Implementing Datalog in Maude[1]

M. Alpuente M. A. Feliú C. Joubert A. Villanueva [2]

Universidad Politécnica de Valencia, DSIC / ELP
Camino de Vera s/n, 46022 Valencia, Spain

Abstract

Transformation of programs among different paradigms has been widely studied in academic research and education. The interest on DATALOG has recently increased as a specification language for expressing, in just a few lines, complex interprocedural analyses involving dynamically created objects. In real-world problems, the DATALOG rules encoding a particular analysis must be solved generally under a huge set of DATALOG facts that are automatically extracted from the analyzed program (e.g. pointer dependencies). In that context, this work aims at exploiting MAUDE's capabilities for supporting efficient evaluation of DATALOG queries. We demonstrate how, starting from an almost straightforward transformation of DATALOG programs into MAUDE specifications, we are able to achieve a highly efficient version.

1 Introduction

DATALOG [11] has lately focused attention as a means to specifying static analysis in a very succinct way (just a few clauses), compared to traditional imperative approaches. 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 code [13]. This static analysis approach implies to recover from the program all the information of interest as a (huge) set of facts which must be handled effectively by the solver. This paper aims at taking advantage of MAUDE's features for the efficient evaluation of DATALOG queries. Transformation of programs among different paradigms, and in particular from logic programs to rewriting theories, has been widely studied in academic research and education. In this work, we demonstrate the impact of different implementation choices (equations *vs* rules, extra conditions, etc.) under our working constraints, i.e., heavy data load (sets of hundreds of facts) and just a few clauses coding the analysis to be carried out.

[1] 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 GV/2009/024, and the Universidad Politécnica de Valencia, under grant PAID-06-07 (TACPAS).

[2] Email: {alpuente,mfeliu,joubert,villanue}@dsic.upv.es

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

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, their operational principle differ: 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 [5,7,10,12]. 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 [12]. 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 direction for binding parameters was presented in [10]. 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 [10] is not concerned with preserving the computed answers, but only the termination behavior. Moreover, [10] does not tackle the problem of efficiently encoding logic programs containing a huge amount of facts in a rewriting-based infrastructure.

In previous work [2], we developed a Datalog query solving technique which is based on *Boolean Equation Systems* (Bess) [3]. 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 of reasoning in the sense that we transform a high-level Datalog program into another high-level Maude program.

In Section 2, we present the running Datalog program example that we use to illustrate the transformations into corresponding Maude specifications. We introduce different transformations by increasing order of efficiency. Section 3 describes a quite straightforward, rule-based transformation and Section 4 incrementally replaces backtracking operations of conditional rules by equations. Finally, we conclude with some experiments and further directions of research in Section 5.

2 The DATALOG example

Let us 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:

```
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 is expected to compute which are the variables in the analyzed program that can point to the object `h5`.

Similarly to [10], our goal is to define a *mode*-independent transformation for definite (Datalog) logic programs in order to keep the possibility of running both kinds of queries. Note that, since variables in rewriting logic are input-only parameters, we cannot use them to encode logic variables of Datalog. We do it by following the standard approach based on defining a ground representation for logic variables [4,6].

3 The rule-based approximation

As explained above, we are interested in recovering all the answers for a given query. The naïve approach consists in encoding the Datalog clauses as Maude conditional rules, and then use the Maude `search` command in order to mimic all possible executions of the Datalog program.

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` represents Datalog terms.

```
sorts Variable Constant Term .
subsort Variable Constant < Term .
```

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 code excerpt above 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 the Datalog clauses is unique to the whole Maude specification.

Having ground terms representing variables, we still lack a way to collect the

answers for an output variable. In our formulation, the 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 (answer constraint) that represents the substitution of (logical) variables by (logical) constants computed during the program execution:

```
op _=_      :  Term Constant -> Constraint .
op success :  -> Constraint [ctor] .
op _,_      :  Constraint Constraint -> Constraint [assoc comm id:  success] .
```

Note that the operator `_,_` has identity element `success` and obeys the laws of associativity and commutativity. A query reduced to `success` represents a successful computation.

Let us now introduce the translation of Datalog programs into Maude. Since we want to simulate the non-determinism of Datalog by using the Maude built-in breadth-first search, we translate as Maude *rules* the original Datalog clauses. Maude's `search` command will take care of trying all the possible combinations of them, providing us with the different solutions. In a Datalog program, the execution of the different clauses or facts for a given predicate symbol is potentially non-deterministic, so we define a one-to-one correspondence between each clause and fact with a Maude rule. Considering the running example, we define the following function symbols in Maude for the predicates `vP/2`, `vP0/2` and `a/2`:

```
op vP vP0 a :  Term Term -> Constraint .
```

Note that it would be useless to carry out partial answers that are not correct. A correct answer is a constraint in which there are no *irreconcilable* variable bindings. In logic programming this is implicit within the unification mechanism. In our transformation, we need an alternative mechanism. We use Maude *conditional rules* to encode simple consistency restrictions.

Given a `Constraint` element, the function `isConsistent` checks whether the constraint is consistent or not:

```
op  isConsistent :  Constraint -> Bool .
eq  isConsistent(success) = true .
ceq isConsistent((X = Cte1) , (X = Cte2) , C) = false if Cte1 =/= Cte2 .
ceq isConsistent((Cte1 = Cte2) , C) = false if Cte1 =/= Cte2 .
ceq isConsistent((T = Cte) , C) = true if isConsistent(C) [owise] .
```

Using this function, we can easily express the *conditional rules* encoding the original Datalog clauses. For the running example, we obtain the following rules:

```
crl vP(T1,T2) => C if vP0(T1,T2) => C /\ isConsistent(C).

crl vP(T1,T2) => (v(T1,T2) = Cte) , C1 , C2 if
  a(T1,v(T1,T2)) => ((v(T1,T2) = Cte) , C1) /\ isConsistent((v(T1,T2) = Cte) , C1) /\
  vP(Cte,T2) => C2 /\ isConsistent((v(T1,T2) = Cte) , C1 , C2) .
```

Each Datalog clause is in one-to-one correspondence with a Maude conditional rule. The first rule rewrites the predicate `vP` to the constraint `C` whenever `C` is a *consistent* result for the term `vP0(T1,T2)`. The second rule has a similar structure. However, since the Datalog clause has two body subgoals, the rule condition is a bit more involved. On one hand, we impose a left-to-right evaluation strategy similar to Prolog. On the other hand, the second Datalog clause in our running example, has a *free* variable, namely `Var2`. As a consequence, the first redex `a(T1,v(T1,T2))` will be reduced to a constraint that assigns a constant to the free variable `v(T1,T2)` representing `Var2`. Then, this value is propagated to the next condition, which is a recursive call to the `vP` predicate.

Up to this point, we have discussed how to encode in Maude the terms and clauses of a Datalog program. Let us now proceed with the translation of the set of (ground) facts of a Datalog program. Each fact represents an assignment of constants to predicate's parameters making the predicate `true`. In our naïve approach, a fact is represented as a Maude rule rewriting the predicate to the assignment of constants to the predicate's arguments. Given the facts of the running example, we obtain the following Maude rules:

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

We use non-conditional Maude rules because substitution's consistency does not need to be checked at the level of facts, being already checked at the clause level. Nevertheless, we have found experimentally that using conditional rules with a simple check of consistency [3] is significantly more efficient than using non-conditional rules. Following this idea, facts are defined in the example as follows:

```
crl a(T1,T2) => ( T1 = 'v1 ) , T2 = 'v2
    if (T1 == 'v1 or T1 ::  Variable) and (T2 == 'v2 or T2 ::  Variable) .
```

Once the transformation of Datalog programs into Maude specifications is performed, we use the Maude `search` command to evaluate the query to be solved. For example, to compute all objects in memory that are pointed by a given variable of the object-oriented program under analysis, we write the following command:

```
search vP(v('variable),v('heap)) =>*
       (v('variable) = C1:Constant) , (v('heap) = C2:Constant) , X:Constraint .
```

This command looks for every possible reduction of the query leading to a constraint that represents the solution for the variable of interest. We have tested this transformation over different sized sets of facts, getting the results shown in Table 1.

Table 1
Execution time (sec.) for the rule-based implementation.

Facts per predicate	100	150	200
Time	1.4 sec.	4.5 sec.	10.9 sec.

Since our final goal is to deal with programs consisting of a huge number of facts (bigger than 10^4 facts), the results of this naïve transformation are not satisfactory. We have learned that the use of rules (especially conditional ones) under backtracking for the exploration of the search space unbearably penalizes the execution time of Maude programs. Hence, one way to improve our transformation is to move to an equation-based representation.

4 The equation-based transformation

Differently from the rule-based approach, we do not check explicitly for the consistency of a constraint. Instead, we make use of simplification equations that collapse every inconsistent constraint into *false* while building (partial) answers. As a consequence, `_,_` becomes a *defined symbol*. We also set a hierarchy of subsorts that allows us to identify *trivial* constraints whenever possible, improving the overall performance. The resulting Maude specification is as follows [4]:

[3] It checks whether the parameters are consistent w.r.t. the corresponding constants of the fact.
[4] The complete specification can be found in [1].

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

op _=_ :  Term Constant -> NonEmptyConstraint .
op T : -> TConstraint .
op F : -> FConstraint .
...
eq (Cte = Cte) = T .                    --- Simplification
eq (Cte1 = Cte2) = F [owise] .          --- Unsatisfiability
eq NEC,NEC = NEC .                      --- Idempotency
eq F,NEC = F .                          --- Zero element
eq F,F = F .                            --- Simplification
eq (V = Cte1),(V = Cte2) = F [owise] .--- Unsatisfiability
```

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, at each execution point, we need to carry on not only a single answer, but all the possible (partial) answers. To this end, we introduce the notion of *set of answer constraints* implementing 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 .     --- Idempotency
```

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 union of constraints. The associativity, commutativity and (the existence of an) identity element properties of `_;_` can be easily expressed by using `ACU` [5] attributes in Maude, thus simplifying the equational specification and achieving better efficiency. We express the idempotency 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 mimic the standard left-to-right [6] 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. Unfortunately, the execution of these kind 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* [8] functions, that stepwisely evaluate each call and propagate the (partially) computed information. We rely on pattern matching to ensure that a call is

[5] Equational axioms of associativity, commutativity and identity.
[6] This is the way some Datalog solvers such as XSB [9] work.

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 this way, 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 union of the answers of functions `vPc1` and `vPc2`[7], 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) .
```

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. First, it contains more than one subgoal, thus we need an auxiliary function to impose the execution order. Second, it contains an existentially quantified variable (not appearing in the head of the clause) that carries information from one subgoal to the next.

```
eq vPc2(T1,T2) = vPc2s1(T1,T2) .
eq vPc2s1(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 `vPc2s1`, whose definition mirrors the execution of the first subgoal, carrying the (partial) set of answers to the second subgoal in the first argument of the call `vPc2s2`. Then, the two arguments of `vPc2s2` contain the (partial) answers resulting from the resolution of `a(T1,v(T1,T2))`, and the list of parameters in the head of the original clause. In the pattern at the left-hand side of the equation `vPc2s2`, the use of the term `v(T1,T2)`, representing the existentially quantified variable `Var2` of the original Datalog program, is the key for carrying the computed information from one subgoal to the subsequent subgoals where the variable occurs. Actually, it allows us to control the recursion over `vP` in `vPc2s2` by constraining it to the value calculated by `a(T1,v(T1,T2))`. Note 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. The formal definition of the transformation can be found in [1].

Similarly to clauses, the set of facts defining a particular predicate represent non-deterministic choices and hence, they cannot be translated one-to-one into equations, since equations are evaluated deterministically. We transform them by joining all non-deterministic choices into a set of answer constraints. Considering the running example, facts are transformed as follows:

```
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 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)) .
```

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

We have run this transformation over different numbers of facts and for the given query we got the results presented in Table 2. These results are much better than the

Table 2
Execution time (sec.) for the equational implementation.

Facts per predicate	100	150	200
Time	0.2 sec.	0.8 sec.	1.7 sec.

previous ones thanks to the use of non-conditional equations. Nevertheless, although encouraging, the results are not yet good enough for the efficiency requirements we have in static analysis for large (Java) programs.

We performed a last optimization with Maude's *memoization* capabilities [4]. As it is expected that the subgoal `vP(X,Y)` will be reduced many times in the recursive calls, we mark the operator representing it as a memoized one:

```
op vP : Term Term -> ConstraintSet [memo].
```

This means that Maude stores each call to `vP(X,Y)` together with its normal form. Thus, when Maude finds a memoized call it won't reduce it but just replace it with its irreducible form, saving a great amount of rewrites.

We have run again the same tests with this little modification and we got the results presented in Table 3. This allows us to gain another order of magnitude in execution time with respect to the previous version.

Table 3
Execution time (sec.) for the equational implementation with memoization.

Facts per predicate	100	150	200	400	800
Time	0.01 sec.	0.01 sec.	0.02 sec.	0.08 sec.	0.6 sec.

We want to mention that, as the number of facts increases, the time required to read them by the Maude interpreter dominates significantly over the resolution time. Execution times presented in Tables 2 and 3 exclude the time spent for loading the specification. As an example, Maude inverts 95% (11.4 sec.) of the total execution time (12 sec.) in reading and preparing a file with 800 facts before rewriting it.

5 Conclusion

In this work we have outlined different transformations from Datalog programs to Maude specifications in the context of Datalog-based static analysis. These transformations are presented from the straightforward implementation to most efficient ones, simulating the steps we followed during the development process. We also describe some encountered difficulties and adopted solutions.

The main advantage of the rule-based approximation is its intuitive correspondence to the original Datalog program, whereas the equational approach, although less straightforward, is several orders of magnitude faster than the latter. Nevertheless, even our best implementation, that handles programs with several hundreds of facts, is not sufficiently competitive compared to traditional solvers.

As a future work, we plan to use a more compact representation of the facts in order to minimize the significant execution time spent in the present version to load them. We can also explore the impact of more sophisticated optimization techniques like tail-recursion or memoization (at the logical level).

Acknowledgments

We gratefully thank Santiago Escobar for his help and careful suggestions with this work.

References

[1] M. Alpuente, M.A. Feliú, C. Joubert, and A. Villanueva. Defining Datalog in Rewriting Logic. In D. de Schreye, editor, *Proceedings of 19th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR'09)*, volume to appear, 2009.

[2] M. Alpuente, M.A. Feliú, C. Joubert, and A. Villanueva. Using Datalog and Boolean Equation Systems for Program Analysis. In *Proceedings of the 13th International Workshop on Formal Methods for Industrial Critical Systems (FMICS 2008)*, volume 5596 of *Lecture Notes in Computer Science*. Springer-Verlag, 2009.

[3] H. R. Andersen. Model checking and boolean graphs. *Theoretical Computer Science*, 126(1):3–30, 1994.

[4] M. Clavel, F. Durán, S. Ejer, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *All About Maude – A High-Performance Logical Framework*, volume 4350 of *Lecture Notes in Computer Science*. Springer-Verlag, 2007.

[5] M. Vam Emden and J.W. Lloyd. A logical reconstruction of Prolog II. *Journal on Logic Programming*, 1, 1984.

[6] P.M. Hill and J.W. Lloyd. Analysis of meta-programs. In H.D. Abramson and M.H. Rogers, editors, *Proceedings of the Meta-Programming in Logic Programming Workshop (Meta88)*, pages 23–52. MIT Press, 1989.

[7] M. Marchiori. Logic Programs as Term Rewriting Systems. In *Proceedings of the 4th International Conference on Algebraic and Logic Programming*, volume 850 of *Lecture Notes In Computer Science*, pages 223– 241. Springer Verlag, 1994.

[8] M. Marchiori. Unravelings and ultra-properties. In M. Hanus and M. Rodríguez-Artalejo, editors, *5th International Conference on Algebraic and Logic Programming, ALP'96*, volume 1039 of *LNCS*, pages 107–121. Springer, 1996.

[9] K. Sagonas, T. Swift, and D.S. Warren. Xsb as an efficient deductive database engine. *SIGMOD Rec.*, 23(2):442–453, 1994.

[10] P. Schneider-Kamp, J. Giesl, A. Serebrenik, and R. Thiemann. Automated Termination Analysis for Logic Programs by Term Rewriting. In *Proceedings of the Logic-Based Program Synthesis and Transformation (LOPSTR'06)*, volume 4407 of *Lecture Notes in Computer Science*, pages 177–193. Springer Verlag, 2007.

[11] J. D. Ullman. *Principles of Database and Knowledge-Base Systems, Volume I and II, The New Technologies*. Computer Science Press, 1989.

[12] Reddy U.S. Transformation of Logic Programs into Functional Programs. In *Proceedings of the Symposium on Logic Programming*, pages 187–197. IEEE Computer Society Press, 1984.

[13] J. Whaley, D. Avots, M. Carbin, and M. S. Lam. Using Datalog with Binary Decision Diagrams for Program Analysis. In *Proc. Third Asian Symp. on Programming Languages and Systems APLAS'05*, volume 3780 of *LNCS*, pages 97–118. Springer-Verlag, 2005.