixpoint? In *ACM Symposium on Principles of Database Systems (PODS)*, pages 231–239, 1988.

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

[Liu96] M. Liu. ROL: A typed deductive object base language. In *Intl. Conference on Database and Expert Systems Applications (DEXA)*, 1996.

[May99a] Wolfgang May. Information extraction and integration with FLORID: The MONDIAL case study. Technical Report 131, Universität Freiburg, Institut für Informatik, 1999. Available from `http://www.informatik.uni-freiburg.de/~may/Mondial/`.

[May99b] Wolfgang May. Modeling and querying structure and contents of the web. In *Workshop on Internet Data Modeling (IDM'99), Proc. DEXA 99 Workshops*, pages 721–725. IEEE Computer Science Press, 1999.

[May00a] W. May. Handling XML with FLORID, 2000. Available from `http://www.informatik.uni-freiburg.de/~dbis/florid`.

[May00b] Wolfgang May. An integrated architecture for exploring, wrapping, mediating and restructuring information from the web. In *Australasian Database Conference*

(ADC 2000). IEEE Computer Science Press, to appear, 2000.

[MHLL99] Wolfgang May, Rainer Himmeröder, Georg Lausen, and Bertram Ludäscher. A unified framework for wrapping, mediating and restructuring information from the web. In *International Workshop on the World-Wide Web and Conceptual Modeling (WWWCM)*, number 1727 in LNCS, pages 307–320, 1999.

[MK98] Wolfgang May and Paul-Th. Kandzia. Nonmonotonic inheritance in object-oriented deductive database languages. Technical Report 114, Universität Freiburg, Institut für Informatik, 1998. Available from `http://www.informatik.uni-freiburg.de/~dbis/Publications/98/Inheritance.html`.

[MLL97] Wolfgang May, Bertram Ludäscher, and Georg Lausen. Well-founded semantics for deductive object-oriented database languages. In *Intl. Conference on Deductive and Object-Oriented Databases (DOOD)*, number 1341 in LNCS, pages 320–336. Springer, 1997.

[MM00] W. May and J. P. Marron. FLORID: User Manual, 2000. Available from `http://www.informatik.uni-freiburg.de/~dbis/florid`.

[MSL97] Wolfgang May, Christian Schlepphorst, and Georg Lausen. Integrating dynamic aspects into deductive object-oriented databases. In *3rd Intl. Workshop on Rules in Database Systems (RIDS)*, number 1312 in LNCS, 1997.

[NT89] S. Naqvi and S. Tsur. *A logical Language for Data and Knowledge Bases*. Computer Science Press, New York, 1989.

[Prz88] Teodor C. Przymusinski. On the declarative semantics of deductive databases and logic programs. In Jack Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 191–216. Morgan Kaufmann, 1988.

[Ull89] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems*, volume 2. Computer Science Press, New York, 1989.

[VGRS91] A. Van Gelder, K. Ross, and J.S. Schlipf. The well-founded semantics for general logic programs. *Journal of the ACM*, 38(3):620 – 650, 1991.