Defining Datalog in Rewriting Logic[⋆]

M. Alpuente, M. A. Feliú, C. Joubert, and A. Villanueva

Universidad Politécnica de Valencia, DSIC / ELP
Camino de Vera s/n, 46022 Valencia, Spain
{alpuente,mfeliu,joubert,villanue}@dsic.upv.es

Abstract. Recently, it has been demonstrated the effective application of logic programming to problems in program analysis. Using a simple relational query language, like DATALOG, complex interprocedural analyses involving dynamically created objects can be expressed in just a few lines. By exploiting the main features of a term rewriting system like MAUDE, we aim at transforming DATALOG programs into efficient rewrite systems. The transformation is proved to be sound and complete. A prototype has been implemented and applied to some real-world DATALOG-based analyses. Experimental results show that solving a DATALOG query using rewriting logic is comparable to state-of-the-art DATALOG solvers.

1 Introduction

DATALOG [1] is a simple relational query language which permits to describe, in an intuitive way, complex interprocedural program analyses involving dynamically created objects. The main advantage of formulating data-flow analyses as a DATALOG query is that analyses that take hundreds of lines of code in a traditional language can be expressed in a few lines of DATALOG [2]. In real-world problems, the DATALOG rules encoding a particular analysis must be solved generally under the huge set of DATALOG facts that are automatically extracted from the analyzed program. In this context, all program updates, like pointer updates, might potentially be inter-related, leading to an exhaustive computation of all results. An important number of optimization techniques for DATALOG has been designed and studied extensively in program analysis, logic programming, and deductive databases [3, 4].

The aim of this paper is to provide efficient DATALOG query answering in Rewriting Logic [5], a very general *logical* and *semantical framework* efficiently implemented in the high-level programming language MAUDE [6]. Our motivation for using Rewriting Logic is to overcome the difficulty to handle metaprogramming features such as reflection in traditional analysis frameworks [7]. Actually, tracking reflective methods invocations requires not just tracking object references through variables but actually tracking method values and method

[⋆] This work has been partially supported by the EU (FEDER), the Spanish MEC/MICINN under grant TIN 2007-68093-C02, the Generalitat Valenciana under grant Emergentes GV/2009/024, and the Universidad Politécnica de Valencia under grant PAID-06-07.

name strings. We consider it a challenge to investigate the interaction of static analysis with metaprogramming frameworks. An additional goal of this work is to evaluate if MAUDE is able to process a sizable number of equations that come from real-life problems, like those from static analysis problems.

In the related literature, the solution for a Datalog query is classically constructed following a bottom-up approach, thus not taking advantage of the information in the query until the model has been constructed [8]. On the contrary, the typical logic programming interpreter would produce the output in a top-down fashion by reasoning backwards from the query. Between these two extremes, there is a whole spectrum of evaluation strategies [4, 9, 10]. While bottom-up computation may be very inefficient, the top-down approach is prone to infinite computations. In this work, we follow a top-down approach equipped with a loop check in order to avoid infinite computations, as in [11].

Logic and functional programming are both instances of rule-based, declarative programming and hence it is not surprising that the relationship between them has been studied. However, the operational principle differs: logic programming is based on *resolution* whereas functional programs are executed by *term rewriting*. There exist many proposals for transforming logic programs into rewriting theories [12–15]. These transformations aim at reusing the infrastructure of term rewriting systems to run the (transformed) logic program while preserving the intended observable behavior (e.g. termination, success set, computed answers, etc). Traditionally, translations of logic programs into functional programs are based on imposing an input/output relation among the parameters of the original program [15]. However, one distinguished feature of DATALOG programs burdening the transformation is that predicate arguments are not *moded*, meaning that they can be used both as input or output parameters.

One recent transformation that does not impose an input/output behavior among parameters was presented in [14]. The authors defined a transformation from definite logic programs into (infinitary) term rewriting for the termination analysis of logic programs. Contrary to our approach, the transformation of [14] is not concerned with preserving the computed answers, but only the termination behavior. Moreover, [14] does not tackle the problem of efficiently encoding logic (DATALOG) programs containing a huge amount of facts in a rewriting-based infrastructure such as MAUDE. After exploring the impact of different implementation choices (equations *vs* rules, extra conditions, etc.) in our working scenario, i.e., heavy data load (sets of hundreds of facts) together with relatively few clauses encoding the analysis to perform, in this work we present an equation-based transformation that leads to efficient MAUDE programs.

In previous work [16], we developed a DATALOG query solving technique based on *Boolean Equation Systems* (BESS) [17]. Although the correspondence between answering a DATALOG query and solving a BES can be established naturally, the main limitation of this approach is in the difficulty to combine indexed and linked data structures in order to schedule suitable optimizations which ensure that only useful combination of facts are simultaneously considered. In this paper, we stay at a higher level in the sense that we transform a high-level Datalog program into another high-level Maude program. The goal is to take advantage of the flexibility and versatility of Maude in order to achieve scalability without losing declaratively.

In Section 2, we present our running example: a program analysis expressed as a Datalog program that we will use to illustrate the general transformation from a Datalog program into a Maude program. In Section 3, we describe the transformation of the example. Section 4 formalizes the general process and prove its correctness and completeness. Section 5 shows experimental results obtained with realistic examples and compares our Maude implementation to state-of-the-art Datalog solvers. We conclude and discuss future work in Section 6.

2 A program analysis written as a Datalog program

Datalog is a relational language using declarative *clauses* to both describe and query a deductive database. A Datalog clause is a function-free Horn clause over a finite alphabet of *predicate* symbols (e.g., relation names or arithmetic predicates, such as $<$) whose *arguments* are either variables or constant symbols. A Datalog program $\mathcal{R}$ is a finite set of Datalog clauses [8].

Definition 1 (Syntax of Rules). *Let $\mathcal{P}$ be a set of* predicate *symbols, $\mathcal{V}$ be a finite set of* variable *symbols, and $\mathcal{C}$ a set of* constant *symbols. A* Datalog clause *r defined over a finite alphabet $P \subseteq \mathcal{P}$ and arguments from $V \cup C$, $V \subseteq \mathcal{V}$, $C \subseteq \mathcal{C}$, has the following syntax:*

$$p_0(a_{0,1}, \ldots, a_{0,n_0}) \;:-\; p_1(a_{1,1}, \ldots, a_{1,n_1}),\; \ldots,\; p_m(a_{m,1}, \ldots, a_{m,n_m}).$$

where $m \geq 0$, and each p_i is a predicate symbol of arity n_i with arguments $a_{i,j} \in V \cup C$ ($j \in [1..n_i]$).

The atom $p_0(a_{0,1}, \ldots, a_{0,n_0})$ in the left-hand side of the clause is the clause's *head*, where p_0 is not arithmetic. The finite conjunction of *subgoals* in the right-hand side of the clause is the clause's *body*, *i.e.*, a sequence of atoms that contain all variables appearing in the head. Following logic programming terminology, a clause with empty body ($m = 0$) is called a *fact*. A clause with empty head and $m > 0$ is called a *query*, and $\Box$ denotes the empty clause. A syntactic object (argument, atom, or clause) that contains no variables is called *ground*. Moreover, an *existentially quantified variable* is a variable that appears in the body of a clause and does not occur in its head. The variables appearing in a query are called *output* variables.[1]

Given a Datalog program $\mathcal{R}$ and a query q, we follow a top-down approach and use SLD-resolution to compute the set of answers of q in $\mathcal{R}$. Given the successful derivation $\mathcal{D} \equiv q \Rightarrow^{\theta_1}_{SLD} q_1 \Rightarrow^{\theta_2}_{SLD} \cdots \Rightarrow^{\theta_n}_{SLD} \Box$, the answer computed by $\mathcal{D}$ is $\theta_1\theta_2 \ldots \theta_n$ restricted to the variables occurring in q.

[1] In the sequel of the paper, Datalog programs are considered to be as defined in this section.

Let us now introduce the running DATALOG program example that we use along the paper. This program defines a simple context-insensitive inclusion-based pointer analysis for an object-oriented language such as JAVA. This analysis is defined by the following predicate `vP/2` representing the fact that a program variable points directly (via `vP0/2`) or indirectly (via `a/2`) to a given position in the heap. The second clause states that `Var1` points to `Heap` if `Var2` points to `Heap` and `Var2` is assigned to `Var1`:

```
vP(Var,Heap)  :- vP0(Var,Heap).
vP(Var1,Heap) :- a(Var1,Var2),vP(Var2,Heap).
```

The predicates `a/2` and `vP0/2` are defined extensionally by a number of facts that are automatically extracted from the original program being statically analyzed. The intuition is that the `a/2` predicate represents a direct assignment from a program variable to another variable, whereas `vP0/2` represents newly created pointers within the analyzed (object-oriented) program from a program variable to the heap. The following code excerpt contains a number of DATALOG facts complementing the above pointer analysis description for a particular object-oriented example program.

```
a(v1,v2).
a(v1,v3).
vP0(v2,h5).
vP0(v3,h4).
```

In the considered DATALOG analysis program, a query typically consists in computing the objects in the heap pointed by a specific variable. We write such a query as `?- vP(v1,Heap).`. The expected outcome of this query is the set of all possible answers, i.e., the set of substitutions mapping the variable `Heap` to constants satisfying the query. In the example, the set of computed answers for the considered query is {{`Heap/h4`},{`Heap/h5`}}.

Another possible query is `?- vP(Var,h5).`, where `h5` stands for a heap object. The solver should compute which are the variables in the analyzed program that can point to the object `h5`.

Similarly to [14], our goal is to define a *mode*-independent transformation for (DATALOG) logic programs in order to keep the possibility of running both kinds of queries. Since variables in rewriting logic are input-only parameters, we cannot use them to encode logic variables of DATALOG. We follow the standard approach based on defining a ground representation for logic variables [6, 18].

3 From DATALOG to MAUDE

As explained above, we are interested in computing by rewriting all answers for a given query. A naïve approach is to translate DATALOG clauses into MAUDE rules, similarly to [14], and then use the `search`[2] command of MAUDE in order to mimic all possible executions of the original DATALOG program. However, in the context of program analysis, this approach results in poor performance.

[2] Intuitively, `search` $t \to t'$ explores the whole rewriting space from the term t to any other terms that match t' [6].

In this section, we first formulate a suitable representation in Maude of the Datalog computed answers. Then, we informally introduce the transformation by means of the running example. Section 4 formalizes the translation procedure and proves its correctness.

3.1 Answers representation

Let us first introduce our representation of variables and constants of a Datalog program as *ground terms* of a given sort in Maude. We define the sorts `Variable` and `Constant` to specifically represent in Maude the variables and constants of the original Datalog program, whereas the sort `Term` (resp. `TermList`) represents Datalog terms (resp. lists of terms, built by simple juxtaposition):

```
sorts Variable Constant Term TermList .
subsort Variable Constant < Term .
subsort Term < TermList .
op __ : TermList TermList -> TermList [assoc] .
op nil : -> TermList .
```

For instance, `T1 T2` represents the list of terms `T1` and `T2`. In order to construct the elements of the `Variable` and `Constant` sorts, we introduce two constructor symbols: Datalog constants are represented as Maude *Quoted Identifiers* (`Qid`s), whereas logical variables are encoded in Maude by means of the constructor symbol `v`. These constructor symbols are specified in Maude as follows:

```
subsort Qid < Constant .         --- Every Qid is a Constant
op v : Qid -> Variable [ctor] . --- v(q) is a Variable if q is a Qid
op v : Term Term -> Variable [ctor] .
```

The last line of the above code excerpt allows us to build variable terms of the form `v(T1,T2)` where both `T1` and `T2` are `Term`s. This is used to ensure that the ground representation in Maude for existentially quantified variables appearing in the body of Datalog clauses is unique to the whole Maude program.

Having ground terms representing variables, we still lack a way to collect the answers for an output variable. In our formulation, answers are stored within the term representing the ongoing partial computation of the Maude program. Thus, we represent a (partial) answer for the original Datalog query as a sequence of equations (called answer constraint) that represents the substitution of (logical) variables by (logical) constants computed during the program execution. We define the sort `Constraint` representing a single answer for a Datalog query, but we also define a hierarchy of subsorts (e.g., the sort `FConstraint`, at the bottom of the hierarchy, represents inconsistent solutions) that allows us to identify the inconsistent as well as the *trivial* constraints (`Cte = Cte`) whenever possible. This hierarchy allows us to simplify constraints as soon as possible and to improve performance. The resulting Maude program is as follows:

```
sorts Constraint EmptyConstraint NonEmptyConstraint TConstraint FConstraint .
subsort EmptyConstraint NonEmptyConstraint < Constraint .
subsort TConstraint FConstraint < EmptyConstraint .
```

```
op _=_ : Term Constant -> NonEmptyConstraint .
op T : -> TConstraint .
op F : -> FConstraint .
op _,_ : Constraint Constraint -> Constraint              [assoc comm id: T] .
op _,_ : FConstraint Constraint -> FConstraint                       [ditto] .
op _,_ : TConstraint TConstraint -> TConstraint                      [ditto] .
op _,_ : NonEmptyConstraint TConstraint -> NonEmptyConstraint        [ditto] .
op _,_ : NonEmptyConstraint FConstraint -> FConstraint               [ditto] .
op _,_ : NonEmptyConstraint NonEmptyConstraint -> NonEmptyConstraint [ditto] .

var NEC           : NonEmptyConstraint .
var V             : Variable .
var Cte Cte1 Cte2 : Constant .

eq (Cte = Cte) = T .                 --- Simplification
eq (Cte1 = Cte2) = F [owise] .       --- Unsatisfiability
eq NEC,NEC = NEC .                   --- Idempotence
eq F,NEC = F .                       --- Zero element
eq F,F = F .                         --- Simplification
eq (V = Cte1),(V = Cte2) = F [owise] .--- Unsatisfiability
```

Note that the conjunction operator `_,_` has identity element `T` and obeys the laws of associativity and commutativity. We express the idempotence property of the operator by a specific equation on variables from the `NonEmptyConstraint` subsort `NEC`. A query reduced to `T` represents a successful computation.

Since equations in MAUDE are run deterministically, all the non-determinism of the original DATALOG program has to be embedded into the carried constraints themselves. This means that we need to carry on not only a single answer, but all the possible (partial) answers at a given execution point. To this end, we introduce the notion of *set of answer constraints*, and implement in MAUDE a new sort called `ConstraintSet`:

```
sorts ConstraintSet EmptyConstraintSet NonEmptyConstraintSet .
subsort EmptyConstraintSet NonEmptyConstraintSet < ConstraintSet .
subsort NonEmptyConstraint TConstraint < NonEmptyConstraintSet .
subsort FConstraint < EmptyConstraintSet .

op _;_ : ConstraintSet ConstraintSet -> ConstraintSet [assoc comm id: F] .
op _;_ : NonEmptyConstraintSet ConstraintSet -> NonEmptyConstraintSet [assoc comm id: F] .

var NECS : NonEmptyConstraintSet .

eq NECS ; NECS = NECS .     --- Idempotence
```

It is easy to grasp the intuition behind the different sorts and subsort relations in the above fragment of MAUDE code. The operator `_;_` represents the disjunction of constraints. The associativity, commutativity and (the existence of an) identity element properties of `_;_` can be easily expressed by using `ACU` attributes in MAUDE, thus simplifying the equational specification and achieving better efficiency. We express the idempotence property of the operator `_;_` by a specific equation on variables from the `NonEmptyConstraintSet` subsort.

In order to incrementally add new constraints along the program execution, we define the composition operator `x` as follows:

```
op _x_ : ConstraintSet ConstraintSet -> ConstraintSet [assoc] .

var CS            : ConstraintSet .
var NECS1 NECS2   : NonEmptyConstraintSet .
var NEC NEC1 NEC2 : NonEmptyConstraint .

eq F x CS = F .                                            --- L-Zero element
eq CS x F = F .                                            --- R-Zero element
eq F x F = F .                                             --- Double-Zero
eq NEC1 x (NEC2 ; CS) = (NEC1 , NEC2) ; (NEC1 x CS) .      --- L-Distributive
eq (NEC ; NECS1) x NECS2 = (NEC x NECS2) ; (NECS1 x NECS2) . --- R-Distributive
```

In order to keep information consistent, some simplification equations are automatically applied. These equations make every inconsistent constraint collapse into a `F` value, and trivial constraints are simplified.

3.2 A glimpse of the transformation

In order to mimic the execution order of the subgoals in the body of the Datalog clauses, the first naïve idea is trying to translate each Datalog clause into a conditional equation. The execution of these kinds of equations suffers an important penalty within the rewriting machinery of Maude that dramatically slows down the overall performance of the computation. In order to obtain better performance, we disregard conditional equations in favor of non-conditional ones and impose an evaluation order by means of some auxiliary *unraveling* [19] functions that stepwisely evaluate each call and propagate the (partially) computed information. We rely on pattern matching to ensure that a call is executed only when the previous one has been solved.

For each Datalog predicate, we introduce one equation representing the disjunction of the possible answers delivered by all the clauses defining that predicate. In the case of predicates defined by facts, each fact can be represented as a `Constraint` term in our setting. Thus, we transform the set of facts defining a particular predicate as a single equation whose rhs consists of the disjunction of `Constraint` terms representing each particular Datalog fact. Considering the running example, facts are transformed to:

```
eq a(T1,T2)   = ((T1 = 'v1) , (T2 = 'v2)) ; ((T1 = 'v1) , (T2 = 'v3)) .
eq vP0(T1,T2) = ((T1 = 'v2) , (T2 = 'h5)) ; ((T1 = 'v3) , (T2 = 'h4)) .
```

In the case of predicates defined by clauses with non-empty body, we generate as many auxiliary functions as different clauses define the Datalog predicate. For instance, the answers for `vP/2` in the example are the disjunction of the answers of functions `vPc1` and `vPc2`,[3] representing the calls to the first and second Datalog clauses of the running example, respectively:

```
eq vP(T1,T2) = vPc1(T1,T2) ; vPc2(T1,T2) .
```

[3] The `c` in `vPc1` and `vPc2` stands for *clause*.

The specification for the first clause `vPc1` is given by

```
eq vPc1(T1,T2) = vP0(T1,T2) .
```

The transformation for the second clause of the program, represented by `vPc2`, is a bit more elaborated since, first, it contains more than one subgoal, thus we need an auxiliary function to impose the execution order. Moreover, it contains an existentially quantified variable (not appearing in the head of the clause) that carries information from one subgoal to the other.

```
eq vPc2(T1,T2) = vPc2s2(a(T1,v(T1,T2)), T1 T2) .
eq vPc2s2(((v(T1,T2) = Cte) , C) ; CS, T1 T2) =
        (vP(Cte,T1 T2) x ((v(T1,T2) = Cte) , C)) ; vPc2s2(CS,T1 T2) .
eq vPc2s2(F,T1 T2) = F .
```

As one can observe, `vPc2` calls to `vPc2s2`, whose first argument represents the execution of the first subgoal and the second one is the list of parameters in the head of the original clause. The pattern in the first argument in the lhs of the equation for `vPc2s2` forces to compute the (partial) answers resulting from the resolution of `a(T1,v(T1,T2))` first to proceed. The use of the term `v(T1,T2)`, representing the existentially quantified variable `Var2` of the original DATALOG program, in the pattern of the specification of the equation `vPc2s2` is the key for carrying the computed information from one subgoal to the subsequent subgoals where the variable occurs. The idea is that `vPc2s2` is defined to receive the value of the shared variable on the pattern `((V = Cte) , C) ; CS)`. The recursion over `vPc2s2` is needed because its first argument represents all the possible answers computed by `a(T1,v(T1,T2))`, thus we recursively compute each solution and use the constraints composition operator previously defined to combine them.

In order to execute a query in the transformed program, we call the MAUDE `reduce` command. The query that computes all positions to which each variable can point-to can be written in MAUDE as follows:

```
reduce vP(v('variable),v('heap)) .
```

The answers to this query are shown below. The first sentence specifies the term which has been reduced. The second sentence shows the number of rewrites and the execution time that MAUDE invested to perform the reduction. The last sentence, written in several lines for the sake of readability, shows the result of the reduction together with its type.

```
reduce in ANALYSIS : vP(v('v), v('h)) .
rewrites: 39 in 0ms cpu5 (0ms real) (  rewrites/second)
result NonEmptyConstraintSet:
  ((v('h) = 'h4),v('v) = 'v3) ;
  ((v('h) = 'h5),v('v) = 'v2) ;
  ((v('h) = 'h4),(v('v) = 'v1),v(v('v), v('h)) = 'v3) ;
  (v('h) = 'h5),(v('v) = 'v1),v(v('v), v('h)) = 'v2
```

As it was expected, four answers have been returned: the first two are obtained by the auxiliary function `vPc1`, whereas the other two are computed by the function `vPc2`.

4 Formal definition of the transformation

In this section, we first give a formal description of the new transformation from a Datalog program into a Maude program. Then, in Section 4.2 we prove the correctness and completeness of the transformation.

4.1 The transformation

Let P be a Datalog program defining predicate symbols $p_1 \ldots p_n$. Before describing the transformation process, we introduce some auxiliary notations. $|p_i|$ is the number of facts or clauses defining the predicate symbol p_i. Following the Datalog standard, we assume without loss of generality that a predicate p_i is defined only by facts, or only by clauses [8]. The arity of p_i is ar_i.

Let us start by describing the case when predicates are defined by facts. We transform the whole set of facts defining a given predicate symbol p_i into a single equation by means of a disjunction of answer constraints. Formally, for each p_i with $1 \le i \le n$ that is defined in the Datalog program only by facts, we write the following snippet of Maude code, where the symbol $c_{i,j,k}$ is the k-th argument of the j-th fact defining the predicate symbol p_i:

```
var T$_{i,1}$ ... T$_{i,ar_i}$ : Term .
eq p$_i$(T$_{i,1}$, ... ,T$_{i,ar_i}$) = (T$_{i,1}$ = $c_{i,1,1}$, ... , T$_{i,ar_i}$ = $c_{i,1,ar_i}$) ; ...
                              ; (T$_{i,1}$ = $c_{i,|p_i|,1}$, ... , T$_{i,ar_i}$ = $c_{i,|p_i|,ar_i}$) .
```

Similarly, our transformation for Datalog clauses with non-empty body combines, in a single equation, the disjunction of the calls to all functions representing the different clauses for the considered predicate symbol p_i. For each p_i with $1 \le i \le m$ with non empty body, we have the following Maude piece of code:

```
var T$_{i,1}$ ... T$_{i,ar_i}$ : Term .
eq p$_i$(T$_{i,1}$, ... ,T$_{i,ar_i}$) = p$_{i,1}$(T$_{i,1}$, ... ,T$_{i,ar_i}$) ; ...
                              ; p$_{i,|p_i|}$(T$_{i,1}$,...,T$_{i,ar_i}$) .
```

Each call $\mathtt{p}_{i,j}$ with $1 \le j \le |p_i|$ produces the answers computed by the j-th clause of the predicate symbol. Now we need to define how each of these clauses is transformed. Notation $\tau^a_{i,j,s,k}$ denotes the name of the variable or constant symbol appearing in the k-th argument of the s-th subgoal in the j-th clause defining the i-th predicate of the original Datalog program. When $s = 0$ then the function refers to the arguments in the head of the clause.

Let us start by considering the case when the body has just one subgoal. We define the function $\tau^p_{i,j,s}$ that returns the predicate symbol appearing in the s-th subgoal of the j-th clause defining the i-th predicate in the Datalog program. For each clause having just a subgoal, we get the following transformation:

```
eq p$_{i,j}$($\tau^a_{i,j,0,1}$,...,$\tau^a_{i,j,0,ar_i}$) = $\tau^p_{i,j,1}$($\tau^a_{i,j,1,1}$,...,$\tau^a_{i,j,1,ar_l}$) .
```

In the case where more than one subgoal appears in the body of a clause, we want to impose a left-to-right evaluation strategy. We use auxiliary functions defined by patterns to force such an execution order. In particular, we set that a subgoal cannot be invoked until the variables in its arguments that also occur in previous subgoals have been instantiated. We call these variables *linked* variables.

Let us first introduce some definitions and functions that will be used in our transformation.

Definition 2 (linked variable). *A variable is called* linked variable *iff it does not occur in the head of a* DATALOG *clause, and occurs in two or more subgoals of the clause's body.*

Definition 3 (function linked). *Let C be a* DATALOG *clause. Then the function* $\mathsf{linked}(C)$ *is the function that returns the list of pairs containing in the first component a linked variable, and in the second component the list of positions where such a variable occurs in the body of the clause*[4].

Example 1. For example, given the DATALOG clause

$C =$ `p(X1,X2) :- p1(X1,X3), p2(X3,X4), p3(X4,X2).`

we have that

$\mathsf{linked}(C)$ `= [(X3,[1.2,2.1]),(X4,[2.2,3.1])]`

Now we define the notion of *relevant* linked variables for a given subgoal, namely the linked variables of a subgoal appearing also in some previous subgoal.

Definition 4 (Relevant linked variables). *Given a clause C and an integer number n, we define the function* relevant *that returns the variables that are common for the n-th subgoal and some previous subgoal:*

$$\mathit{relevant}(n, C) = \{X \mid (X, LX) \in \mathsf{linked}(C), \text{ and there exists } m < n, \exists j \text{ s.t. } m.j \in LX\}$$

Note that, similarly to [14], we are not marking the input/output positions of predicates, as required in more traditional transformations. We are just identifying which are the variables whose values must be propagated for evaluating the subsequent subgoals following the evaluation strategy.

Now we are ready to address the problem of transforming a clause with more than one subgoal (and maybe existentially quantified variables) into a set of equations. Intuitively, the main function initially calls to an auxiliary function that undertakes the execution of the first subgoal. We have as many auxiliary functions as subgoals in the original clause, and in the rhs's of the auxiliary functions, the execution order of the successive subgoals is implictly controlled by passing the results of each subgoal as a parameter to the subsequent function.

Let the function $\mathtt{p}_{i,j}$ generate the solutions calculated by the j-th clause of the predicate symbol p_i. We state that $\mathtt{ps}_{i,j,s}$ represents the auxiliary function corresponding to the s-th subgoal of the j-th clause defining the predicate p_i. Then, for each clause we have the following translation, where the variables $\mathtt{X}_1 \ldots \mathtt{X}_N$ of each equation are calculated by the function $\mathsf{relevant}(k, \mathsf{linked}(\mathit{clause}(i,j)))$[5] and transformed into the corresponding MAUDE terms.

[4] Positions extend to goals in the natural way.

[5] $\mathit{clause}(i,j)$ represents the j-th DATALOG clause defining the predicate symbol p_i.

The first equation reduces the considered DATALOG predicate to a call to the first auxiliary function that calculates the (partial) answers for the second subgoal, first computing the answers from the first subgoal $\tau^{p}_{i,j,1}$ in its first argument. The second argument of the equations represents the list of terms in the initial predicate call that, together with the information retrieved from Definitions 3 and 4, allow us to correctly build the patterns and function calls during the transformation.

```
eq p$_{i,j}$($\tau^a_{i,j,0,1}$,...,$\tau^a_{i,j,0,ar_i}$) = ps$_{i,j,2}$($\tau^p_{i,j,1}$($\tau^a_{i,j,1,1}$,...,$\tau^a_{i,j,1,r}$), $\tau^a_{i,j,0,1}$ ... $\tau^a_{i,j,0,ar_i}$) .
```

where r is the arity of the predicate $\tau^{p}_{i,j,1}$. Then, for each auxiliary function, first we declare as many constants as relevant variables the given subgoal has. The left hand side of the equation is defined with patterns that adjust the relevant variables to the values already computed by the execution of a previous subgoal. Note that we may have more assignments in the constraint, represented by `C`, and that we may have more possible solutions in `CS`. The auxiliary equation `ps'`$_{i,j,s}$ takes each possible (partial) solution and combines it with the solutions given by the s-th subgoal in the clause (whose predicate symbol is $\tau^{p}_{i,j,s}$). Note that we propagate the instantiation of the relevant variables by means of a substitution.

```
var C$_1$ ...C$_N$ : Constant .
var NECS : NonEmptyConstraintSet .

eq ps$_{i,j,s}$(NECS, T$_1$...T$_{ar_i}$) = ps$_{i,j,s+1}$(ps'$_{i,j,s}$(NECS, T$_1$...T$_{ar_i}$), T$_1$...T$_{ar_i}$) .
eq ps$_{i,j,s}$(F , LL) = F .

eq ps'$_{i,j,s}$(((X$_1$=C$_1$,...,X$_N$=C$_N$, C) ; CS), T$_1$...T$_{ar_i}$) =
        (($\tau^p_{i,j,s}$($\tau^v_{i,j,s,1}$,...,$\tau^v_{i,j,s,r}$)[X$_1$\C$_1$,...,X$_N$\C$_N$]) x (X$_1$=C$_1$,...,X$_N$=C$_N$, C)) ;
        ps'$_{i,j,s}$(CS, T$_1$...T$_{ar_i}$) .
eq ps'$_{i,j,s}$((T ; CS), T$_1$...T$_{ar_i}$) =
        $\tau^p_{i,j,s}$($\tau^v_{i,j,s,1}$,...,$\tau^v_{i,j,s,r}$) ; ps'$_{i,j,s}$(CS, T$_1$...T$_{ar_i}$) .
eq ps'$_{i,j,s}$(F , LL) = F .
```

The equation for the last subgoal in the clause is slightly different, since we need not invoke the following auxiliary function. Assuming that g denotes the number of subgoals in a clause, we define

```
eq ps$_{i,j,g}$(((X$_1$=C$_1$,...,X$_N$=C$_N$, C) ; CS) , T$_1$...T$_{ar_i}$) =
        (($\tau^p_{i,j,g}$($\tau^v_{i,j,g,1}$,...,$\tau^v_{i,j,g,r}$)[X$_1$\C$_1$,...,X$_N$\C$_N$]) x (X$_1$=C$_1$,...,X$_N$=C$_N$, C)) ;
        ps$_{i,j,g}$(CS , T$_1$...T$_{ar_i}$) .
eq ps$_{i,j,g}$((T ; CS) , T$_1$...T$_{ar_i}$) =
        $\tau^p_{i,j,g}$($\tau^v_{i,j,g,1}$,...,$\tau^v_{i,j,g,r}$) ; ps$_{i,j,g}$(CS , T$_1$...T$_{ar_i}$) .
eq ps$_{i,j,g}$(F , LL) = F .
```

Finally, we define the transformation for the DATALOG query $q(X_1, \ldots, X_n)$ where X_i, $1 \leq i \leq n$ are DATALOG variables or constants. The MAUDE encoding of the query is `q`(τ^q_1,...,τ^q_n) where τ^q_i, $1 \leq i \leq n$ is the transformation of the corresponding X_i.

4.2 Correctness of the transformation

We have defined a transformation from DATALOG programs specifying static analyses into MAUDE programs in such a way that the normal form computed for a term of the `ConstraintSet` sort represents the set of computed answers for a query of the original Datalog program. In this section we show that the transformation is sound and complete w.r.t. computed answers.

We first introduce some notation. Let `CS` be a `ConstraintSet` of the form $\mathtt{C}_1$; $\mathtt{C}_2$; ...; $\mathtt{C}_n$ where each $\mathtt{C}_i$, $i \geq 1$ is a `Constraint` in normal form ($\mathtt{C}_1$ = $\mathtt{Cte}_1$,...,$\mathtt{C}_m$ = $\mathtt{Cte}_m$), and V be a list of variables. We write $\mathtt{C}_i|_V$ to the restriction of the constraint $\mathtt{C}_i$ to the variables in V. We extend the notion to sets of constraints in the natural way, and denote it as $\mathtt{CS}|_V$. Given two terms t and t', we write $t \rightarrow^*_S t'$ when there exists a rewriting sequence from t to t' in the Maude program S. Also, $var(t)$ is the set of variables occurring in t.

Now we define a suitable notion of *(rewriting) answer constraint*:

Definition 5 (Answer Constraint Set). *Given a* Maude *program S as described in this work and a input term t, we say that the* answer constraint set *computed by $t \rightarrow^*_S$ `CS` is $\mathtt{CS}|_{var(t)}$.*

There is a natural isomorphism between the equational constraint C and an idempotent substitution $\theta = \{X_1/C_1, X_2/C_2, \ldots, X_n/C_n\}$, given by: `C` is equivalent to θ iff ($\mathtt{C} \Leftrightarrow \hat{\theta}$), where $\hat{\theta}$ is the equational representation of θ. By abuse, given a disjunction `CS` of equational constraints and a set of idempotent substitutions ($\Theta = \cup_{i=1}^n \theta_i$), we define $\Theta \equiv$ `CS` iff $\mathtt{CS} \Leftrightarrow \bigvee_{i=1}^n \hat{\theta}_i$

Next we prove that for a given query and Datalog program, each answer constraint set computed for the corresponding input term in the transformed Maude program is equivalent to the set of computed answers of the original Datalog program. The proof of this result is given in [21].

Theorem 1 (Correctness and completeness). *Consider a* Datalog *program P together with the query q. Let $\mathcal{T}(P)$ be the corresponding, transformed* Maude *program, and $\mathcal{T}_g(q)$ be the corresponding, transformed input term. Let Θ be the set of computed answers of P for the query q, and $\mathtt{CS}|_{var(\mathcal{T}_g(q))}$ be the answer constraint set computed by $\mathcal{T}_g(q) \rightarrow^*_{\mathcal{T}(P)}$ `CS`. Then, $\Theta \equiv \mathtt{CS}|_{var(\mathcal{T}_g(q))}$.*

5 Experimental results

This section reports on the performance of our prototype implementing the transformation. First, we compare the efficiency of our implementation with respect to a naïve transformation to rewriting logic documented in [20]; then, we evaluate the performance of our prototype by comparing it to three state-of-the-art Datalog solvers.

All experiments were conducted using Java JRE 1.6.0, Joeq version 20030812, on a Mobile AMD Athlon XP2000+ (1.66GHz) with 700 Megabytes of RAM, running Ubuntu Linux 8.04.

5.1 Comparison w.r.t. the previous rewriting-based versions

We implemented several versions of transformation from Datalog programs to Maude programs before the one presented in this paper [20]. The first attempt consisted on a transformation based on a one-to-one correspondence of

DATALOG rules and MAUDE conditional rules. Then we tried to get rid of all the non-determinism introduced by conditional equations and rules. In the following, we briefly present the results obtained by using the rule-based approach, the equational-based approach, and the equational-based approach improved by using the memoization capability of MAUDE [6]. MAUDE is able to store each call to a given function (in the running example `vP(X,Y)`) together with its normal form. Thus, when MAUDE finds a memoized call it won't reduce it but just replaces it with its normal form, saving a great amount of rewrites.

Table 1 shows the resolution times of the three selected versions. The sets of initial DATALOG facts (`a/2` and `vP0/2`) are extracted by the JOEQ compiler from a JAVA program (with 374 lines of code) implementing a tree visitor. The DATALOG clauses are those of our running example: a simple context-insensitive inclusion-based pointer analysis. The evaluated query is `?- vP(Var,Heap).`, i.e., all possible answers that satisfy the predicate `vP/2`.

Table 1. Number of initial facts (`a/2` and `vP0/2`) and computed answers (`vP/2`), and resolution time (in seconds) for the three implementations.

a/2	vP0/2	VP/2	rule-based	equational	equational+memo
100	100	?	6.00	0.67	0.02
150	150	?	20.59	2.23	0.04
200	200	?	48.48	6.11	0.10
403	399	?	382.16	77.33	0.47
807	1669	?	4715.77	1098.64	3.52

The results obtained with the equational implementation are an order of magnitude better than those obtained by the naïve transformation based on rules. These results are due to the fact that backtracking operations associated to the non-determinism of rules and conditional equations and rules penalize the naïve version. We can also observe that using memoization enables us to gain another order of magnitude in execution time with respect to the basic equational implementation. These results confirm that the equational implementation fits our stated purpose, namely program analysis, and that it is likely to provide a useful way forward, compared to other implementations of DATALOG.

5.2 Comparison w.r.t. other state-of-the-art DATALOG solvers

The same sets of initial facts were used to compare our prototype (the equational-based version with memoization) with three state-of-the-art DATALOG solvers, namely XSB 3.2 [6], DATALOG 1.4 [7], and IRIS 0.58 [8]. Average resolution times of three runs for each solver are shown in Figure 1.

In order to evaluate the performance of our implementation with respect to the other DATALOG solvers, only resolution times are presented in Figure 1. This

[6] `http://xsb.sourceforge.net`

[7] `http://datalog.sourceforge.net`

[8] `http://iris-reasoner.sourceforge.net`

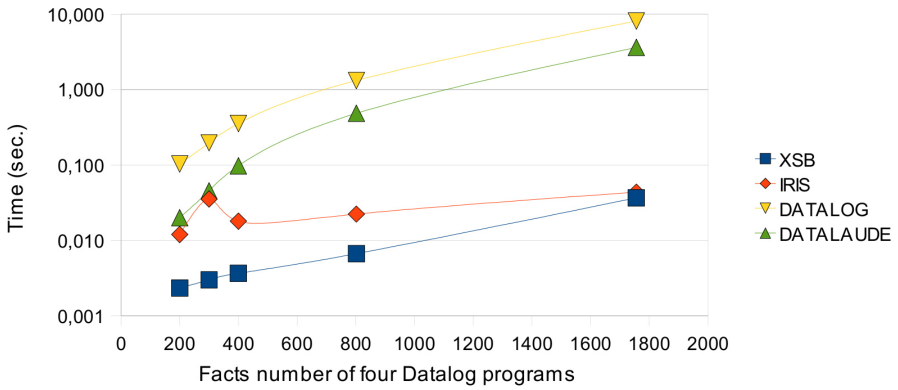

Fig. 1. Average resolution times of four Datalog solvers (logarithmic time).

means that initialization operations, like loading and compilation, are not taken into account in the results. Although being slower than XSB or IRIS, from our figures we can conclude that MAUDE behaves similarly to optimized deductive database systems, like DATALOG 1.4, which is implemented in C. This validates that MAUDE is able to process, under a good transformation such as the equational implementation extended with memoization, a large number of equations extracted from real programs in the context of static program analysis. Our rewrite system could be even more enhanced with the incorporation of efficient BDD representation [22] of the input data.

6 Conclusion

In this work, we have defined and implemented an efficient transformation from definite DATALOG programs into MAUDE programs in the context of DATALOG-based static analysis. We have formalized and proved the correctness of the transformation, and compared the implementation to standard DATALOG solvers. We evaluated that MAUDE was able to process a sizable number of equations, that come from real-life problems, like those from the static analysis of programs.

As a future work, we plan to use a more compact representation of the facts, such as BDDs, in order to minimize the significant loading time and size of the manipulated term in the rewriting system. We also plan to explore the impact of more sophisticated optimization techniques like tail-recursion or memoization (at the logical level) and other specific DATALOG optimizations [23]. Our final goal is to explore the impact of using the metalevel capabilities of rewriting logic for the analysis of object-oriented programs that include metaprogramming features such as reflection.

References

1. Ullman, J.D.: Principles of Database and Knowledge-Base Systems, Vol. I and II, The New Technologies. Computer Science Press (1989)
2. Whaley, J., Avots, D., Carbin, M., Lam, M.: Using Datalog with Binary Decision Diagrams for Program Analysis. In: Proc. of 3rd Asian Symp. on Programming Lang. and Systems (APLAS'05). Vol. 3780 of LNCS, Springer-Verlag (2005) 97-118

3. Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases. Addison Wesley (1995)
4. Ceri, S., Gottlob, G., Tanca, L.: Logic Programming and Databases. Springer-Verlag (1990)
5. Meseguer, J.: Conditional Rewriting Logic as a Unified Model of Concurrency. Theoretical Computer Science **96**(1) (1992) 73–155
6. Clavel, M., Durán, F., Ejer, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: All About Maude – A High-Performance Logical Framework. Vol. 4350 of LNCS. Springer-Verlag (2007)
7. Livshits, B., Whaley, J., Lam, M.: Reflection Analysis for Java. In: Proc. of the 3rd Asian Symp. on Programming Lang. and Systems (APLAS'05). (2005) 139–160
8. Leeuwen, J., ed.: Formal Models and Semantics. Volume B. Elsevier, The MIT Press (1990)
9. Bancilhon, F., Maier, D., Sagiv, Y., Ullman, J.D.: Magic Sets and Other Strange Ways to Implement Logic Programs. In: Proc. of the 5th ACM SIGACT-SIGMOD Symp. on Principles of Database Systems (PODS'86), ACM Press (1986) 1–15
10. Vieille, L.: Recursive Axioms in Deductive Databases: The Query/Subquery Approach. In: Proc. of the 1st Int'l Conf. on Expert Database Systems (EDS'86). (1986) 253–267
11. Sagonas, K.F., Swift, T., Warren, D.S.: XSB as an Efficient Deductive Database Engine. In: Proc. of the 1994 ACM SIGMOD Int'l Conf. on Management of Data, ACM Press (1994) 442–453
12. Emden, M., Lloyd, J.: A logical reconstruction of Prolog II. Journal on Logic Programming **1** (1984)
13. Marchiori, M.: Logic Programs as Term Rewriting Systems. In: Proc. of the 4th Int'l Conf. on Algebraic and Logic Programming (ALP'94. Vol. 850 of LNCS., Springer-Verlag (1994) 223– 241
14. Schneider-Kamp, P., Giesl, J., Serebrenik, A., Thiemann, R.: Automated Termination Analysis for Logic Programs by Term Rewriting. In: Proc. of the 16th Int'l Symp. on Logic-Based Program Synthesis and Transformation (LOPSTR'06). Vol. 4407 of LNCS., Springer-Verlag (2007) 177–193
15. Reddy, U.: Transformation of Logic Programs into Functional Programs. In: Proc. of the Symp. on Logic Programming (SLP'84), IEEE Computer Society Press (1984) 187–197
16. Alpuente, M., Feliú, M., Joubert, C., Villanueva, A.: Using Datalog and Boolean Equation Systems for Program Analysis. In: Proc. of the 13th Int'l Workshop on Formal Methods for Industrial Critical Systems (FMICS'08). Vol. 5596 of LNCS., Springer-Verlag (2009) 215–231
17. Andersen, H.R.: Model checking and boolean graphs. Theoretical Computer Science **126**(1) (1994) 3–30
18. Hill, P.M., Lloyd, J.W.: Analysis of Meta-Programs. In: Proc. of the First Int'l Workshop on Meta-Programming in Logic (META'88). (1988) 23–51
19. Marchiori, M.: Unravelings and ultra-properties. In: Proc. of the 5th Int'l Conf. on Algebraic and Logic Programming (ALP'96). Vol. 1039 of LNCS., Springer-Verlag (1996) 107–121
20. Alpuente, M., Feliú, M., Joubert, C., Villanueva, A.: Implementing Datalog in Maude. In: Proc. of the IX Jornadas sobre Programación y Lenguajes (PROLE'09) and I Taller de Programación Funcional (TPF'09). (September 2009) To appear.
21. Alpuente, M., Feliú, M., Joubert, C., Villanueva, A.: Defining Datalog in Rewriting Logic. Technical Report DSIC, Universidad Politécnica de Valencia. `http://www.dsic.upv.es/~villanue/pub/AFJV09-techrep.pdf`

22. Zantema, H., Pol, J.: A rewriting approach to binary decision diagrams. Journal of Logic and Algebraic Programming **49** (2001) 61–86
23. Liu, Y., Stoller, S.: From Datalog Rules to Efficient Programs with Time and Space Guarantees. ACM Transactions on Programming Languages and Systems (2009) To appear.