

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. In recent work, the effectiveness of using declarative languages has been demonstrated for many 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 power of the Rewriting Logic language MAUDE, we aim at transforming DATALOG programs into efficient rewrite systems that compute the same answers. A prototype has been implemented and applied to some real-world DATALOG-based analyses. Experimental results show that the performance of solving DATALOG queries in rewriting logic is comparable to state-of-the-art DATALOG solvers.

1 Introduction

DATALOG [1] is a simple relational query language that allows complex interprocedural program analyses involving dynamically created objects to be described in an intuitive way. 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 that encode 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], which is a very general *logical* and *semantical framework* that is efficiently implemented in the high-level programming language MAUDE [6]. Our motivation for using Rewriting Logic is to overcome the difficulty of handling 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

^{*} 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.

method name strings. Unless reflective calls are interpreted during the computation, analysis tools run the danger of incorrectness and incompleteness, and we consider it a challenge to investigate the interaction of static analysis with metaprogramming frameworks [6]. An additional goal of this work is to determine whether MAUDE is able to process a sizable number of constraints that arise in real-life problems, like the static analysis of JAVA programs.

In the related literature, the solution for a DATALOG query is classically constructed following a bottom-up approach, thus the information in the query is not taken advantage of until the model has been built [8]. In contrast, the typical top-down, logic programming interpreter would produce the output by reasoning backwards from the query. Between these two extremes, there is a whole spectrum of evaluation strategies [4, 9, 10], but in this work, we have considered a top-down approach for developing our transformation, since it is closer to MAUDE's evaluation principle that is based on (non-deterministic) rewriting.

Logic and functional programming are both instances of rule-based, declarative programming; 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 [11–14]. 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 [14]. However, one distinguished feature of DATALOG programs that burdens 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 [13]. 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 [13] is not concerned with preserving the computed answers, but only the termination behavior. Moreover, [13] 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 that encode the analysis to perform, in this work, we present an equation-based transformation that leads to efficient MAUDE-programs.

In previous work [15], we developed a DATALOG query solving technique based on *Boolean Equation Systems* (BESS) [16]. Although the correspondence between answering a DATALOG query and solving a BES can be established naturally, the main limitation of this approach is the difficulty of combining indexed and linked data structures in order to schedule suitable optimizations that ensure that only useful combination of facts are simultaneously considered. In this paper, we work

at a higher level in the sense that we transform a high-level DATALOG program into another high-level MAUDE program. Our goal is to take advantage of the flexibility and versatility of MAUDE in order to achieve scalability and meta-programming capabilities without losing the declarative nature of specifying program analyses in DATALOG.

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 such transformation. Section 4 formalizes the general process and establishes 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. Missing proofs can be found in Appendix A.

2 A program analysis written as a DATALOG program

DATALOG is a relational language that uses 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 $V \cup C$, $V \subseteq \mathcal{V}$ and $C \subseteq \mathcal{C}$, has the following syntax:*

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

where $m \geq 0$, and each p_i is a predicate symbol of arity n_i with arguments $a_{i,j} \in V \cup C$ ($j \in [1..n_i]$), where p_0 is not arithmetic.

The atom $p_0(a_{0,1}, \dots, a_{0,n_0})$ in the left-hand side (*lhs*) of the clause is the clause's *head*. The finite conjunction of *subgoals* in the right-hand side (*rhs*) 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 \square 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 in the 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 $\mathcal{D} \equiv q \Rightarrow_{SLD}^{\theta_1} q_1 \Rightarrow_{SLD}^{\theta_2} \dots \Rightarrow_{SLD}^{\theta_n} \square$, the answer computed by \mathcal{D} is $\theta_1\theta_2 \dots \theta_n$ restricted to the variables occurring in q .

¹ In the rest of the paper, DATALOG programs are considered to be as defined here.

Let us now introduce the running DATALOG program example that we use throughout 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 $\text{vP}/2$ representing the fact that a program variable points directly (via $\text{vP0}/2$) or indirectly (via $\text{a}/2$) 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 :

$$\begin{aligned} \text{vP}(\text{Var}, \text{Heap}) & \text{ :- } \text{vP0}(\text{Var}, \text{Heap}) . \\ \text{vP}(\text{Var1}, \text{Heap}) & \text{ :- } \text{a}(\text{Var1}, \text{Var2}), \text{vP}(\text{Var2}, \text{Heap}) . \end{aligned}$$

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

$$\begin{array}{ll} \text{a}(\text{v1}, \text{v2}) . & \text{vP0}(\text{v2}, \text{h5}) . \\ \text{a}(\text{v1}, \text{v3}) . & \text{vP0}(\text{v3}, \text{h4}) . \end{array}$$

In the considered DATALOG analysis program, a query typically consists in computing the objects in the heap pointed by a specific variable, for example v1 . We write such a query as $?- \text{vP}(\text{v1}, \text{Heap}) .$ 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 $\{\{\text{Heap}/\text{h4}\}, \{\text{Heap}/\text{h5}\}\}$. Another possible query is $?- \text{vP}(\text{Var}, \text{h5}) .$, where h5 stands for a heap object. The solver should determine which variables in the analyzed program can point to the object h5 .

Similarly to [13], 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, 17].

3 From DATALOG to MAUDE

As explained above, we are interested in computing all answers for a given query by term rewriting. A naïve approach is to translate DATALOG clauses into MAUDE rules, 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 with a huge number of facts, this approach results in poor performance [18]. This is because *rules* are handled non-deterministically in MAUDE whereas *equations* are applied deterministically [6].

² Intuitively, `search` $t \rightarrow t'$ 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 our equation-based transformation by means of the running example.

3.1 Answer 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 the variables and constants of the original DATALOG program in MAUDE, 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* (**Qids**), 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 **Terms**. This is used to ensure that the ground representation in MAUDE for existentially quantified variables that appear in the body of DATALOG clauses is unique to the whole MAUDE program.

With 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 Cte Cte1 Cte2 : Constant . var NEC : NonEmptyConstraint .
var V : Variable .
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 we implement 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 the subsort relations in the above fragment of MAUDE code. The operator `;-` represents the disjunction of constraints. The properties of associativity, commutativity and identity

element 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 throughout 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 an `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 a single equation that represents 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:

$$\text{eq } vP(T1, T2) = vPc1(T1, T2) ; vPc2(T1, T2) .$$

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

$$\text{eq } vPc1(T1, T2) = vP0(T1, T2) .$$

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

$$\begin{aligned} \text{eq } vPc2(T1, T2) &= vPc2s2(a(T1, v(T1, T2)), T1 T2) . \\ \text{eq } vPc2s2(((v(T1, T2) = Cte) , C) ; CS, T1 T2) &= \\ &\quad (vP(Cte, T1 T2) \times ((v(T1, T2) = Cte) , C)) ; vPc2s2(CS, T1 T2) . \\ \text{eq } vPc2s2(F, T1 T2) &= F . \end{aligned}$$

As can be observed, $vPc2$ calls to $vPc2s2$, whose first argument represents the execution of the first subgoal and the second argument 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 the computation of the (partial) answers resulting from the resolution of $a(T1, v(T1, T2))$ first in order to proceed. The use of the term $v(T1, T2)$, which represents the existentially quantified variable Var2 of the original DATALOG program, in the pattern 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 above 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:

$$\text{reduce } vP(v('variable), v('heap)) .$$

The answers to this query are shown below. The first sentence specifies the term that 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, which is written in several lines for the sake of readability, shows the result of the reduction together with its sort.

³ The c in $vPc1$ and $vPc2$ stands for *clause*.

```

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 expected, four answers were returned: the first two were obtained by the function `vPc1`, whereas the other two were 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 that delivers the same answers. The correctness and completeness of the transformation is given in Section 4.2.

4.1 The transformation

Let P be a DATALOG program defining predicate symbols $p_1 \dots p_n$. 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 $1 \leq i \leq n$ 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 Ti,1 ... Ti,ari : Term .
eq pi(Ti,1, ... ,Ti,ari) = (Ti,1 = ci,1,1, ... , Ti,ari = ci,1,ari) ; ...
; (Ti,1 = ci,|pi|,1, ... , Ti,ari = ci,|pi|,ari) .

```

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 $1 \leq i \leq m$ with non empty body, we have the following piece of code:

```

var Ti,1 ... Ti,ari : Term .
eq pi(Ti,1, ... ,Ti,ari) = pi,1(Ti,1, ... ,Ti,ari) ; ...
; pi,|pi|(Ti,1, ... ,Ti,ari) .

```

Each call $p_{i,j}$ with $1 \leq j \leq |p_i|$ 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 $\tau_{i,j,s,k}^a$ 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 of just one subgoal in the body. We define the function $\tau_{i,j,s}^p$, which returns the predicate symbol that appears in the s -th subgoal of the j -th clause that defines the i -th predicate in the DATALOG program. For each clause having just one subgoal, we get the following transformation:

$$\text{eq } p_{i,j}(\tau_{i,j,0,1}^a, \dots, \tau_{i,j,0,ar_i}^a) = \tau_{i,j,1}^p(\tau_{i,j,1,1}^a, \dots, \tau_{i,j,1,ar_l}^a) \ .$$

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. Specifically, 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 notions that are used in our transformation.

Definition 2 (linked variable). *A variable is called linked variable if and only if (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 $\text{linked}(C)$ is the function that returns the list of pairs containing a linked variable in the first component, and the list of positions where such a variable occurs in the body of the clause in the second component⁴.*

Example 1. For example, given the DATALOG clause

$$C = p(X1, X2) \text{ :- } p1(X1, X3), p2(X3, X4), p3(X4, X2) \ .$$

we have that $\text{linked}(C) = [(X3, [1.2, 2.1]), (X4, [2.2, 3.1])]$

Now we define the notion of *relevant* linked variables for a given subgoal, namely the linked variables of a subgoal that also appear in a 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:*

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

Note that, similarly to [13], we are not marking the input/output positions of predicates, as required in more traditional transformations. We are just identifying the variables whose values must be propagated in order to evaluate 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

⁴ Positions extend to goals in the natural way.

undertakes the execution of the first subgoal. We have as many auxiliary functions as subgoals in the original clause. Also, in the *rhs* of the auxiliary functions, the execution order of the successive subgoals is implicitly controlled by passing the results of each subgoal as a parameter to the subsequent function call.

Let the function $p_{i,j}$ generate the solutions calculated by the j -th clause of the predicate symbol p_i . We state that $ps_{i,j,s}$ 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 $X_1 \dots X_N$ of each equation are calculated by the function $\text{relevant}(s, \text{linked}(\text{clause}(i,j)))$ ⁵ and transformed into the corresponding MAUDE terms.

The equation for $p_{i,j}$ below reduces the considered DATALOG predicate to a call to the first auxiliary function that calculates the (partial) answers for the second subgoal by first computing the answers from the first subgoal $\tau_{i,j,1}^p$ 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.

$$\text{eq } p_{i,j}(\tau_{i,j,0,1}^a, \dots, \tau_{i,j,0,ar_i}^a) = ps_{i,j,2}(\tau_{i,j,1}^p(\tau_{i,j,1,1}^a, \dots, \tau_{i,j,1,r}^a), \tau_{i,j,0,1}^a \dots \tau_{i,j,0,ar_i}^a) .$$

In the equation, r is the arity of the predicate $\tau_{i,j,1}^p$. Then, for each auxiliary (unraveling) function, we declare as many constants as there are relevant variables in the corresponding subgoal. The left hand side of the equation for this auxiliary function 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, which is 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 $\tau_{i,j,s}^p$). Note that we propagate the instantiation of the relevant variables by means of a substitution.

```

var C1 ... CN : Constant .
var NECS : NonEmptyConstraintSet .
eq psi,j,s(NECS, T1...Tari) = psi,j,s+1(ps'_{i,j,s}(NECS, T1...Tari), T1...Tari) .
eq psi,j,s(F , LL) = F .

eq ps'_{i,j,s}(((X1=C1, ..., XN=CN, C) ; CS), T1...Tari) =
  ((τi,j,sp(τi,j,s,1v, ..., τi,j,s,rv)[X1\C1, ..., XN\CN] x (X1=C1, ..., XN=CN, C)) ;
  ps'_{i,j,s}(CS, T1...Tari) .
eq ps'_{i,j,s}((T ; CS), T1...Tari) =
  τi,j,sp(τi,j,s,1v, ..., τi,j,s,rv) ; ps'_{i,j,s}(CS, T1...Tari) .
eq ps'_{i,j,s}(F , LL) = F .

```

The equation for the last subgoal in the clause is slightly different, since we do not need to recursively invoke the auxiliary equation $ps'_{i,j,s}$. Assuming that g denotes the number of subgoals in a clause, we define

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

eq $\text{ps}_{i,j,g}(((X_1=C_1, \dots, X_N=C_N, C) ; \text{CS}) , T_1 \dots T_{ar_i}) =$
 $((\tau_{i,j,g}^p(\tau_{i,j,g,1}^v, \dots, \tau_{i,j,g,r}^v) [X_1 \setminus C_1, \dots, X_N \setminus C_N]) \times (X_1=C_1, \dots, X_N=C_N, C)) ;$
 $\text{ps}_{i,j,g}(\text{CS} , T_1 \dots T_{ar_i}) .$
 eq $\text{ps}_{i,j,g}((T ; \text{CS}) , T_1 \dots T_{ar_i}) =$
 $\tau_{i,j,g}^p(\tau_{i,j,g,1}^v, \dots, \tau_{i,j,g,r}^v) ; \text{ps}_{i,j,g}(\text{CS} , T_1 \dots T_{ar_i}) .$
 eq $\text{ps}_{i,j,g}(\text{F} , \text{LL}) = \text{F} .$

Finally, we define the transformation for the DATALOG query $q(X_1, \dots, X_n)$ (where $X_i, 1 \leq i \leq n$ are DATALOG variables or constants) as the MAUDE code $q(\tau_1^q, \dots, \tau_n^q)$, where $\tau_i^q, 1 \leq i \leq n$ is the transformation of the corresponding X_i .

4.2 Correctness of the transformation

We have defined a transformation from DATALOG programs 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. the observable of computed answers.

We first introduce some notation. Let **CS** be a **ConstraintSet** of the form $C_1 ; C_2 ; \dots ; C_n$ where each $C_i, i \geq 1$ is a **Constraint** in normal form ($C_i = \text{Cte}_1, \dots, C_m = \text{Cte}_m$), and let V be a list of variables. We write $C_i|_V$ to the restriction of the constraint C_i to the variables in V . We extend the notion to sets of constraints in the natural way, and denote it as $\text{CS}|_V$. 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, $\text{var}(t)$ 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 an input term t , we say that the answer constraint set computed by $t \rightarrow_S^* \text{CS}$ is $\text{CS}|_{\text{var}(t)}$.*

There is a natural isomorphism between the equational constraint C and an idempotent substitution $\theta = \{X_1/C_1, X_2/C_2, \dots, X_n/C_n\}$, which is given by the following: C is equivalent to θ iff $(C \Leftrightarrow \hat{\theta})$, 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 $\Theta \equiv \text{CS}$ iff $\text{CS} \Leftrightarrow \bigvee_{i=1}^n \hat{\theta}_i$

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 Appendix A.

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

5 Experimental results

This section reports on the performance of our prototype, called `DATALAUDE`⁶, implementing the transformation. First, we compare the efficiency of our implementation with respect to a naïve transformation to rewriting logic documented in [18]; then, we evaluate the performance of our prototype by comparing it to three state-of-the-art `DATALOG` solvers. All the 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. a previous rewriting-based implementation

We implemented several transformations from `DATALOG` programs to `MAUDE` programs before developing the one presented in this paper [18]. The first attempt consisted of a one-to-one mapping from `DATALOG` rules into `MAUDE` conditional rules. Then, in order to get rid of all the non-determinism caused by conditional equations and rules in `MAUDE`, we restricted our transformation to produce only unconditional equations as defined in the previous section. 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 doesn't reduce it but it just replaces it with its normal form, saving a great number 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+memoization
100	100	144	6.00	0.67	0.02
150	150	222	20.59	2.23	0.04
200	200	297	48.48	6.11	0.10
403	399	602	382.16	77.33	0.47
807	1669	2042	4715.77	1098.64	3.52

⁶ <http://users.dsic.upv.es/users/elp/dataalaude>

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 the backtracking associated to the non-deterministic evaluation penalizes the naïve version. It can also be observed that using memoization allows 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 program analysis purposes better, and provides a versatile and competitive DATALOG solver as 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 3.2⁷, DATALOG 1.4⁸, and IRIS 0.58⁹. Average resolution times of three runs for each solver are shown in Figure 1.

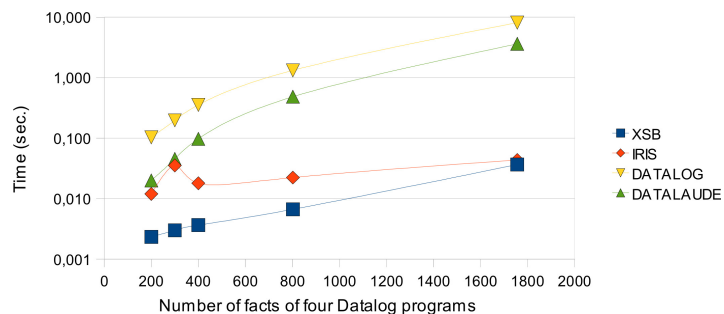


Fig. 1. Average resolution times of four DATALOG solvers (logarithmic time).

In order to evaluate the performance of our implementation with respect to the other DATALOG solvers, only resolution times are presented in Figure 1 since the compared implementations are quite different in nature. This means that initialization operations, like loading and compilation, are not taken into account in the results. Our experiments conclude that DATALAUE performs similarly to optimized deductive database systems like DATALOG 1.4, which is implemented in C, although it is slower than XSB or IRIS. Therefore, under a suitable transformation scheme such as the equational implementation extended with memoization, MAUDE is able to process a large number of equations extracted from statically analyzed, real JAVA programs. Our rewriting logic implementation might be further improved by using a BDD-based representation [20], which we plan to address

⁷ <http://xsb.sourceforge.net>

⁸ <http://datalog.sourceforge.net>

⁹ <http://iris-reasoner.sourceforge.net>

as future work. However, our purpose is not to produce the faster DATALOG solver ever, but to provide a tool that supports sophisticated analyses with reasonable performance in a clean way.

5.3 Analyzing Java programs with reflection

Addressing reflection is considered a difficult problem in the static analysis of JAVA programs, which is generally handled in an unsound or ad-hoc manner [7]. Reflection in JAVA is a powerful technique that is used when a program needs to examine or modify the runtime behavior of applications running in the JAVA virtual machine. For example, by using reflection, it is possible to write to object fields and invoke methods that are not known at compile time. JAVA provides a set of methods to handle reflection. These methods are found in `java.lang.reflect`.

In Figure 2 we show a simple example. We define a class `PO` with two fields: `c1` and `c2`. In the `Main` class, an object `u` of class `PO` is created by using the constructor method `new`, which assigns the empty string to the two fields of `u`. Then, `r` is defined as a field of a class, specifically, as the field `c1` of an object of class `PO` since `v` stores the value `"c1"`. The sentence `r.set(u, w)` states that `r` is the field object `c1` of `u`, and its value is that of `w`, i.e., `"c2"`. Finally, the last instruction sets the new value of `v` to the value of `u.c1`, i.e., `"c2"`.

<pre>class PO { PO (String c1, String c2) { this.c1 = c1; this.c2 = c2; } public String c1; public String c2; }</pre>	<pre>public class Main { public static void main(String[] args) { PO u = new PO("", ""); String v = "c1"; String w = "c2"; java.lang.reflect.Field r = PO.class.getField(v); r.set(u, w); v = u.c1; } }</pre>
---	---

Fig. 2. JAVA reflection example

A pointer flow-insensitive analysis of this program would tell us that `r` may point not only to the field object `u.c1`, but also `u.c2` since `v` in the argument of the reflective method `getField` may be assigned both to string `"c1"` and `"c2"`.

The key point for the reflective analysis is the fact that we don't have all the basic information for the points-to analysis at the beginning of the computation. In fact, the variables that occur in the methods handling reflection may generate new basic information. A sound proposal for handling JAVA reflection is proposed in [7], which is essentially achieved by first annotating the DATALOG program so that it is subsequently transformed by means of an external (to DATALOG) engine. As in [7], we assume we know the name of the methods and objects that may be used in the invocations. In our approach, we use the MAUDE reflection capability to automatically generate the rules that represent new deduced information without resorting to any ad-hoc notation or external artifact.

Let's start by showing which pointer-analysis information JOEQ would extract from our example. We enforce the fact that we work at the *bytecode level*, so some

JAVA instructions are converted into more than one bytecode instructions and some new auxiliary variables —in the example \$0— are introduced.

Java Code	Extracted Information
<code>P0 u = new P0("", "");</code>	<code>vP0(u,0).</code> <code>vT(u,P0).</code>
<code>String v = "c1";</code>	<code>vP0(v,12).</code> <code>vT(v,string).</code>
<code>String w = "c2";</code>	<code>vP0(w,15).</code> <code>vT(w,string).</code>
<code>java.lang.reflect.Field r = P0.class.getField(v);</code>	<code>vP0(\$0,18).</code> <code>vT(\$0,ClassP0).</code> <code>vT(r,field).</code> <code>mI(main,21,getField).</code> <code>iRet(21,r).</code> <code>actual(21,0,\$0).</code> <code>actual(21,1,v).</code>
<code>r.set(u, w);</code>	<code>mI(main,30,set).</code> <code>actual(30,0,r).</code> <code>actual(30,1,u).</code> <code>actual(30,2,w).</code>
<code>v = u.c1;</code>	<code>l(u,c1,v).</code>

All these predicates state properties or actions performed to references and heap objects.

`vP0(V,H)` States that there is a creation of a new object `H` —where `H` is the position of the call to the object’s constructor in the code— and that is referenced by the variable `V`.

`vT(V,T)` States that the declared type of variable `V` is `T`.

`hT(H,T)` States that the object `H` has type `T`.

`actual(I,N,V)` States that the variable `V` is used as the actual parameter number `N` at the *invocation point*¹⁰ `I`.

`mI(M,I,N)` States that at invocation point `I` of method `M` there is a method call to be resolved with the name `N`.

`iRet(I,V)` States that variable `V` will receive the return value of the invocation at point `I`.

`l(V1,F,V2)` States that the value of the field `F` of variable `V1` is assigned to variable `V2`.

`s(V1,F,V2)` States that the value of variable `V2` is assigned to the field `F` of variable `V1`.

With this kind of information, it is easy to specify a non-reflective pointer analysis by means of DATALOG clauses as in [2]. The analysis would mimic any possible flow

¹⁰ An *invocation point* is either a method call, a static call or a special call at the bytecode level.

of pointers in the code. Nevertheless, the analysis would be missing some hidden flow of pointers related to the use of reflection. Following the code execution with the semantics of the reflection API of JAVA in mind, v is the name of the field represented by the reflective object r . Then, the instruction $r.set(u,w)$ stores the value of w in the field $c1$ (represented by r) of the object pointed by u , and this would be resumed in the DATALOG fact $s(u,c1,w)$. However, this behaviour is *dynamic* because it depends on the runtime values of the variable v , and so we have no way of knowing what objects v can point to at compile time. For example, if v points to the string "c1", as it does in the example, a new *reflective* object which represents a "c1" field of objects of class $P0$ would be created and assigned to the variable r . Any call to the method `set` on the previous object would store on the field "c1" of the first parameter the content of the second parameter. Because v could potentially point to many other strings representing fields, r could point to many reflective objects representing correspondent fields, and so calls to method `set` on r could mean many different kinds of stores $s(V1,F,V2)$.

The reflective analysis as proposed by [7] uses additional information (extracted by JOEQ) regarding which calls are done to the reflective API. This enriches the analysis allowing us to deduce new "on-the-fly" (at analysis time) facts that in the basic, non-reflective analysis were considered static information. For example, store facts $s(V1,F,V2)$ can also be deduced by the clause:

$$s(V1,F,V2) :- iE(I,'Field.set') , actual(I,0,V) , vP(V,H) , \\ fieldObject(H,F) , actual(I,1,V1) , actual(I,2,V2) .$$

Let us present the new predicates that appear in this rule:

$iE(I,M)$ States that there is a call to the *resolved method*¹¹ M at the invocation point I . This predicate represents an approximation to the program's *call-graph*.

$fieldObject(H,F)$ States that the object H is a reflective object representing the field F .

These predicates are also derived from other facts. The meaning of the clause is straightforward: we state that $V2$ is stored in the field F of $V1$ if there is call to `Field.set` over a reflective object representing field F ($fieldObject(H,F)$) and the first and second parameters of that call are respectively $V1$ and $V2$.

In our reflective setting, we have followed the direct approach of translating DATALOG clauses into MAUDE rules as in [18] in order to ease the manipulation of modules at the metalevel. In this approach, each DATALOG clause is translated into a MAUDE conditional rule. Therefore, checking that the clause body is satisfiable equals to checking if the condition of that rule holds. Following this idea, facts are translated into non-conditional rules in one-to-one correspondence. Consequently, deducing information equals to rewriting queries into assignments to its arguments. The rule above is translated into MAUDE as:

$$cr1 \ s(V1,F,V2) \Rightarrow V1 \rightarrow CteV1 , F \rightarrow CteF , V2 \rightarrow CteV2$$

¹¹ A *resolved method* means specific code from a certain class.

```

if iE(I,'Field.set) => I -> CteI , 'Field.set -> 'Field.set
  /\ isConsistent I -> CteI
/\ actual(CteI,'0,V) => CteI -> CteI , '0 -> '0 , V -> CteV
  /\ isConsistent V -> CteV
/\ vP(CteV,H) => CteV -> CteV , H -> CteH
  /\ isConsistent H -> CteH
/\ fieldObject(CteH,F) => CteH -> CteH , F -> CteF
  /\ isConsistent F -> CteF
/\ actual(CteI,'1,V1) => CteI -> CteI , '1 -> '1 , V1 -> CteV1
  /\ isConsistent V1 -> CteV1
/\ actual(CteI,'2,V2) => CteI -> CteI , '2 -> '2 , V2 -> CteV2
  /\ isConsistent V2 -> CteV2 .

```

With this transformation, it can be seen that the structure of the resulting MAUDE code is very close to the original DATALOG program. The novelty in the reflective analysis is on the need of new information to support the analysis, such as identifiers of reflective methods and string constants representing names of reflective objects. In our proof-of-concept prototype, we have considered field-reflection analysis. This implies that JOEQ must recover facts for the following two predicates:

stringToField(H,F) States that object H is a string representation of field F.
getField(M) States that method M is a reflective method which returns a reflective object representing a field.

Adding these extra information to the basic, non-reflective analysis we can deduce new reflective information which enriches the basic analysis. Then, the enriched basic analysis allows us to deduce new reflective information starting an iterative process until a fixpoint is reached.

Rewriting logic is reflective in a precise mathematical way: there is a finitely presented rewrite theory \mathcal{U} that is universal in the sense that we can represent (as data) any finitely presented rewrite theory \mathcal{R} in \mathcal{U} (including \mathcal{U} itself), and then mimic the behavior of \mathcal{R} in \mathcal{U} . The fact that rewriting logic is a reflective logic and the fact that MAUDE effectively supports reflective rewriting logic computation make reflective design (in which theories become data at the metalevel) ideally suited for manipulation tasks in MAUDE.

MAUDE's reflection is systematically exploited in our tool. On one hand, we can easily define new rules to be included in the specification by manipulating term meta-representations of rules and modules. On the other hand, by virtue of our reflective design, our metatheory of program analysis (which includes a common fixpoint infrastructure) is made accessible to the user who writes a particular analysis in a clear and principled way.

We have endowed our prototype implementation with the capability to carrying on reflection analysis for JAVA. The extension essentially consists of a module at the MAUDE meta-level that implements a generic infrastructure to deal with reflection. Figure 3 shows the structure of a typical reflection analysis to be run in our tool.

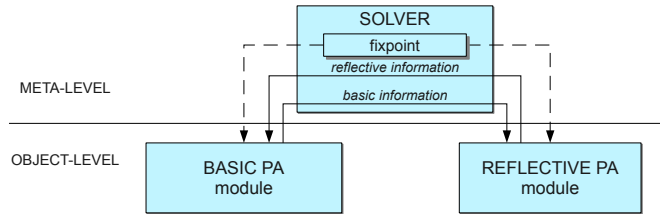


Fig. 3. The structure of the reflective analysis.

The static analysis is specified in two object-level modules, a *basic module* and a *reflective module*, that can be written in either DATALOG or MAUDE, since DATALOG analyses are automatically compiled into MAUDE code. The *basic program analysis (PA)* module contains the rules for the classical analysis (that neglects reflection) whereas the *reflective program analysis* module contains the part of the analysis dealing with the reflective components of the considered JAVA program. For example, the rule representing the reflective clause $s(V1, F, V2)$ would be included in the reflective program analysis module.

The module called *solver* deals with the program analysis modules at the meta-level. It consists of a generic fixpoint algorithm that feeds the reflective module with the information that can be inferred by the basic analysis and vice versa. Our implementation of the fixpoint is the following:

```

op fixpoint : Module Module -> Module .

var M1 M2 M3 : Module .

ceq fixpoint(M1,M2) = fixpoint(M3,M1)
  if M3 := closure(M1,M2)
  /\ M3 /= M2 .
eq fixpoint(M1,M2) = closure(M1,M2) [owise] .

```

The `closure` function infers all the information from the module given as its first parameter and adds it to the module given as its second parameter, returning the modified module. In order to do that, `closure` queries the first module, translates the solutions into rules, and finally adds them to the second module.

For the points-to analysis with field reflection, the reflective and basic modules contain 11 rules each, whereas the generic solver is written in just 50 rules (including those that generate rules from the new computed information). The fact of separating the specification of the analysis into several modules enhances its comprehension and allows us to easily compose analysis on demand.

6 Conclusions

In this work, we have defined and implemented a powerful 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 we have compared our implementation to standard DATALOG solvers. We confirmed that MAUDE is able to process a sizable number of constraints that arise in real-life problems, like the static analysis of JAVA programs. By taking advantage of MAUDE's reflective design, we have also demonstrated how it is possible to perform efficient reflection analyses of JAVA programs in MAUDE that are, at one time, declarative, accurate, and sound.

As future work, we plan to use a more compact, BDD-based representation of program rules in order to minimize both loading time and memory consumption. We also plan to explore the impact of adapting to our context more sophisticated DATALOG optimizations [21].

References

1. Ullman, J.D.: Principles of Database and Knowledge-Base Systems, Volume I and II, The New Technologies. Computer Science Press (1989)
2. Whaley, J., Avots, D., Carbin, M., Lam, M.S.: Using Datalog with Binary Decision Diagrams for Program Analysis. In: Proceedings of the Third Asian Symposium on Programming Languages and Systems (APLAS'05). Volume 3780 of Lecture Notes in Computer Science., 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. Volume 4350 of Lecture Notes in Computer Science. Springer-Verlag (2007)
7. Livshits, B., Whaley, J., Lam, M.: Reflection Analysis for Java. In: Proceedings of the Third Asian Symposium on Programming Languages 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: Proceedings of the 5th ACM SIGACT-SIGMOD Symposium 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: Proceedings of the 1st International Conference on Expert Database Systems (EDS'86). (1986) 253–267
11. Emden, M., Lloyd, J.: A logical reconstruction of Prolog II. Journal on Logic Programming **1** (1984)
12. Marchiori, M.: Logic Programs as Term Rewriting Systems. In: Proceedings of the 4th International Conference on Algebraic and Logic Programming (ALP'94). Volume 850 of Lecture Notes In Computer Science., Springer-Verlag (1994) 223–241

13. Schneider-Kamp, P., Giesl, J., Serebrenik, A., Thiemann, R.: Automated Termination Analysis for Logic Programs by Term Rewriting. In: Proceedings of the 16th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR'06). Volume 4407 of Lecture Notes in Computer Science., Springer-Verlag (2007) 177–193
14. Reddy, U.: Transformation of Logic Programs into Functional Programs. In: Proceedings of the Symposium on Logic Programming (SLP'84), IEEE Computer Society Press (1984) 187–197
15. Alpuente, M., Feliú, M., Joubert, C., Villanueva, A.: Using Datalog and Boolean Equation Systems for Program Analysis. In Cofer, D., Fantechi, A., eds.: Proceedings of the 13th International Workshop on Formal Methods for Industrial Critical Systems (FMICS'08). Volume 5596 of Lecture Notes in Computer Science., Springer-Verlag (2009) 215–231
16. Andersen, H.R.: Model checking and boolean graphs. *Theoretical Computer Science* **126**(1) (1994) 3–30
17. Hill, P.M., Lloyd, J.W.: Analysis of Meta-Programs. In: Proceedings of the First International Workshop on Meta-Programming in Logic (META'88). (1988) 23–51
18. Alpuente, M., Feliú, M., Joubert, C., Villanueva, A.: Implementing Datalog in Maude. In Peña, R., ed.: Proceedings of the IX Jornadas sobre Programación y Lenguajes (PROLE'09) and I Taller de Programación Funcional (TPF'09). (September 2009) 15–22
19. Marchiori, M.: Unravelings and ultra-properties. In Hanus, M., Rodríguez-Artalejo, M., eds.: Proceedings of the 5th International Conference on Algebraic and Logic Programming (ALP'96). Volume 1039 of Lecture Notes in Computer Science., Springer-Verlag (1996) 107–121
20. Zantema, H., Pol, J.: A rewriting approach to binary decision diagrams. *Journal of Logic and Algebraic Programming* **49** (2001) 61–86
21. 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.

A Proof

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 let $\mathcal{T}_g(q)$ be the corresponding, transformed input term. Let Θ be the set of computed answers of P for the query q , and let $\mathbf{CS}|_{\text{var}(\mathcal{T}_g(q))}$ be the answer constraint set computed by $\mathcal{T}_g(q) \rightarrow_{\mathcal{T}(P)}^* \mathbf{CS}$. Then, $\Theta \equiv \mathbf{CS}|_{\text{var}(\mathcal{T}_g(q))}$.

Proof of Theorem 1

(\Leftarrow) We proceed by induction on both the structure of the clauses and the length of the computations.

We should prove that if $\mathcal{T}_g(q) \rightarrow_{\mathcal{T}(P)}^! \mathbf{CS}$, then for every \mathbf{C} in the answer constraint set \mathbf{CS} , there exists a computed answer θ for q and P such that $\mathbf{C}|_{\text{var}(\mathcal{T}_g(q))} \equiv \theta$.

Let us first consider the case when q is defined only by facts.

By the definition of our transformation, when the predicate symbol (of arity m) of the query q is defined by facts¹², there exists an equation in $\mathcal{T}(P)$, whose left hand side is of the form $\mathbf{q}(\mathbf{T1}, \dots, \mathbf{Tm})$, that rewrites to an answer constraint set that contains as many answer constraints as facts define the predicate in the DATALOG program. Again by definition, each answer constraint corresponds to one (ground) fact in the DATALOG program instantiating each argument of the predicate to the appropriate constant.

In this case, the rewriting sequence for the initial term $\mathcal{T}_g(q)$ is

$$\mathcal{T}_g(q) \rightarrow_{\mathcal{T}(P)} \mathbf{C}_1; \dots; \mathbf{C}_n \rightarrow_{\mathcal{T}(P)}^! \mathbf{C}_v; \dots; \mathbf{C}_w$$

where n is the number of facts defining the DATALOG predicate and $v, \dots, w \in \{1, \dots, n\}$. Each answer constraint in $\mathbf{C}_1; \dots; \mathbf{C}_n$ comes up from one DATALOG fact. The second part of the sequence is the simplification for the union operator $;$. The simplification consists in removing duplicate elements and collapsing inconsistent constraints to \mathbf{F} . The inconsistent constraints appear when a single variable is equaled to two different values or when two different constants are equaled. This case may only occur when the query is partially (or totally) instantiated. In this case, all the answer constraints that are incompatible with the passed value are collapsed to \mathbf{F} . In the DATALOG setting, this corresponds with failing to unify the query with the facts generating these answers. It is easy to observe that the DATALOG resolution is able to compute each of these consistent solutions.

Now we consider the case when q is defined by n clauses with non-empty bodies. By definition of our transformation, the initial term rewrites as follows.

$$\mathcal{T}_g(q) \rightarrow_{\mathcal{T}(P)} \mathbf{q}_1(\mathbf{T1}, \dots, \mathbf{Tm}); \dots; \mathbf{q}_n(\mathbf{T1}, \dots, \mathbf{Tm})$$

Again by definition, each function q_i can be defined in our transformation in two different ways, depending on the number of subgoals in the clause represented by q_i .

¹² Remember that, in DATALOG, predicates are defined by facts or by clauses but not by both.

Let us consider the case of a clause having a single subgoal. Let the equation defining the function symbol q_i be

$$\text{eq } q_i(U_1, \dots, U_m) = p(V_1, \dots, V_z)$$

where U_1, \dots, U_m and V_1, \dots, V_z are the terms in the DATALOG clause. Therefore, many of them may coincide, and the set of variables in V_1, \dots, V_z subsumes the set of variables in U_1, \dots, U_m (we are considering *safe* DATALOG programs).

Hence, the rewriting sequence given by the equation shown above is as follows:

$$q_i(T_1, \dots, T_m) \rightarrow_{\mathcal{T}(P)} p(W_1, \dots, W_z)$$

Notice that p is a predicate symbol in the DATALOG program that is also transformed. By induction hypothesis, $p(W_1, \dots, W_z)$ rewrites to the set of its correct answer constraints $\mathcal{C}'_1; \dots; \mathcal{C}'_w \mid_{\text{var}(p(W_1, \dots, W_z))}$. Since we are considering safe DATALOG programs, we know that all the variables T_1, \dots, T_m occur in the arguments of the body subgoals and are thus in the set of variables $\{W_1, \dots, W_z\}$. Therefore, the correct answer constraint for the query is $\mathcal{C}'_1; \dots; \mathcal{C}'_w \mid_{\text{var}(q(T_1, \dots, T_m))}$

Let us now proceed with the general case when the clause body contains more than one subgoal. In this case, the rewriting sequence starts by rewriting to an auxiliary function s_2 which represents the execution of the second subgoal, after having reduced the first one (on the first argument of s_2). This is ensured by the operational semantics of MAUDE and the patterns in the definition of that auxiliary function (and of those of successive subgoals). The second part of the sequence below corresponds to the computation of the first subgoal:

$$q_i(T_1, \dots, T_m) \rightarrow_{\mathcal{T}(P)} s_2(p(W_1, \dots, W_z), T_1 \dots T_m) \rightarrow_{\mathcal{T}(P)}^* s_2(\mathcal{C}_1; \dots; \mathcal{C}_w, T_1 \dots T_m)$$

By induction hypothesis, the set $\mathcal{C}_1; \dots; \mathcal{C}_w$ contains correct answer constraints for $p(W_1, \dots, W_z)$. At this execution point, following the definition of our transformation there are two possibilities, depending on whether or not there are more subgoals (Case 2), and (Case 1), respectively. Let us assume that we are dealing with the i -th subgoal (function symbol s_i).

Case 1 In this case, the computation may proceed in two different ways:

1. There is no solution for $p(W_1, \dots, W_z)$; thus, the answer constraint set is \mathbf{F} . In this case, the rewriting sequence is:

$$s_i(\mathcal{C}_1; \dots; \mathcal{C}_w, T_1 \dots T_m) \rightarrow_{\mathcal{T}(P)} \mathbf{F}$$

Therefore, there exists no solution for the first subgoal and the computation of the query trivially fails, which corresponds with the DATALOG resolution.

2. Consider the case when there are w different answer constraints for $p(W_1, \dots, W_z)$. The rewriting sequence following the definition of our transformation is:

$$s_i(\mathcal{C}_1; \dots; \mathcal{C}_w, T_1 \dots T_m) \rightarrow_{\mathcal{T}(P)} s_{i+1}(s'_i(\mathcal{C}_1; \dots; \mathcal{C}_w, T_1 \dots T_m), T_1 \dots T_m)$$

Note that to compute the answer constraints for the third subgoal (s_{i+1}), we first have to rewrite the second one by reducing the redex s'_i that contains the partially accumulated answer constraint set. Depending on the form of this constraint set, we have three possible rewritings:

- (a) The first answer constraint (\mathcal{C}_1) is \mathbf{T} (which is an **EmptyConstraint**); thus, the computation of the previous subgoal (which is ground) performed no substitution of variables:

$$\begin{aligned}
& s_{i+1}(s'_i(\mathbf{T} \quad ; \quad \dots \quad ; \quad \mathbf{C}_w, T_1 \dots T_m), T_1 \dots T_m) \quad \rightarrow_{\mathcal{T}(P)} \\
& s_{i+1}(\mathbf{q}(Q_1, \dots, Q_z); (s'_i(\mathbf{C}_2 \quad ; \quad \dots \quad ; \quad \mathbf{C}_w, T_1 \dots T_m), T_1 \dots T_m) \quad \rightarrow_{\mathcal{T}}^* \\
& s_{i+1}(\mathbf{C}'_1; \dots; \mathbf{C}'_{w'}; (s'_i(\mathbf{C}_2 \quad ; \quad \dots \quad ; \quad \mathbf{C}_w, T_1 \dots T_m), T_1 \dots T_m)
\end{aligned}$$

By induction hypothesis, $\mathbf{C}'_1, \dots, \mathbf{C}'_{w'}$ are correct answer constraints for the i -th subgoal (whose function symbol is \mathbf{q} , given by the function τ^P of our transformation). Intuitively, this rewriting step represents the propagation of variable assignments to the following subgoals. The recursive call of s'_i propagates not only the information from the first answer constraint, but also to the information needed to proceed with the computation of the rest of solutions. We will come back to this point of the proof after introducing the rest of cases.

- (b) The first answer constraint is not \mathbf{T} , generating the following rewriting sequence:

$$\begin{aligned}
& s_{i+1}(s'_i(\mathbf{C}_1 \quad ; \quad \dots \quad ; \quad \mathbf{C}_w, T_1 \dots T_m), T_1 \dots T_m) \quad \rightarrow_{\mathcal{T}(P)} \\
& s_{i+1}(\mathbf{q}(Q_1, \dots, Q_z)[Q_j \setminus X_j, \dots, Q_k \setminus X_k] \times \mathbf{C}_1; \\
& (s'_i(\mathbf{C}_2 \quad ; \quad \dots \quad ; \quad \mathbf{C}_w, T_1 \dots T_m), T_1 \dots T_m) \quad \rightarrow_{\mathcal{T}(P)}^* \\
& s_{i+1}((\mathbf{C}'_1; \dots; \mathbf{C}'_{w'}) \times \mathbf{C}_1; (s'_i(\mathbf{C}_2 \quad ; \quad \dots \quad ; \quad \mathbf{C}_w, T_1 \dots T_m), T_1 \dots T_m)
\end{aligned}$$

Note that, by definition, the substitution $\mathbf{q}(Q_1, \dots, Q_z)[Q_j \setminus X_j, \dots, Q_k \setminus X_k]$ replaces each relevant variable of \mathbf{q} X_j by its computed value, captured in the pattern of the lhs of the corresponding transformation equation. The constraints for these values are also in the computed answer \mathbf{C}_1 . By induction hypothesis, $\mathbf{C}'_1, \dots, \mathbf{C}'_{w'}$ are correct answer constraints for the term $\mathbf{q}((Q_1, \dots, Q_z)[Q_j \setminus X_j, \dots, Q_k \setminus X_k])$. Then, the \times operator combines each solution of the second subgoal with the information in \mathbf{C}_1 . Since we have passed the shared information with the applied substitution before the subsequent reduction step, we know that the shared variables have the same value; thus, the new combined solutions are consistent for the conjunction of the two (or more) subgoals. Note that the only case when inconsistencies may arise (and be simplified) by the \times operator is when both sets of answers contain an output variable and each one computes a different value for it. This inconsistent case is reduced to false, so no inconsistent answer constraint is carried on.

- (c) There are no answer constraints to proceed; thus, the first argument is \mathbf{F} and the rewriting sequence is:

$$s_{i+1}(s'_i(\mathbf{F}, T_1 \dots T_m), T_1 \dots T_m) \rightarrow_{\mathcal{T}(P)} s_{i+1}(\mathbf{F}, T_1 \dots T_m)$$

This last case is the base case for the recursion appearing in the two previous ones. By induction on the number of elements in the answer constraint set $\mathbf{C}_1 \quad ; \quad \dots \quad ; \quad \mathbf{C}_w$, we can see that the subterm $(s'_i(\mathbf{C}_2 \quad ; \quad \dots \quad ; \quad \mathbf{C}_w, T_1 \dots T_m), T_1 \dots T_m)$ in the cases (a) and (b) is a smaller recursive call.

Hence, we are at the point in which we have computed all the accumulated answer constraints up to the i -th subgoal:

$$s_{i+1}(\mathbf{C}_1; \dots; \mathbf{C}_n, T_1 \dots T_m)$$

Case 2 In this case, s_i is the last subgoal, so no propagation of information is performed. Let us recall the term that had to be reduced

$$s_i(\mathbf{C}_1 ; \dots ; \mathbf{C}_w, T_1 \dots T_m)$$

Also in this case, there are three possible paths:

1. The first answer constraint (\mathbf{C}_1) is **T** (which is an **EmptyConstraint**), thus the computation of the previous subgoal (which is ground) performed no substitution of variables:

$$\begin{aligned} & s_i(\mathbf{T} ; \dots ; \mathbf{C}_w, T_1 \dots T_m) \\ & \rightarrow_{\mathcal{T}(P)} \mathbf{q}(Q_1, \dots, Q_n) ; s_i(\mathbf{C}_2 ; \dots ; \mathbf{C}_w, T_1 \dots T_m) \rightarrow_{\mathcal{T}(P)}^* \\ & \mathbf{C}'_1 ; \dots ; \mathbf{C}'_{w'} ; s_i(\mathbf{C}_2 ; \dots ; \mathbf{C}_w, T_1 \dots T_m) \end{aligned}$$

By induction hypothesis, $\mathbf{C}'_1 ; \dots ; \mathbf{C}'_{w'}$ are the correct answer constraints of $\mathbf{q}(Q_1, \dots, Q_n)$. For the recursive call, we will come back to this point of the proof after having introduced the rest of cases.

2. The first answer constraint is not **T**, generating the following rewriting sequence:

$$\begin{aligned} & s_i(\mathbf{C}_1 ; \dots ; \mathbf{C}_w, T_1 \dots T_m) \rightarrow_{\mathcal{T}(P)} (\mathbf{q}(Q_1, \dots, Q_z)[Q_j \setminus X_j, \dots, Q_k \setminus X_k]) \times \mathbf{C}_1 ; s_i(\mathbf{C}_2 ; \dots ; \mathbf{C}_w, T_1 \dots T_m) \rightarrow_{\mathcal{T}(P)}^* \\ & (\mathbf{C}'_1 ; \dots ; \mathbf{C}'_{w'}) \times \mathbf{C}_1 ; (s_i(\mathbf{C}_2 ; \dots ; \mathbf{C}_w, T_1 \dots T_m)) \end{aligned}$$

Similarly to Case (1.2.b) above, the X_j are the linked variables that have already been instantiated, and their value is propagated to the corresponding Q_j . The X_j variables are computed in \mathbf{C}_1 . By induction hypothesis, $\mathbf{C}'_1, \dots, \mathbf{C}'_{w'}$ are correct answer constraints for the term $\mathbf{q}(Q_1, \dots, Q_z)[Q_j \setminus X_j, \dots, Q_k \setminus X_k]$. Then, the \times operator combines each solution of the second subgoal with the information in \mathbf{C}_1 . Since we have passed the shared information with the substitution before reduction, we know that the shared variables have the same value; thus, no inconsistency comes up due to these. The only case when inconsistencies may arise (and be simplified) by the \times operator is when both sets of answers contain an output variable and each one computes a different value for it. This inconsistent case is reduced to false, so no inconsistent answer constraint is carried on.

As in the previous case, we will consider the recursive call after having presented the three cases.

3. There are no answer constraints to proceed; thus, the first argument is **F** and the rewriting sequence is:

$$s_i(\mathbf{F}, T_1 \dots T_m) \rightarrow_{\mathcal{T}(P)} \mathbf{F}$$

This last case is the base case for the recursion appearing in the two previous ones. By induction on the cardinality of the set of answer constraints $\mathbf{C}_1 ; \dots ; \mathbf{C}_w$, we can see that the subterm $(s_i(\mathbf{C}_2 ; \dots ; \mathbf{C}_w, T_1 \dots T_m), T_1 \dots T_m)$ is a smaller recursive call, thus at some point will reach the base case.

Hence, we are at the point in which we have computed all the accumulated answer constraints up to the last i -th subgoal:

$$\mathbf{C}_1 ; \dots ; \mathbf{C}_n$$

(\Rightarrow) We proceed by induction on both the structure of the clauses and the length of the computations.

We must prove that for each computed answer θ for q and P , then after the reduction $\mathcal{T}_g(q) \rightarrow_{\mathcal{T}(P)}^! \text{CS}$, there exists a \mathbf{C} in the answer constraint set CS such that $\mathbf{C}|_{\text{var}(\mathcal{T}_g(q))} \equiv \theta$.

Let us first consider the case when q is defined by facts. For each fact defining the predicate of the query in the DATALOG program, there are two cases:

1. It is possible to unify the query with the fact, getting a computed answer given by the substitution θ .
2. The query does not unify with the fact, so there is no computed answer for this execution branch.

The second case may occur only when some of the arguments of the query are instantiated and do not coincide with the corresponding arguments in the given fact.

By definition, our transformation generates an answer constraint for each fact. Assume that the query has the form $\mathbf{q}(A_1, \dots, A_m)$ where each A_i , $1 \leq i \leq m$ is a variable or a constant. Given a fact $\mathbf{q}(t_1, \dots, t_m)$, by definition in our transformation, there exists a \mathbf{C} in CS of the form, $\bigwedge_{1 \leq i \leq m} A_i = t_i$. If we are in the first case above, then clearly $\theta \equiv \mathbf{C}$ in normal form (i.e., after having simplified the constraints of the form $\mathbf{Cte} = \mathbf{Cte}$ when the argument in the query is instantiated). When we are in the second case above, then there exists an equality constraint $\mathbf{Cte} = \mathbf{Cte}'$ for two different constants; therefore, after normalization the answer constraint reduces to \mathbf{F} (correctness).

The rewriting sequence for the initial term $\mathcal{T}_g(q)$ is

$$\mathcal{T}_g(q) \rightarrow_{\mathcal{T}(P)} \mathbf{C}_1; \dots; \mathbf{C}_n \rightarrow_{\mathcal{T}(P)}^! \mathbf{C}_v; \dots; \mathbf{C}_w$$

where n is the number of facts defining the DATALOG predicate and $v, \dots, w \in \{1, \dots, n\}$. Each answer constraint in $\mathbf{C}_1; \dots; \mathbf{C}_n$ comes up from one DATALOG fact. The second part of the sequence is the simplification for the union operator.

Now we consider the case when q is defined by n clauses with non-empty bodies. We must ensure that each of these solutions is included in CS , the set of answer constraints. By definition of our transformation, the set of answer constraints for q is the disjunction of the sets of answer constraints generated for each clause. Let us consider the solutions computed by each clause independently.

We recall the first step of the initial MAUDE term rewriting sequence:

$$\mathcal{T}_g(q) \rightarrow_{\mathcal{T}(P)} q_1(T_1, \dots, T_m); \dots; q_n(T_1, \dots, T_m)$$

Next we prove that the solutions computed from the i -th clause are included in the set of answer constraints computed by the function $q_i(T_1, \dots, T_m)$. By definition, each function q_i can be defined in our transformation in two different ways, depending on the number of subgoals in the clause represented by q_i .

Let us consider the case of a clause having a single subgoal. Assume that the term on the *rhs* of that clause is a predicate call with predicate symbol \mathbf{p} and z arguments: $\mathbf{p}(V_1, \dots, V_z)$. By definition of our transformation, the equation for this clause is the following one, where \mathbf{p} is now a defined function symbol:

$$\text{eq } \mathbf{q}_i(U_1, \dots, U_m) = \mathbf{p}(V_1, \dots, V_z)$$

where U_1, \dots, U_m and V_1, \dots, V_z are the terms in the DATALOG clause. Therefore, many of them may coincide, and the set of variables in V_1, \dots, V_z subsumes the set of variables in U_1, \dots, U_m (we are considering *safe* DATALOG programs).

The rewriting sequence given by the equation shown above is as follows:

$$q_i(T_1, \dots, T_m) \rightarrow_{\mathcal{T}(P)} \mathbf{p}(W_1, \dots, W_z) \xrightarrow{!}_{\mathcal{T}(P)} \mathbf{C}'_1; \dots; \mathbf{C}'_w$$

By induction hypothesis, for each computed answer θ for the query $\mathbf{p}(W_1, \dots, W_z)$, there exists an answer constraint \mathbf{C}'_i , $1 \leq i \leq w$ such that $\theta \equiv \mathbf{C}'_i$. Since the names of arguments in the DATALOG program are preserved in the MAUDE code, the computed answers restricted to the variables of the initial query form the answers for the MAUDE query. It is clear that if the same restriction is applied to the answer constraint, the DATALOG answers are still equivalent to the restricted answer constraint.

Let us now proceed with the general case when the clause body contains more than one subgoal. In this case, the chosen top-down left-to-right DATALOG strategy states that for computing the answers for the query, the answers for the first subgoal must be computed first. Then, the rest of the body with the corresponding substitutions (from the resolution of the first subgoal) must be resolved. As in the above case, we prove that each computed answer for this specific clause has an equivalent answer constraint computed by the corresponding q_i function.

Following our transformation, the rewriting sequence starts by rewriting to an auxiliary function s_2 . This function represents the execution of the second subgoal after having reduced the first subgoal (on the first argument of s_2). This is ensured by the operational semantics of MAUDE, the definition of linked and relevant variables, and the patterns in the definition of that auxiliary function (and of those of successive subgoals). The second part of the sequence below corresponds to the computation of that first subgoal:

$$\begin{array}{l} q_i(T_1, \dots, T_m) \quad \rightarrow_{\mathcal{T}(P)} \quad s_2(\mathbf{p}(W_1, \dots, W_z), T_1 \dots T_m) \quad \rightarrow_{\mathcal{T}(P)}^* \\ s_2(\mathbf{C}_1; \dots; \mathbf{C}_w, T_1 \dots T_m) \end{array}$$

By induction hypothesis, for each computed answer θ of the DATALOG query $\mathbf{p}(W_1, \dots, W_z)$, there exists an answer constraint C_i in the set $\mathbf{C}_1; \dots; \mathbf{C}_w$ such that $\theta \equiv C_i$. At this execution point, following the definition of our transformation there are two possibilities, depending on whether or not there are more subgoals (Case 2), and (Case 1), respectively. Let us assume that we are dealing with the i -th subgoal (function symbol s_i).

Case 1 In this case, the computation may proceed in two different ways:

1. There is no solution for $\mathbf{p}(W_1, \dots, W_z)$ (and this is correct by induction hypothesis). Therefore, the answer constraint set is of the form \mathbf{F} . In this case, the rewriting sequence is:

$$s_i(\mathbf{C}_1; \dots; \mathbf{C}_w, T_1 \dots T_m) \rightarrow_{\mathcal{T}(P)} \mathbf{F}$$

This means that there is no solution for the first subgoal, so this case is trivially proved.

2. Consider the case when there are w different answer constraints for $\mathbf{p}(W_1, \dots, W_z)$ (that by induction hypothesis include the equivalent answer constraints for each DATALOG computed answer). The rewriting sequence following the definition of our transformation is:

$$s_i(\mathbf{C}_1; \dots; \mathbf{C}_w, T_1 \dots T_m) \rightarrow_{\mathcal{T}(P)} s_{i+1}(s'_i(\mathbf{C}_1; \dots; \mathbf{C}_w, T_1 \dots T_m), T_1 \dots T_m)$$

Note that to compute the answer constraints for the third subgoal (s_{i+1}), we first have to rewrite the second one by reducing the redex s'_i that contains the partially accumulated answer constraint set. Depending on the form of this constraint set, we have three possible rewritings:

- (a) The first answer constraint for the previous subgoal (C_1) is **T** (which is an **EmptyConstraint**). Therefore, the computation of the previous subgoal (which is ground) performed no substitution of variables:

$$\begin{aligned} & s_{i+1}(s'_i(\mathbf{T} \quad ; \quad \dots \quad ; \quad C_w, T_1 \dots T_m), T_1 \dots T_m) \quad \rightarrow_{\mathcal{T}(P)} \\ & s_{i+1}(\mathbf{q}(Q_1, \dots, Q_z); (s'_i(C_2 \quad ; \quad \dots \quad ; \quad C_w, T_1 \dots T_m)), T_1 \dots T_m) \quad \rightarrow_{\mathcal{T}}^* \\ & s_{i+1}(C'_1; \dots; C'_{w'}; (s'_i(C_2 \quad ; \quad \dots \quad ; \quad C_w, T_1 \dots T_m)), T_1 \dots T_m) \end{aligned}$$

By induction hypothesis, for each computed answer θ for the call $\mathbf{q}(Q_1, \dots, Q_z)$, there exists an answer constraint C_i in the set $C'_1, \dots, C'_{w'}$ such that $\theta \equiv C_i$. These are all answers for the i -th subgoal (whose function symbol is \mathbf{q} , given by the function τ^p of our transformation). Intuitively, this rewriting step represents the propagation of variable assignments to the following subgoals. It can be seen that since no substitution needed to be propagated, all the answer constraints are also answer constraints for the query consisting of the conjunction of the previous subgoal(s) and the present one. Therefore, no solution is lost.

The recursive call of s'_i propagates not only the information from the first answer constraint, but also the information needed to proceed with the computation of the rest of the solutions. We will come back to this point of the proof after introducing the rest of the cases in order to prove that answers are also preserved for them.

- (b) The first answer constraint is not **T** but a set $C_1 \quad ; \quad \dots \quad ; \quad C_w$, which by hypothesis includes the equivalent answer constraints for the computed answers of the i -th clause. The rewriting sequence is:

$$\begin{aligned} & s_{i+1}(s'_i(C_1 \quad ; \quad \dots \quad ; \quad C_w, T_1 \dots T_m), T_1 \dots T_m) \quad \rightarrow_{\mathcal{T}(P)} \\ & s_{i+1}((\mathbf{q}(Q_1, \dots, Q_z)[Q_j \setminus X_j, \dots, Q_k \setminus X_k]) \mathbf{x} C_1; \\ & (s'_i(C_2 \quad ; \quad \dots \quad ; \quad C_w, T_1 \dots T_m)), T_1 \dots T_m) \quad \rightarrow_{\mathcal{T}(P)}^* \\ & s_{i+1}((C'_1; \dots; C'_{w'}) \mathbf{x} C_1; (s'_i(C_2 \quad ; \quad \dots \quad ; \quad C_w, T_1 \dots T_m)), T_1 \dots T_m) \end{aligned}$$

where the X_j are the linked variables that have already been instantiated, and their value is propagated to the corresponding Q_j . The X_j variables are computed in C_1 . By induction hypothesis, for each computed answer θ for $\mathbf{q}(Q_1, \dots, Q_z)[Q_j \setminus X_j, \dots, Q_k \setminus X_k]$, there exists a C'_i in $C'_1, \dots, C'_{w'}$ such that $\theta \equiv C'_i$. Then, the \mathbf{x} operator combines each solution of the second subgoal with the information in C_1 . Since we have passed the shared information with the applied substitution before the subsequent reduction step, we know that the shared variables have the same value. Therefore, the new combined solutions are consistent for the conjunction of the two (or more) subgoals. Note that the only case when inconsistencies may arise (and be simplified) by the \mathbf{x} operator is when both sets of answers contain an output variable

and each one computes a different value for it. This inconsistent case is reduced to false, so no consistent answer constraint is deleted.

- (c) There are no answer constraints to proceed, so the first argument is **F** and the rewriting sequence is:

$$s_{i+1}(s'_i(\mathbf{F}, T_1 \dots T_m), T_1 \dots T_m) \rightarrow_{\mathcal{T}(P)} s_{i+1}(\mathbf{F}, T_1 \dots T_m)$$

This last case is the base case for the recursion appearing in the two previous ones. By induction on the number of elements in the answer constraint set $C_1 ; \dots ; C_w$, it can be observed that the subterm $(s'_i(C_2 ; \dots ; C_w, T_1 \dots T_m), T_1 \dots T_m)$ in the cases (a) and (b) is a smaller recursive call. Therefore, at some point the sequence will reach the base case.

Hence, we are at the point in which we have computed all the accumulated answer constraints up to the i -th subgoal and they include the equivalent answer constraints to the computed answers of the DATALOG query:

$$s_{i+1}(C_1 ; \dots ; C_n, T_1 \dots T_m)$$

Case 2 In this case, s_i is the last subgoal, so no propagation of information must be performed. We now also prove that, in this case, for each computed answer of the query, there exists an equivalent answer constraint as the result of the rewriting until normalization of the corresponding transformed query.

Remember that the term that had to be reduced at this point and that should generate the answer constraints for the considered DATALOG clause is

$$s_i(C_1 ; \dots ; C_w, T_1 \dots T_m),$$

where $C_1 ; \dots ; C_w$ include the equivalent answer constraints for the computed answers of $\mathbf{p}(W_1, \dots, W_z)$. Similarly to Case 1, in this case, there are also three possible paths:

1. The first answer constraint for the previous subgoal (C_1) is **T** (which is an **EmptyConstraint**); thus, the computation of the previous subgoal (which is ground) performed no substitution of variables:

$$s_i(\mathbf{T} ; \dots ; C_w, T_1 \dots T_m) \rightarrow_{\mathcal{T}(P)} \mathbf{q}(Q_1, \dots, Q_n) ; s_i(C_2 ; \dots ; C_w, T_1 \dots T_m) \rightarrow_{\mathcal{T}(P)}^* C'_1 ; \dots ; C'_{w'} ; s_i(C_2 ; \dots ; C_w, T_1 \dots T_m)$$

By induction hypothesis, for each computed answer θ for the call $\mathbf{q}(Q_1, \dots, Q_z)$, there exists an answer constraint C_i in the set $C'_1, \dots, C'_{w'}$ such that $\theta \equiv C_i$. These are all answers for the i -th subgoal (whose function symbol is \mathbf{q} , given by the function τ^P of our transformation).

For the recursive call, we will come back to this point of the proof after introducing the rest of the cases to prove that answers are also preserved for them.

2. The first answer constraint is not **T** but a set $C_1 ; \dots ; C_w$ that by hypothesis includes the equivalent answer constraints for the computed answers of the i -th clause. The rewriting sequence is:

$$s_i(C_1 ; \dots ; C_w, T_1 \dots T_m) \rightarrow_{\mathcal{T}(P)} (\mathbf{q}(Q_1, \dots, Q_z)[Q_j \setminus X_j, \dots, Q_k \setminus X_k]) \times C_1 ; s_i(C_2 ; \dots ; C_w, T_1 \dots T_m) \rightarrow_{\mathcal{T}(P)}^* (C'_1 ; \dots ; C'_{w'}) \times C_1 ; (s_i(C_2 ; \dots ; C_w, T_1 \dots T_m))$$

Similarly to Case (1.2.b) above, the X_j are the linked variables that have already been instantiated, and their value is propagated to the corresponding Q_j . The X_j variables are computed in C_1 . By induction hypothesis,

for each computed answer θ for $q((Q_1, \dots, Q_z)[Q_j \setminus X_j, \dots, Q_k \setminus X_k])$, there exists a C'_i in C'_1, \dots, C'_w , such that $\theta \equiv C'_i$. Then, the x operator combines each solution of the second subgoal with the information in C_1 . Since we have passed the shared information with the applied substitution before the subsequent reduction step, we know that the shared variables have the same value, thus the new combined solutions are consistent for the conjunction of the two (or more) subgoals. We note that the only case when inconsistencies may arise (and be simplified) by the x operator is when both sets of answers contain an output variable and each one computes a different value for it. This inconsistent case is reduced to false, so no consistent answer constraint is deleted.

As in the previous case, we will consider the recursive call after having presented the three cases.

3. There are no answer constraints to proceed, thus the first argument is F and the rewriting sequence is:

$$s_i(F, T_1 \dots T_m) \rightarrow_{\mathcal{T}(P)} F$$

This last case is the base case for the recursion appearing in the two previous ones. By induction on the cardinality of the set of answer constraint $C_1 ; \dots ; C_w$, it can be observed that the subterm $(s'_i(C_2 ; \dots ; C_w, T_1 \dots T_m), T_1 \dots T_m)$ in the cases (a) and (b) is a smaller recursive call. Therefore, at some point the sequence will reach the base case. Hence, we are at the point in which we have computed all the accumulated answer constraints up to the i -th subgoal and they include the equivalent answer constraints to the computed answers of the DATALOG query:

$C_1 ; \dots ; C_n$