Universidad Politécnica de Valencia

Departamento de Sistemas Informáticos y Computación

Máster en Ingeniería del Software, Métodos Formales y Sistemas de Información

Master Thesis

Evaluation of Datalog queries and its application to the static analysis of Java code

Candidate:
Marco A. Feliú

Supervisors:
María Alpuente
Christophe Joubert
Alicia Villanueva

Academic Year 2010/2011

Departamento de Sistemas Informáticos y Computación
Universidad Politécnica de Valencia
Camino de Vera, s/n
46022 Valencia
España

Contents

Introduction

In the transition from an industrial economy to a knowledge-based global economy, computer technologies have become a determining factor in productivity advances and, as a result, in economic growth. Works like [JV05] are conclusive regarding the fact that, starting from the second half of the nineties, computer technologies have played a progressively important role in the productivity advances of the G7 countries [VT06]. Moreover, they have become crucial for our lives, and they are present in everyday objects (mobile devices, cars) and in strategic areas (health, transportation), frequently in a critical way w.r.t. safety, economy or security.

The main problem in the development of software systems is the higher and higher complexity of analyzing and guaranteeing the reliability of their behaviour. Defects in these –more and more complex and heterogeneous– software systems cause enormous personal, environmental and economic damage, and also make software development and maintenance extremely expensive. Formal methods have the potential to guarantee the absence of defects, but today they do not meet software industry needs like cost effectiveness, time-to-market, reusability, or versatility. The common feature of formal methods is that those techniques are all based on logical methods.

Logical methods in computer science

Logic-based theory, techniques and tools are having an increasingly big impact on different computer science areas, as well as on finding solutions to numerous computational problems that arise in industry and other sciences like Biology. There are several reasons for explaining this boom in logic-based methods. One reason is that, since computer science is still a young science, its numerous ad hoc techniques are still evolving into more general and better-studied common foundations that usually turn out to be based on logics. Another reason is a theoretical one: most formalisms (automata, languages, complexity classes, etc.) have their logical counterparts, and there exist correspondences between computational mechanisms and logics, like the ones established by the Curry-Howard isomorphism. But, above all those reasons we have to highlight their practical repercussions, as Alan Turing already predicted:

"I expect that digital computing machines will eventually stimulate a considerable interest in symbolic logic (...). The language in which one communicates with these machines (...) forms a sort of symbolic logic."

Similarly, McCarthy wrote, back in the sixties, that logics would have an importance in computer science similar to the importance that mathematical analysis had in Physics in the XIX century. Along the same lines, Manna and Waldinger [MW85] called logics *"The Calculus of Computer Science"*, since its role in computer science, both at the theoretical and practical level, is similar to the one of Mathematics in Physics and, in particular, to the one of calculus in engineering.

Architects and engineers analyze the mathematical properties of their constructions, and similarly, computer scientists can analyze the logical properties of their systems while designing, developing, verifying or maintaining them, especially when it comes to deal with systems that are critical in terms of economy, security or privacy. But this is also the case for systems whose *efficiency* is critical, since logical analysis might be revealing. More generally in all sorts of systems, it is widely accepted that methods and tools based on logics are able to improve their quality and reduce their cost.

This view is widely documented in the paper *"On the Unusual Effectiveness of Logic in Computer Science"* [HHI^{+}01], where the crucial role of logics in areas like databases, programming languages, complexity, agent systems and verification is exposed by worldwide experts in each of these areas; see also `http://www.cs.rice.edu/~vardi/logic/`.

Analogously, logics is playing a central role in several important recent applications. A perfect example is the so-called *semantic web*, that provides Internet (and intranet) web pages with semantic information that allows one to use semantically-based search criteria, deductive mechanisms, consistency or integrity constraints, etc. All these ideas are tightly related to the study of *description logics*, that correspond to certain decidable subclasses of first-order logic[BCM^{+}03]. One last present-day example is the use of logics for cryptographic protocol verification, for applications like authentication or anonymous electronic money.

Static analysis

Static (program) analysis in computer science is the field of study that deals with the analysis of programs without executing them. Static analysis provides static compile-time techniques for predicting safe and computable approximations of the set of values or behaviours arising dynamically at run-time when executing a program on a computer. Its formal foundations lie in the theory of Abstract Interpretation [CC77], which is a a theory of sound approximation of the semantics of computer programs, based on monotonic functions over ordered sets, especially lattices.

One branch of logic-based methods lies on logic programming languages. Recently, the logic language Datalog has been used to specify many computer-intensive static analyses. One of the benefits of this approach are the succinctness and the understandability of the specification, which benefits the development of progressively more sophisticated analysis. The other great benefit of the Datalog-based static analysis is that it clearly separates the analysis specification from its efficient execution. As an immediate consequence, a broad set of static analyses (those that can be expressed as Datalog programs) can be directly endowed with an efficient implementation, thanks to the techniques for efficiently executing Datalog programs. Due to this fact, many members of the static analysis community have focused their interest in the development of progressively more efficient methods for solving Datalog queries.

Our solution

We propose two different Datalog query answering techniques that are specially-tailored to object-oriented program analysis. Our techniques essentially consist of transforming the original Datalog program into a suitable set of rules which are then executed under an optimized top-down strategy that caches and reuses "rewrites" in the target language.

We use two different formalisms for transforming any given set of definite Datalog clauses into an efficient implementation, namely Boolean Equation Systems (Bes) [And94a] and Rewriting Logic (Rwl) [Mes92], a very general *logical* and *semantical framework* that is efficiently implemented in the high-level executable specification language Maude [CDE+07a].

In the Bes-based program analysis methodology, the Datalog clauses that encode a particular analysis, together with a set of Datalog facts that are automatically extracted from program source code, are dynamically transformed into a Bes whose local resolution corresponds to the demand-driven evaluation of the program analysis. This approach allows us to reuse existing general purpose analysis and verification toolboxes such as Cadp, which provides local Bes resolution with linear-time complexity. Similarly to the *Query/Subquery* technique [Vie86], computation proceeds with a set of tuples at a time. This can be a great advantage for large datasets since it makes disk access more efficient.

Our motivation for developing our second, Rwl-based query answering technique for Datalog was to provide purely declarative yet efficient program analyses that overcome the difficulty of handling meta-programming features such as reflection in traditional analysis frameworks [LWL05]. Tracking reflective method invocations requires not just tracking object references through variables but actually tracking method values and method name strings. The interaction of static analysis with meta-programming frameworks is non-trivial, and analysis tools risk losing correctness and completeness, particularly when reflective calls are improperly interpreted during the computation. By transforming Datalog programs into MAUDE programs, we take advantage of the flexibility and versatility of MAUDE in order to achieve meta-programming capabilities, and we make significant progress towards scalability without losing the declarative nature of specifying complex program analyses in Datalog. The current version of MAUDE can do more than 3 million rewritings per second on standard PCs, so it can be used as an implementation language [RH05]. Also, as a means to scale up towards handling real programs, we wanted to determine to what extent MAUDE is able to process a sizable number of constraints that arise in real-life problems, like the static analysis of JAVA programs. After exploring the impact of different implementation choices (equations *vs* rules, unraveling *vs* conditional term rewriting systems, explicit *vs* implicit consistency check, etc.) in our working scenario (i.e., sets of hundreds of facts and a few clauses that encode the analysis), we elaborate on an equation-based transformation that leads to efficient transformed MAUDE-programs.

Original contributions

The contributions presented in this thesis have given birth to some publications that we enumerate below:

- "Using Datalog and Boolean Equation Systems for Program Analysis" [AFJV09c] and "DATALOG_SOLVE: A Datalog-Based Demand-Driven Program Analyzer" [AFJV09a] present how to transform Datalog programs into BES, and how to solve them in the context of static analysis with our prototype DATALOG_SOLVE.

- "Implementing Datalog in Maude" [AFJV09b] illustrates the iterative process, aimed at optimizing the running time, that we followed for transforming Datalog programs into RWL theories.

- "Defining Datalog in Rewriting Logic" [AFJV10] formally presents our transformation from Datalog into RWL.

- "Datalog-based program analysis with BES and RWL" [?] is an overview of the two approaches and their current state of development.

This thesis provides a comprehensive view of these techniques, which are fully automatable.

Plan of the thesis

This document progressively introduces the necessary notions for the comprehension of the work done.

The present introduction serves as a motivation and as a general view of the work done for the fulfillment of this thesis. The first Chapter introduces the theoretical background necessary to understand our contribution: Datalog, *static analyses*, *Boolean Equation Systems*, and *Rewriting Logic*. In Chapter 2 we explain the transformation into BES for executing Datalog programs, as well as our prototype DATALOG_SOLVE and the experimental results that we obtained with it. Chapter 3 presents the RWL approach for executing Datalog programs and its application to the analysis of JAVA *reflective* programs, as well as our prototype DATALAUDE together with the experimental results obtained with it. Finally, in the conclusions we summarize the work presented in this thesis and give further ideas to explore as future work.

Chapter 1

Preliminaries

This Chapter introduces the background knowledge necessary to understand the work presented in this thesis. We will present concepts related to the Datalog logic language, *Boolean Equation Systems* (BESs), *rewriting logic* (RWL) as implemented in MAUDE, and static analysis, specifically, pointer analysis.

1.1 Datalog

Datalog [Ull85] is a relational language that uses declarative *clauses* to both describe and query a deductive database. It is a language that uses a PROLOG-like notation, but whose semantics is far simpler than that of PROLOG.

Predicates, atoms and literals. The basic elements of Datalog are *atoms* of the form $p(X_1, X_2, ..., X_n)$ where:

1. p is a *predicate symbol* — a symbol that represents an assertion concerning the arguments given between parenthesis at its right.

2. $X_1, X_2, ..., X_n$ are terms (*i.e.*, variables or constants) that act as arguments of the predicate.

A *ground atom* is an atom with only constants as arguments. Every ground atom asserts a particular fact, and its value is either true or false. It is often convenient to represent a predicate by a *relation*, or table of its true ground atoms. Each ground atom is represented by a single row, or tuples, of the relation. The columns of the relation are its attributes, and each tuple has a component for each attribute. The attributes correspond to the argument positions of the predicate represented by the relation. Any ground atom present in the relation is true and we will call it a *fact*, whereas ground atoms not in the relation are false. From now on, we will use relation p and predicate p interchangeably.

Rules. Rules (also called clauses) are a way of expressing logical inferences, and suggest how a computation of the true facts should be carried out. The form of a rule is:

$$H \ :- \ B_1 \ , \ B_2 \ , \ \dots \ , \ B_n. \tag{1.1.1}$$

The components are as follows:

- H, B_1, B_2, ..., B_n are *literals*, *i.e.*, either atoms or negated atoms.
- H is the *head* and B_1 , B_2 ... , B_n form the *body* of the clause.
- Each of the B_i's is called a *subgoal* or *hypothesis* of the rule.

We should read the $:-$ symbol as "if". The "," operators separating each subgoal of the body is a logical *and* operator. Thus, the meaning of a rule is "the head is true if the body is true". More precisely, a rule is *applied* to a given set of ground atoms as follows. Consider all possible substitutions of constants for the variables of the rule. If a certain substitution makes every subgoal of the body true, *i.e.*, each subgoal is in the given set of ground atoms — which are supposed to be true —, then we can infer that the head with this substitution applied is a true fact.

Programs. A Datalog *program* is a collection of rules together with the "data", in the form of an initial set of facts for some of the predicates. The semantics of the program is the set of ground atoms inferred by using the facts and applying the rules until no more inferences can be made. The initial set of facts of a Datalog program is called the *extensional database*, whereas the set of facts inferred by means of clauses with non-empty bodies is called the *intensional database*. In this way a predicate defined in the extensional or intensional databases is called *extensional* or *intensional* predicate, respectively.

Queries. In a *demand-driven* context, that is, under the assumption that we are not interested in everything that can be inferred, queries allow us to restrict the information we want to compute. By restricting the information to be inferred, the execution improves in terms of (execution) time and (memory) space.

A Datalog *goal* has this form:

$$:- \ B_1 \ , \ B_2 \ , \ \dots \ , \ B_n. \tag{1.1.2}$$

The structure of a goal is analogous to the one of a rule body, but each of the B_i's is called a *subgoal*. We can read a query as a question "there exists a substitution of the variables used in the goal that make true all the subgoals?".

There are many approaches for the evaluation of Datalog programs. The two basic ones are the *top-down* and the *bottom-up* strategies. The *top-down* approach solves queries by reasoning backwards, whereas the *bottom-up* approach blindly infers all the program facts and then checks if the query has been previously inferred.

In this thesis we use the top-down approach, and we introduce some more detailed notions about it in Chapters 2 and 3.

1.2 Static analysis

Static (program) analysis in computer science is the field of study that deals with the analysis of programs without executing them. Static analysis provides static compile-time techniques for predicting safe and computable approximations of the set of values or behaviours arising dynamically at run-time when executing a program on a computer. Its formal foundations lie in the theory of Abstract Interpretation [CC77], which is a a theory of sound approximation of the semantics of computer programs, based on monotonic functions over ordered sets, especially lattices.

Static analyses have a very wide field of applications. In fact, any property of interest that a computer program may have is a possible target for static analysis. The verification of these properties and, in particular, safety and security properties are one of the most important domains of application.

Traditionally, static analysis have been defined by providing an abstract interpreter, or by transforming the program into an abstract one. The idea is to mimic an *abstract execution* of the program, which can be seen as a superset of the possible behaviours of the original programs.

Static analysis deals with real programs, implemented in complex programming languages, making more difficult their specification. However, when someone wants to analyze a program he usually is interested only on a specific aspect of the program semantics, thus, ignoring many other of its dimensions.

A recent proposal [WACL05] for analyzing programs consists in extracting the analysis relevant information from the program, and specifying the analysis logic by using *more natural* formalism, in this case, Datalog. This approach partially overcomes the problem of the high complexity of static analysis implementations. However, it is necessary the existence of highly efficient solvers for the formalism, in order for the approach to be competitive. Hence the interest in new optimizations for Datalog solvers.

1.2.1 Datalog-based static analysis

The Datalog approach to static program analysis [WACL05] can be summarized as follows. On one hand, each program element, namely variables, types, and code locations are grouped in their respective *domains*. Thus, each argument of a predicate symbol is typed by a domain of values. Each program statement is decomposed into *basic program operations* such as load, store, and assignment operations. Each kind of basic operation is described by a relation in a Datalog program. By considering only finite program domains, and applying standard loop-checking techniques, Datalog program execution is ensured to terminate. On the other hand, the static analysis logic (algorithm) can be declaratively expressed in the form of Datalog clauses. These Datalog clauses infer relations that represents the information of interest resulting from the execution of the analysis.

Pointer-analysis. Pointer analysis is a family of static analysis focused on answering the question “which things can point to which things?”. In an object oriented programming language like JAVA, the “things” that are able to point to something are *variables* and *fields* inside objects; whereas the “things” can be pointed to are *objects* located in the heap. So, in this case, pointer analysis on JAVA approximates all possible flow through variables and objects of object references. In order to describe the transformations from Datalog programs into BES and RWL, let us introduce our running example: a version of Andersen’s [And94b] points-to analysis, which is a context-insensitive points-to analysis borrowed from [WACL05].

Example 1.2.1 The upper left side of Figure 1.1 shows a simple JAVA program where `o1` and `o2` are heap allocations (extracted by a JAVA compiler from the corresponding bytecode). The Datalog pointer analysis approach consists in first extracting Datalog facts (relations at the upper right side of the figure) from the program. For instance, the relation `vP0` represents the direct points-to information of a program, i.e., `vP0(v,h)` holds if there exists a direct assignment of heap (abstraction) object reference `h` to program variable `v`. Other Datalog relations such as `store`, `load` and `assign` relations are inferred similarly from the code.

Using these extracted facts, the analysis deduces further pointer-related information, like points-to relations from local variables and method parameters to heap objects (`vP(V1,H1)` in Figure 1.1) as well as points-to relations between heap objects through field identifiers (`hP(H1,F,H2)` in Figure 1.1).

A Datalog query consists of a goal over the relations defined in the Datalog program, *e.g.*, `:- vP(X,Y)`. This goal aims at computing the complete set of program variables in the domain of `X` that may point to any heap object `Y` during program execution. In the example above, the query computes the following answers: {`X/p,Y/o1`}, {`X/q,Y/o2`}, and {`X/r,Y/o2`}. ■

```
public A foo { ...  p = new Object(); /* o1 */       vPO(p,o1).
                    q = new Object(); /* o2 */       vPO(q,o2).
                    p.f = q;                         store(p,f,q).
                    r = p.f; ...  }                  load(p,f,r).
```

```
      vP(V1,H1)      :- vPO(V1,H1).
      vP(V1,H1)      :- assign(V1,V2), vP(V2,H1).
      hP(H1,F,H2)    :- store(V1,F,V2), vP(V1,H1), vP(V2,H2).
      vP(V1,H1)      :- load(V2,F,V1), vP(V2,H2), hP(H2,F,H1).
```

Figure 1.1: Datalog specification of a context-insensitive points-to analysis.

1.3 Parameterised Boolean Equation Systems

Parameterised Boolean Equation Systems (Pbess) are a low-level formalism that has been largely studied in the context of formal verification. There exist very efficient tools for solving Pbes in an industrial setting [GMLS07]. In the following, we present the basic notions for working with Pbes.

Given $\mathcal{X}$ a set of boolean variables and $\mathcal{D}$ a set of data terms, a *Parameterised Boolean Equation System* [Mat98] (Pbes) $B = (x_0, M_1, ..., M_n)$ is a set of n blocks M_i, each one containing $p_i \in \mathbb{N}$ fixpoint equations of the form

$$x_{i,j}(\vec{d}_{i,j} : \vec{D}_{i,j}) \overset{\sigma_i}{=} \phi_{i,j}$$

with $j \in [1..p_i]$ and $\sigma_i \in \{\mu, \nu\}$, also called *sign* of equation i, the least (μ) or greatest (ν) fixpoint operator. Each $x_{i,j}$ is a boolean variable from $\mathcal{X}$ that binds zero or more data terms $d_{i,j}$ of type $D_{i,j}$[1] which may occur in the *boolean formula* $\phi_{i,j}$ (from a set Φ of boolean formulae). $x_0 \in \mathcal{X}$, defined in block M_1, is a boolean variable whose value is of interest in the context of the local resolution methodology. Boolean formulae $\phi_{i,j}$ are formally defined as follows.

Definition 1.3.1 (Boolean Formula) *A* boolean formula ϕ*, defined over an alphabet of (parameterised) boolean variables* $X \subseteq \mathcal{X}$ *and data terms* $D \subseteq \mathcal{D}$*, has the following syntax given in positive form:*

$$\phi, \phi_1, \phi_2 ::= \mathsf{true} \mid \mathsf{false} \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid X(e) \mid \forall d \in D.\ \phi \mid \exists d \in D.\ \phi$$

where boolean constants and operators have their usual definition, e *is a data term (constant or variable of type* D*),* $X(e)$ *denotes the* call *of a boolean variable* X *with parameter* e*, and* d *is a term of type* D*.*

[1]To simplify our description in the rest of the paper, we intentionally restrict to one the maximum number of data term parameter $d : D$.

A *boolean environment* $\delta \in \Delta$ is a partial function mapping each (parameterised) boolean variable $x(d : D)$ to a predicate $\delta(x) : \mathcal{X} \to (D \to \mathbb{B})$, with $\mathbb{B} = \{\mathsf{true}, \mathsf{false}\}$. *Boolean constants* true and false abbreviate the empty conjunction $\wedge\emptyset$ and the empty disjunction $\vee\emptyset$ respectively. A *data environment* $\varepsilon \in \mathcal{E}$ is a partial function mapping each data term e of type D to a value $\varepsilon(e) : D \to D$, which forms the so-called support of ε, noted *supp*(ε). Note that $\varepsilon(e) = e$ when e is a constant data term. The *overriding* of ε_1 by ε_2 is defined as $(\varepsilon_1 \oslash \varepsilon_2)(x) = \mathit{if}\ x \ \in \mathit{supp}(\varepsilon_2)\ \mathit{then}\ \varepsilon_2(x)\ \mathit{else}\ \varepsilon_1(x)$. The *interpretation function* $[\![\phi]\!]\delta\varepsilon$, where $[\![.]\!] : \Phi \to \Delta \to \mathcal{E} \to \mathbb{B}$, gives the truth value of boolean formula ϕ in the context of δ and ε, where all free boolean variables x are evaluated by $\delta(x)$, and all free data terms d are evaluated by $\mathcal{E}(d)$.

Definition 1.3.2 (Semantics of Boolean Formula) *Let* $\delta : \mathcal{X} \to (D \to \mathbb{B})$ *be a* boolean environment *and* $\varepsilon : D \to D$ *be a data environment. The semantics of a boolean formula* ϕ *is inductively defined by the following interpretation function:*

$$\begin{array}{rcl}
[\![\mathsf{true}]\!]\delta\varepsilon & = & \mathsf{true} \\
[\![\mathsf{false}]\!]\delta\varepsilon & = & \mathsf{false} \\
[\![\phi_1 \wedge \phi_2]\!]\delta\varepsilon & = & [\![\phi_1]\!]\delta\varepsilon \wedge [\![\phi_2]\!]\delta\varepsilon \\
[\![\phi_1 \vee \phi_2]\!]\delta\varepsilon & = & [\![\phi_1]\!]\delta\varepsilon \vee [\![\phi_2]\!]\delta\varepsilon \\
[\![x(e)]\!]\delta\varepsilon & = & (\delta(x))(\varepsilon(e)) \\
[\![\forall d \in D.\ \phi]\!]\delta\varepsilon & = & \forall\, v \in D,\ [\![\phi]\!]\delta(\varepsilon \oslash [v/d]) \\
[\![\exists d \in D.\ \phi]\!]\delta\varepsilon & = & \exists\, v \in D,\ [\![\phi]\!]\delta(\varepsilon \oslash [v/d])
\end{array}$$

Definition 1.3.3 (Semantics of Equation Block) *Given a* Pbes $B = (x_0, M_1, ..., M_n)$ *and a boolean environment* δ*, the solution* $[\![M_i]\!]\delta$ *to a block* $M_i = \{x_{i,j}(d_{i,j} : D_{i,j}) \overset{\sigma_i}{=} \phi_{i,j}\}_{j\in[1,p_i]}$ *(*$i \in [1..n]$*) is defined as follows:*

$$[\![\{x_{i,j}(d_{i,j} : D_{i,j}) \overset{\sigma_i}{=} \phi_{i,j}\}_{j\in[1,p_i]}]\!]\delta = \sigma_i \Psi_{i\delta}$$

where $\Psi_{i\delta} : (D_{i,1} \to \mathbb{B}) \times\ \ldots\ \times\ (D_{i,p_i} \to \mathbb{B}) \to (D_{i,1} \to \mathbb{B}) \times\ \ldots\ \times\ (D_{i,p_i} \to \mathbb{B})$ *is a vectorial functional defined as*

$$\Psi_{i\delta}(g_1, ..., g_{p_i}) = (\lambda v_{i,j} : D_{i,j}.[\![\phi_{i,j}]\!](\delta \oslash [g_1/x_{i,1}, ..., g_{p_i}/x_{i,p_i}])[v_{i,j}/d_{i,j}])_{j\in[1,p_i]}$$

where $g_i : D_i \to \mathbb{B}$*,* $i \in [1..p_i]$*.*

A Pbes is *alternation-free* if there are no mutual recursion between boolean variables defined by least ($\sigma_i = \mu$) and greatest ($\sigma_i = \nu$) fixpoint boolean equations. In this case, equation blocks can be sorted topologically such that the resolution of a block M_i only depends upon variables defined in a block M_k with $i < k$. A block M_i is *closed* when the resolution of all its boolean formulae $\phi_{i,j}$ only depends upon boolean variables $x_{i,k}$ from M_i.

Definition 1.3.4 (Semantics of alternation-free PBES) *Given an alternation-free* PBES $B = (x_0, M_1, ..., M_n)$ *and a boolean environment* δ*, the semantics* $[\![B]\!]\delta$ *to* B *is the value of its main variable* x_0 *given by the semantics of* M_1*,* i.e., $\delta_1(x_0)$*, where the contexts* δ_i *are calculated as follows:*

$$\begin{array}{rcl} \delta_n & = & [\![M_n]\!][] \text{ (the context is empty because } M_n \text{ is closed)} \\ \delta_i & = & ([\![M_i]\!]\delta_{i+1}) \oslash \delta_{i+1} \text{ for } i \in [1, n-1] \end{array}$$

where each block M_i *is interpreted in the context of all blocks*

1.4 Rewriting Logic

Rewriting logic is a powerful logical framework that allows us to formally represent a wide range of systems [Mes92], including models of concurrency, distributed algorithms, network protocols, semantics of programming languages, and models of cell biology. Rewriting logic is also an expressive universal logic, thus being flexible logical framework in which many different logics and inference systems can be represented and mechanized.

A rewrite theory is a tuple $R = (\sigma, E, R)$, with:

- (σ, E) an equational theory with function symbols σ and equations E; and
- R a set of labeled rewrite rules of the general form

$$r : t \longrightarrow t' \tag{1.4.1}$$

  with t, t', σ-terms which may contain variables in a countable set X of variables.

Intuitively, R specifies a concurrent system, whose states are elements of the initial algebra $T_{\sigma/E}$ specified by (σ, E), and whose concurrent transitions are specified by the rules R. The equations E may be decomposes as a union $E = E_0 \cup A$, where A is a (possibly empty) set of structural axioms (such as associativity, commutativity, and identity axioms).

Rewriting logic expresses an equivalence between logic and computation in a particularly simple way. Namely, system states are in bijective correspondence with formulas (modulo whatever structural axioms are satisfied by such formulas: for example, modulo the associativity and commutativity of a certain operator) and concurrent computations in a system are in bijective correspondence with proofs (modulo appropriate notions of equivalence among computations and among proofs).

Given this equivalence between computation and logic, a rewriting logic axiom of the form:

$$t \;\rightarrow\; t'$$

has two readings. Computationally, it means that a fragment of a system's state that is an instance of the pattern t can change to the corresponding instance of t' concurrently with any other state changes; that is, the computational reading is that of a local concurrent transition. Logically, it just means that we can derive the formula t' from the formula t; that is, the logical reading is that of an inference rule.

Rewriting logic is entirely neutral about the structure and properties of the formulas/states t. They are entirely *user definable* as an algebraic data type satisfying certain equational axioms, so that rewriting deduction takes place modulo such axioms. Because of this neutrality, rewriting logic has good properties: as a logical framework, many other logics can be naturally represented in it; as a semantic framework, many different system styles, models of concurrent computation, and languages can be naturally expressed.

1.4.1 Maude

MAUDE[2] [CDE+07b] is a very efficient implementation of *rewriting logic*. As it will be presented in this section, MAUDE is a programming language that uses rewriting rules, similarly to the so-called functional languages like HASKELL, ML, SCHEME, or LISP. In the following we briefly present some of the features of this language that have been used in our work.

A MAUDE program is made up of different *modules*. Each module can include:

- *sort* (or *type*) declarations;
- *variable* declarations;
- *operator* declarations;
- *rules* and/or *equations* describing the behaviour of the system operators, *i.e.*, the *functions*.

MAUDE, mainly distinguishes two kinds of modules depending on the constructions they define and on their expected behaviour. Functional modules do not contain rules and the behaviour of their equations is expected to be confluent and terminating. On the contrary, system modules can contain both equations and rules and, though the behaviour of their equations is also expected to be confluent and terminating, the behaviour of its rules may be non-confluent and non-terminating. A functional module is limited by the reserved keywords `fmod` and `endfm`, whereas a system module is defined in between `mod` and `endm`.

[2]`http://maude.cs.uiuc.edu/`

Sorts. A sort declaration looks like:

```
sort T .
```

where `T` is the identifier of the newly introduced sort `T`. MAUDE *identifiers* are sequences of ASCII characters without white spaces, nor the special characters '`{`','`}`','`(`','`)`','`[`', and '`]`' unless they are escaped with the back-quote character '`` ` ``'. If we want to introduce many sorts `T1 T2 ... Tn` at the same time, we write:

```
sorts T1 T2 ... Tn .
```

After having declare the sorts, we can define operators on them.

Operators. Operators are declared as follows:

```
op C : T1 T2 ... Tn -> T .
```

where `T1 T2 ... Tn` are the sorts of the arguments for operator `C`, and `T` is the resulting sort for the operator. We can also declare at the same time many operators `C1 C2 ... Cn` having the same signature (*i.e.*, arguments and resulting sorts) at the same time:

```
op C1 C2 ... Cn : T1 T2 ... Tm -> T .
```

Operators can represent two kinds of objects: constructors and defined symbols. Constructors constitute the *ground terms* or *data* associated to a sort, whereas defined symbols represent functions whose behaviour will be specified by means of equations or rules. The rewriting engine of MAUDE does not distinguish between constructors or defined symbols, so there is no real syntactic difference between them. However, for documentation (and debugging) purposes operators that are used as constructors can be labeled with the attribute `ctor`.

Operator attributes. Operator attributes are labels associated to an operator which provide additional information about the operator: semantic, syntactic, pragmatic, etc. All such attributes are declared within a single pair of enclosing square brackets "`[`" and "`]`":

```
op C1 C2 ... Cn : T1 T2 ... Tm -> T [A1 ... Ao] .
```

where the `Ai` are attribute identifiers.

Mix-fix notation. Another interesting feature of operators in MAUDE is *mix-fix* notation. Every operator defined as above is declared in *prefix* notation, that is, its arguments are given separated by commas, and enclosed in parenthesis, following the operator symbol, as in:

```
C(t1, t2, ... , tn)
```

where `t1`, `t2`,..., and `tn` are, respectively, terms of sorts `T1`, `T2`,..., and `Tn`. Nevertheless, MAUDE provides a powerful and tunable syntax analyzer that allows us to declare operators composed of different identifiers separated by its arguments. The arguments can be set in any position, in any order, and even separated by white spaces. Mix-fix operators are identified by the sequence of its component identifiers, with characters '_' inserted in the place each argument is expected to be, as in:

```
op if_then_else_fi : Bool Exp Exp -> Exp .
op __ : Element List -> List .
```

The first line above defines and if-then-else operator, while the second one defines `Lists` of juxtaposed (i.e., separated by white spaces) `Elements`.

Sort orders. Sorts can be organized into *hierarchies* with `subsort` declarations. In:

```
subsort T1 < T2 .
```

we state that each element in `T1` is also in `T2`. For example, we can define natural numbers by considering their classification as positives or as the zero number in this way:

```
sorts Nat Zero NonZeroNat .
subsort Zero < Nat .
subsort NonZeroNat < Nat .
op 0 : -> Zero [ctor] .
op s : Nat -> NonZeroNat [ctor].
```

MAUDE also provides operator *overloading.* For example, if we add:

```
sort Binary .
op 0 : -> Binary [ctor] .
op 1 : -> Binary [ctor] .
```

to the previous declarations, the operator `0` is used to construct values both for the `Nat` and for the `Binary` sorts.

Structural axioms. The language allows the specification of structural axioms over operators, *i.e.*, certain algebraic properties like *associativity*, *commutativity* and *identity element* that operators may satisfy. Structural axioms serve to perform the computation on equivalence classes of expressions, instead of on concrete expressions. In order to carry out computations on equivalence classes, Maude chooses a canonical representative of each class and uses it for the computation. Thanks to the structural information given as operator attributes, Maude can also choose specific data structures to give an efficient low-level representation of expressions.

For example, let us define a list of natural numbers separated by colons:

```
sorts NatList EmptyNatList NonEmptyNatList .
subsort EmptyNatList < NatList .
op nil : -> EmptyNatList [ctor] .
subsort Nat < NonEmptyNatList .
subsort NzNat < NatList .
op _:_ : NatList NatList -> NonEmptyNatList [assoc] .
```

The operator "_:_" is declared as *associative* by means of its attribute `assoc`. Associativity means that the value of an expression is not dependent on the subexpression grouping considered, that is, the places where the parenthesis are inserted. Thus, if "_:_" is associative Maude will consider the following expressions as equivalent:

```
s(0) : s(s(0)) : nil
(s(0) : s(s(0))) : nil
s(0) : (s(s(0)) : nil)
```

As another example, let us define an associative list with `nil` as its identity element:

```
sort NatList .
subsort Nat < NatList .
op nil : -> NatList [ctor] .
op _:_ : NatList NatList -> NatList [assoc id: nil] .
```

The operator "_:_" is declared as having `nil` as its *identity element* by means of its attribute `id: nil`. Having an identity element e means that the value of an expression is not dependent on the presence of e's as subexpressions, that is, it is possible to insert e's without changing the meaning of the expression. Thus, if "_:_" is associative and has `nil` as its identity element, Maude will consider the following expressions (and an infinite number of similar ones) as equivalent:

```
s(0) : s(s(0))
nil : s(0) : s(s(0))
s(0) : nil : s(s(0))
s(0) : s(s(0)) : nil
nil : s(0) : nil : s(s(0)) : nil
⋮
```

For that reason, in the canonical representative MAUDE deletes `nil`, unless if it appears alone as an expression. As the last example, let us introduce how we define a multi-set, that is, an associative and commutative list with `nil` as its identity element:

```
sort NatMultiSet .
subsort Nat < NatMultiSet .
op nil : -> NatMultiSet [ctor] .
op _:_ : NatList NatList -> NatList [assoc comm id: nil] .
```

The operator "`_:_`" is declared as *commutative* by means of the attribute `comm`. Commutativity means that the value of an expression is not dependent on the order of its subexpressions, that is, it is possible to change the order of subexpressions without changing the meaning of the expression. Thus, if "`_:_`" is a commutative and associative operator, MAUDE will consider the following expressions equivalent:

```
s(0) : s(s(0)) : s(s(0))
s(s(0)) : s(0) : s(s(0))
s(s(0)) : s(s(0)) : s(0)
```

The structural properties presented are efficiently built in MAUDE. Additional structural properties can be defined by means of equations, as we will see further forward.

Rules and equations. In MAUDE, *rules* or *equations* characterize the behaviour of certain operators, the *defined symbols*. Both language constructions have a similar structure:

```
rl l => r .
eq l = r .
```

`l` and `r` are `terms`, *i.e.*., expressions recursively built by nesting correctly typed operators and variables inside an operator's arguments. `l` is called the left-hand side of a rule or equation, whereas `r` is its right-hand side. Variables can be declared on-the-fly when they are used in an expression with the structure *name*`:`*sort*, or in variable declarations:

```
var N1 N2 ... Nm : S .
```

where `N1`, `N2`, ..., and `Nm` are variables names, and `S` is a sort. Terms form patterns which may represent many *ground terms* (terms without variables). The *pattern* nature of terms allows them to be *matched* with other terms. The *pattern-matching* process between a (pattern) term p and another term t consisting of finding the *substitution* θ from variables in p to terms such that $p\theta = t$, *i.e.*, substitution θ applied to p makes the result equal to t.

Definition 1.4.1 (Rewriting semantics) *The semantics of rewriting a certain term t with a rule or an equation in a rewriting system P is that of replacing a subterm t_{sub} that matches a left-hand side l with substitution θ by $r\theta$, that is, the respective right-hand side r with substitution θ applied. Thus, after rewriting subterm t_{sub} inside term t we get another term t' equal to t except for the rewritten subterm:*

$$t \ \rightarrow_P \ t'$$

If a term to be rewritten does not match the left-hand side of any rule or equation, then the term cannot be further rewritten and is called a canonical form*.*

In Maude, it can be specified that an equation should only be used for rewriting if none of the rest can. To do that, we label (with the same syntax of operators) the equation of interest with the reserved keyword `owise`.

As an example, let us define an equation that represents the *idempotency* structural property of sets:

```
eq X:Nat : X:Nat = X:Nat .
```

If `_:_` is declared as a commutative and associative operator, each time that two identical elements are found in the set, only one is kept.

Chapter 2

The Bes-based Datalog evaluation approach

This chapter summarizes how Datalog queries can be solved by means of Boolean Equation System [And94a] (Bes) resolution. The key idea of our approach is to translate the Datalog specification representing a specific analysis into an implicit Bes, whose resolution corresponds to the execution of the analysis [AFJV09c]. This technique has been implemented in the Datalog solver Datalog_Solve[1] [AFJV09a] that is based on the well-established verification toolbox Cadp [GMLS07], which provides a generic library for local Bes resolution.

A Boolean Equation System is a set of equations defining boolean variables that can be solved with linear-time complexity. Parameterised Boolean Equation System [Mat98] (Pbes) are defined as Bes with typed parameters. Since Pbes are a more compact representation than Bess for a system, we first present an elegant and natural intermediate representation of a Datalog program as a Pbes. Then, we establish a precise correspondence between Datalog query evaluation and Pbes resolution, which is formalized as a linear-time transformation from Datalog to Pbes, and vice-versa.

2.1 From Datalog to Bes.

In the following, we informally illustrate how a Pbes can be obtained from a Datalog program in an automatic way. In Figure 2.1 we introduce a simplified version of the Andersen points-to analysis, previously given in Figure 1.1, that contains four facts and the first two clauses that define the predicate `vP`:

Given the query `:- vP(V,o2).` and the Datalog program shown in Figure 2.1, our transformation constructs the Pbes shown in Figure 2.2, where the boolean variable x_0 and three

[1] `http://www.dsic.upv.es/users/elp/datalog_solve`

```
vP0(p,o1).
vP0(q,o2).
assign(r,q).
assign(w,r).
vP(V,H) :- vP0(V,H).
vP(V,H) :- assign(V,V2), vP(V2,H).
```

Figure 2.1: Datalog (partial) context-insensitive points-to analysis.

parameterised boolean variables (x_{vP0}, x_{assign} and x_{vP}) are defined. Parameters of these boolean variables are defined on a specific domain and may be either variables or constants. The domains in the example are the heap domain ($D_h = \{o1, o2\}$) and the source program variable domain ($D_v = \{p, q, r, w\}$). PBES are evaluated by a least fixpoint computation (μ) that sets the variable x_0 to *true* if there exists a value for V that makes the parameterised variable $x_{vP}(V, o2)$ true. Logical connectives are interpreted as usual.

$$\begin{array}{lcl}
x_0 & \overset{\mu}{=} & \exists V \in D_v \,.\, x_{vP}(V, o2) \\
x_{vP0}(p, o1) & \overset{\mu}{=} & \mathsf{true} \\
x_{vP0}(q, o2) & \overset{\mu}{=} & \mathsf{true} \\
x_{assign}(r, q) & \overset{\mu}{=} & \mathsf{true} \\
x_{assign}(w, r) & \overset{\mu}{=} & \mathsf{true} \\
x_{vP}(V\colon D_v, H\colon D_h) & \overset{\mu}{=} & x_{vP0}(V, H) \vee \exists V2 \in D_v.(x_{assign}(V, V2) \wedge x_{vP}(V2, H))
\end{array}$$

Figure 2.2: PBES representing the points-to analysis in Figure 2.1.

Intuitively, the Datalog query is transformed into the *relevant* variable x_0, i.e., the variable that will guide the PBES resolution. Each Datalog fact is transformed into an *instantiated* parameterised boolean variable (no variables appear in the parameters), whereas each predicate symbol defined by Datalog clauses (different from facts) is transformed into a parameterised boolean variable (in the example $x_{vP}(V\colon D_v, H\colon D_h)$). This parameterised boolean variable is defined as the disjunction of the boolean variables that represent the bodies of the corresponding Datalog clauses. Variables that do not appear in the parameters of the boolean variable are existentially quantified on the specific domain (in the example $\exists V \in D_v$ and $\exists V2 \in D_v$).

Among the different known techniques for solving a PBES (see [DPW08] and the references therein), we consider the resolution method based on transforming the PBES into an alternation-free parameterless boolean equation system (BES) that can be solved by linear time and memory algorithms when data domains are finite [Mat98].

The first step towards the resolution of the analysis is to write the PBES in a simpler format. This simplification step consists of introducing new variables so that each formula at the right-hand side of a boolean equation contains at most one operator. Hence, boolean formulae are restricted to pure disjunctive or conjunctive formulae.

Then, by applying the instantiation algorithm of Mateescu [Mat98], we obtain a parameterless BES where all possible values of each typed data term have been enumerated over their corresponding finite data domains. Actually, we do not explicitly construct the parameterless BES. Instead, an implicit representation of the instantiated BES is defined. This implicit representation is then used by the CADP toolbox to generate the explicit parameterless BES on-the-fly. Intuitively, the construction of the BES can be seen as the resolution of the analysis.

2.2 A complete Datalog to BES transformation

We propose a transformation of the Datalog query into a related logical query, naturally expressed as a parameterised boolean variable of interest and a PBES, which is subsequently evaluated using traditional PBES evaluation techniques. To simplify our description, in the rest of the chapter we intentionally restrict to one both the maximum number of data term parameters $d : D$ that a boolean variable $x \in \mathcal{X}$ can have, and the arity of predicate symbols.

Firstly, let us formalize the syntax and semantics of Datalog in an appropriate way for our setting.

Definition 2.2.1 (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* rule *r, also called* clause, *defined over a finite alphabet $P \subseteq \mathcal{P}$ and arguments from $V \cup C$, $V \subseteq \mathcal{V}$, $C \subseteq \mathcal{C}$, has the following syntax:*

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

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

The *Herbrand Universe* of a Datalog program R defined over P, V and C, denoted U_R, is the finite set of all ground arguments, *i.e.*, constants of C. The *Herbrand Base* of R, denoted B_R, is the finite set of all ground atoms that can be built by assigning elements of U_R to the predicate symbols in P. A *Herbrand Interpretation* of R, denoted I (from a set $\mathcal{I}$ of Herbrand interpretations, $\mathcal{I} \subseteq B_R$), is a set of ground atoms.

Definition 2.2.2 (Fixed point semantics) *Let R be a Datalog program. The least* Herbrand model *of R is a Herbrand interpretation I of R defined as the least fixed point of a* *monotonic, continuous operator* $T_R : \mathcal{I} \rightarrow \mathcal{I}$ *known as the* immediate consequences operator *and defined by:*

$$\begin{array}{rcl} T_R(I) & = & \{h \in B_R \mid \ h : -b_1, ..., b_m \textit{ is a ground instance of a rule in } R, \\ & & \textit{with } \ b_i \in I, i = 1..m, m \geq 0\} \end{array}$$

The number of Herbrand models being finite for a Datalog program R, there always exists a least fixed point for T_R, denoted μT_R, which is the least Herbrand model of R. In practice, one is generally interested in the computation of some specific atoms, called *queries*, and not in the whole database of atoms. Hence, queries may be used to prevent the computation of facts that are irrelevant for the atoms of interest, *i.e.*, facts that are not derived from the query.

Definition 2.2.3 (Query Evaluation) *A* Datalog query q *is a pair* $\langle G, R \rangle$ *where:*

- R *is a Datalog program defined over* P, V *and* C,
- G *is a set of* goals.

Given a query q*, its* evaluation *consists in computing* $\mu T_{\{q\}}$, $\{q\}$ *being the extension of the Datalog program* R *with the Datalog rules in* G.

The evaluation of a Datalog program augmented with a set of goals deduces all the different constant combinations that, when assigned to the variables in the goals, can make one of the goal clauses true,*i.e.*, all atoms b_i in its body are satisfied.

Proposition 2.2.4 *Let* $q = \langle G, R \rangle$ *be a* Datalog *query, defined over* P, V *and* C*, and* $B_q = (x_0, M_1)$*, with* $\sigma_1 = \mu$*, be a* PBES *defined over a set* $\mathcal{X}$ *of boolean variables* x_p *in one-to-one correspondence with predicate symbols* p *of* P *plus a special variable* x_0*, a set* $\mathcal{D}$ *of data terms in one-to-one correspondence with variable and constant symbols of* $V \cup C$*, and* M_1 *the block containing exactly the following equations, where fresh variables are existentially quantified after the transformation:*

$$x_0 \overset{\mu}{=} \bigvee_{:-\ q_1(d_1),\ ...,\ q_m(d_m).\ \in G} \ \bigwedge_{i:=1}^{m} x_{q_i}(d_i) \tag{2.2.1}$$

$$\{x_p(d : D) \overset{\mu}{=} \bigvee_{p(d)\ :-\ p_1(d_1),...\ p_m(d_m).\ \in R} \ \bigwedge_{i:=1}^{m} x_{p_i}(d_i) \mid p \in P\} \tag{2.2.2}$$

Then q *is satisfiable if and only if* $[\![B_q]\!]\delta(x_0) = \mathsf{true}$.

The boolean variable x_0 encodes the set of Datalog goals G, whereas the (parameterized) boolean variables $x_p(d : D)$ represent the set of Datalog rules R modulo renaming.

In our framework, the reverse direction of reducibility consists in the transformation of a parameterised boolean variable of interest, defined in a PBES, into a related relation of interest expressed as a Datalog query, which could be evaluated using traditional Datalog evaluation techniques.

Proposition 2.2.5 *Let $B = (x_0, M_1)$, with $\sigma_1 = \mu$, be a* PBES *defined over a set $\mathcal{X}$ of boolean variables and a set $\mathcal{D}$ of data terms, and $q_B = \langle G, R\rangle$ be a* Datalog *query defined over a set P of predicate symbols p in one-to-one correspondence with boolean variables x_p of $\mathcal{X} \setminus \{x_0\}$, a set $V \cup C$ of variable and constant symbols in one-to-one correspondence with data terms of D, and $\langle G, R\rangle$ containing exactly the following* Datalog *rules:*

$$G = \left\{ \begin{array}{rl} :- & q_{1,1}(d_{1,1}),\ \ldots\ , q_{1,n_1}(d_{1,n_1}).,\\ & \vdots \\ :- & q_{m_0,1}(d_{m_0,1}),\ \ldots\ , q_{m_0,n_{m_0}}(d_{m_0,n_{m_0}}). \end{array} \right| \left. x_0 \overset{\mu}{=} \bigvee_{i=1}^{m_0} \bigwedge_{j=1}^{n_i} x_{q_{i,j}}(d_{i,j}) \in M_1 \right\}$$

$$R = \left\{ \begin{array}{rl} p(d) :- & p_{1,1}(d_{1,1}),\ \ldots\ , p_{1,n_1}(d_{1,n_1}).,\\ & \vdots \\ p(d) :- & p_{m_p,1}(d_{m_p,1}),\ \ldots\ , p_{m_p,n_{m_p}}(d_{m_p,n_{m_p}}). \end{array} \right| \left. x_p(d) \overset{\mu}{=} \bigvee_{i=1}^{m_p} \bigwedge_{j=1}^{n_i} x_{p_{i,j}}(d_{i,j}) \in M_1 \right\}$$

Then $[\![B]\!]\delta(x_0) = \mathsf{true}$ if and only if $q_B = \langle G, R\rangle$ is satisfiable.

Example 2.2.6 We illustrate the reduction method from Datalog to PBES by means of our running example. Let $q = \langle G, R\rangle$ be the following Datalog query with domains $D_h = \{o1, o2\}$) and $D_v = \{p, q, r, w\}$:

```
:- vP (V, o2).
vP0(p,o1).
vP0(q,o2).
assign(r,q).
assign(w,r).
vP(V,H) :- vP0(V,H).
vP(V,H) :- assign(V,V2), vP(V2,H).
```

By using Proposition 2.2.4, we obtain the following PBES:

$$\begin{array}{lcl} x_0 & \overset{\mu}{=} & \exists V \in D_v \,.\, x_{vP}(V, o2) \\ x_{vP0}(p, o1) & \overset{\mu}{=} & \mathsf{true} \\ x_{vP0}(q, o2) & \overset{\mu}{=} & \mathsf{true} \\ x_{assign}(r, q) & \overset{\mu}{=} & \mathsf{true} \\ x_{assign}(w, r) & \overset{\mu}{=} & \mathsf{true} \\ x_{vP}(V\colon D_v, H\colon D_h) & \overset{\mu}{=} & x_{vP0}(V, H) \vee \exists V2 \in D_v.(x_{assign}(V, V2) \wedge x_{vP}(V2, H)) \end{array}$$

■

In the rest of the chapter, we will develop our methodology for using PBESs in order to solve Datalog queries.

2.2.1 Instantiation to parameterless BES

Among the different known techniques for solving a PBES [DPW08], such as Gauss elimination with symbolic approximation, and use of patterns, under/over approximations, or invariants, we consider the resolution method based on transforming the PBES into an alternation-free parameterless boolean equation system (BES) that can be solved by linear time and memory algorithms [Mat98, DPW08] when data domains are finite.

Definition 2.2.7 (Boolean Equation System) *A* Boolean Equation System *(*BES*)* $B = (x_0, M_1, ..., M_n)$ *is a* PBES *where data domains are removed and boolean variables, being independent from data parameters, are considered to be* propositional*.*

To obtain a direct transformation into a parameterless BES, we first described the PBES in a simpler format. This simplification step consists in introducing new variables, such that each formula at the right-hand side of a boolean equation only contains at most one operator. Hence, boolean formulae are restricted to pure disjunctive or conjunctive formulae.

Given a Datalog query $q = \langle G, R \rangle$, by applying this simplification to the PBES of Proposition 2.2.4, we obtain the following PBES:

$$\begin{aligned}
x_0 &\stackrel{\mu}{=} \bigvee_{:-\ q_1(d_1),\ldots,q_m(d_m).\ \in G} g_{q_1(d_1),\ldots,q_m(d_m)} \\
g_{q_1(d_1),\ldots,q_m(d_m)} &\stackrel{\mu}{=} \bigwedge_{i:=1}^{m} x_{q_i}(d_i) \\
x_p(d : D) &\stackrel{\mu}{=} \bigvee_{p(d)\ :-\ p_1(d_1),\ldots,p_m(d_m).\ \in R} r_{p_1(d_1),\ldots,p_m(d_m)} \\
r_{p_1(d_1),\ldots,p_m(d_m)} &\stackrel{\mu}{=} \bigwedge_{i:=1}^{m} x_{p_i}(d_i)
\end{aligned}$$

By applying the instantiation algorithm of Mateescu [Mat98], we eventually obtain a parameterless BES, where all possible values of each typed data terms have been enumerated over their corresponding finite data domains.

The resulting implicit parameterless BES is defined as follows, where $\preceq$ is the standard preorder of relative generality (instantiation ordering).

$$x_0 \stackrel{\mu}{=} \bigvee_{:-\ q_1(d_1),\dots,q_m(d_m).\ \in G} g_{q_1(d_1),\dots,q_m(d_m)} \tag{2.2.3}$$

$$g_{q_1(d_1),\dots,q_m(d_m)} \stackrel{\mu}{=} \bigvee_{1 \le i \le m,\, e_i \in D_i \wedge d_i \preceq e_i} g^i_{q_1(e_1),\dots,q_m(e_m)} \tag{2.2.4}$$

$$g^i_{q_1(e_1),\dots,q_m(e_m)} \stackrel{\mu}{=} \bigwedge_{i:=1}^{m} x_{q_i(e_i)} \tag{2.2.5}$$

$$x_{p(d)} \stackrel{\mu}{=} \bigvee_{p(d)\ :-\ p_1(d_1),\dots,p_m(d_m).\ \in R} r_{p_1(d_1),\dots,p_m(d_m)} \tag{2.2.6}$$

$$r_{p_1(d_1),\dots,p_m(d_m)} \stackrel{\mu}{=} \bigvee_{1 \le i \le m,\, e_i \in D_i \wedge d_i \preceq e_i} r^i_{p_1(e_1),\dots,p_m(e_m)} \tag{2.2.7}$$

$$r^i_{p_1(e_1),\dots,p_m(e_m)} \stackrel{\mu}{=} \bigwedge_{i:=1}^{m} x_{p_i(e_i)} \tag{2.2.8}$$

Observe that Equation 2.2.1 is transformed into a set of parameterless equations (2.2.3, 2.2.4, 2.2.5). First, Equation 2.2.3 describes the set of parameterised goals $g_{q_1(d_1),\dots,q_m(d_m)}$ of the query. Then, Equation 2.2.4 represents the instantiation of each variable parameter d_i to the possible values e_i from the domain. Finally, Equation 2.2.5 states that each instantiated goal $g^i_{q_1(e_1),\dots,q_m(e_m)}$ is satisfied whenever the values e_i make all predicates q_i of the goal true. Similarly, Equation 2.2.2 (describing Datalog rules) is encoded into a set of parameterless equations (2.2.6, 2.2.7, 2.2.8).

Example 2.2.8 Let us instantiate the PBES obtained in Example 2.2.6. By applying Mateescu instantiation algorithm, we obtain the following BES:

$$\begin{aligned}
x_0 &\stackrel{\mu}{=} g_{vP(V,o2)} \\
g_{vP(V,o2)} &\stackrel{\mu}{=} \bigvee g^i_{vP}(p,o2) \ \vee\ g^i_{vP}(q,o2) \ \vee\ g^i_{vP}(r,o2) \\
&\quad \vee\ g^i_{vP}(w,o2) \\
g^i_{vP}(p,o2) &\stackrel{\mu}{=} x_{vP(p,o2)} \\
x_{vP(p,o2)} &\stackrel{\mu}{=} \bigvee r_{vP0(p,o2)} \ \vee\ r_{assign(p,V2),vP(V2,o2)} \\
r_{vP0(p,o2)} &\stackrel{\mu}{=} r^i_{vP0(p,o2)} \\
r^i_{vP0(p,o2)} &\stackrel{\mu}{=} x_{vP0(p,o2)} \\
x_{vP0(p,o2)} &\stackrel{\mu}{=} \mathsf{false} \\
r_{assign(p,V2),vP(V2,o2)} &\stackrel{\mu}{=} \bigvee r^i_{assign(p,p),vP(p,o2)} \ \vee\ r^i_{assign(p,q),vP(q,o2)} \\
&\quad \vee\ r^i_{assign(p,r),vP(r,o2)} \ \vee\ r^i_{assign(p,w),vP(w,o2)}
\end{aligned} \tag{2.2.9}$$

$$
\begin{array}{rcl}
r^i_{assign(p,p),vP(p,o2)} & \overset{\mu}{=} & \bigwedge x_{assign(p,p)} \ \wedge \ x_{vP(p,o2)} \\
x_{assign(p,p)} & \overset{\mu}{=} & \mathsf{false} \\
r^i_{assign(p,q),vP(q,o2)} & \overset{\mu}{=} & \bigwedge x_{assign(p,q)} \ \wedge \ x_{vP(q,o2)} \\
x_{assign(p,q)} & \overset{\mu}{=} & \mathsf{false} \\
r^i_{assign(p,r),vP(r,o2)} & \overset{\mu}{=} & \bigwedge x_{assign(p,r)} \ \wedge \ x_{vP(r,o2)} \\
x_{assign(p,r)} & \overset{\mu}{=} & \mathsf{false} \\
r^i_{assign(p,w),vP(w,o2)} & \overset{\mu}{=} & \bigwedge x_{assign(p,w)} \ \wedge \ x_{vP(w,o2)} \\
x_{assign(p,w)} & \overset{\mu}{=} & \mathsf{false} \\
g^i_{vP}(q,o2) & \overset{\mu}{=} & x_{vP(q,o2)} \\
x_{vP(q,o2)} & \overset{\mu}{=} & \bigvee r_{vP0(q,o2)} \ \vee \ r_{assign(q,V2),vP(V2,o2)} \\
r_{vP0(q,o2)} & \overset{\mu}{=} & r^i_{vP0(q,o2)} \\
r^i_{vP0(q,o2)} & \overset{\mu}{=} & x_{vP0(q,o2)} \\
x_{vP0(q,o2)} & \overset{\mu}{=} & r_{\varnothing} \\
r_{\varnothing} & \overset{\mu}{=} & \mathsf{true} \\
g^i_{vP}(r,o2) & \overset{\mu}{=} & x_{vP(r,o2)} \\
x_{vP(r,o2)} & \overset{\mu}{=} & \bigvee r_{vP0(r,o2)} \ \vee \ r_{assign(r,V2),vP(V2,o2)} \\
r_{vP0(r,o2)} & \overset{\mu}{=} & r^i_{vP0(r,o2)} \\
r^i_{vP0(r,o2)} & \overset{\mu}{=} & x_{vP0(r,o2)} \\
x_{vP0(r,o2)} & \overset{\mu}{=} & \mathsf{false} \\
r_{assign(r,V2),vP(V2,o2)} & \overset{\mu}{=} & \bigvee r^i_{assign(r,p),vP(p,o2)} \ \vee \ r^i_{assign(r,q),vP(q,o2)} \\
 & & \vee \ r^i_{assign(r,r),vP(r,o2)} \ \vee \ r^i_{assign(r,w),vP(w,o2)} \\
r^i_{assign(r,p),vP(p,o2)} & \overset{\mu}{=} & \bigwedge x_{assign(r,p)} \ \wedge \ x_{vP(p,o2)} \\
x_{assign(r,p)} & \overset{\mu}{=} & \mathsf{false} \\
r^i_{assign(r,q),vP(q,o2)} & \overset{\mu}{=} & \bigwedge x_{assign(r,q)} \ \wedge \ x_{vP(q,o2)} \\
x_{assign(r,q)} & \overset{\mu}{=} & r_{\varnothing} \\
g^i_{vP}(w,o2) & \overset{\mu}{=} & \ldots \\
 & \vdots &
\end{array}
$$

■

Optimizations. The parameterless BES described above is rather inefficient since it adopts a brute-force approach that, at the very first steps of the computation (Equation 2.2.4), enumerates all possible tuples of the query (see Equation 2.2.9 in Example 2.2.8). It is well-known that a general Datalog program runs in $O(n^k)$ time, where k is the largest number of variables in any single rule, and n is the number of constants in the facts and rules. Similarly, for a simple query like `:- vP(V,H).`, with `V` and `H` respectively being elements of domains D_v and D_h, each one of size 10 000, Equation 2.2.4 will generate D^2, *i.e.*, 10^8, boolean variables representing all possible combinations of values `V` and `H` in the relation `vP`. Usually, for each atom in a Datalog program, the number of facts that are given or inferred by the Datalog rules is much lower than the product of the $domain's\ sizes$ of its corresponding arguments. Ideally, the Datalog query evaluation should enumerate (given or inferred) facts only *on-demand.*

Among the existing optimizations for top-down evaluation of Datalog queries, the so-called *Query-Sub-Query* [Vie86] technique consists in minimizing the number of tuples derived by a rewriting of the program based on the propagation of bindings. Basically, the method aims at keeping the bindings of variables between atoms $p(a)$ in a rule. In our Datalog evaluation technique based on BES, we adopt a similar approach: two boolean equations (Equations 2.2.4 and 2.2.7 slightly modified) only enumerate the values of variable arguments that appear more than once in the body of the corresponding Datalog rule; otherwise, arguments are kept unchanged. Moreover, if the atom $p(a)$ is part of the extensional database, the only possible values of its variable arguments are values that reproduce a given fact of the Datalog program. We denote as D_i^p the subdomain of D that contains all possible values of the i^{th} variable argument of p if p is in the extensional database, otherwise $D_i^p = D$. Hence, the resulting BES resolution is likely to process fewer facts and be more efficient than the brute-force approach.

Following this optimization technique, a parameterless BES can directly be derived from the previous BES representation which we define as follows:

$$x_0 \overset{\mu}{=} \bigvee_{:-\ q_1(d_1),\ldots,q_m(d_m).\ \in G} g_{q_1(d_1),\ldots,q_m(d_m)} \tag{2.2.10}$$

$$g_{q_1(d_1),\ldots,q_m(d_m)} \overset{\mu}{=} \bigvee_{\substack{\{a_1,\ \ldots,\ a_m\}\in(\{V\cup D_1^{q_1}\}\times\ldots\times\{V\cup D_1^{q_m}\})\ \mid \\ if\ (\exists\ j\in[1..m],\ j\neq i\ \mid\ d_i=d_j\wedge d_i\in V) \\ then\ a_i\in D_1^{q_i}\ \wedge\ (\forall\ j\in[1..m],\ d_i=d_j\ \mid\ a_j:=a_i)\ else\ a_i:=d_i}} g^{pi}_{q_1(a_1),\ldots,q_m(a_m)} \tag{2.2.11}$$

$$g^{pi}_{q_1(a_1),\ldots,q_m(a_m)} \overset{\mu}{=} \bigwedge_{i:=1}^{m} x_{q_i(a_i)} \tag{2.2.12}$$

$$x_{q(a)} \overset{\mu}{=} x^f_{q(a)} \vee x^c_{q(a)} \tag{2.2.13}$$

$$x^f_{q(a)} \overset{\mu}{=} \bigvee_{(e:=a\ \wedge\ a\in C)\ \vee\ (e\in D_1^q\ \wedge\ a\in V)\ \mid\ q(e).\in R} x^i_{q(e)} \tag{2.2.14}$$

$$x^i_{q(e)} \stackrel{\mu}{=} \mathsf{true} \tag{2.2.15}$$

$$x^c_{p(a)} \stackrel{\mu}{=} \bigvee_{p(a)\ :-\ p_1(d_1),\ldots,p_m(d_m).\ \in R} r_{p_1(d_1),\ldots,p_m(d_m)} \tag{2.2.16}$$

$$r_{p_1(d_1),\ldots,p_m(d_m)} \stackrel{\mu}{=} \bigvee_{\substack{\{a_1,\ \ldots,\ a_m\}\in(\{V\cup D_1^{p1}\}\times\ldots\times\{V\cup D_1^{pm}\})\ \mid \\ \mathit{if}\ (\exists\ j\in[1..m],\ j\neq i\ \mid\ d_i=d_j\wedge d_i\in V) \\ \mathit{then}\ a_i\in D_1^{p_i}\ \wedge\ (\forall\ j\in[1..m],\ d_i=d_j\ \mid\ a_j:=a_i)\ \mathit{else}\ a_i:=d_i}} r^{pi}_{p_1(a_1),\ldots,p_m(a_m)} \tag{2.2.17}$$

$$r^{pi}_{p_1(a_1),\ldots,p_m(a_m)} \stackrel{\mu}{=} \bigwedge_{i:=1}^{m} x_{p_i(a_i)} \tag{2.2.18}$$

Boolean variables whose name starts with x are those that correspond to the goal and subgoals of the original program and we will call them *original* variables, whereas boolean variables starting with r or g are auxiliary variables that are defined during unfolding and instantiation of (sub)goals. Observe that Equations 2.2.10, 2.2.12, 2.2.16 and 2.2.17 respectively correspond to Equations 2.2.3, 2.2.5, 2.2.6 and 2.2.8 of the previous Bes definition with only a slight renaming of generated boolean variables. The important novelty is that, instead of enumerating all possible values of the domain, as it is done in Equation 2.2.4, the corresponding new Equation 2.2.11 only enumerates the values of variable arguments that are repeated in the body of a rule; otherwise, variable arguments are kept unchanged *i.e.*, $a_i := d_i$. Actually, the generated boolean variables $g^{pi}_{q_1(a_1),\ldots,q_m(a_m)}$, where pi stands for *partially instantiated*, may still refer to atoms containing variable arguments. Thus, the combinatorial explosion of possible tuples is avoided at this point and delayed to future steps. Equation 2.2.13 generates two boolean successors for variable $x_{q(a)}$: $x^f_{q(a)}$ when q is a relation that is part of the extensional database, and $x^c_{q(a)}$ when q is defined by Datalog clauses. In Equation 2.2.14, each value of a variable or constant that leads to a given fact $q(e)$ of the program generates a new boolean variable $x^i_{q(e)}$, where i stands for (fully) *instantiated*, that is true by definition of a fact. Equation 2.2.16 infers Datalog rules whose head is p_a. Note that Equations 2.2.11, 2.2.14, and 2.2.17 enumerate possible values of subdomains $D_1^{p_i}$ instead of full domain D. For the Datalog program described in Figure 2.1, this restriction would consist in using four new subdomains ${D_{v}}_1^{vP0} = \{p, q\}$, ${D_{h}}_2^{vP0} = \{o1, o2\}$, ${D_{v}}_1^{assign} = \{r, w\}$, and ${D_{v}}_2^{assign} = \{q, r\}$, instead of full domains D_h and D_v for the values of each variable argument in relations $vP0$ and $assign$.

Example 2.2.9 To illustrate the idea behind this optimized version of the generated Bes we show (a part of) the Bes that results from our running example.

$$
\begin{array}{lcl}
x_0 & \overset{\mu}{=} & g_{vP(V,o2)} \\
g_{vp(V,o2)} & \overset{\mu}{=} & g^{pi}_{vP(V,o2)} \\
g^{pi}_{vP(V,o2)} & \overset{\mu}{=} & x_{vP(V,o2)} \\
x_{vP(V,o2)} & \overset{\mu}{=} & x^{f}_{vP(V,o2)} \vee x^{c}_{vP(V,o2)} \\
 & \vdots & \\
x^{c}_{vP(V,H)} & \overset{\mu}{=} & r_{vP0(V,H)} \vee r_{assign(V,V2),vP(V2,H)} \\
r_{vP0(V,H)} & \overset{\mu}{=} & r^{pi}_{vP0(V,H)} \\
 & \vdots & \\
r^{pi}_{vP0(V,H)} & \overset{\mu}{=} & x_{vP0_{V,H}} \\
x_{vP0_{V,H}} & \overset{\mu}{=} & x^{f}_{vP0_{V,H}} \vee x^{r}_{vP0_{V,H}} \\
x^{f}_{vP0_{V,H}} & \overset{\mu}{=} & x^{c}_{vP0_{p,o1}} \vee x^{c}_{vP0_{q,o2}} \\
x^{c}_{vP0_{p,o1}} & \overset{\mu}{=} & \mathsf{true} \\
x^{c}_{vP0_{q,o2}} & \overset{\mu}{=} & \mathsf{true} \\
r_{assign(V,V2),vP(V2,H)} & \overset{\mu}{=} & r^{pi}_{assign(V,q),vP(q,H)} \vee r^{pi}_{assign(V,r),vP(r,H)} \\
r^{pi}_{assign(V,q),vP(q,H)} & \overset{\mu}{=} & x_{assign(V,q)} \wedge x_{vP(q,H)} \\
x_{assign(V,q)} & \overset{\mu}{=} & x^{f}_{assign(V,q)} \vee x^{c}_{assign(V,q)} \\
x^{f}_{assign(V,q)} & \overset{\mu}{=} & x^{i}_{assign(r,q)} \\
x^{i}_{assign(r,q)} & \overset{\mu}{=} & \mathsf{true} \\
 & \vdots &
\end{array}
$$

Each *original* variable is defined as the disjunction of r (or g) boolean variables that represent the body of one of the clauses that define the corresponding predicate (see equation for variable $x^{c}_{vP(V,H)}$). Then, each r (or g) variable is defined as the disjunction of the different possible instantiations of the query on the shared variables (see equation for variable $r_{assign(V,V2),vP(V2,H)}$). These *partial instantiations* are represented by r^{pi} (or g^{pi}) boolean variables. The r^{pi} variables are defined as the conjunction of the subqueries, which are represented by (*original*) x variables. Finally, the original variables x are defined as the disjunction of the boolean variables that correspond to querying the *facts* x^{f} and querying the *clauses* x^{c} (see equation for $x_{assign(V,q)}$).

As stated above, when the r^{pi} variables are generated, only variables that are shared by two or more subgoals in the body of the Datalog program are instantiated, and only values that appear in the corresponding parameters of the program facts are used. In other words, we do not generate spurious variables, such as $r^{pi}_{assign(V,w),vP(w,H)}$, which can never be true. ■

2.2.2 Solution extraction

Considering the optimized parameterless Bes defined above, the query satisfiability problem is reduced to the local resolution of boolean variable x_0. The value (true or false) computed for x_0 indicates whether there exists at least one satisfiable goal in G. We can remark that the Bes representing the evaluation of a Datalog query is only composed of one equation block that contains alternating dependencies between disjunctive and conjunctive variables. Hence, it can be solved by optimized depth-first search (DFS) for such a type of equation block. However, since the DFS strategy can only conclude the existence of a solution to the query by computing a minimal number of boolean variables, it is necessary to use a breadth-first search (BFS) strategy to compute all the different solutions to a Datalog query. Such a strategy will "force" the resolution of all boolean variables that have been put in the BFS queue, even if the satisfiability of the query has been computed in the meantime. Consequently, the solver will compute all possible boolean variables $x^i_{q(e)}$, which are potential solutions for the query. Upon termination of the Bes resolution (ensured by finite data domains and table-based exploration), query solutions, *i.e.*, combinations of variable values $\{e_1, \ldots, e_m\}$, one for each atom of the query that lead to a satisfied query, are extracted from all boolean variables $x^i_{q(e)}$ that are reachable from boolean variable x_0 through a path of true boolean variables.

2.3 The prototype Datalog_Solve

We implemented the Datalog query transformation to Bes in a powerful, fully automated Datalog solver tool, called Datalog_Solve, developed within the Cadp verification toolbox. Without loss of generality, in this section, we describe the Datalog_Solve tool focusing on Java program analysis. Other source languages and classes of problems can be specified in Datalog and solved by our tool as well.

Datalog_Solve takes three different inputs (see Figure 2.3): the (optional) domain definitions (`.map`), the Datalog constraints or *facts* (`.tuples`), and a Datalog query $q = \langle G, R \rangle$ (`.datalog`, *e.g.* `pa.datalog` in Figure 2.4). The domain definitions state the possible values for each predicate's argument of the query. These are meaningful names for the numerical values that are used to efficiently described the Datalog constraints. For example, in the context of pointer analyses, variable names (`var.map`) and heap locations (`heap.map`) are two domains of interest. Each line of a `.map` file represents a different domain element. For efficiency reasons, a domain element is identified by its line number, thus its human-readable description is provided by the content of its `.map` file's associated line. The Datalog

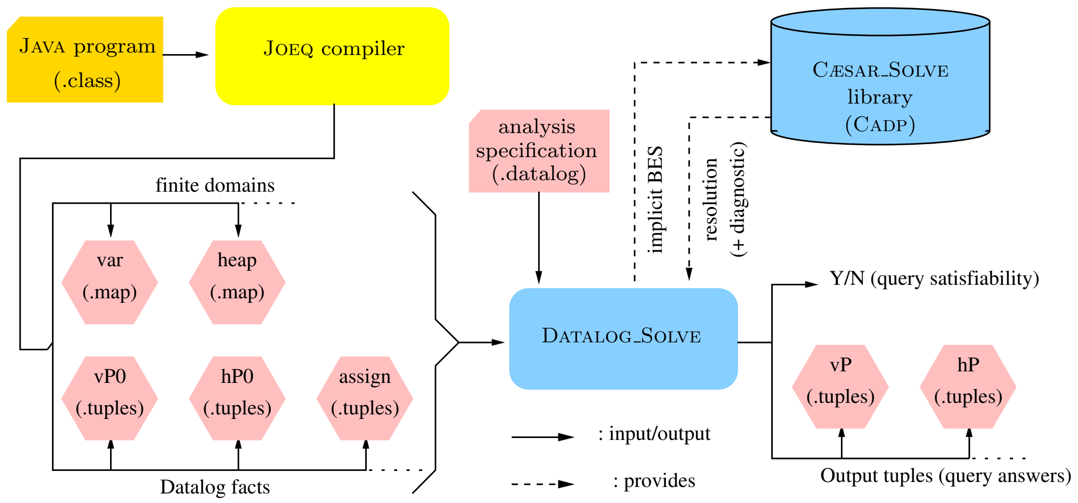

Figure 2.3: JAVA program analysis using the DATALOG_SOLVE tool.

constraints represent information relevant for the analysis. For instance, `vP0.tuples` gives all direct references from variables to heap objects in a given program. These combinations are described by numerical values in the range 0..(*domain size* − 1), which represents domain elements identifiers thanks to the use of the `.map` files.

```
### Domains
V 262144 variable.map
H 65536 heap.map
F 16384 field.map
### Relations
vP_0      (variable :  V, heap :  H)            inputtuples
store     (base :  V, field :  F, source :  V)  inputtuples
load      (base :  V, field :  F, dest :  V)    inputtuples
assign    (dest :  V, source :  V)              inputtuples
vP        (variable :  V, heap :  H)            outputtuples
hP        (base :  H, field :  F, target :  H)  outputtuples
### Rules
vP (V1, H1)     :- vP_0(V1, H1).
vP (V1, H1)     :- assign(V1, V2), vP(V2, H2).
hP (H1, F1, H2) :- store(V1, F1, V2), vP(V1, H1), vP(V2, H2).
vP (V2, H2)     :- load (V1, F1, V2), vP(V1, H1), hP(H1, F1, H2).
```

Figure 2.4: DATALOG_SOLVE input file specifying Andersen's points-to analysis.

Both, domain definitions and facts are specified in the `.datalog` input file (see Figure 2.4) and they are automatically extracted from program source code by using the JOEQ compiler framework [Wha03] that we slightly modified to generate *tuple-based* instead of BDD-*based* input relations. The `.datalog` input file has three sections separated by its corresponding headers:

Domains Declares a domain on each line by means of three consecutive fields: the domain identifier, the domain size, and the domain `.map` file.

Relations Declares the predicate symbols used in the program by means of their identifiers, the association of their arguments to previously declared domains, and stating whether they are part of the extensional (`inputtuples`) or the intensional (`outputtuples`) databases. If a predicate p is declared as extensional, a file named p`.tuples` will be used to load the facts associated with p.

Rules States the rules which specify the analysis to be performed.

Datalog_Solve 1.0 (120 lines of Lex, 380 lines of Bison and 3 500 lines of C code) proceeds in two steps:

1. The front-end of Datalog_Solve constructs the optimized implicit Bes representation given by Equations 2.2.10-2.2.18 from the inputs.
2. The back-end of our tool carries out the demand-driven generation, resolution and interpretation of the Bes by means of the generic Cæsar_Solve library of Cadp, devised for local Bes resolution and diagnostic generation.

This architecture clearly separates the implementation of Datalog-based static analyses from the resolution engine, which can be extended and optimized independently. We will further discuss some optimizations in Section 2.4.

Upon termination (ensured by safe input Datalog programs), Datalog_Solve returns both the query's satisfiability and the computed answers represented in various output files (`.tuples` files) numerically. The tool takes as a default query the computation of the least set of facts that contains all the facts that can be inferred using the given rules. This represents the worst case of a demand-driven evaluation and computes all the information derivable from the considered Datalog program.

2.4 Experimental results

The Datalog_Solve tool was applied to a number of Java programs by computing the context-insensitive pointer analysis described in Figure 2.4.

Table 2.1: Description of the Java projects used as benchmarks.

Name	Description	Classes	Methods	Vars	Allocs
freetts (1.2.1)	speech synthesis system	215	723	8K	3K
nfcchat (1.1.0)	scalable, distributed chat client	283	993	11K	3K
jetty (6.1.10)	server and servlet container	309	1160	12K	3K
joone (2.0.0)	Java neural net framework	375	1531	17K	4K

To test the scalability and applicability of the transformation, we applied our technique to four of the most popular 100% Java projects on Sourceforge that could compile directly as standalone applications and were previously used as benchmarks for the Bddbddb tool [WACL05]. They are all real applications with tens of thousands of users each. Projects vary in the number of classes, methods, variables, and heap allocations. The information details, shown on Table 2.1, are calculated on the basis of a context-insensitive callgraph precomputed by the Joeq[2] compiler. All experiments were conducted using Java JRE 1.5, Joeq version 20030812, on a Intel Core 2 T5500 1.66GHz with 3 Gigabytes of RAM, running Linux Kubuntu 8.04.

Table 2.2: Times (in seconds) and peak memory usages (in megabytes) for each benchmark and context-insensitive pointer analysis.

Name	time (sec.)	memory (Mb.)
freetts (1.2.1)	10	61
nfcchat (1.1.0)	8	59
jetty (6.1.10)	73	70
joone (2.0.0)	4	58

The analysis time and memory usage of our context insensitive pointer analysis, shown on Table 2.2, illustrate the scalability of our Bes resolution and validate our theoretical results on real examples. Datalog_Solve solves the (default) query for all benchmarks in a few seconds. The computed results were verified by comparing them with the solutions computed by the Bddbddb tool on the same benchmark of Java programs and analysis.

Further Improvements. Recently, a new Bes-based approach for the resolution of Datalog programs has been developed by the author, in joint work with Christophe Joubert and Fernando Tarín [FJT10b, FJT10a]. Our contribution is a novel bottom-up evaluation strategy

[2]`http://joeq.sourceforge.net/`

specially tailored for Datalog-based program analysis. Our work is based on the evaluation strategy presented by Liu and Stoller in their PPDP'2003 paper, and further detailed in their TOPLAS article [LS09]. Their strategy is a generalization of the systematic algorithm development method of Paige et al. [PK82], which transforms extensive set computations like set union, intersection, and difference into incremental operations. Incremental operations are supported by sophisticated data structures with constant access time. They derive an imperative resolution algorithm, which computes a fixpoint over all (preformatted) rules from an input Datalog program by first considering input predicates, then considering rules with one subgoal, and finally considering rules with two subgoals.

Our novel proposal is to enhance this evaluation strategy by means of the following:

1. A declarative description of the bottom-up resolution strategy that is separate from the fixpoint computation. This is achieved by transforming Datalog programs to *Boolean Equation Systems* (BESs) and evaluating the resulting BESs by standard solvers.

2. A predicate order that is employed to simplify the BES by removing various set operations. This order is determined by the dependency between predicate symbols and the number of times that rules are fired.

3. A sophisticated data-structure with quicker access time and lower memory consumption. This efficient data-structure is based on a complex representation of a *trie*. *Tries*, also called prefix trees, are ordered tree data structures where each node position in the tree is the key that is associated to this node. This structure has faster look-up keys than binary search trees and imperfect hash tables.

We endowed the DATALOG_SOLVE prototype with the new evaluation strategy applied to the evaluation of Andersen's points-to analysis encoded as a BES. DATALOG_SOLVE 2.0 does not depend on CADP, and uses a simple and fast specific BES solver. Facts are extracted by an extended version of SOOT [3] from the JAVA programs of the DACAPO [4] benchmark with JDK 1.6. We tested the efficiency and feasibility of our implementation by comparing it to two state-of-the-art Datalog solvers XSB 3.2[5] and the prototype of Liu and Stoller[6], which in the rest of the chapter we will call TOPLAS. In Figures 2.5 and 2.6, performance results are presented in terms of evaluation user time and peak memory consumption. All experiments were performed on an Intel Core 2 duo E4500 2.2 GHz, with 2048 KB cache, 4 GB of RAM, and running Linux Ubuntu 10.04. DATALOG_SOLVE and XSB solver were compiled using gcc

[3]http://www.sable.mcgill.ca/soot
[4]http://voxel.dl.sourceforge.net/sourceforge/dacapobench/dacapo-2006-10-MR2-xdeps.zip
[5]http://xsb.sourceforge.net
[6]Provided by the authors of [LS09]

4.4.1. Python 2.6.4 was used for the Toplas solver. The measures do not include the time needed by Xsb and Toplas to precompile the facts. The analysis results were verified by comparing the outputs of all solvers.

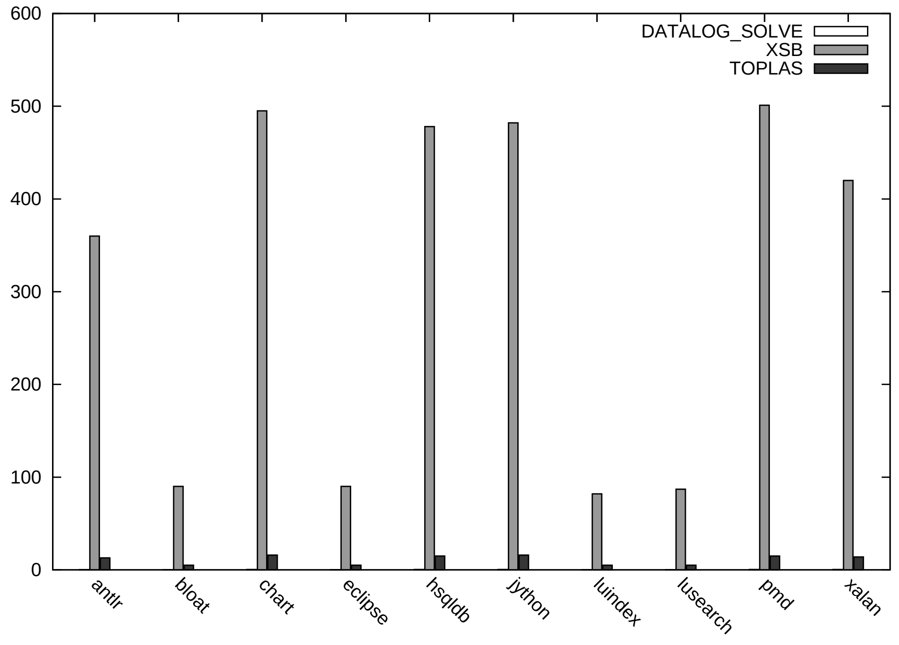

Figure 2.5: Dacapo analysis times (sec.) of various datalog implementations.

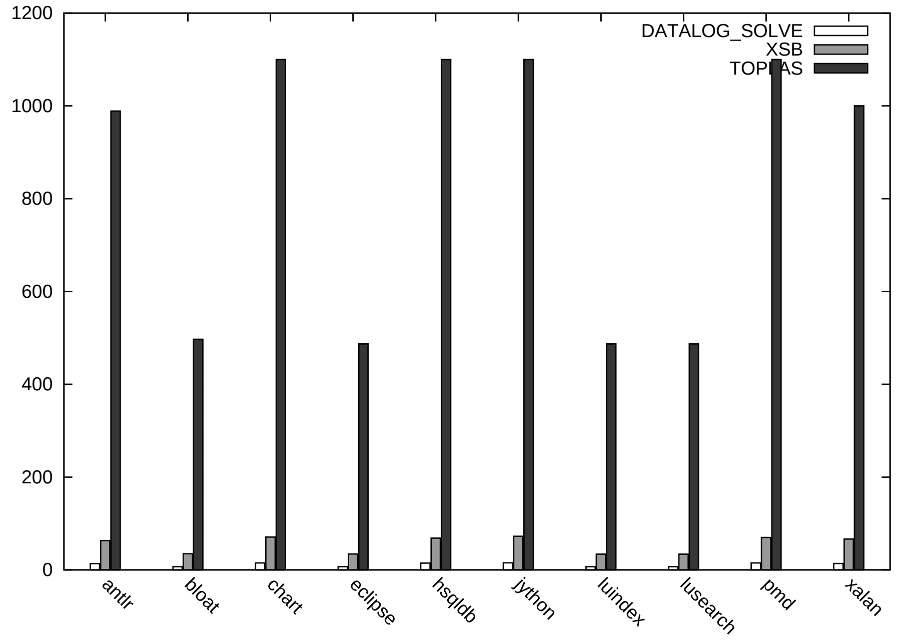

Figure 2.6: Dacapo memory usage (MB.) of various datalog implementations.

DATALOG_SOLVE 2.0 evaluates the whole benchmark in only 3 seconds with a mean-time of 0.3 seconds per program. This explains why the time measures for DATALOG_SOLVE are hardly visible in Figure 2.5. Our experiments demonstrate that XSB is much slower than TOPLAS, which is in turn an order of magnitude slower than DATALOG_SOLVE. For the *pmd* example, XSB evaluated the points-to analysis in 501 seconds, TOPLAS solved it in 15 seconds, and DATALOG_SOLVE took 0.391 seconds to solve it. With respect to memory consumption, TOPLAS consumes significantly more than XSB and DATALOG_SOLVE. For the *pmd* example, TOPLAS required 1.1 GB of memory, while XSB consumed 70 MB, and DATALOG_SOLVE consumed 15 MB. These performance results show that the BES-based evaluation strategy together with an optimized data-structure scales really well for very large programs regarding Andersen's points-to analysis.

2.5 Related Work.

Recently, BESs with typed parameters [Mat98], called PBES, have been successfully used to encode several hard verification problems such as the first-order value-based modal μ-calculus model-checking problem [MT08], and the equivalence checking of various bisimulations [CPvW07] on (possibly infinite) labeled transition systems. However, PBESs were not used to compute complex interprocedural program analyses involving dynamically created objects until our work in [AFJV09c]. The work that is most closely related to the BES-based analysis approach of ours is [LS98], where Dependency Graphs (DGs) are used to represent satisfaction problems, including propositional Horn Clauses satisfaction and BES resolution. A linear time algorithm for propositional Horn Clause satisfiability is described in terms of the least solution of a DG equation system. This corresponds to an alternation-free BES, which can only deal with propositional logic problems. The extension of Liu and Smolka's work [LS98] to Datalog query evaluation is not straightforward. This is testified by the encoding of data-based temporal logics in equation systems with parameters in [MT08], where each boolean variable may depend on multiple data terms. DGs are not sufficiently expressive to represent such data dependencies on each vertex. Hence, it is necessary to work at a higher level, on the PBES representation.

A very efficient Datalog program analysis technique based on binary decision diagrams (BDDs) is available in the BDDBDDB system [WACL05], which scales to large programs and is competitive w.r.t. the traditional (imperative) approach. The computation is achieved by a fixpoint computation starting from the everywhere false predicate (or some initial approximation based on Datalog facts). Datalog rules are then applied in a bottom-up manner until saturation is reached so that all the solutions that satisfy each relation of a Datalog program are exhaustively computed. These sets of solutions are then used to answer complex formulas. In contrast, our approach focuses on demand-driven techniques to solve the considered query with no *a priori* computation of the derivable atoms. In the context of program analysis, note that all program updates, like pointer updates, might potentially be inter-related, leading to an exhaustive computation of all results. Therefore, improvements to top-down evaluation are particularly important for program analysis applications. Recently, Zheng and Rugina [ZR08] showed that demand-driven CFL-reachability with worklist algorithm compares favorably with an exhaustive solution. Our technique to solve Datalog programs based on local BES resolution goes in the same direction and provides a novel approach to demand-driven program analyses almost for free.

2.6 Conclusions

We have presented a transformation from Datalog to BES in the context of Datalog-based static analysis. The transformation carries Datalog to a powerful framework such as BESs, which have been widely used for verification of industrial critical systems, and for which many efficient resolution algorithms exists.

We have presented some experimental results which show that the presented transformation is quite efficient. We have also briefly discussed how we have improved this transformation as future work, thus showing the progress reached by our approximation.

Chapter 3

The Rwl-based Datalog evaluation approach

With the aim to achieve higher expressiveness for static-analysis specification, in this chapter we present a translation of Datalog into a powerful and highly extensible framework, namely, rewriting logic (Rwl). Due to the high level of expressiveness of Rwl, many ways for translating Datalog into Rwl can be considered. Because efficiency does matter in the context of Datalog-based program analysis, our proposed transformation is the result of an iterative process that is aimed at optimizing the running time of the transformed program. The basic idea of the translation is to automatically compile Datalog clauses into deterministic equations, having queries and answers consistently represented as terms so that the query is evaluated by reducing its term representation into a *constraint set* that represents the answers. This chapter summarizes how Datalog queries can be solved by means of Rwl rewriting.

3.1 From Datalog to Rwl.

In the following, we informally illustrate how a Rwl *term rewriting system* can be obtained from a Datalog program in an automatic way. Let us recall the simplified version of the Andersen points-to analysis that we introduced in Figure 2.1 (Chapter 2):

```
vP0(p,o1).
vP0(q,o2).
assign(r,q).
assign(w,r).
vP(V,H) :- vP0(V,H).
vP(V,H) :- assign(V,V2), vP(V2,H).
```

In the following, we illustrate the transformation by means of this example. We first show values, variables and answers are represented in Maude. Then, the resulting Maude program is presented by showing how each datalog clause in the example is transformed.

Datalog *answers* are expressed as equational *constraints* that relate the variables of the queries to values. Values are represented as *ground terms* of sort `Constant` that are constructed by means of MAUDE *Quoted Identifiers* (`Qids`). Since logical variables cannot be represented with rewriting rule variables because of their dual input-output nature, we give a representation for them as ground terms of sort `Variable` by means of the overloaded `vrbl` constructor. A `Term` is either a `Constant` or a `Variable`. These elements are represented in MAUDE as follows:

```
sorts Variable Constant Term .
subsort Variable Constant < Term .
subsort Qid < Constant .
op vrbl :  Term -> Variable [ctor] .
```

In our formulation, answers are recorded within the term that represents the ongoing partial computation of the MAUDE program. Thus, we represent a (partial) answer for the original Datalog query as a conjunction of equational constraints (called answer constraints) that represent the substitution of (logical) variables by (logical) constants that are incrementally computed during the program execution. We define the sort `Constraint` whose values represent single answers for a Datalog query as follows:

```
sort Constraint .
op _=_ :  Term Constant -> Constraint .
op T : -> Constraint .
op F : -> Constraint .
op _,_ :  Constraint Constraint -> Constraint [assoc comm id:  T] .
eq F, C:Constraint = F .                        --- Zero element
```

Constraints are constructed by the conjunction (`_,_`) of solved equations of the form `T:Term = C:Constant`, the *false* constraint `F`, or the *true* constraint `T`.[1] Note that the conjunction operator `_,_` obeys the laws of associativity and commutativity.[2] `T` is defined as the identity of `_,_`, and `F` is used as the zero element.

Unification of expressions is performed by combining the corresponding answer constraints and checking the satisfiability of the compound. Simplification equations are introduced in order to simplify trivial constraints by reducing them to `T`, or to detect inconsistencies (unification failure) so that the whole conjunction can be drastically replaced by `F`, as shown in the following code excerpt:

[1]The actual transformation defines a more complex hierarchy of sorts in order to obtain simpler equations and improve performance that will be presented in Section 3.2.

[2]Associativity, commutativity, and identity are easily expressed by using `ACI` attributes in MAUDE, thus simplifying the equational specification and also achieving a more efficient implementation.

```
var Cst Cst1 Cst2 :  Constant .  var V : Variable .
eq (V = Cst)  , (V = Cst)  = (V = Cst) , T . --- Idempotence
eq (V = Cst1) , (V = Cst2) = F [owise] .     --- Unsatisfiability
```

In our setting, a failing computation occurs when a query is reduced to F. If a query is reduced to T, then the original (ground) query is proven to be satisfiable. On the contrary, if the query is reduced to a set of solved equations, then the computed answer is given by a substitution $\{x_1/t_1, \ldots, x_n/t_n\}$ that is expressed as an equation set in solved form by the computed normal form x_1 = t_1 , ... , x_n = t_n.

Since equations in MAUDE are run deterministically, all the non-determinism of the original Datalog program has to be embedded into the term under reduction. This means that we need to carry all the possible (partial) answers at a given execution point. To this end, we introduce the notion of *set of answer constraints*, and we define a new sort called ConstraintSet as follows:

```
sorts ConstraintSet .
subsort Constraint < ConstraintSet .
op _;_ : ConstraintSet ConstraintSet -> ConstraintSet  [assoc comm id: F] .
```

The set of constraints is constructed as the (possibly empty) disjunction _;_ of accumulated constraints. The disjunction operator _;_ obeys the laws of associativity and commutativity and is also given the identity element F.

Now we are ready to show how the predicates are transformed. Predicates are naturally expressed as functions (with the same arity) whose codomain is the ConstraintSet sort. They will be reduced to the set of constraints that represent the satisfiable instantiations of the original query. The three predicates of our running example are represented in MAUDE as follows:

```
op vP vPO assign :  Term Term -> ConstraintSet .
```

In order to incrementally add new constraints throughout the program execution, we define the composition operator x for constraint sets as follows:

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

The composition operator x allows us to combine (partial) solutions of the subgoals in a clause body.

Example 3.1.1 Let us illustrate the transformation by evaluating different queries in our running example. For instance, by executing the Datalog query :- vPO(p,Y) on the program in Figure 2.1, we obtain the solution {Y/o1}. Here, vPO is a predicate defined only by facts, so the answers to the query represent the variable instantiations as given by the existing facts. Thus, we would expect the query's RWL representation `vP0('p, vrbl('Y))` to be reduced to the `ConstraintSet` (with just one constraint) `vrbl('Y) = 'o1`. This is accomplished by representing facts according to the following equation pattern:

```
var T0 T1 :  Term .
eq vP0(T0,T1) = (T0 = 'p , T1 = 'o1) ; (T0 = 'q , T1 = 'o2) .
eq assign(T0,T1) = (T0 = 'r , T1 = 'q) ; (T0 = 'w , T1 = 'r) .
```

The right-hand side of the RWL equation that is used to represent the facts that define a given predicate (in the example `vP0` and `assign`) consists of the set of constraints that express the satisfiable instantiations of the original predicate. As it can be observed, arguments are propagated to the constraints, thus allowing the already mentioned equational unification and simplification process on the constraints to happen.

For the considered goal, the reduction of the transformed Datalog query `vP0('p, vrbl('Y))` proceeds as follows:

```
vP0('p,vrbl('Y))
  → ('p = 'p , vrbl('Y) = 'o1) ; ('p = 'q , vrbl('Y) = 'o2)
  →* (T , vrbl('Y) = 'o1) ; (F , vrbl('Y) = 'o2)
  →* vrbl('Y) = 'o1 ; F
  → vrbl('Y) = 'o1
```

■

Example 3.1.2 Another example of Datalog query is `:- vP(V,o2)`, whose execution for the leading example delivers the solutions {`V/q`,`V/r`,`V/w`}. Thus, we expect `vP(vrbl('V),'o2)` to be reduced to the set of constraints `(vrbl('V) = 'q) ; (vrbl('V) = 'r) ; (vrbl('V) = 'w)`. In this case, `vP` is a predicate defined by clauses, so the answers to the query are the disjunction of the answers provided by all the clauses defining it. This is represented in RWL by introducing auxiliary functions to separately compute the answers for each clause, and the equation to join them, which is defined as follows:

```
op vP-clause-1 vP-clause-2 :  Term Term -> ConstraintSet .
var V H : Term .
eq vP(V,H) = vP-clause-1(V , H) ; vP-clause-2(V , H) .
```

In order to compute the answers delivered by a clause, we look for the satisfiable instantiations of its body's subgoals. In our translation, we explore the possible instantiations from the leftmost subgoal to the rightmost one. In order to impose this left-to-right exploration, we create a different (auxiliary) unraveling function for each subgoal. Each of these auxiliary functions computes the partial answer depending on the corresponding and previous subgoals and propagates it to the subsequent unraveling function[3]. Additionally, existential variables that occur only in the body of original Datalog clauses, e.g., `V2`, are introduced by using a ground representation that is parameterised with the corresponding call pattern in order to generate fresh variables (in the example below `vrbl-V2(V,H)`).

As shown in the following code excerpt, in our example, the first Datalog clause can be transformed without using unraveling functions. For the second Datalog clause (with two subgoals) only one unraveling function is needed in order to force the reduction of the first subgoal.

```
op vrbl-V2 :  Term Term -> Variable .
op unrav :  ConstraintSet TermList -> ConstraintSet .

eq vP-clause-1(V,H) = vP0(V,H) .
eq vP-clause-2(V,H) = unrav( assign(V, vrbl-V2(V,H)) , V H ) .
```

The `unrav` function has two arguments: a `ConstraintSet`, which is the first (reduced) subgoal (the original subgoal `assign(V,V2)` in this case); and the `V H` call pattern. This function is defined as follows:

```
var Cnt :  Constant .  var TS : TermList .
var C : Constraint .  var CS : ConstraintSet .

eq unrav( ( (vrbl-V2(V,H) = Cnt , C) ; CS ) , V H ) =
    ( vP(Cnt,H) x (vrbl-V2(V,H) = Cnt , C) ) ; unrav( CS , V H ) .
eq unrav( F , TS ) = F .
```

The unraveling function (in the example `unrav`) takes a set of partial answers as its first argument. It requires the partial answers to be in solved equation form by pattern matching, thus ensuring the left-to-right execution of the goals. The second argument is the call pattern of the translated clause and serves to reference the introduced existential variables. The propagated call pattern is represented as a `TermList`, that is, a juxtaposition (__ operator) of `Terms`. The two `unrav` equations (recursively) combine each (partial) answer obtained from the first subgoal with every (partial) answer computed from the (instantiated) subsequent subgoal.

Consider again the Datalog query `:- vP(V,o2)`. We undertake each possible query reduction by using the equations above. Given the size of the execution trace, we will use the following abbreviations: `V` stands for `vrbl('V)`, `vPci` for `vP-clause-i`, and `V2-V-H` for `vrbl-V2(V,H)`.

[3]Conditional equations could also be used to impose left-to-right evaluation, but in practice they suffer from poor performance as our experiments revealed.

```
vP(V,’o2 )
  → vPc1(V,’o2) ; vPc2(V,’o2)
  →* vP0(V,’o2) ; unrav( assign(V,V2-V-o2) , V ’o2 )
  →* ((V = ’p , ’o2 = ’o1) ; (V = ’q , ’o2 = ’o2))
     ; unrav( ((V = ’r , V2-V-o2 = ’q) ; (V = ’w , V2-V-o2 = ’r)) , V ’o2 )
  →* (F ; (V = ’q , T)) ; (vP(’q,’o2) x (V = ’r , V2-V-o2 = ’q))
     ; unrav( (V = ’w , V2-V-o2 = ’r) , V ’o2 )
  →* (V = ’q) ; ((vPc1(’q,’o2) ; vPc2(’q,’o2)) x (V = ’r , V2-V-o2 = ’q))
     ; (vP(’r,’o2) x (V = ’w , V2-V-o2 = ’r)) ; unrav( F , V ’o2 )
  ...
  →* (V = ’q) ; (V = ’r) ; (V = ’w)
```

■

As it can be seen, the evaluation of a Datalog query is naturally transformed to the process of reducing that query into its solutions.

3.2 A complete Datalog to Rwl transformation

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`[4] 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 [AFJV09b]. This is because *rules* are handled non-deterministically in Maude whereas *equations* are applied deterministically [CDE+07a].

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

In this section, we formulate a complete representation in Maude of the Datalog computed answers, and then, we give a formal description of our equation-based transformation together with its correctness and completeness results.

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 that are built by simple juxtaposition):

[4]Intuitively, `search` $t \rightarrow t'$ explores the whole rewriting space from the term t to any other terms that match t' [CDE+07a].

```
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 `vrbl`. These constructor symbols are specified in MAUDE as follows:

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

The last line of the above code excerpt allows us to build variable terms of the form `vrbl(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[5].

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 MAUDE code that implements these features is as follows:

[5]Actually, the definition of `vrbl` is a bit more complicated to ensure freshness. The interested reader may access the code in the prototype website.

```
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] .
```

As we have said before, a query reduced to `T` represents a successful computation, whereas a failing computation is represented by a final `F` term. Note that the conjunction operator `_,_` has identity element `T` and obeys the laws of associativity and commutativity. The properties of associativity, commutativity and identity element can be easily expressed by using `ACU` attributes in MAUDE, thus simplifying the equational specification and achieving better efficiency. Other properties of the constraint-builder operators must be expressed with equations: for example, we express the idempotency property of the operator by a specific equation on variables from the `NonEmptyConstraint` subsort `NEC`. Moreover, in order to keep information consistent and without redundancy, additional simplification equations are automatically applied. These equations make every inconsistent constraint collapse into an `F` value, and simplify every redundant or trivial constraint. The MAUDE code implementing this features is:

```
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
```

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. It is an associative and commutative operator that has `F` as its identity element. 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 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
```

The transformation of clauses. Let P be a Datalog program defining predicate symbols $p_1 \ldots p_n$. Before describing the transformation process, we introduce some auxiliary notations. $|p_i|$ is the number of facts or clauses defining the predicate symbol p_i. Following the Datalog standard, we assume without loss of generality that a predicate p_i is defined only by facts, or only by clauses [Lee90]. 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 $T_{i,1}$ ... $T_{i,ar_i}$ :  Term .
eq $p_i$($T_{i,1}$, ... ,$T_{i,ar_i}$) = ($T_{i,1}$ = $c_{i,1,1}$, ... , $T_{i,ar_i}$ = $c_{i,1,ar_i}$) ; ...
                          ; ($T_{i,1}$ = $c_{i,|p_i|,1}$, ... , $T_{i,ar_i}$ = $c_{i,|p_i|,ar_i}$) .
```

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

$$\begin{array}{l}
\texttt{var } \texttt{T}_{i,1} \texttt{ ... } \texttt{T}_{i,ar_i} \texttt{ : Term .} \\
\texttt{eq } \texttt{p}_i\texttt{(T}_{i,1}\texttt{, ... ,T}_{i,ar_i}\texttt{) = p}_{i,1}\texttt{(T}_{i,1}\texttt{, ... ,T}_{i,ar_i}\texttt{) ; ...} \\
\qquad\qquad\qquad\qquad\texttt{; p}_{i,|p_i|}\texttt{(T}_{i,1}\texttt{,...,T}_{i,ar_i}\texttt{) .}
\end{array}$$

Each call to a function $\texttt{p}_{i,j}$ 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 $\tau^a_{i,j,s,k}$ denotes the name of the variable or constant symbol appearing in the k-th argument of the s-th subgoal in the j-th clause defining the i-th predicate of the original Datalog program. When $s = 0$, then the function refers to the arguments in the head of the clause.

Let us start by considering the case of just one subgoal in the body. We define the function $\tau^p_{i,j,s}$, 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:

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

In our formalization, r is used to denote the arity of the predicate in whose arguments appears (e.g. $\tau^p_{i,j,1}$).

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 with specific patterns to force such an execution order. Specifically, we impose 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 formalize the auxiliary notions that we need for our transformation.

Definition 3.2.1 (linked variable) *A variable is called* linked variable *if and only if (*iff*) it occurs in two or more subgoals of the clause's body.*

Definition 3.2.2 (function linked) *Let C be a* Datalog *clause. Then the function* 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 component*[6]*.*

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

Example 3.2.3 For example, given the Datalog clause

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

we have that linked(C) `= [(X1,[1.1,2.1]),(X3,[1.2,2.2]),(X4,[2.3,3.1])]` ■

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

$$\mathit{relevant}(n,C)=\{X\,|\,(X,LX)\in\mathsf{linked}(C), \exists i,j,m\,/\,n.i\in LX, m.j\in LX, m<n\}$$

Note that, similarly to [SKGST07], 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 undertakes the execution of the first subgoal. We have as many auxiliary functions as subgoals in the original clause minus one. Also, in the right-hand side (*rhs*s) of the auxiliary functions definitions, 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 $\mathtt{p}_{i,j}$ generate the solutions calculated by the j-th clause of the predicate symbol p_i. We state that $\mathtt{ps}_{i,j,s}$ represents the auxiliary function corresponding to the s-th subgoal of the j-th clause defining the predicate p_i. Then, for each clause, we have the following translation, where the variables $\mathtt{X}_1 \ldots \mathtt{X}_N$ of each equation are calculated by the function relevant(s,*clause*(i,j))[7] and transformed into the corresponding MAUDE terms.

The equation for $\mathtt{p}_{i,j}$ below calls the first auxiliary function ($\mathtt{ps}_{i,j,2}$) that calculates the (partial) answers for the second subgoal by first computing the answers from the first subgoal $\tau^{p}_{i,j,1}$ in its first argument. The second argument of the call to $\mathtt{ps}_{i,j,2}$ represents the list of terms in the initial predicate call that, together with the information retrieved from Definitions 3.2.2 and 3.2.4, allow us to correctly build the patterns and function calls during the transformation.

[7]The notation *clause*(i,j) represents the j-th Datalog clause defining the predicate symbol p_i.

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

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 $\tau^p_{i,j,s}$). Note that we propagate the instantiation of the relevant variables by means of a substitution.

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

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

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

The equation for the last subgoal in the clause is slightly different, since we 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

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

Query representation. Finally, we define the transformation for the Datalog query $q(X_1, \dots, X_n)$ (where X_i, $1 \le i \le n$ are Datalog variables or constants) as the MAUDE code `q(`τ^q_1`,...,`τ^q_n`)`, where τ^q_i, $1 \le i \le n$ is the transformation of the corresponding X_i.

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

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

Definition 3.2.5 (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 $t \rightarrow_S^*$ `CS` is* $\texttt{CS}|_{var(t)}$.

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

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.

Theorem 3.2.6 (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 $\texttt{CS}|_{var(\mathcal{T}_g(q))}$ be the answer constraint set computed by $\mathcal{T}_g(q) \rightarrow_{\mathcal{T}(P)}^*$ `CS`. Then, $\Theta \equiv \texttt{CS}|_{var(\mathcal{T}_g(q))}$.*

Proof of Theorem 3.2.6

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

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

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 facts[8], there exists an equation in $\mathcal{T}(P)$, whose left hand side is of the form `q(T1,..., 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

$\mathcal{T}_g(q) \rightarrow_{\mathcal{T}(P)} \mathtt{C}_1;\ldots;\mathtt{C}_n \rightarrow^!_{\mathcal{T}(P)} \mathtt{C}_v;\ldots;\mathtt{C}_w$

where n is the number of facts defining the Datalog predicate and $v,\ldots,w \in \{1,\ldots,n\}$. Each answer constraint in $\mathtt{C}_1;\ldots;\mathtt{C}_n$ comes up from one Datalog fact. The second part of the sequence is the simplification for the union operator `;` and constraint constructors. The simplification consists in removing duplicate elements and collapsing inconsistent constraints to `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 occur when a query is partially (or totally) instantiated and/or when it has a variable that appears multiple times. In this case, all the answer constraints that are incompatible with the passed value are collapsed to `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.

$\mathcal{T}_g(q) \rightarrow_{\mathcal{T}(P)} \mathtt{q_1}(\mathtt{T1},\ldots,\mathtt{Tm});\ldots;\mathtt{q_n}(\mathtt{T1},\ldots,\mathtt{Tm})$

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.

Let us consider the case of a clause having a single subgoal. Let the equation defining the function symbol $\mathtt{q_i}$ be

$\mathsf{eq}\ \mathtt{q_i}(U_1,\ldots,U_m) = \mathtt{p}(V_1,\ldots,V_z)$

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

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

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

Notice that `p` is a predicate symbol in the Datalog program that is also transformed. By induction hypothesis, $\mathtt{p}(W_1,\ldots,W_z)$ rewrites to the set of its correct answer constraints

[8]Remember that, in Datalog, predicates are defined by facts or by clauses but not by both.

$\mathtt{C'}_1; \ldots ; \mathtt{C'}_w|_{var(\mathtt{p}(W_1,\ldots,W_z))}$. Since we are considering safe Datalog programs, we know that all the variables $T_1, \ldots, T_m$ occur in the arguments of the body subgoals and are thus in the set of variables $\{W_1, \ldots, W_z\}$. Therefore, the correct answer constraint for the query is $\mathtt{C'}_1; \ldots ; \mathtt{C'}_w|_{var(\mathtt{q}(T_1,\ldots,T_m))}$.

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:

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

By induction hypothesis, the set $\mathtt{C}_1; \ldots ; \mathtt{C}_w$ contains correct answer constraints for $\mathtt{p}(W_1, \ldots, W_z)$. 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 $\mathtt{p}(W_1, \ldots, W_z)$; thus, the answer constraint set is $\mathtt{F}$. In this case, the rewriting sequence is:

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

   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 $\mathtt{p}(W_1, \ldots, W_z)$. The rewriting sequence following the definition of our transformation is:

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

   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 ($\mathtt{C}_1$) is $\mathtt{T}$ (which is an `EmptyConstraint`); thus, the computation of the previous subgoal (which is ground) performed no substitution of variables:

   $$s_{i+1}(s'_i(\mathtt{T} \; ; \; \ldots \; ; \; \mathtt{C}_w, T_1 \ldots T_m)), T_1 \ldots T_m)$$

$\rightarrow_{\mathcal{T}(P)}$ $s_{i+1}(\mathtt{q}(Q_1,\ldots,Q_z); (s'_i(\mathtt{C}_2 \;;\; \ldots \;;\; \mathtt{C}_w, T_1\ldots T_m)), T_1\ldots T_m)$
$\rightarrow^*_{\mathcal{T}(P)}$ $s_{i+1}(\mathtt{C}'_1;\ldots;\mathtt{C}'_{\mathtt{w}'}; (s'_i(\mathtt{C}_2 \;;\; \ldots \;;\; \mathtt{C}_w, T_1\ldots T_m)), T_1\ldots T_m)$

By induction hypothesis, $\mathtt{C}'_1,\ldots,\mathtt{C}'_{\mathtt{w}'}$ are correct answer constraints for the i-th subgoal (whose function symbol is 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 set of answer constraints for the first subgoal, but also the call pattern. We will come back to this point of the proof after introducing the rest of cases.

(b) The first answer constraint is not T, generating the following rewriting sequence:

$s_{i+1}(s'_i(\mathtt{C}_1 \;;\; \ldots \;;\; \mathtt{C}_w, T_1\ldots T_m)), T_1\ldots T_m)$
$\rightarrow_{\mathcal{T}(P)}$ $s_{i+1}(\mathtt{q}(Q_1,\ldots,Q_z)[Q_j\backslash X_j,\ldots,Q_k\backslash X_k])$ x $\mathtt{C}_1$;
$(s'_i(\mathtt{C}_2 \;;\; \ldots \;;\; \mathtt{C}_w, T_1\ldots T_m)), T_1\ldots T_m)$
$\rightarrow^*_{\mathcal{T}(P)}$ $s_{i+1}((\mathtt{C}'_1;\ldots;\mathtt{C}'_{\mathtt{w}'})$ x $\mathtt{C}_1; (s'_i(\mathtt{C}_2 \;;\; \ldots \;;\; \mathtt{C}_w, T_1\ldots T_m)), T_1\ldots T_m)$

Note that, by definition, the substitution $\mathtt{q}(Q_1,\ldots,Q_z)[Q_j\backslash X_j,\ldots,Q_k\backslash X_k]$ replaces each relevant variable X_j of q 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 $\mathtt{C}_1$. By induction hypothesis, $\mathtt{C}'_1,\ldots,\mathtt{C}'_{\mathtt{w}'}$ are correct answer constraints for the term $\mathtt{q}((Q_1,\ldots,Q_z)[Q_j\backslash X_j,\ldots,Q_k\backslash X_k])$. Then, the x operator combines each solution of the second subgoal with the information in $\mathtt{C}_1$. 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 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 inconsistent answer constraint is carried on.

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

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

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 $\mathtt{C}_1 \;;\; \ldots \;;\; \mathtt{C}_w$, we can see that the subterm $(s'_i(\mathtt{C}_2 \;;\; \ldots \;;\; \mathtt{C}_w, T_1\ldots T_m)), T_1\ldots T_m)$ 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:

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

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

$s_i(\mathtt{C}_1 \ ; \ \ldots \ ; \ \mathtt{C}_w, T_1 \ldots T_m)$

Also in this case, there are three possible paths:

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

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

By induction hypothesis, $\mathtt{C'}_1 \ ; \ \ldots \ ; \ \mathtt{C'}_{w'}$ are the correct answer constraints of $\mathtt{q}(Q_1, \ldots, Q_n)$. For the recursive call, the proof is perfectly analogous to the one for the other cases.

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

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

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 $\mathtt{C}_1$. By induction hypothesis, $\mathtt{C'_1}, \ldots, \mathtt{C'_{w'}}$ are correct answer constraints for the term $\mathtt{q}(Q_1, \ldots, Q_z)[Q_j \backslash X_j, \ldots, Q_k \backslash X_k]$. Then, the `x` operator combines each solution of the second subgoal with the information in $\mathtt{C}_1$. 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 `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 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:

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

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 $\mathtt{C}_1$; ... ; $\mathtt{C}_w$, we can see that the subterm $(s_i(\mathtt{C}_2$; ... ; $\mathtt{C}_w, T_1 ... T_m)), T_1 ... T_m)$ 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:

$\mathtt{C}_1; ... ; \mathtt{C}_n$

($\Rightarrow$) 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 $\mathcal{T}_g(q) \rightarrow^!_{\mathcal{T}(P)} \mathtt{CS}$, there exists a `C` in the answer constraint set `CS` such that $\mathtt{C}|_{var(\mathcal{T}_g(q))}) \equiv \theta$.

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 (1) when a query is partially (or totally) instantiated and the given values do not coincide with those in the corresponding facts; or (2) when a query has a variable that appears multiple times in its arguments and a single fact assigns two different values to such variable at the same time.

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

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

$\mathcal{T}_g(q) \rightarrow_{\mathcal{T}(P)} \mathtt{C}_1; \ldots ; \mathtt{C}_n \rightarrow^!_{\mathcal{T}(P)} \mathtt{C}_v; \ldots ; \mathtt{C}_w$

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

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:

$\mathcal{T}_g(q) \rightarrow_{\mathcal{T}(P)} q_1(T1, \ldots , Tm); \ldots ; q_n(T1, \ldots , Tm)$

Next we prove that the solutions computed from the i-th clause are included in the set of answer constraints computed by the function $q_i(T1, \ldots , Tm)$. 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 `p` and z arguments: $\mathtt{p}(V_1, \ldots , V_z)$. By definition of our transformation, the equation for this clause is the following one, where `p` is now a defined function symbol:

$\mathtt{eq}\ \mathtt{q_i}(U_1, \ldots ,U_m) = \mathtt{p}(V_1, \ldots ,V_z)$

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

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

$q_i(T_1, \ldots ,T_m) \rightarrow_{\mathcal{T}(P)} \mathtt{p}(W_1, \ldots ,W_z) \rightarrow^!_{\mathcal{T}(P)} \mathtt{C'}_1; \ldots ; \mathtt{C'}_w$

By induction hypothesis, for each computed answer θ for the query $\mathtt{p}(W_1, \ldots ,W_z)$, there exists an answer constraint $\mathtt{C'}_i$, $1 \leq i \leq w$ such that $\theta \equiv \mathtt{C'_i}$. 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:

$\mathtt{q_i}(T_1,\ldots,T_m) \rightarrow_{\mathcal{T}(P)} s_2(\mathtt{p}(W_1,\ldots,W_z),T_1\ldots T_m) \rightarrow^*_{\mathcal{T}(P)} s_2(C_1;\ldots;C_w,T_1\ldots T_m)$

By induction hypothesis, for each computed answer θ of the Datalog query $\mathtt{p}(W_1,\ldots,W_z)$, there exists an answer constraint C_i in the set $C_1;\ldots;C_w$ 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 $\mathtt{p}(W_1,\ldots,W_z)$. Therefore, the answer constraint set is of the form F. In this case, the rewriting sequence is:

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

   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 $\mathtt{p}(W_1,\ldots,W_z)$ (that by induction hypothesis include the equivalent answer constraints for each Datalog computed answer). The rewriting sequence following the definition of our transformation is:

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

   Note that, in order 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 ($\mathtt{C}_1$) is `T` (which is an `EmptyConstraint`). Therefore, the computation of the previous subgoal (which is ground) performed no substitution of variables:

$$\begin{array}{ll} & s_{i+1}(s'_i(\mathtt{T}\ ;\ \ldots\ ;\ \mathtt{C}_w, T_1 \ldots T_m)), T_1 \ldots T_m) \\ \to_{\mathcal{T}(P)} & s_{i+1}(\mathtt{q}(Q_1, \ldots, Q_z); (s'_i(C_2\ ;\ \ldots\ ;\ C_w, T_1 \ldots T_m)), T_1 \ldots T_m) \\ \to^*_{\mathcal{T}} & s_{i+1}(C'_1; \ldots; C'_{w'}; (s'_i(C_2\ ;\ \ldots\ ;\ C_w, T_1 \ldots T_m)), T_1 \ldots T_m) \end{array}$$

By induction hypothesis, for each computed answer θ for the call $\mathtt{q}(Q_1, \ldots, Q_z)$, there exists an answer constraint C_i in the set $C'_1, \ldots, C'_{w'}$ such that $\theta \equiv C_i$. These are all answers for the i-th subgoal (whose function symbol is `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 $C_1\ ;\ \ldots\ ;\ C_w$, which by hypothesis includes the equivalent answer constraints for the computed answers of the i-th subgoal. The rewriting sequence is:

$$\begin{array}{ll} & s_{i+1}(s'_i(C_1\ ;\ \ldots\ ;\ C_w, T_1 \ldots T_m)), T_1 \ldots T_m) \\ \to_{\mathcal{T}(P)} & s_{i+1}((\mathtt{q}(Q_1, \ldots, Q_z)[Q_j \backslash X_j, \ldots, Q_k \backslash X_k])) \ \mathtt{x}\ C_1; \\ & \quad (s'_i(C_2\ ;\ \ldots\ ;\ C_w, T_1 \ldots T_m)), T_1 \ldots T_m) \\ \to^*_{\mathcal{T}(P)} & s_{i+1}((C'_1; \ldots; C'_{w'})\ \mathtt{x}\ C_1; (s'_i(C_2\ ;\ \ldots\ ;\ C_w, T_1 \ldots T_m)), T_1 \ldots T_m) \end{array}$$

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 $\mathtt{q}(Q_1, \ldots, Q_z)[Q_j \backslash X_j, \ldots, Q_k \backslash X_k]$, there exists a C'_i in $C'_1, \ldots, C'_{w'}$ such that $\theta \equiv C'_i$. Then, the `x` operator combines each solution of the second subgoal with the information in $\mathtt{C}_1$. 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 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:
$s_{i+1}(s'_i(\mathtt{F}, T_1...T_m)), T_1...T_m) \rightarrow_{\mathcal{T}(P)} s_{i+1}(\mathtt{F}, T_1...T_m)$

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 $C_1 \ ; \ \ldots \ ; C_w$, it can be observed that the subterm $(s'_i(C_2 \ ; \ \ldots \ ; C_w, T_1...T_m)), T_1...T_m)$ 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:
$s_{i+1}(C_1; \ldots ; C_n, T_1...T_m)$

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

$s_i(C_1 \ ; \ \ldots \ ; C_w, T_1...T_m)$,

where $C_1 \ ; \ \ldots \ ; C_w$ include the equivalent answer constraints for the computed answers of $\mathtt{p}(W_1, \ldots, W_z)$. 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:

$$\begin{aligned} & s_i(\mathtt{T} \ ; \ \ldots \ ; C_w, T_1...T_m) \\ \rightarrow_{\mathcal{T}(P)} \ & \mathtt{q}(Q_1, \ldots, Q_n) \ ; \ s_i(C_2 \ ; \ \ldots \ ; C_w, T_1...T_m) \\ \rightarrow^*_{\mathcal{T}(P)} \ & C'_1 \ ; \ \ldots \ ; C'_{w'} \ ; \ s_i(C_2 \ ; \ \ldots \ ; C_w, T_1...T_m) \end{aligned}$$

By induction hypothesis, for each computed answer θ for the call $\mathtt{q}(Q_1,\ldots,Q_z)$, there exists an answer constraint C_i in the set $C'_1,\ldots,C'_{w'}$ such that $\theta \equiv C_i$. These are all answers for the i-th subgoal (whose function symbol is 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 ; ... ; C_w that by hypothesis includes the equivalent answer constraints for the computed answers of the i-th subgoal. The rewriting sequence is:

$$\begin{array}{l} \quad s_i(\mathtt{C}_1 \ ; \ \ldots \ ; \mathtt{C}_w, T_1\ldots T_m) \\ \to_{\mathcal{T}(P)} (\mathtt{q}(Q_1,\ldots,Q_z)[Q_j\backslash X_j,\ldots,Q_k\backslash X_k]))\ \mathtt{x}\ \mathtt{C}_1; s_i(\mathtt{C}_2 \ ; \ \ldots \ ; \mathtt{C}_w, T_1\ldots T_m) \\ \to^*_{\mathcal{T}(P)} (\mathtt{C}'_1;\ldots;\mathtt{C}'_{\mathtt{w}'})\ \mathtt{x}\ \mathtt{C}_1; (s_i(\mathtt{C}_2 \ ; \ \ldots \ ; \mathtt{C}_w, T_1\ldots T_m)) \end{array}$$

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 $\mathtt{q}((Q_1,\ldots,Q_z)[Q_j\backslash X_j,\ldots,Q_k\backslash X_k])$, there exists a C'_i in $C'_1,\ldots,C'_{w'}$ such that $\theta \equiv C'_i$. Then, the x operator combines each solution of the second subgoal with the information in $\mathtt{C}_1$. 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:

$s_i(\mathtt{F}, T_1\ldots T_m) \to_{\mathcal{T}(P)} \mathtt{F}$

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 C_1 ; ... ; C_w, it can be observed that the subterm $(s'_i(C_2 \ ; \ \ldots \ ; C_w, T_1\ldots T_m)), T_1\ldots T_m)$ in the cases (a) and

(b) is a smaller recursive call. Therefore, at some point the sequence will reach the base case.

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

$\mathtt{C_1}; \ldots ; \mathtt{C_n}$

This concludes the proof.

3.3 Dealing with JAVA 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 [LWL05]. 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 the package java.lang.reflect.

In Figure 3.1 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"`.

```
class PO {
      PO (String c1, String c2) {
         this.c1 = c1;
         this.c2 = c2;
      }
      public String c1;
      public String c2;
}
```

```
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;
       } }
```

Figure 3.1: 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 do not 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 [LWL05], 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 [LWL05], 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 us 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

`PO u = new PO("","");`	`vPO(u,0).`
	`vT(u,PO).`

`String v = "c1";`	`vPO(v,12).`
	`vT(v,string).`

`String w = "c2";`	`vPO(w,15).`
	`vT(w,string).`

`java.lang.reflect.Field r`
`= PO.class.getField(v);`
	`vPO($0,18).`
	`vT($0,ClassPO).`
	`vT(r,field).`
	`mI(main,21,getField).`
	`iRet(21,r).`
	`actual(21,0,$0).`
	`actual(21,1,v).`

`r.set(u, w);`	`mI(main,30,set).`
	`actual(30,0,r).`
	`actual(30,1,u).`
	`actual(30,2,w).`

`v = u.c1;`	`l(u,c1,v).`

The following predicates state properties or actions performed to references and heap objects.

`vPO(V,H)`: A new object `H` is created —where `H` is the position of the call to the object's constructor in the code— and is referenced by the variable `V`.

`vT(V,T)`: The declared type of variable `V` is `T`.

`hT(H,T)`: The object `H` has type `T`.

`actual(I,N,V)`: The variable `V` is used as the actual parameter number `N` at the *invocation point*[9] `I`.

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

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

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

`s(V1,F,V2)`: 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 [WACL05]. The analysis would then mimic any possible flow 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 to know 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 `PO` would be created and assigned to the variable `r`. Any call to the method `set` on the previous object would store within 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 proposed by [LWL05] uses additional information (extracted by the Joeq compiler) 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:

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

`iE(I,M)`: There is a call to the *resolved method*[10] `M` at the invocation point `I`. This predicate represents an approximation to the program's *call-graph*.

`fieldObject(H,F)`: 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 `V1` and `V2`, respectively.

In our reflective setting, we have followed the direct approach of translating Datalog clauses into Maude rules as in [AFJV09b] 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:

```
crl s(V1,F,V2) => V1 -> CteV1 , F -> CteF , V2 -> CteV2
    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 in the need for 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:

[10]A *resolved method* refers to specific code from a certain class.

`stringToField(H,F)`: The object `H` is a string representation of field `F`.

`getField(M)`: The method `M` is a reflective method which returns a reflective object representing a field.

From our example, the following field-reflection information would be extracted:

```
stringToField(12,c1).
stringToField(15,c2).
getField(Class.getField).
```

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 3.2 shows the structure of a typical reflection analysis to be run in our tool.

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 feds the reflective module with the information

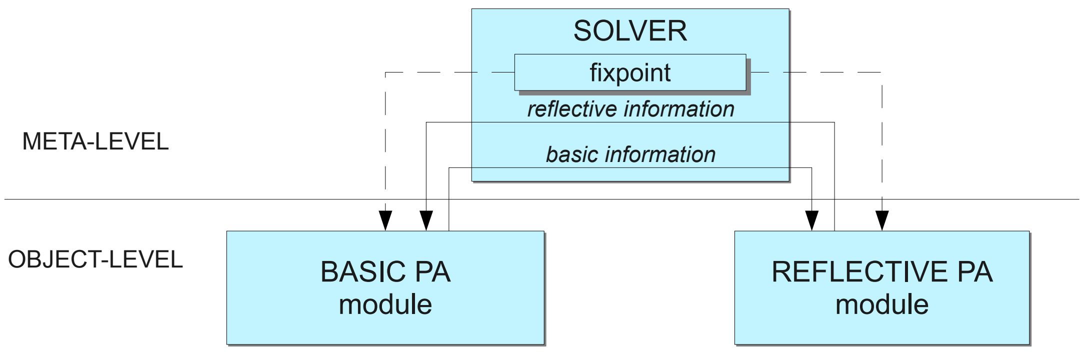

Figure 3.2: The structure of the reflective analysis.

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.

3.4 The prototype Datalaude

Datalaude[11] is a Haskell program that implements the Datalog transformation to Rwl we have presented in this chapter.

[11]Datalaude is accessible via a web interface in `http://www.dsic.upv.es/users/elp/datalaude`.

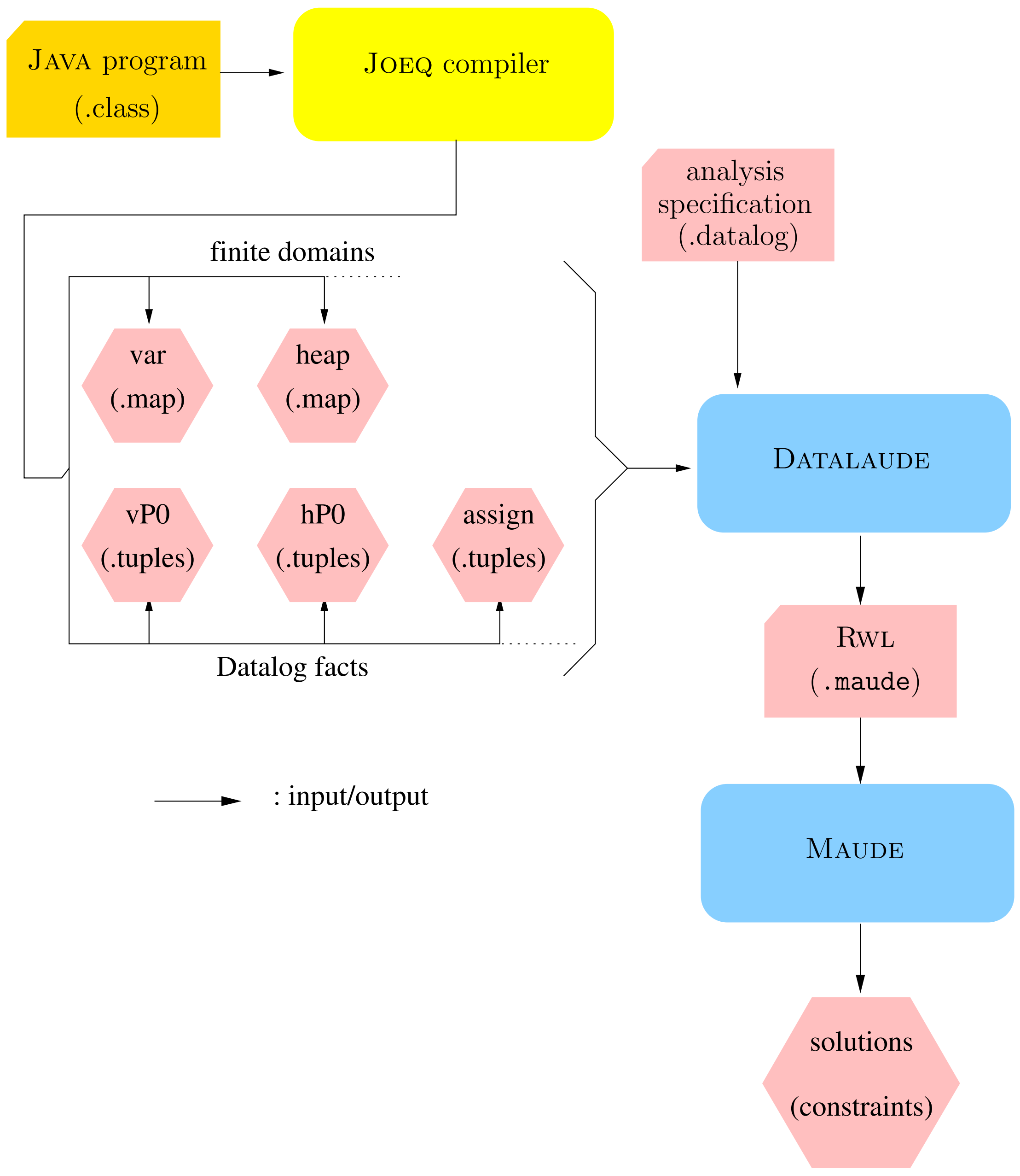

Figure 3.3: JAVA program analysis using DATALAUDE.

As can be seen in Figure 3.3 DATALAUDE takes the same input files as DATALOG_SOLVE (as explained in Section 2.3), but its output is a MAUDE program. This program can subsequently be used by the MAUDE interpreter to reduce Datalog queries into sets of constraints representing the corresponding solutions to the original Datalog query. To do so, first the user should load the `.maude` file obtained from DATALAUDE into the interpreter, and then ask MAUDE to reduce the necessary queries.

Example 3.4.1 If DATALAUDE is fed with the classical Andersen points-to analysis, we obtain a file called `andersen.maude`. From the MAUDE interpreter we should load the transformation with the command:

```
load andersen.maude .
```

To execute the query `:- vP(V,o2).`, which is naturally written in MAUDE as `vP(vrbl('V),'o2)`, we would write the following:

```
reduce vP(vrbl('V),'o2) .
```

The output of MAUDE is shown below. The first part specifies the term that has been reduced (first line). The second part shows the number of rewrites and the execution time that MAUDE invested to perform the reduction (second line). The last part, which is written in several lines for the sake of readability, shows the result of the reduction (i.e., the set of answer constraints) together with its sort.

```
reduce in ANALYSIS : vP(vrbl('v), 'o2) .
rewrites:  39 in 0ms cpu5 (0ms real) (  rewrites/second)
result NonEmptyConstraintSet:
  vrbl('v) = 'q ;
  vrbl('v) = 'r , vrbl(vrbl('v) , 'o2) = 'q ;
  vrbl('v) = 'v1, vrbl(vrbl('v) , 'o2) = 'q , vrbl('q , 'o2) = 'r)
```

Notice that the constraints obtained reference not only the variables present in the query, but also the existential variables used to infer the solutions.

■

3.5 Experimental results

This section reports on the performance of our prototype, DATALAUDE, implementing the transformation. First, we compare the efficiency of our implementation with respect to a naïve transformation to rewriting logic documented in [AFJV09b] and shown in Section 3.3; then, we evaluate the performance of our prototype by comparing it to three public 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.

3.5.1 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 thesis [AFJV09b]. 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 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 [CDE+07a]. MAUDE is able to store each call to a given function (in the running example `vP(V,H)`) together with its normal form. Thus, when MAUDE finds a memoized call it does not reduce it but it just replaces it with its normal form, saving a great number of rewrites.

Table 3.1 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: the Andersen points-to analysis. The evaluated query is `?- vP(Var,Heap).`, i.e., all possible answers that satisfy the predicate `vP/2`.

Table 3.1: 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

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.

3.5.2 Comparison w.r.t. other Datalog solvers

The same sets of initial facts were used to compare our prototype (the equational-based version with memoization) with three state-of-the-art Datalog solvers, namely XSB 3.2 [12], Datalog 1.4 [13], and IRIS 0.58 [14]. Average resolution times of three runs for each solver are shown in Figure 3.4.

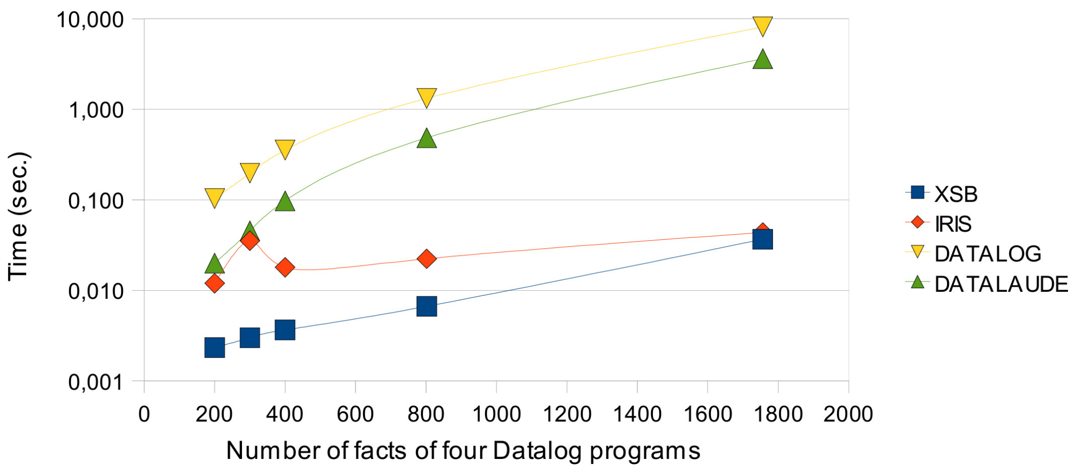

Figure 3.4: Average resolution times of four Datalog solvers (logarithmic time).

In order to evaluate the performance of our implementation with respect to the other Datalog solvers, only resolution times are presented in Figure 3.4 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 DATALAUDE performs similarly to optimized deductive database systems like Datalog 1.4, which is implemented in C, although it is slower than XSB or IRIS. 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.

3.6 Related Work.

The RWL-based approach to Datalog evaluation essentially consists of a suitable transformation from Datalog into MAUDE. Since the operational principles of logic programming (*resolution*) and functional programming (*term rewriting*) share some similarities [Han94], many proposals exist for transforming logic programs into term rewriting systems

[12]http://xsb.sourceforge.net
[13]http://datalog.sourceforge.net
[14]http://iris-reasoner.sourceforge.net

[Mar94, Red84, SKGST07]. These transformations aim at reusing the term rewriting infrastructure 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 (mode) on the parameters of the original program [Red84]. 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 modes on parameters was presented in [SKGST07]. 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 [SKGST07] is not concerned with preserving the computed answers, but only the termination behavior. Moreover, [SKGST07] 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.

3.7 Conclusions

We have presented a transformation from Datalog to RWL in the context of Datalog-based static analysis. The transformation carries Datalog to a powerful framework such as RWL preserving its declarative nature. Reflection is a key capability of RWL specially suited to implement the evolution of systems. We have applied reflection to formalize a way of implementing static analyses that deal with JAVA reflection in a declarative way.

We have also presented some experimental results which show that, under a suitable transformation scheme (such as the equational implementation extended with memoization), MAUDE can process a large number of equations extracted from statically analyzed, real JAVA programs. Our purpose has not been to produce the faster Datalog solver ever, but to provide a tool that supports sophisticated analyses with reasonable performance in a purely declarative way.

Conclusions and future work

In this thesis, we have presented two different Datalog query answering techniques that are specially-tailored to object-oriented program analysis. These techniques essentially consist in transforming the original Datalog program into a suitable set of rules which are then executed under an optimized top-down strategy that caches and reuses “rewrites” in the target language.

We have formalized the transformation of any given set of definite Datalog clauses into two efficient implementations, namely Boolean Equation Systems (Bes) [And94a] and Rewriting Logic (Rwl) [Mes92].

In the Bes-based program analysis methodology, the Datalog clauses that encode a particular analysis, together with a set of Datalog facts that are automatically extracted from program source code, are dynamically transformed into a Bes whose local resolution corresponds to the demand-driven evaluation of the program analysis. This approach has allowed us to reuse existing general purpose analysis and verification toolboxes such as Cadp, which provides local Bes resolution with linear-time complexity. We have implemented this technique into a prototype called Datalog_Solve that shows a good performance on our setting. As future work on the Bes approach we envisage two directions. First, it would be interesting to optimize the transformation, as we have already done in [FJT10b]. The other direction consists in distributing the resolution of the Bes between different machines. The distribution of the resolution could be done at the Bes level [JM06] or at the Datalog level [AU10].

The second, Rwl-based, query answering technique for Datalog was developed in order to provide purely declarative yet efficient program analyses that overcome the difficulty of handling meta-programming features such as reflection in traditional analysis frameworks [LWL05]. By transforming Datalog programs into Maude programs, we take advantage of the flexibility and versatility of Maude in order to achieve meta-programming capabilities, and we make significant progress towards scalability without losing the declarative nature of specifying complex program analyses in Datalog. We have implemented this technique into a prototype called Datalaude, and we have concluded that it is competitive w.r.t. other optimized deductive database systems. Without being one of the fastest Datalog solvers, we have provided a tool that supports sophisticated analyses with reasonable performance in a clean way. As future work on the Rwl approach we envisage two directions. First, just as before, the transformation can surely be optimized since there is still literature on *rewriting techniques* to cover. Second, the use of the Maude meta-level endows us with a fine-grained control of the Rwl execution, thus making possible the implementation of resolution strategies with cost guarantees [LS09], or even compositional reasoning over Datalog programs [BJ03].

Bibliography

[AFJV09a] M. Alpuente, M. A. Feliú, C. Joubert, and A. Villanueva. Datalog_solve: A datalog-based demand-driven program analyzer. *Electronic Notes in Theorertical Computer Science*, 248:57–66, 2009.

[AFJV09b] M. Alpuente, M. A. Feliú, C. Joubert, and A. Villanueva. Implementing Datalog in Maude. In R. Peña, editor, *Proceedings of the IX Jornadas sobre Programación y Lenguajes (PROLE'09) and I Taller de Programación Funcional (TPF'09)*, pages 15–22, September 2009.

[AFJV09c] M. Alpuente, M. A. Feliú, C. Joubert, and A. Villanueva. Using Datalog and Boolean Equation Systems for Program Analysis. In D. Cofer and A. Fantechi, editors, *Proceedings of the 13th International Workshop on Formal Methods for Industrial Critical Systems (FMICS'08)*, volume 5596 of *Lecture Notes in Computer Science*, pages 215–231. Springer-Verlag, 2009.

[AFJV10a] M. Alpuente, M. A. Feliú, C. Joubert, and A. Villanueva. Defining Datalog in Rewriting Logic. In D. De Schreye, editor, *19th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR 2009). Revised Selected Papers*, volume 6037 of *Lecture Notes in Computer Science*, pages 188–204. Springer-Verlag, 2010.

[AFJV10b] M. Alpuente, M. A. Feliú, C. Joubert, and A. Villanueva. Datalog-based Program Analysis with BES and RWL. In *Proceedings of the Workshop on Datalog 2.0*, to appear in *Lecture Notes in Computer Science*. Springer-Verlag, 2010.

[And94a] H. R. Andersen. Model checking and boolean graphs. *Theoretical Computer Science*, 126(1):3–30, 1994.

[And94b] L. O. Andersen. Program Analysis and Specialization for the C Programming Language. Ph.D thesis, DIKU, Unversity of Copenhagen, 1994.

[AU10] F. N. Afrati and J. D. Ullman. Optimizing joins in a map-reduce environment. In I. Manolescu, S. Spaccapietra, J. Teubner, M. Kitsuregawa, A. Léger, F. Naumann, A. Ailamaki, and F. Özcan, editors, *EDBT*, volume 426 of *ACM International Conference Proceeding Series*, pages 99–110. ACM, 2010.

[BCM$^+$03] F. Baader, D. Calvanese, D. L. McGuinness, D. Nardi, and P. F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation, and Applications.* Cambridge University Press, 2003.

[BJ03] F. Besson and T. Jensen. Modular class analysis with datalog. In *SAS'03: Proceedings of the 10th international conference on Static analysis*, volume 2694 of *Lecture Notes in Computer Science*, pages 19–36, Springer-Verlag, 2003.

[CC77] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the 4th ACM Symposium on ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*, pages 238–252, 1977.

[CDE$^+$07a] 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.

[CDE$^+$07b] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C.Talcott. *All About Maude: A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*, volume 4350 of *Lecture Notes in Computer Science.* Springer-Verlag, 2007.

[CPvW07] T. Chen, B. Ploeger, J. van de Pol, and T. A. C. Willemse. Equivalence Checking for Infinite Systems Using Parameterized Boolean Equation Systems. In *Proceedings 18th International Conference on Concurrency Theory CONCUR'07*, volume 4703 of *Lecture Notes in Computer Science*, pages 120–135. Springer-Verlag, 2007.

[DPW08] A. van Dam, B. Ploeger, and T.A.C. Willemse. Instantiation for Parameterised Boolean Equation Systems. In *Proceedings 5th International Colloquium on Theoretical Aspects of Computing ICTAC'08*, volume 5160 of *Lecture Notes in Computer Science.* Springer-Verlag, 2008.

[FJT10a] M. A. Feliú, C. Joubert, and F. Tarín. Efficient BES-based Bottom-Up Evaluation of Datalog Programs. In *Proceedings of the X Jornadas sobre Programación y Lenguajes (PROLE'10)*, pages 165–176, 2010.

[FJT10b] M. A. Feliú, C. Joubert, and F. Tarín. Evaluation strategies for datalog-based points-to analyses. In *Proceedings of the 10th Workshop on Automated Verification of Critical Systems (AVoCS'2010)*, page To appear, 2010.

[GMLS07] H. Garavel, R. Mateescu, F. Lang, and W. Serwe. CADP 2006: A Toolbox for the Construction and Analysis of Distributed Processes. In *Proceedings 19th International Conference on Computer Aided Verification CAV'07*, volume 4590 of *Lecture Notes in Computer Science*, pages 158–163. Springer-Verlag, 2007.

[Han94] M. Hanus. The Integration of Functions into Logic Programming: From Theory to Practice. *Journal on Logic Programming*, 19&20:583–628, 1994.

[HHI$^+$01] J. Y. Halpern, R. Harper, N. Immerman, P. G. Kolaitis, M. Y. Vardi, and V. Vianu. On the unusual effectiveness of logic in computer science. *Bulletin of Symbolic Logic*, 7(2):213–236, 2001.

[JM06] C. Joubert and R. Mateescu. Distributed On-the-Fly Model Checking and Test Case Generation. In *Proceedings 13th International SPIN Workshop on Model Checking of Software SPIN'06*, volume 3925 of *Lecture Notes in Computer Science*, pages 126–145. Springer-Verlag, 2006.

[JV05] D. W. Jorgenson and K. Vu. Information technology and the world economy. *Scandinavian Journal of Economics*, 107:631–650, Dec 2005.

[Lee90] J. van Leeuwen, editor. *Formal Models and Semantics*, volume B. Elsevier, The MIT Press, 1990.

[LS98] X. Liu and S. A. Smolka. Simple Linear-Time Algorithms for Minimal Fixed Points. In *Proceedings 25th International Colloquium on Automata, Languages, and Programming ICALP'98*, volume 1443 of *Lecture Notes in Computer Science*, pages 53–66. Springer-Verlag, 1998.

[LS09] Y. A. Liu and S. D. Stoller. From datalog rules to efficient programs with time and space guarantees. *ACM Trans. Program. Lang. Syst.*, 31(6), 2009.

[LWL05] B. Livshits, J. Whaley, and M.S. Lam. Reflection Analysis for Java. In *Proceedings of the Third Asian Symposium on Programming Languages and Systems (APLAS'05)*, pages 139–160, 2005.

[Mar94] M. Marchiori. 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*, pages 223– 241. Springer-Verlag, 1994.

[Mat98] R. Mateescu. Local Model-Checking of an Alternation-Free Value-Based Modal Mu-Calculus. In *Proceedings 2nd International Workshop on Verication, Model Checking and Abstract Interpretation VMCAI'98*, 1998.

[Mes92] J. Meseguer. Conditional Rewriting Logic as a Unified Model of Concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[MMW85] Z. Manna, and R. Waldinger. *The Logical Basis for Computer Programming.* Addison-Wesley, 1985.

[MT08] R. Mateescu and D. Thivolle. A Model Checking Language for Concurrent Value-Passing Systems. In *Proceedings 15th International Symposium on Formal Methods FM'08*, volume 5014 of *Lecture Notes in Computer Science.* Springer-Verlag, 2008.

[PK82] R. Paige and S. Koenig. Finite differencing of computable expressions. *ACM Transactions on Programming Languages and Systems*, 4(3):402–454, 1982.

[Red84] U. S. Reddy. Transformation of Logic Programs into Functional Programs. In *Proceedings of the Symposium on Logic Programming (SLP'84)*, pages 187–197. IEEE Computer Society Press, 1984.

[RH05] G. Rosu and K. Havelund. Rewriting-Based Techniques for Runtime Verification. *Automated Software Engineering*, 12(2):151–197, 2005.

[SKGST07] P. Schneider-Kamp, J. Giesl, A. Serebrenik, and R. Thiemann. 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*, pages 177–193. Springer-Verlag, 2007.

[Ull85] J. D. Ullman. Implementation of logical query languages for databases. *ACM Transactions on Database Systems*, 10(3):289–321, 1985.

[Vie86] L. Vieille. Recursive Axioms in Deductive Databases: The Query/Subquery Approach. In *Proceedings 1st International Conference on Expert Database Systems EDS'86*, pages 253–267, 1986.

[VT06] J. Vilaseca i Requena and J. Torrent i Sellens. Tic, conocimiento y crecimiento economico: Un análisis empírico, agregado e internacional sobre las fuentes de la productividad. *Economía Industrial*, 360:41–60, 2006.

[WACL05] J. Whaley, D. Avots, M. Carbin, and M. S. Lam. 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*, pages 97–118. Springer-Verlag, 2005.

[Wha03] J. Whaley. Joeq: a Virtual Machine and Compiler Infrastructure. In *Proceedings Workshop on Interpreters, Virtual Machines and Emulators IVME'03*, pages 58–66. ACM Press, 2003.

[ZR08] X. Zheng and R. Rugina. Demand-driven alias analysis for C. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages POPL'08*, pages 197–208. ACM Press, 2008.