A short version of the paper was presented at
5th Intl. Conf. on Deductive and Object-Oriented Databases
(DOOD'97), December 8–12, 1997, Montreux, Switzerland.

Albert-Ludwigs-Universität Freiburg, Institut für Informatik,

Technical Report 82 (revised 1997)

Towards Efficient Evaluation of Methods by Reduction

Paul-Th. Kandzia
Institut für Informatik, Albert-Ludwigs-Universität,
D-79110 Freiburg, Germany,
kandzia@informatik.uni-freiburg.de

Abstract

Bottom-up evaluation of DOOD programs is improved by adding functional reduction as call–by–need computation for scalar methods. To that aim methods, conceived as a functional language, have to satisfy certain properties, like uniqueness and directed use from call to result. In this paper we propose declarative and operational semantics of such methods, formalize the required properties, and show the integration of reduction and bottom-up evaluation. In an experiment this approach shows a better performance than a top-down technique and magic set optimization.

1 Introduction

During the last decade much research has been done to develop logics for object–oriented databases. Typically such approaches comprise object–identity, methods, classes, and inheritance. Moreover, in advanced languages like F-logic [4, 8], schema and data are handled at a uniform level, enabling among others, the data–dependent definition of classes. If such a logic is used as a deductive rule language, due to the rich semantics, its efficient evaluation becomes a major challenge. In this paper for two reasons a Datalog–like bottom–up technique is considered:

- The advantages of bottom-up reasoning for deductive databases are well known: The set–at–a–time access to the extensional database allows for the application of relational optimization techniques, besides there is no influence of the ordering of rules in the program or subgoals within a rule body upon the termination of a program. Furthermore, since the whole minimal model is computed, materialization of the intensional database is simple.
- To find an efficient top–down evaluation for general DOOD–languages seems to be difficult. First, the search space tends to be very huge compared with the SLD–trees of Prolog resolution. As an example, consider the following F-logic query, searching for all scalar methods with result object `r` defined on object `o`:[1]

[1]As usual capitals denote variables.

?– o[M→r].

To answer the query it is not sufficient to resolve – like Prolog – with all rule heads which unify with the query, in particular those of the form o[m→r] for any constant m. Since a class C1 of o may exist which defines an appropriate inheritable method, we also have to consider the following query (with $\bullet\!\!\rightarrow$ denoting an inheritable method):

?– o:C1, C1[M$\bullet\!\!\rightarrow$r].

Note that in F-logic class information like o:C1 is defined by rules, too. Moreover, for complex object expressions only complete sets of most general unifiers rather than a unique mgu and therefore no polynomial algorithm may exist [14]. The example assumes that no overriding or inheritance conflicts occur. However, some advanced semantics for multiple inheritance [8] are exclusively based on bottom–up evaluation, because all possibly inheritable results for a method have to be known in advance to choose the right one, which gives a second objection to resolution.

For a bottom–up evaluation optimization is essential. Unfortunately, an adaptation of the *syntactical* methods well–understood in Datalog (e.g. magic–set or seminaive evaluation) runs into similar obstacles as resolution: unification is not sufficient to decide dependencies between queries and rule heads.
Hence attention should be payed to specialized optimizations which take advantage from the *semantical* knowledge about certain parts of a program. In this paper we focus on scalar methods. As a simple example consider railway connections: There are objects of class station which have a method next giving the next station depending on one parameter, the line. A method dist is defined on lines which, for any two stations, gives the number of stations between them. With F-logic rules, the definition is as follows:

$$\begin{array}{ll} \mathsf{L[dist@(X,X)\rightarrow 0]} & \mathsf{:-\ L:line,\ X:station.} \\ \mathsf{L[dist@(X,Y)\rightarrow D]} & \mathsf{:-\ X[next@(L)\rightarrow Z],\ D = L.dist@(Z,Y) + 1.} \end{array} \tag{1.1}$$

A pure bottom–up evaluation will deduce all distances between arbitrary stations and store them in the computed model, even if the user queries only one certain distance. If magic–set optimization is used, due to the recursive definition, still the distances of stations "along the way" are deduced, claiming space and iterations of the consequence operator.
To overcome this problem we propose to complement the general bottom–up fixpoint evaluation by functional reduction as a kind of subordinate computation taking place whenever a call to a so called *computed method* occurs. Hence, to evaluate a computed method neither the storage of results nor iterations of the consequence operator are needed. The idea is similar to the aim of functional–logic programming [6] (including the language *Life* [3], which additionaly supports attributes, classes and inheritance), but there are crucial differences:

- *Declarative semantics:* In functional–logic languages functions are interpreted by an equality relation over the Herbrand–universe. To model the function *mother* we could state:

  mother(mary) = jane.

In an Herbrand–model the terms mother(mary) and jane have to be mapped to the same element of the universe, consequently a factorization of the universe in congruence classes according to the equations is neccesary. In contrast, the method definition

mary[mother→jane].

implies no equality over objects of the universe (here mother, mary and jane). Rather, we can describe the semantics of a method definition as a set of additional facts extending a given Herbrand–interpretation.

- *Operational semantics:* Functional–logic programming is based on the top–down principle: SLD–resolution is extended by functional reduction, called *narrowing* [6]. But as aforementioned our interest is on bottom-up algorithms.

The main contributions of our work can be summarized as follows: Computed methods of a DOOD can be considered as algorithms executed when needed, as suggested in [1, 11]. Their evaluation is based on fast functional reduction, improving efficiency but fully preserving the declarative minimal model semantics of a program. Moreover, termination of some logical programs is achieved which typically are *unsafe* otherwise.
An earlier approach discusses reduction rules in Datalog [5]. However, the declarative semantics is as in other functional-logic languages (see above), there is no object orientation, and functional reduction is not used as a "by need" computation, which is supported by our approach.

2 F-logic

Because of its generality we use F-logic as a formal framework. In favor of a concise presentation the definition of only a small kernel (classes and scalar methods) is summed up, since the rest is well documented [4, 8] and our ideas can be extended to the whole language straightforwardly.

2.1 Syntax

Objects are (named) entities of the modelled world which are arranged in *classes* and have a structure formed by *methods*. To handle objects a generalisation of the widespread concept of path expressions, called *references* [4], is used. References allow for two kinds of object access: navigation along methods and association by properties. We distinguish the following kinds of references:
ID-terms serve as names of objects, classes and methods. They can be constants (i.e. 0-ary constructors), such as john and 24, or composed of constructors, such as cons(1,nil). Furthermore, *variables*, as usual denoted by capitals, are ID-terms.
Constructor–references like $f(r_1, \ldots, r_n)$ combine n references $r_1, \ldots, r_n$, $n \geq 0$, using an n-ary constructor f.[2]
Path–references of the form $(h.m@(p_1, \ldots, p_n))$ (where $h, m, p_1, \ldots, p_n$, $n \geq 0$, are references) denote the result of applying a (scalar) method m to an object h with parameters $p_1, \ldots, p_n$. If there are no parameters ($n = 0$) the part @() is omitted. The term X.salary yields the result

[2] Id-terms are constructor references, too.

of applying the method salary to the object bound to the variable X, mary.salary@(1994) gives the salary of Mary in year 1994.
Properties of objects can be specified in the following two ways: *Data–references* of the form $(o[m@(p_1,\dots,p_n)\to r])$ for $n \geq 0$, where o, m, r and the p_i are references, define methods. r is the result of applying the method m to the host object o with parameters $p_1,\dots,p_n$, e.g. mary[age→32]. As above an empty parameter list is omitted.
Isa–references have the form $(a : b)$ where a and b are references and express that a is an instance of class b, e.g. mary:person.
For convenience, references are read from left to right, hence many parantheses are not necessary. In a data– or isa–reference the expression $[m@(p_1,\dots,p_n)\to r]$ or $: b$, respectively, is called a *filter*; references of which another reference is composed are called *subreferences*.
References get their distinctive flexibility not only from the possibility of variables on method or class position but especially since they can be nested arbitrarily. Consider the following example:

X : bike.wheel[supplier→S].axis : quicklock[supplier→S].

Finding the meaning of such a reference can be understood as moving in two dimensions. The path operator $r_1.r_2$ leads into "depth". Starting from the object described by r_1 the method name obtained from r_2 is applied yielding a new object as result. Between such method applications properties of the involved objects are specified by filters, which opens the "breadth" dimension. In the example the depth–dimension is the reference X.wheel.axis, describing the *object value* of the reference, provided that X:bike, X.wheel[supplier→S], X.wheel.axis:quicklock and X.wheel.axis[supplier→S] are satisfied. Hence, the reference describes Bikes X having a wheel with a quicklock axis where the supplier S of the axis is also the supplier of the wheel. Both dimensions of a reference, an object valued depth and a truth valued breadth are reflected by the semantics (see below).
We distinguish a kind of basic references, the *atoms*.

Definition 1 *Atoms* are references of the form $o[m@(t_1,\dots,t_n)\to r]$ or $o : c$ where $o, m, t_1,\dots,t_n$, r and c are ID-terms. □

Next we introduce logical rules and queries which are essentially the same as in Datalog, using references instead of predicates. Once more to keep the framework simple, we do not consider negation.

Definition 2 Let h be an atom[3], $l_1,\dots,l_n$, $n \geq 1$, arbitrary references.
A *(logical) rule* has the form

$$h :- l_1,\dots,l_n. \quad \text{or} \quad h.$$

The second form is called a *fact*. A *logic program* is a set of rules. A *query* has the form

$$?\!-\ l_1,\dots,l_n.$$

□

[3]Taking into bargain more involved definitions we could as well allow complex references here.

2.2 Semantics

For our purpose we will not consider arbitrary interpretations but only Herbrand–structures (H-structures), the standard meaning in logic programming.
In F-logic, an H-structure is a set of ground atoms describing an object world. Contrasting Prolog and Datalog, this set has to satisfy several *closure axioms* related to general object–oriented properties. However, in the restricted version of this paper only the uniqueness property of methods has to hold.

Definition 3 Let H be a (possibly infinite) set of ground atoms. H is an *H-structure*, if there are no ID-terms $u_0, \ldots, u_n, u_m, u_r, u'_r$ such that

$$u_0[u_m@(u_1, \ldots, u_n){\rightarrow}u_r] \in H, \quad u_0[u_m@(u_1, \ldots, u_n){\rightarrow}u'_r] \in H \quad \text{and} \quad u_r \neq u'_r$$

□

As a consequence of the closure axioms certain programs do not have a Herbrand-model, e.g. the following two facts:

mary[age→31].
mary[age→32].

Such programs are called *inconsistent.*
To evaluate arbitrary references in an H-structure first the variables are mapped to ground ID-terms; then this mapping is inductively extended to references. As sketched above, for a variable valuation (i.e. ground substitution) the semantics of a reference is a set of one object if all filters are satisfied in the H-structure, otherwise the empty set[4]. The following definition is called *direct valuation* as opposed to the extended valuation of the next section.

Definition 4 Let H be a set of ground atoms, β a ground substitution, $t_0, \ldots, t_n$, $n \geq 0$, m, and r references.
For a reference t the *direct valuation* $\beta_H(t)$ is defined inductively as follows:

1. If $t = X$ is a variable then
$$\beta_H(t) = \{\beta(X)\}.$$

2. If $t = f(t_1, \ldots, t_n)$ is a constructor–reference with n-ary constructor f, $n \geq 0$, then
$$\beta_H(t) = \{f(u_1, \ldots, u_n) \mid u_i \in \beta_H(t_i) \text{ for all } i = 1, \ldots, n\}.$$

3. If $t = t_0.m@(t_1, \ldots, t_n)$ is a path–reference then
$$\beta_H(t) = \{u \mid u_m \in \beta_H(m), u_i \in \beta_H(t_i) \text{ for all } i = 0, \ldots, n \text{ and } u_0[u_m@(u_0, \ldots, u_n){\rightarrow}u] \in H\}.$$

4. If $t = t_0 : t_1$ is an isa–reference then
$$\beta_H(t) = \{u_0 \mid u_0 \in \beta_H(t_0),\ u_1 \in \beta_H(t_1) \text{ and } u_0 : u_1 \in H\}.$$

[4] In full F-logic, due to multivalued methods, sets with more than one object are possible as meaning of a reference.

5. If $t = t_0[m@(t_1, \ldots, t_n) \to r]$, $n \geq 0$, is a data–reference then

$$\beta_H(t) = \{u_0 \mid u_m \in \beta_H(m), u_r \in \beta_H(r), u_i \in \beta_H(t_i) \text{ for all } i = 0, \ldots, n \text{ and } u_0[u_m@(u_0, \ldots, u_n) \to u_r] \in H\}.$$

□

For usage in rules and queries, references have a second, truth valued meaning. A reference is satisfied if an appropriate object is found, i.e. its direct valuation is not empty.

Definition 5 Let r be a reference, H be an H-structure, and β a ground substitution. Then H *satisfies* r, written $H \models_\beta r$, if $\beta_H(r) \neq \emptyset$. □

Formulas are interpreted as usual, in particular the minimal model of a consistent logic program P, denoted M_P, is defined, which coincides with the least fixpoint of the immediate consequence operator associated with the program.
We stress the fact that this framework can be extended to the full F-logic comprising signatures, multi–valued methods, subclass–hierarchies, inheritance, predicates and negation. However, these details have no influence to the principles developed in the next section.

3 Definition of computed methods

In this section *computed methods* are introduced which are evaluated by functional reduction. Hence, to specify computed methods, the following issues are essential:

1. A definition shows a direction from call to result, i.e. has the form of a *reduction rule.*

2. Definitions depend on object properties, in particular there are conditions answered with respect to the logical part of a program.

3. Method definitions must not have "side effects" like the following example:

   $$\mathsf{X[apply@(F)}\to\mathsf{X.F]}. \tag{3.1}$$

   The fact defines the method apply which applies a method F to an object X. However, since the variables are universally quantified the subexpression X.F states that *all Id-terms* are defined as a method on *all objects*, what obviously the programmer did not have in mind. A correct definition could look like the following rule:

   X[apply@(F)→Y] :– X[F→Y].

4. The results of computed methods have to be unique. For general F-logic programs the uniqueness of methods is not decidable [2], but once a model is computed it can be checked for inconsistencies. However, since computed methods are not materialized we need sufficient syntactical restrictions to ensure their uniqueness. For example the following definition is ambigous:

   M[selfRef→O] :– O[M→O].

The method selfRef yields all objects on which a method M is the identity. Clearly, in general selfRef can give multiple results for a given method. Such cases will be excluded by stating a variable condition and a restriction concerning a set of method definitions.

5. To avoid difficulties of the operational semantics the conditions in definitions of computed methods must not itself use computed methods. Consider the following rule:

   o[m→1] :– o[m→1].

   If the call o.m has to be computed in a functional manner using the above definition, the condition, i.e. the rule body, has to be evaluated first. However, to that aim the result of method m on object o must be known, which closes a cycle!

The following definitions respect the above demands.

Definition 6 Let m be a ground ID-term, $h, p_1, \ldots, p_n$, $n \geq 0$ ID-terms, $c_1, \ldots, c_k$, $k \geq 0$, literals without variables on method position, and rhs a reference without a filter.

- A variable X satisfies the *variable condition* if one of the following conditions holds:
  - X occurs in one of the expressions $h, p_1, \ldots, p_n$.
  - Let $t = t_0[m'@(t_1, \ldots, t_l) \rightarrow r]$ be a data reference within one of the $c_j, j \in 1, \ldots, k$, such that r is an ID-term in which X occurs. Further, the variables in $t_0, \ldots, t_l$ and m' satisfy the variable condition. Then X satisfies the variable condition, too.
- The formula

  $$h.m@(p_1, \ldots, p_n) \stackrel{\cdot}{\longrightarrow} rhs | c_1, \ldots, c_k$$

  is called *M-reduction rule* (for **M**ethod–reduction) if all variables in rhs satisfy the variable condition. We say, the formula *defines* m which is called a *computed method*; lhs (rhs) is called the *left (right) hand side* of the M-reduction rule. The *condition part* $|c_1, \ldots, c_k$ can be omitted denoting an unconditional rule.

  The declarative semantics of an M-reduction rule is given by the formula[5]:

  $$h[m@(p_1, \ldots, p_n) \rightarrow rhs] :- c_1, \ldots, c_k, rhs \qquad ((3.2))$$

□

Note that the second part of the variable condition makes use of the uniqueness property of methods, a feature not found in the variable condition of functional-logic languages, e.g. [12]. Moreover, the occurence of rhs in the premise of formula (3.2) excludes the "side effect" problem of (3.1), in particular, object creation in the sense of [4] is not possible by M-reduction rules, because all subreferences occuring in the head of formula (3.2) also occur in the body. Next we discuss method definitions comprising more than one M-reduction rule. Here the last two goals, uniqueness and the condition parts, have to be considered.

[5] Since rhs may contain a path operator the formula is not a logical rule according to Definition 2.

Definition 7 Let R be a set of M-reduction rules. R is *unambigous*, if for every two rules r and r'

$$
\begin{aligned}
r &= lhs \dot{\longrightarrow} rhs \mid cond \\
r' &= lhs' \dot{\longrightarrow} rhs' \mid cond'
\end{aligned}
$$

defining the same method one of the following conditions hold:

- The left hand sides lhs and lhs' do not unify.
- If σ is a most general unifier of lhs and lhs' then the conjunction of the condition instances $cond\ \sigma \wedge cond'\ \sigma$ is not satisfiable (in the modelled application).

$\square$

Definition 8 A set R of M-reduction rules is an M-ruleset if the following conditions hold:

- The condition parts of the M-reduction rules in R contain no occurence of a computed method of R.
- In every left hand side $h.m@(p_1, \ldots, p_n)$ of a M-reduction rule in R, m is the only occurence of a computed method of R.
- R is unambigous.

$\square$

The next example illustrates the definitions.

Example 1 A manufacturer maintains a product database (class product). Every product has a method parts yielding a list[6] of its subparts and a method cost which gives the assembly costs. A typical product description could look like

wheel[parts→<spoke,axis>; cost→50].

The total cost of a product is the sum of the total costs of its subparts plus the assembly cost. Note, that using a multivalued method parts instead of lists would require non–trivial aggregations. The following M-ruleset defines an appropriate method totalcost:[7]

P.totalcost $\dot{\longrightarrow}$ P.parts.map@(totalcost).sum + P.cost | P:product

nil.map@(F) $\dot{\longrightarrow}$ nil
cons(X,L).map@(F) $\dot{\longrightarrow}$ cons(X.F,L.map@(F))

nil.sum $\dot{\longrightarrow}$ 0
cons(X,L).sum $\dot{\longrightarrow}$ X + L.sum

For the first M-reduction rule we show the declarative semantics (see (3.2)), too:

P[totalcost→P.parts.map@(totalcost).sum + P.cost] :– P.parts.map@(totalcost).sum + P.cost, P:product.

[6]Using the usual constructors cons and nil and the abbreviation <...>, respectively.
[7]We assume + as a built-in method in infix-notation.

The last formula is equivalent to the rule:

$\mathsf{P[totalcost{\rightarrow}T]} :- \mathsf{P.parts.map@(totalcost).sum} + \mathsf{P.cost} = \mathsf{T}, \mathsf{P{:}product}.$

□

The next step is to define a minimal model semantics for M-rulesets. Of course we always have to consider the method definitions together with a known object base (more exactly an H-structure). The M-ruleset defines a kind of closure over those facts, comprising exactly the data–references related to computed methods.

Example 2 Consider the very simple M-ruleset

$\mathsf{X.allowance} \stackrel{\cdot}{\longrightarrow} 500 \mid \mathsf{X{:}parent}$

and the following facts:

mary:parent.
donald:parent.

The closure adds the following facts:

mary[allowance→500].
donald[allowance→500].

□

We now give a formal definition of the R-closure:

Definition 9 Let R be an M-ruleset with an *appropriate* H-structure H, that is, all methods which are computed in R are undefined in H. Then the *R-closure* of H, H_R, is defined as the minimal model of R which is a superset of H. □

The next theorem shows that M-rulesets and their semantics are well defined.

Theorem 1 *For any M-ruleset R and appropriate H-structure H there exists a unique R-closure H_R.*

Proof: An M-reduction rule is equivalent to a logical rule: Since all path expressions in the right hand side (the head in the logical form) also appear in the body (see formula (3.2)) they can be substituted by new variables bound in the body, which results in a rule according to Definition 2. Essentially, this is tantamount to the proposition that method definitions are not "creating" objects by path references. Further, due to the uniqueness properties, M-rulesets can never imply inconsistencies; H is consistent and the methods defined in H are disjoint from the ones computed in R, since H is appropriate. Consequently, H_R is consistent. The rest is analogous to equivalent results for first order horn–clause logic [10]. □

4 Program Evaluation

We have seen how computed methods can be defined such that their consistency is ensured. Now an operational semantics is introduced which computes by functional reduction exactly those results which are valid in the R-closure. In a second step the reduction will be integrated in the valuation of references as well as in the bottom-up computation of programs.

4.1 M-Reduction

In principle, M-reduction is the substitution of a call, either by the right hand side of a fitting M-reduction rule if the method is computed, or by the result given in the underlying H-structure if it is not computed. An M-reduction rule fits a call of a computed method if the left hand side of a rule instance agrees with the call and its condition part is valid in H. The rewriting step is iterated until either no method call occurs anymore, that is, an ID-term is found as a result, or no fitting rule exists, in which case the call is undefined in the respective R-closure.

Definition 10 Let H be an H-structure, R an M-ruleset, and t a ground reference without a filter. Three cases are distinguished:

1. $t = h.m@(p_1, \ldots, p_n)$ where m is a computed method (i.e. defined in R) and $h, p_1, \ldots, p_n$ are ID-terms. Moreover, there is an M-reduction rule

   $$ru = h'.m@(p'_1, \ldots, p'_n) \dot{\longrightarrow} rhs | c_1, \ldots, c_k$$

   in R and a ground substitution σ such that the following conditions hold:

   (a) $h'\sigma = h,\ p'_1\sigma = p_1,\ \ldots,\ p'_n\sigma = p_n$ and
   (b) $H \models c_i\sigma \quad$ for all $i = 1, \ldots, k$.

   Then we define $t \stackrel{\cdot}{\Longrightarrow}_{R,H}\ rhs\ \sigma$ and say: *ru fits t*.

2. $t = h.m@(p_1, \ldots, p_n)$, where $h, m, p_1, \ldots, p_n$ are ID-terms, m is not a computed method and there is a ground ID-term r such that

   $$h[m@(p_1, \ldots, p_n) \rightarrow r] \in H$$

   Then we define $t \stackrel{\cdot}{\Longrightarrow}_{R,H}\ r$

3. t_s is the leftmost subreference of t such that $t_s \stackrel{\cdot}{\Longrightarrow} t'_s$ according to 1. or 2. Then we define $t \Longrightarrow t'$, where t' is obtained by substituting t'_s for t_s in t.

Furthermore, for a reference t we define the *normalform* $t\!\downarrow$ obtained by applying the relation $\Longrightarrow_{R,H}$ to t as long as possible. The normalform is unique due to the uniqueness properties of R. □

Note that for M-reduction only classical unification between ID-terms is used which is accomplished in polynomial time.

4.2 The extended valuation

The valuation introduced in Definition 4 evaluates all references directly in an H-structure H. This function can be extended by an M-ruleset R, used to compute the normalform of method calls. If the normalform is an id-term it represents the result, otherwise the call is undefined. This extended valuation in H with respect to R is equivalent to the direct valuation in H_R. Note the crucial difference: to use the extended evaluation only the facts of H have to be known explicitly whereas the R-closure remains virtual in form of R; on the other hand the direct valuation needs the entire set H_R. Hence, used for an interpreter, the extended valuation can handle infinite method definitions whereas the direct valuation is restricted to finite H-structures.
The extended valuation differentiates from the direct one only in cases 3. and 5. which regard to methods.

Definition 11

3. If $t = t_0.m@(t_1, \ldots, t_n)$, $n \geq 0$, is a path reference then

$$\begin{aligned}
\beta_{H,R}(t) \quad = \{u \mid \quad & u_m \in \beta_{H,R}(m), u_i \in \beta_{H,R}(t_i) \text{ for all } i = 0, \ldots, n, \\
& u_m \text{ is computed in } R, u_0.u_m@(u_1, \ldots, u_n){\downarrow}= u \\
& \text{and } u \text{ is an id-term } \} \\
\cup \{u \mid \quad & u_m \in \beta_{H,R}(m), u_i \in \beta_{H,R}(t_i) \text{ for all } i = 0, \ldots, n, \\
& u_m \text{ is not computed in } R \text{ and} \\
& u_0[u_m@(u_1, \ldots, u_n) \rightarrow u] \in H \ \}
\end{aligned}$$

5. If $t = t_0[m@(t_1, \ldots, t_n) \rightarrow r]$ is a data reference then

$$\begin{aligned}
\beta_{H,R}(t) \quad = \{u_0 \mid \quad & u_m \in \beta_{H,R}(m), u \in \beta_{H,R}(r), u_i \in \beta_{H,R}(t_i) \text{ for all } i = 0, \ldots, n, \\
& u_m \text{ is computed in } R, u_0.u_m@(u_1, \ldots, u_n){\downarrow}= u \\
& \text{and } u \text{ is an id-term } \} \\
\cup \{u_0 \mid \quad & u_m \in \beta_{H,R}(m), u \in \beta_{H,R}(r), u_i \in \beta_{H,R}(t_i) \text{ for all } i = 0, \ldots, n, \\
& u_m \text{ ist not computed in } R \text{ and} \\
& u_0[u_m@(u_1, \ldots, u_n) \rightarrow u] \in H \ \}
\end{aligned}$$

The rest is as in Definition 4. □

With the extended valuation, logical entailment is defined accordingly:

Definition 12 Let H be an H-structure, R an M-ruleset, β a variable valuation and t a reference. H and R satisfy t, written $(H, R) \models_\beta \phi$, if $\beta_{H,R}(\phi) \neq \emptyset$. □

An example shows how the extended valuation works:

Example 3 The reference bike.totalcost is evaluated with the M-ruleset of Example 1 and the following H-structure:

```
bike:product[parts→<wheel,saddle,brake> ;cost→200].
wheel:product[parts→<spoke,axis> ;cost→50].
spoke:product[cost→10;parts→nil].
axis:product[cost→20;parts→nil].
saddle:product[cost→50;parts→nil].
brake:product[cost→80;parts→nil].
```

The first steps of the reduction are as follows (the reduced subreferences are underlined):

bike.$\underline{\text{totalcost}}$
$\Downarrow$ rule 1, $\sigma = \{\text{P/bike})\}$
bike.$\underline{\text{parts}}$.map@(totalcost).sum + bike.cost
$\Downarrow$ not computed
$<$ wheel, saddle, brake $>$ $\underline{\text{.map}}$@(totalcost).sum + bike.cost
$\Downarrow$ rule 3, $\sigma = \{\text{X/wheel, } \overline{\text{L}}\text{/ <saddle,brake>}\}$
cons(wheel.$\underline{\text{totalcost,}}$ <saddle,brake>.map@(totalcost).sum + bike.cost
...

After several more steps an integer is obtained which, being an id-term, is the result of the method call. □

The extended valuation is sound and complete:

Theorem 2 *For any M-ruleset R, appropriate H-structure H, ground substitution β, and reference t the following holds:*

$$\begin{array}{rcl} \beta_{H_R}(t) & = & \beta_{H,R}(t) \\ H_R \models_\beta t & \textit{iff} & (H,R) \models_\beta t \end{array}$$

Proof: The latter proposition is a consequence of the first. To prove the first, the main step is to show that for ground id-terms $u_0, \ldots, u_n, u_m$, where u_m is a computed method, $\beta_{H_R}(u_0.u_m@(u_1, \ldots, u_n)) = \{u\}$ iff $u_0.u_m@(u_1, \ldots, u_n){\downarrow_{R,H}} = u$ is an id-term; i.e. that the results of computed methods in the R-closure are exactly the normalforms which are id-terms (implying that calls not reducible to an id-term are undefined in the R-closure). The if–direction is shown using the fixpoint characterization of H_R (see proof of Theorem 1) by induction over the iterations of the consequence operator associated with R. Here again it is a crucial condition that the right hand side of an M-reduction rule appears in the body of its logical definition (see formula (3.2)). For the only-if–direction an induction over the length of the reduction takes place. Both inductions are lengthy but straightforward. □

4.3 Bottom–up evaluation with computed methods

The extended valuation can be used within a Datalog–like immediate consequence operator for logical rules, which finds answer substitutions for rule bodies not in a partial model alone, but rather in its R-closure given by the computed methods. If the iteration of the operator reaches a fixpoint containing no inconsistency, the R-closure of the fixpoint is the common

minimal model of the logical rules and the M-ruleset (which exists since M-reduction rules are equivalent to logical rules).
In the following, programs of the form $P = L \cup R$ are considered, where L is a set of logical rules and R an M-reduction ruleset; the computed methods defined in R are not allowed in the rule heads of L. This condition ensures that all partial models of the program are appropriate for R (see Definition 9).

Example 4 Let R be the M-reduction rules for totalcost. Besides facts like those shown in Example 3, L may comprise the following rules which use the computed method totalcost in a subgoal and, vice versa, define object properties (the method parts) used in a condition of R.

P:smallpart :– P:product.totalcost < 10.
P[parts→nil] :– P:basepart.

□

The (extended) immediate consequence operator is defined as follows:

Definition 13 Let H be a set of ground atoms and $P = L \cup R$ as above. We define the *extended immediate consequence operator* $T_{L/R}$:

- $T_{L/R}(H) = \{h \mid$ h $\leftarrow b_1, \ldots, b_n$ is a ground instance of a rule in L such that $(H, R) \models b_i$ for all $i = 1, \ldots, n\}$
- $T_{L/R} \uparrow 0 = \emptyset$
  $T_{L/R} \uparrow n = T_{L/R}(T_{L/R} \uparrow n - 1)$
  $T_{L/R} \uparrow \omega = \bigcup_{i=0}^{\infty} T_{L/R} \uparrow i$

□

Answers of queries to P are computed as follows: first $T_{L/R}$ is iterated until a fixpoint is reached. If this fixpoint is inconsistent P has no model. Otherwise, the meaning of a reference can be obtained by applying the extended valuation in the fixpoint with respect to R.

Theorem 3 *Let P be a program as above, $FP = T_{L/R} \uparrow \omega$.*

- *If FP is not inconsistent then the following holds for any reference t and formula ϕ:*

$$\begin{array}{rcl} \beta_{FP,\,R}(t) & = & \beta_{M_P}(t) \\ (FP, R) \models \phi & \textit{iff} & M_P \models \phi \end{array}$$

- *If FP is inconsistent then P has no model.*

Proof: Again the proof is tedious but straightforward, considering that the R-closure of a partial model can not introduce any inconsistencies (Theorem 1), that is, an inconsitency of a program is explicitly found in FP. The rest follows mainly from the equivalence of extended valuation in a partial model and direct evaluation in its R-closure (Theorem 2). □

Regarding to computability of the operator $T_{L/R}$ three kinds of problems may arise:

- Variables occuring in the head of a rule but not in the body give rise to an infinite set of consequences.
- A non–terminating computed method leads to a non–terminating operator $T_{L/R}$.
- Calls of computed methods must be instantiated prior to their reduction since the reduction relation is only defined for ground terms. Moreover, a non–ground call can lead to an infinite set of answer substitutions, as in the following query (with sum defined as in Example 1:

  ?– L[sum→5].

  Obviously there is an infinite number of lists having the element sum of 5.

The first problem is exactly as in Datalog, the last similar to the problem of built–ins. Both are avoided by a *safeness condition.* The second problem is undecidable and we leave it to the programmer's responsibility. We restrict the definition of safeness to rules consisting of atoms only, the extension to the general case is straightforward.

Definition 14 Consider a conjunction of atoms. A variable X is *limited* if one of the following conditions holds:

1. X occurs in an isa-reference $o : c$.
2. X occurs in a data reference $t_0[m@(t_1, \ldots, t_n)\rightarrow r]$ such that m is ground and not computed.
3. X occurs in a data–reference $t_0[m@(t_1, \ldots, t_n)\rightarrow r]$ in the result r where all variables in $t_0, m, t_1, \ldots, t_n$ are limited.

□

Definition 15

- A query or rule body (consisting of atoms only) is *safe* if all its variables are limited.
- A logical rule (consisting of atoms only) is *safe* if
  - all variables in the body are limited and
  - all variables in the head also occur in the body.
- A program is *safe* if all of its logical rules are safe.

□

Note that no safeness condition for M-rulesets is needed.

4.4 Inheritance

As already mentioned, M-reduction can be extended to full F-logic. An especially important feature is inheritance. Generally, in F-logic classes inherit *results* of methods, not *algorithms* (the latter can be simulated, though) [8]. The results of methods are computed for classes, and, once they are found, inherited to the instances. Consequently, the evaluation of methods is completely unrelated to inheritance, which takes place afterwards. Hence, inheritable methods can be defined by M-rulesets the same way as non-inheritable ones and provided with an arbitrary semantics for inheritance, e.g. the monotonic or the trigger semantics.

4.5 Experimental Results

To confirm the theoretical work different implementations of typical algorithms are compared:

- an F-logic program with *computed methods* running on an XSB–prototype using a straightforward (naive) implementation of the presented evaluation,
- a Datalog program running on Coral [13] using two stategies: the system–generated *magic set optimization* and a top-down technique called *Pipelining*,
- an F-logic program without computed methods on the system FLORID [7] working seminaive bottom-up.

To enable a comparison the examples do not rely on the object oriented constructs and therefore take no advantage of the specific power of the language. Moreover, note that the three systems are quite different: FLORID and Coral provide full fledged C++–implementations whereas only a "rapid prototype" was at hand for computed methods.

Distance in a Graph A variant of this problem is used as a benchmark for Coral. The EDB consists of facts describing direct connections between nodes. The method (predicate) dist gives the distance of arbitrary connected nodes.

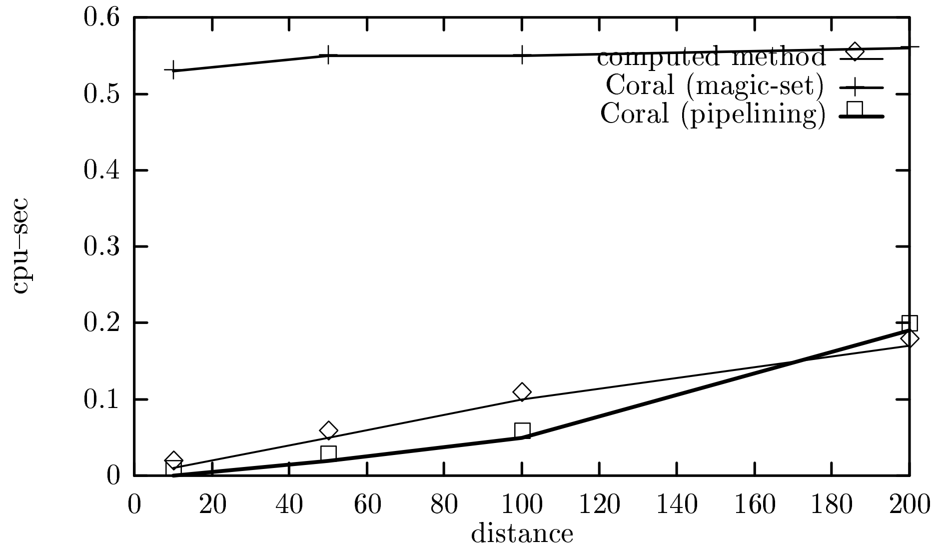
```
/* version with computed methods */
0[link→1].
1[link→2].
...

X.dist@(X)  -·-> 0
X.dist@(Y)  -·-> Z.dist@(Y) + 1 | X[link→Z]

/* Datalog version for Coral */
link(0,1).
link(1,2).
...

dist(X,X,0).
dist(X,Y,D) :– link(X,Z), dist(Z,Y,D1), D = D1 + 1.
```

```
/* version with F-logic rules for Florid*/
X[dist@(X)→0] :– X[link→Y].
X[dist@(Y)→D1] :– X[link→Z], Z[dist@(Y)→D], D1 = D + 1.
```

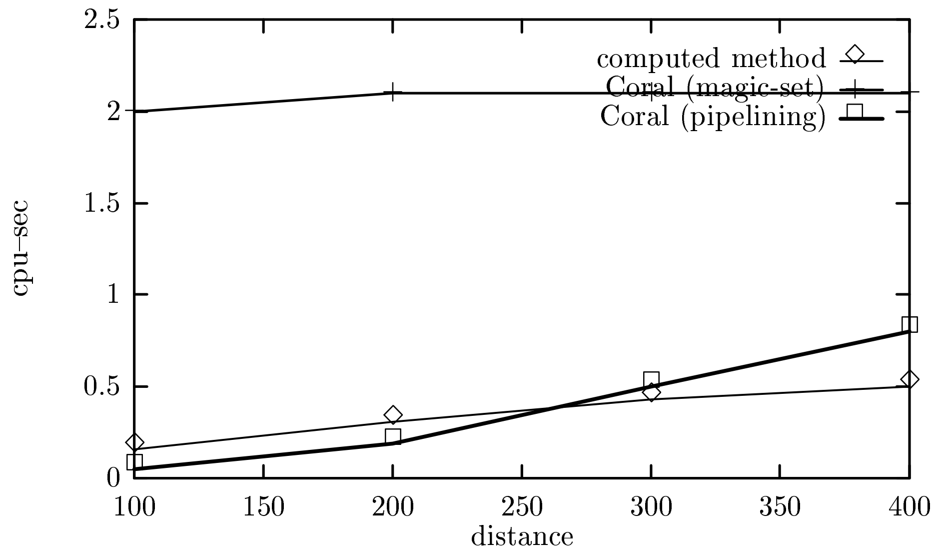
Time (in Cpu-seconds) was measured dependent on the size of the EDB and the queried distance.

facts	distance	comp. method	Coral		Florid
			magic-set	pipelining	
50	10	0.01	0.05	0.005	1.8
	50	0.03	0.06	0.02	1.8
100	10	0.01	0.17	0.005	6.3
	50	0.04	0.17	0.02	6.3
	100	0.07	0.17	0.05	6.3
200	10	0.01	0.53	0	30
	50	0.05	0.55	0.02	30
	100	0.1	0.55	0.05	30
	200	0.17	0.56	0.19	30
400	100	0.16	2.0	0.05	180
	200	0.31	2.1	0.19	180
	300	0.43	2.1	0.5	180
	400	0.5	2.1	0.8	180

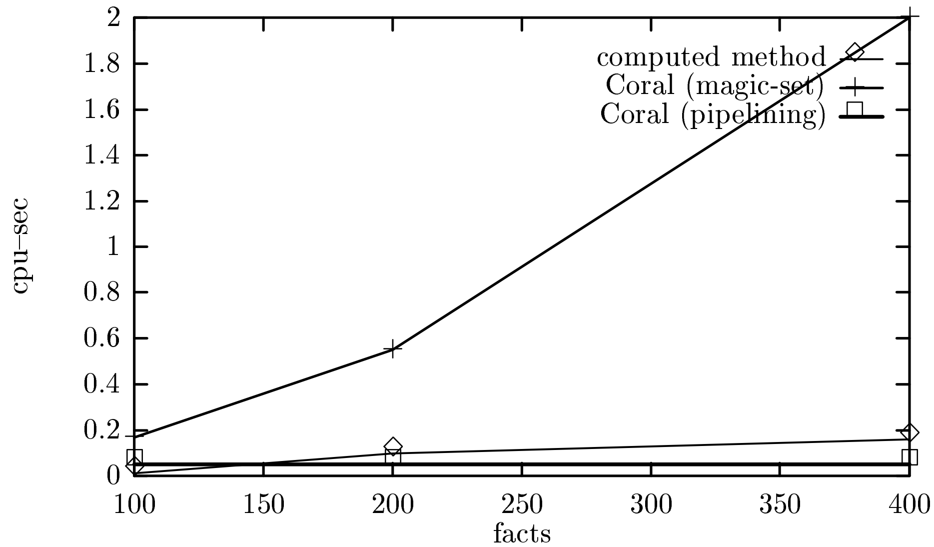
Plot for 200 facts:



Plot for 400 facts:



Plot for distance 100:



For larger distances (compared to the number of facts) the computed method is the most efficient implementation, for increasing EDB the break-even point with pipelining occurs earlier. In all cases computed methods perform better than magic set!

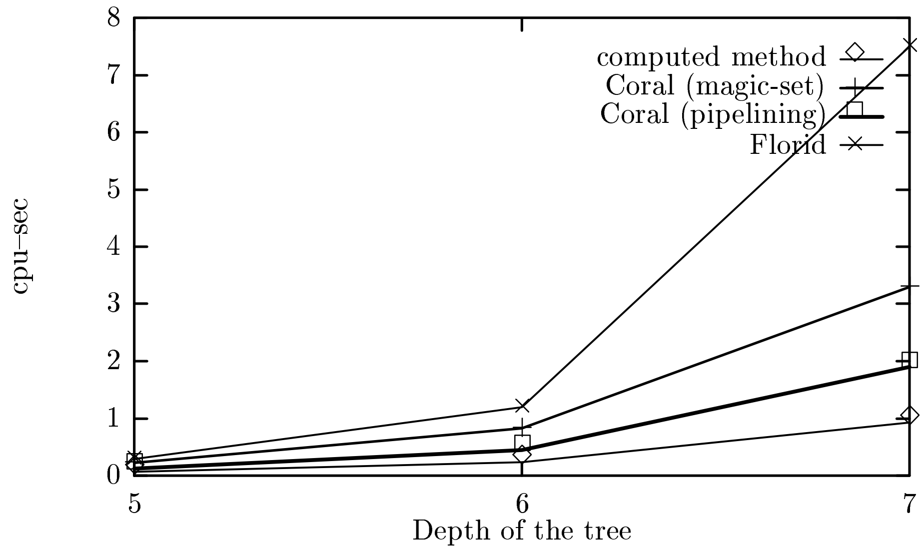
Number of leaves in a tree In a complete binary tree the number of leaves is computed. The relation "left (right) neighbour" is given in the EDB.

/* version with computed method */
X.leafnum $\dot{\longrightarrow}$ 1 | X:leaf
X.leafnum $\dot{\longrightarrow}$ X.left.leafnum + X.right.leafnum | X:inner

```
/* Datalog version for Coral */
leafnum(X,1) :– leaf(X).
leafnum(X,N) :–
        inner(X),
        left(X,XL),
        right(X,XR),
        leafnum(XL,NL),
        leafnum(XR,NR),
        N = NL + NR.


/* version with F-logic rules for Florid*/
X[leafnum→1] :– X:leaf.
X[leafnum→L] :– X:inner, X.left.leafnum + X.right.leafnum = L.
```

Time is taken for trees of different depth.

Depth	comp. method	Coral		Florid
		magic-set	pipelining	
5	0.07	0.23	0.12	0.3
6	0.24	0.84	0.46	1.2
7	0.94	3.3	1.9	7.5



In this example always the computed method version works fastest.

5 Conclusion

Computed methods can significantly improve the performance of DOOD-programs under bottom-up evaluation. We briefly sum up the distinctive features:

- A programmer can use a computed method whenever certain restrictions conforming to the functional paradigm apply:
  - A *safeness* condition ensuring directed use from call to result, that is, no inversion of a computed method is possible,
  - A syntactical *uniqueness* property ensuring consistency of the method definition.
- If a method is specified as a computed one, it is evaluated by functional reduction rather than by explicit bottom-up deduction, hence the following advantages are achieved:
  - If a computed method as well as a direct formulation with logical rules is possible, using a computed method can save much time and space as shown by the experiment.
  - Frequently termination problems are avoided. Consider again the method totalcost (Example 1). It is not at all an easy task to write logical rules for map or sum which will terminate when evaluated bottom-up; certainly such a program will not be just as concise and clear.
- Computed methods work together with all essential features of F-logic; they will probably be applicable to many other DOOD-languages as well. Also, other optimizations like seminaive evaluation may be used additionally.

We think that the first experimental results – though being far away from a thorough complexity analysis – are very promising and that future research on applications of computed methods will be worthwhile.

Acknowledgements My thanks go to Georg Lausen for suggesting the topic and useful discussion.

References

[1] S. Abiteboul. Towards a deductive object-oriented database language. In Kim et al. [9], pages 419 – 438.

[2] S. Abiteboul and R. Hull. Data functions, datalog and negation. In H. Boral and P.-A. Larson, editors, *Proc. ACM SIGMOD Intl. Conference on Management of Data*, pages 143–153, 1988.

[3] H. Aït-Kaci and A. Podelski. Towards a Meaning of Life. In J. Maluszyński and M. Wirsing, editors, *Programming Language Implementation and Logic Programming (PLILP)*, number 528 in LNCS. Springer, 1991.

[4] J. Frohn, G. Lausen, and H. Uphoff. Access to objects by path expressions and rules. In J. B. Bocca, M. Jarke, and C. Zaniolo, editors, *Proc. Intl. Conference on Very Large Data Bases (VLDB)*, Santiago de Chile, 1994.

[5] S. Grumbach. Integration of Functions Defined with Rewriting Rules in Datalog. In Kim et al. [9].

[6] M. Hanus. The integration of functions into logic programming: from theory to practice. *Journal of Logic Programming*, 19/20, 1994.

[7] P.-T. Kandzia. Nonmonotonic Reasoning in Florid. In *Proc. Intl. Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR)*, number 1265 in LNAI. Springer, 1997.

[8] M. Kifer, G. Lausen, and J. Wu. Logical foundations of object-oriented and frame-based languages. *Journal of the ACM*, 42(4):741–843, July 1995.

[9] W. Kim, J.-M. Nicolas, and S. Nishio, editors. *Proc. Intl. Conference on Deductive and Object-Oriented Databases (DOOD)*. North-Holland/Elsevier Science Publishers, 1989.

[10] J. W. Lloyd. *Foundations of Logic Programming*. Springer, 2nd edition, 1987.

[11] Y. Lou and Z. M. Ozsoyoglu. LLO: An object oriented deductive language with methods and method inheritance. In J. Clifford and R. King, editors, *Proc. ACM SIGMOD Intl. Conference on Management of Data*, 1991.

[12] J. Moreno-Navarro and M. Rodriguez-Artalejo. Logic programming with functions and predicates: The language BABEL. *Journal of Logic Programming*, 12(3&4), 1992.

[13] R. Ramakrishnan, D. Srivastava, S. Sudarshan, and P. Seshadri. The CORAL deductive system. *The VLDB Journal*, 3(2):161–210, Apr. 1994.

[14] Z. Xie and J. Han. Normalization and compilation of deductive and object-oriented database programs for efficient query evaluation. In T. W. Ling, A. O. Mendelzon, and L. Vieille, editors, *Proc. Intl. Conference on Deductive and Object-Oriented Databases (DOOD)*, number 1013 in LNCS, pages 485 – 502, Singapore, 1995. Springer.