The Semantics of Datalog for the Evidential Tool Bus* (Extended Abstract)

Simon Cruanes[1], Stijn Heymans[2], Ian Mason[3], Sam Owre[3], and Natarajan Shankar[3]

[1] Ecole Polytechnique, Palaiseau, France
[2] Artificial Intelligence Center, SRI International, Menlo Park, CA 94025, USA
[3] Computer Science Laboratory, SRI International, Menlo Park, CA 94025, USA
simon.cruanes.2007@polytechnique.org
Stijn.Heymans@sri.com
{Iam, Owre, Shankar}@csl.sri.com

Dedicated to Kokichi Futatsugi for his inspiring vision and generous spirit.

Abstract. The Evidential Tool Bus (ETB) is a distributed framework for tool integration for the purpose of building and maintaining assurance cases. ETB employs Datalog as a metalanguage both for defining workflows and representing arguments. The application of Datalog in ETB differs in some significant ways from its use as a database query language. For example, in ETB Datalog predicates can be tied to external tool invocations. The operational treatment of such external calls is more expressive than the use of built-in predicates in Datalog. We outline the semantic characteristics of the variant of Datalog used in ETB and describe an abstract machine for evaluating Datalog queries.

1 Introduction

Software is an important component of many modern safety-critical systems, and its reliability must therefore be certified to very high levels of assurance. It is quite common for an assurance case for software to be developed using workflows that integrate multiple formal, semi-formal, and informal tools. The capabilities offered by these tools span the software lifecycle from requirements capture and validation, to design and verification, and eventually system integration and testing. At SRI, we have been developing a framework for software assurance called the Evidential Tool Bus (ETB) [5]. The ETB middleware can be used to integrate external tools through tool wrappers, to define workflows, and to

* This work was supported by NSF Grant CSR-EHCS(CPS)-0834810, NASA Cooperative Agreement NNA10DE73C, and by DARPA under agreement number FA8750-12-C-0284. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of NSF, NASA, DARPA or the U.S. Government. We are grateful for the insightful feedback we received from the anonymous referees and from Mark Utting of the University of Waikato.

collect claims and evidence in support of a well-defined argument. The Datalog fragment of Horn clause programming is at the core of ETB. Datalog is used as the metalanguage, both for scripting workflows that incorporate multiple tools, and for representing assurance arguments. ETB differs from other application of Datalog in some subtle but significant ways. Since we are using Datalog for developing assurance cases, it is important to capture the semantic details of the language in a rigorous manner. We outline the semantic peculiarities of ETB Datalog and define an abstract machine for the evaluation of Datalog programs in the context of a distributed computation.

Workflows for software assurance involve *semi-formal* steps for validation, testing, and hazard analysis; *formal* steps for verification, synthesis, and test generation; and *informal* steps such as checklists and human inputs. From the viewpoint of assurance, the end result of such a workflow must be a certifiable assurance case consisting of claims supported by arguments and evidence. Many verification workflows involve multiple tools: type checkers, static analyzers, SAT and SMT solvers, interactive and automated theorem provers, and symbolic and explicit-state model checkers. The tools and inference rules used in the argument must be expressly qualified for use in the assurance case. Each of the different tools might work only with certain languages and representations, so that translations between different representations will also be a key part of the workflow. An assurance case constructed from the workflow is a collection of artifacts (files, properties, metrics, etc.) along with claims about these artifacts, and arguments in support of these claims. For the purpose of certification, it is desirable that arguments representing the assurance case be replayable. It should also be possible to maintain the argument against changes to inputs (e.g., requirements) as well as modifications to the tools.

The Evidential Tool Bus framework has been outlined in an earlier paper [5]. We summarize the key points below. ETB is a distributed framework for tool integration. An ETB installation is a network of ETB servers as shown in Figure 1, where each server can offer specific services. ETB uses Datalog as the scripting language for defining workflows as well as the metalanguage for building arguments. Services are packaged as Datalog predicates. Each ETB server runs a Datalog engine that can be used to implement workflows integrating different services. The claims are maintained together with their proofs. A service is associated with a Datalog predicate by means of a wrapper. For example, the Yices SMT solver can be offered as a service through the Datalog predicate $yices(F, S, M)$, where the variable[4] F represents a file containing a formula in the Yices input language, S is the result, *sat* or *unsat*, of the satisfiability check, and M is the model when S is *sat*. If *a.ys* is a Yices file containing a formula, then the query $yices(a.ys, S, M)$ invokes the Yices solver to bind the variables S and M.

A workflow is defined as a Datalog program consisting of Horn clauses. For example, a workflow that generates a test input from a formula in a file F and executes it on a program P can be defined by the Horn clause below, where

[4] As is conventional in logic programming, identifiers starting with uppercase letters are variables.

$A :- B, C, D$ represents the clause $(B \wedge C \wedge D) \implies A$.

$$gentest(F, P, Result) :- yices(F, S, M), equal(S, sat), test(P, M, Result)$$

If $a.ys$ is a file defining a Yices formula and b is a file containing a program, then $gentest(a.ys, b, Result)$ executes the workflow on these files and binds the test results to the variable $Result$. A query is invoked by a client and is evaluated by a Datalog engine at a server in the ETB network. The server can invoke services that are available at other nodes in the network. The *gentest* query above is evaluated at a specific server, but might use the *yices* service at a remote server by copying the input files to the remote server and copying back any files representing the results.

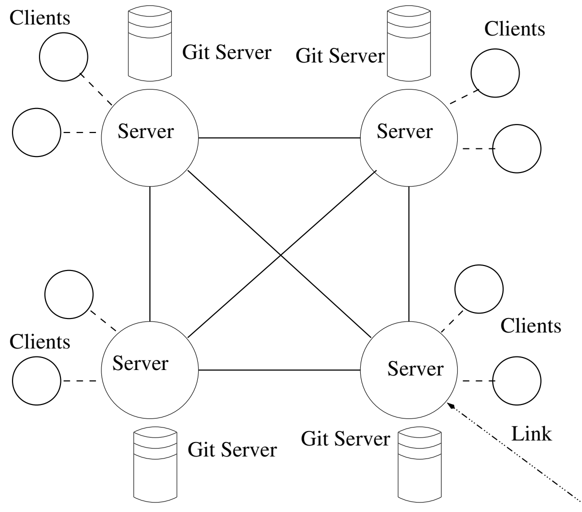

Fig. 1. The ETB Client-Server Architecture

The Datalog variant used in ETB serves as an integration language for a distributed network of services. It is also used as a representation language for assurance arguments that build in calls to certain trusted services. External services are invoked through queries with external predicates such as the *yices* predicate. Such invocations are operationally quite similar to the evaluation of internal predicates, e.g., *gentest*, in the sense that the evaluation step returns a (possibly empty) set of clauses whose head atoms are instances of the query atom. This leads to a richer notion of external predicates than the traditional *built-in* predicates in Datalog. Furthermore, the similarity between the evaluation of internal and external predicates yields a somewhat uniform denotational and operational semantics for the Datalog variant used in ETB.

There is a large volume of work on Datalog related to its use in Databases. This work has its origins in a workshop on *Logic and Data Bases* organized by Gallaire and Minker [6]. Details of the use and implementation of Datalog as a programming language are covered in the book *Foundations of Databases* by Abiteboul, Hull, and Vianu [1], and in several survey articles [3,7,8]. Our focus here is on Datalog extended with a specific mechanism for service invocation that is richer than the interpretation of *built-in* predicates. We present the semantics of this version of Datalog that is suitable for use in the ETB framework. The abstract machine we present employs *tabling* to memoize the computation of repeated subgoals, and the presentation here is somewhat related to the abstract machine for tabled Datalog defined by Sagonas and Swift [10].

We describe the peculiarities of ETB Datalog in Section 3. We then present the semantics of ETB Datalog in Section 4. Concluding observations and future work are presented in Section 6.

2 A Brief Overview of Datalog

Though Datalog was first introduced as a deductive language for defining database queries, it has found applications in a number of other areas such as declarative networking [9], static analysis [12], distributed computing[2], and parallel programming [4]. The core of Datalog is a Horn clause programming language where the terms are either variables or constants. Typical applications of Datalog employ a fragment that includes a notion of negation, but we restrict ourselves to the positive fragment.

The Datalog language is specified relative to a set of *constants* C, a set of *variables* V, and a set of *predicates* Σ, where each predicate has an arity. An *atom* is of the form $p(a_1, \ldots, a_n)$, where p is an n-ary predicate in Σ, and each a_i is a *term*, i.e., either a variable in V or a constant in C. A *ground* atom is an atom that contains no variables. A *rule* is of the form $A :- Q$, where A is the *head* atom of the rule, and the *body* Q is a (possibly empty) sequence of atoms $A_1, \ldots, A_n$. The set of variables occurring in the head A must be a subset of those occurring in the body Q. A program R is a set of rules. A predicate p is defined in R by the set of rules in R of the form $p(a_1, \ldots, a_n) :- Q$.

For example, the program below defines a *sibling* relation given the *father* and *mother* relations. The parent relation is defined as the union of the father and mother relations.

$$\begin{aligned} &\mathit{sibling}(X, Y) :- \mathit{parent}(Z, X), \mathit{parent}(Z, Y) \\ &\mathit{parent}(X, Y) :- \mathit{father}(X, Y) \\ &\mathit{parent}(X, Y) :- \mathit{mother}(X, Y) \end{aligned}$$

A Datalog program will contain both rules, such as the definitions shown above, as well as facts which are just ground atoms such as

$$\begin{array}{l} \mathit{father}(joe, \mathit{bill}) \\ \mathit{mother}(mary, joe) \\ \mathit{father}(jim, mary) \\ \mathit{father}(jim, bob) \end{array}$$

Informally, a Datalog program R consisting of rules and facts entails a set of ground atoms H, i.e., every atom in H holds in every model of R. A query is a negated atom of the form $\neg p(a_1, \ldots, a_n)$. The answers to such a query are the ground instances of $p(a_1, \ldots, a_n)$ in H, namely, the refutations of the query. For example, if the query is $\neg \mathit{sibling}(mary, x)$, then the answers are $\mathit{sibling}(mary, mary)$ (which is counterintuitive, but what the definition implies) and $\mathit{sibling}(mary, bob)$.

The definition of the *sibling* relation can already be formulated in first-order logic, but Datalog can also capture recursive definitions that are not expressible in first-order logic. The *ancestor* relation can be given the recursive Horn clause definition shown below.

$$\begin{array}{l} \mathit{ancestor}(X, Y) :- \mathit{parent}(X, Y) \\ \mathit{ancestor}(X, Y) :- \mathit{parent}(Z, Y), \mathit{ancestor}(X, Z) \end{array}$$

For example, the query $\neg \mathit{ancestor}(X, \mathit{bill})$ yields the answers

1. $\mathit{ancestor}(joe, \mathit{bill})$
2. $\mathit{ancestor}(mary, \mathit{bill})$
3. $\mathit{ancestor}(jim, \mathit{bill})$

3 Datalog as used in ETB

As a metalanguage for ETB, Datalog offers a simple semantic framework for expressing claims, composing arguments, and defining workflows that direct the flow of information and work to and from the external tools. For example, the following ETB Datalog program generates all the satisfying assignments for a Boolean formula.

$$\begin{array}{rl} sat(F, M) :- & yices(F, S, M), equal(S, sat) \\ unsat(F) :- & yices(F, S, M), equal(S, unsat) \\ allsat(F, Answers) :- & sat(F, M), \\ & negateModel(F, M, NewF), \\ & allsat(NewF, T), \\ & cons(M, T, Answers) \\ allsat(F, Answers) :- & unsat(F), nil(Answers) \end{array}$$

The query $\neg yices(f, S, M)$ triggers the invocation of the Yices SMT solver on the Yices formula in the file corresponding to the file handle f to return the

result *sat* or *unsat* binding S. In the latter case, the model M is irrelevant and is bound to the Yices formula $false$. In the former case, the variable M is bound to the model which is given as a Yices formula that is a conjunction of literals. The program for the predicate *allsat* invokes the Yices solver to compute a model M for the formula in the file f. Its negation is conjoined with the formula in F and placed in a new file with the handle $NewF$. The *allsat* procedure is repeated on $NewF$ until the formula becomes unsatisfiable. The list of all the assignments is bound to the variables $Answers$. Note that even the list operations of binding $Answers$ to *nil* and the pairing operation $cons(M, T, Answers)$ are implemented as external calls.

Let f be the file handle for the file containing the input Yices formula. A goal query, e.g., $\neg allsat(f, Answers)$ is evaluated with respect to a set of rules by means of backward chaining. The goal is resolved with all the program clauses where *allsat* is the head predicate, i.e., the predicate of the head atom. There are two such clauses. In both cases, unification binds the variable F with the file handle f. The leading atoms of the body, namely, $sat(f, M)$ and $unsat(f)$, then become new goals. Backward chaining on these goals leads to the evaluation of $yices(f, S, M)$. Although the subgoal $yices(f, S, M)$ occurs twice in the evaluation, it is only evaluated once. If Yices finds the formula in the file (handle) f to be satisfiable, then it binds S to *sat* and M to the resulting model which we can label as m_1. Since the evaluation of $unsat(F)$ returns no bindings, we can terminate the evaluation of the second clause in the definition of *allsat*. The evaluation of the body of the first clause continues with the evaluation of $negateModel(F, M, NewF)$. This creates a new file where the contents of the file handle f have been augmented with the assertion of the negation of the formula corresponding to model m_1. The *allsat* program is now evaluated recursively on this new file. The answer m_1 is added to the list of assignments $m_2, \ldots, m_n$ returned by the recursive evaluation.

The bulk of the computation is carried out by these external tools. Predicates, like *yices*, that invoke external tools are called *interpreted* or *external* predicates. They are similar to built-in predicates in Datalog. However, built-in predicates are usually invoked on ground arguments whereas the invocation in ETB of an interpreted predicate will involve binding the variables to zero or more bindings. The typical evaluation of a query involving an interpreted predicate, such as *yices* will return at most one binding, but there are predicates that can return multiple bindings. Another difference with built-in predicates is that the evaluation of an interpreted predicate can generate further queries. For example, an interpreted query for computing a definite integral of a function over an interval using Risch's algorithm might return a result with the qualification that the function must be defined and continuous over the interval. Queries returned by the external procedure can be used to guard the answers with side conditions or reflect the case analysis in the computation.

We employ standard mathematical notation in presenting the details of ETB Datalog. The metavariables a and b range over Datalog terms, i.e., variables and constants. The metavariable p ranges over Datalog predicates, the metavariable A ranges over atoms, and Q ranges over conjunctions of atoms. In running text,

a clause is bracketed for ease of reading and is represented as $A :- Q$, with head A and body Q.

The Datalog variant used in ETB has several distinctive features:

1. The basic evaluation scheme is backward chaining on rules through resolution with queries. The body literals in the rule are evaluated through backward chaining, left-to-right order. This order of evaluation is significant. Backward chaining is needed to ensure that only the relevant external predicate calls are evaluated. The left-to-right order of evaluation on the body of a rule ensures that external predicates are not evaluated until their preconditions have been verified. For example, it does not make sense to invoke the PVS prover on a formula that has not yet been typechecked.
2. As in tabled evaluation [10], the searches are memoized to avoid repeated computation.
3. An external predicate corresponds to a service that might be available only from specific servers. In order to provide this service, the corresponding server has one or more wrappers associated with the predicate. Each wrapper covers a specific mode for the predicate. The modes specify the arguments to the predicate that are provided as inputs, and some subset of the remaining arguments might be computed through the evaluation of the wrappers. For example, the invocation of the *yices* predicate has the mode $\langle +, -, - \rangle$ in the *allsat* program.
4. Though the external calls can return multiple bindings (i.e., substitutions) for the outputs, most wrappers are expected to return at most one binding. This means that we can adopt a Prolog-style, tuple-at-a-time mode of evaluation rather than computing with sets of tuples.
5. External calls can generate further queries. This means that the evaluation of an external call, e.g., $p(a_1, \ldots, a_n)$ can return clauses of the form $p(b_1, \ldots, b_n) :- Q$. Most implementations of Datalog restrict external calls to ground atoms, i.e., atoms of the form $p(b_1, \ldots, b_n)$ for ground terms $b_1, \ldots, b_n$.
6. The evaluation of Datalog queries relative to a program returns a set of claims. Each claim is supported by a derivation or a proof. The derivation should be replayable, and it should be possible to identify the evidence artifacts such as files (and file contents) that are used in the derivation.
7. For the purpose of developing an assurance argument, we can restrict the external predicates and Datalog rules that are sanctioned for use in the construction of a derivation.
8. Query evaluation is distributed in the sense that the evaluation of external calls can take place at a remote ETB server. This means that the evaluation must be asynchronous — at any given point in the computation, a server can be awaiting results from multiple external calls. In some cases, a service might fail during evaluation or might only become available after a delay.

The above features of ETB Datalog require special treatment that are not offered by existing implementations of Datalog. We present the semantics of ETB Datalog and describe an abstract machine that captures the evaluation of Datalog queries in this framework.

4 Semantics of ETB Datalog

The semantics of the Datalog language can be given by a traditional first-order structure M that maps each element c of C to an element $\mathbf{c}$ in the domain $|M|$, and each n-ary predicate to a subset of the set $|M|^n$ of the n-tuples from $|M|$. The meaning of a rule set R can be given by a set H of ground atoms such that for any model M of R: $M \models A$ for $A \in H$. In this case, we say $R \models A$. Given a goal query $\neg p(a_1, \dots, a_n)$ and a rule set R, a Datalog computation should return the set of valid ground instances of the goal query, i.e., those atoms A that are instances of $p(a_1, \dots, a_n)$ such that $R \models A$.

One way to check $R \models A$ is by computing the minimal Herbrand model for R. This is done starting with H_0 as the empty set. Each successive H_{i+1} is computed by closing under the application of rules from R so that

$$H_{i+1} = \{\hat{B} | \hat{B} = \sigma(B) \text{ for ground } \hat{B}, \sigma(Q) \subseteq H_i, B :- Q \in R\}.$$

Then $H = H_i$ for the least i such that $H_i = H_{i+1}$. It can be checked that $R \models A$ iff $A \in H$. Note also that for a given R, the set H is finite, even if the set of constants C is infinite.

An operational way to compute the valid ground instances of the goal query is through depth-first backward search. We introduce the operations of substitution and unification as a prelude to the operational semantics. A substitution σ is a partial map from variables in V to terms, e.g., $[v_1 \mapsto a_1, v_2 \mapsto a_2]$. For such a partial map, $\sigma(x) = a$ if σ maps x to a, and $\sigma(x) = x$, otherwise. A substitution σ can be applied to an atom A as $\sigma(A)$, a rule body Q as $\sigma(Q)$, or a rule K as $\sigma(K)$. In each case, the result is obtained by substituting $\sigma(x)$ for each occurrence of a variable x. A substitution σ is at least as general as another substitution σ' if $\sigma(x) = \sigma'(x)$ whenever $\sigma(x)$ is defined. An atom A is an *instance* of an atom B if there is a substitution σ such that $\sigma(B) = A$. Conversely, B is said to be a *generalization* of A. A substitution σ is at least as general as another substitution σ' if $\sigma(A)$ is at least as general than $\sigma'(A)$, for any atom A. A substitution σ is a *unifier* of two atoms A and B if $\sigma(B)$, i.e., the *unification*, is an instance of A. We have given an asymmetric definition of unification so that we can avoid renaming the variables in A and B apart. A substitution σ is the *most general* unifier of two atoms A and B if it yields a unification $\sigma(B)$ that is at least as general as the unification resulting from another unifier. The operation $mgu(A, B)$ is the most general unifier of A and B when it exists, and is $\perp$, otherwise.

Unification is used to compute $D_R(A)$, the valid ground instances of A given the rule set R. It is defined mutually recursively with the operation $D_R(Q)$ that computes the valid ground instances of a sequence of atoms Q. We define $D_R(Q)$ as

$$D_R(A, Q) = \{A', Q' | A' \in D_R(A), \sigma(A) = A', Q' \in D_R(\sigma(Q))\},$$

and $D_R(\epsilon) = \epsilon$, where ϵ is the empty sequence. Let $R(A)$ be the set of clauses $\{\sigma(B :- Q) | B :- Q \in R, \sigma = mgu(A, B) \neq \perp\}$. We can complete the mutual

recursion by defining $D_R(A)$ as

$$\{B' | B :- Q \in R(A), Q' \in D_R(Q), \sigma(Q) = Q', B' = \sigma(B)\}.$$

The sets $D_R(A)$ and $D_R(Q)$ are finite and contain all and only the ground instances of A and Q, respectively, that are valid in R.

In ETB, we also include an external oracle E that interprets the *external predicates*, which we take as any predicate that is not defined in R.[5] With external predicates, we give up the property that there is a finite Herbrand model. For example, if we have an external predicate *successor* such that $successor(0, Y)$ returns the binding of 1 to Y, and $successor(K, Y)$ is the successor of the numeral K, then we can write a Datalog program that computes all of the natural numbers:

$$\begin{aligned} &nat(0) \\ &nat(X) :- nat(Y), successor(Y, X) \end{aligned}$$

We can in fact recover the full power of Prolog through external oracles that perform unification.

Examples of atoms in external predicates can range from simple built-in operations such as $less(x, y)$ and $subrange(low, high, i)$ to wrapper calls such as $yices(f, S, M)$. The interpretation $E(p(a_1, \ldots, a_n))$ for an atom is performed by a *wrapper*. For example, the evaluation of the external predicate

$$yices(filename.ys, s, m)$$

invokes a wrapper that executes the Yices SMT solver on the input from the file $filename.ys$, and binds the result, *sat* or *unsat*, to the variable s, and a model, if one exists to the variable m. In general, the interpretation $E(p(a_1, \ldots, a_n))$ returns a (possibly empty) list of clauses where each clause has the form

$$p(b_1, \ldots, b_n) :- Q.$$

The head atom $p(b_1, \ldots, b_n)$ must be an instance of the query atom $p(a_1, \ldots, a_n)$, and the variables in the head must also occur in the body. Most Datalog variants admit only a limited interpretation of external predicates where the queries must be fully grounded, whereas in ETB, we allow a more liberal and expressive interpretation of external predicates that allows further queries to be spawned. This interpretation also makes the behavior of E and R similar with respect to the operational semantics.

Each external predicate can be evaluated under one or more *modes*. An n-ary mode for an n-ary predicate is a sequence of symbols length n, where each symbol is either $+$ or $-$. The positions marked by $+$ are the input arguments for the predicate, and these have to be grounded in the query, whereas the positions marked $-$ are the outputs that are bound during the evaluation of the query. For

[5] We disallow the possibility of a predicate being defined both in R and E since in our semantics, the same effect can be achieved solely through external oracles.

an atom $p(a_1, \ldots, a_n)$, $mode(p(a_1, \ldots, a_n))$ is the sequence $m_1, \ldots, m_n$, where each m_i is either $+$ or $-$, and a_i is a variable exactly when m_i is $-$. An external predicate might have wrappers associated only with specific modes. For example, the query $subrange(0, High, 3)$ is not sensible since the set of bindings for $High$ is infinite. Similarly, $yices(F, unsat, M)$ should not have a wrapper associated with it since it requires finding a file containing an unsatisfiable formula, and there could be unboundedly many such files.

There is a partial ordering on the modes of an external predicate so that one mode is narrower than another if the set of input arguments of the first mode is a superset of the set of input arguments of the second mode. If a mode is interpretable for an external predicate, then any narrowing of this mode obtained by turning outputs arguments into input arguments must also be interpretable. This can be satisfied by interpreting p with a more general mode, i.e., one where some of the input arguments are outputs, and filtering the results relative to the additional input arguments. For example, to compute $E(p(c_1, \ldots, c_n))$, we can instead compute $E(p(a_1, \ldots, a_n))$, where each a_i is either c_i or a fresh variable v_i. The resulting clauses can then instantiated and filtered so that the head atom is always an instance of $p(c_1, \ldots, c_n)$. This ensures, for example, that it is always possible to invoke an external call on a fully grounded atom.

The wrappers for the different modes of an external predicate have to be *compatible*, so that even if there are multiple wrappers for p that can be used to compute $E(p(a_1, \ldots, a_n))$, the set of clauses returned is the same. In ETB, we do not check the compatibility of the wrappers for a given external predicate. Instead, we assume that every external predicate has a wrapper for the fully grounded mode, i.e., one where all the arguments are inputs. This is the only wrapper that needs to be trusted since it will be used to check the arguments associated with the final set of claims.

Herbrand models do not make sense for external oracles since new constants can be generated when an oracle is invoked. We can instead construct a relatively closed Herbrand model where an oracle generates a set of ground external atoms Ω and $E[\Omega]$ is $\bigcup\{E(A) | A \in \Omega\}$. Then, a ground atom A is a consequence of R and E (relative to the oracle Ω) if $R \cup E[\Omega] \models A$. The minimal Herbrand model can be constructed by defining H_0 as the empty set and $H_{i+1} = \{\hat{B} | \hat{B} = \sigma(B)$ for ground $\hat{B}, \sigma(Q) \subseteq H_i, B :- Q \in R \cup E[\Omega]\}$. We can then say that the ground atom A is a consequence of R and E if for *some* set of ground external atoms Ω, A is a consequence of $R \cup E[\Omega]$. The abstract machine in Section 5 defines a specific set Ω from which Herbrand models can be constructed.

The model-theoretic semantics and the minimal Herbrand model do not yield effective operational methods for computing the set of answers to a query relative to a rule set R and an external oracle E. In the next section, we present an abstract machine for answering Datalog queries.

5 An Abstract Machine for ETB Datalog

We define an abstract machine for the ETB Datalog engine and argue for its correctness relative to the semantics given in the previous section. The engine

evaluates a goal query of the form $\neg p(a_1, \ldots, a_n)$ against a set of rules R and external oracle E. The goal is to return all and only those ground instances of $p(a_1, \ldots, a_n)$ that are entailed by the rules R together with the external oracle E.

In order to easily check for equality, all the expressions are maintained in normalized form so that variables are ordered by occurrence so that v_i names the i'th distinct variable occurring in the expression. For example, the atom $p(X, c_1, Y)$ is normalized as $p(v_1, c_1, v_2)$. Similarly, a clause $p(X, c_1, Y) :- p(X, c_2, Z), p(Z, c_3, Y)$ is represented as $p(v_1, c_1, v_2) :- p(v_1, c_2, v_3), p(v_3, c_3, v_2)$.

5.1 An Abstract Inference System

We can first capture the Datalog computation as an abstract inference system [11]. For this we define a *logical state* is a pair $\langle G, J \rangle$, where

1. The set G consists of goals so that G is $\{\neg A_1, \ldots, \neg A_n\}$
2. The set J consists of clauses of the form $B :- Q$ or of the form B, where Q is a nonempty list of atoms. A *claim* is a clause of the form B in J. It must be ground because of the condition that any variables in the head must occur in the body, and the body here is empty.

In each inference step, we perform one of the following steps:

1. **Backchain:** For a clause of the form $B :- A_1, \ldots, A_n$ in J, we add the goal $\neg A_1$ to G if it is not already in G.
2. **Resolve:** For a goal $\neg A$ in G and clause $B :- Q$ in R, we add a new clause K to J, where $\sigma = mgu(A, B) \neq \perp$ and $K = \sigma(B :- Q)$.
3. **External:** For a goal $\neg A$ in G where A is an external atom and a clause K in $E(A)$, we add a new clause K to J.
4. **Propagate:** For a clause in J of the form $B :- A, Q$ and another clause in J of the form A', with σ such that $\sigma(A) = A'$, we add the new clause $\sigma(B :- Q)$ to J.

The initial logic state consists of $G = \{\neg A\}$ where $\neg A$ is the initial goal. The inference procedure terminates when no further inference steps can be applied, i.e., when the logic state is *irreducible*. The result of the computation is the set $\{B \in J | \sigma(A) = B\}$.

The abstract inference system described here is sound and complete with respect to the semantics given above. Given an irreducible state $\langle G, J \rangle$, the set Ω is the set of all claims of the form $\{B \in J | B \text{ is an external atom}\}$. Then, each claim in J is a consequence of $R \cup E[\Omega]$. Additionally, for an initial goal $\neg A$, expanding Ω does not add any new claims of the form B to J where B is an instance of A.

5.2 An Abstract Machine

The abstract inference system above captures the basic idea of using resolution to compute with Datalog programs, but it has a major source of inefficiency.

The logic state is not suitably indexed so that the number of steps for finding an applicable inference step can be quadratic in the number of formulas in the state. The number of formulas can itself be exponential in the size of the universe. We can improve the performance of the abstract machine through better indexing and structuring.

Another problem with the abstract inference system is that our Datalog engine works in a distributed setting where new queries can be added from other nodes in the network. In this case, we might be done processing one goal query but the state might not be irreducible because other goals are still being processed. A termination check is needed to determine if a goal has been fully processed and all of the associated claims have been generated.

We modify the inference system to address these drawbacks. In the extended system, the state now consists of goal nodes G, clause nodes J, and an index T. The goal and clause nodes are enriched with annotations. Each entry g in G now consists of

1. *Literal:* The actual goal literal.
2. *Parents:* A set of the clauses in J from which the goal originated. This entry can be empty if the goal was introduced at the top level. Note that a goal can have multiple parents.
3. *Index:* The index that uniquely identifies a goal node. The *Index* slot is used in timestamps for checking termination.
4. *Claims:* A *sequence* of claims, i.e., clause nodes j where $j.Clause$ is of the form B, instantiating the goal.
5. *Children:* The set of clause nodes obtained by applying R or E to the Atom.
6. *Status:* *Open*, *Resolved*, *Closed*, or *Completed*.[6]

Each entry j in J consists of

1. *Clause:* The actual clause corresponding to the entry.
2. *Goal:* The parent goal in G from which the clause node originates.
3. *Subgoal:* A pointer to the subgoal in G generated from the clause. This slot could be empty. If $j.Clause$ is of the form $B :- A, Q$, then the $j.Subgoal.Literal$ is $\neg A$. Furthermore, $j.Subgoal.Parents$ contains j.
4. *Subclause:* A set of clause nodes that are derived by propagating from j.
5. *SubgoalIndex:* The number of claims corresponding to the subgoal that have already been propagated. It is initially zero when node j is created and is bumped up by one for each claim that is propagated from the subgoal.

We say that one goal h is an *immediate subgoal* of another goal g if there is some j such that $j.Goal = g$ and $j.Subgoal = h$. The inference steps can now be rewritten to operate on the annotated logic state.

1. **Backchain:** For a clause node j in J with $j.Clause$ of the form $B :- A_1, \ldots, A_n$, where $j.Subgoal$ is empty,

[6] In the implementation, a goal may also be *Stuck* if there is neither a rule nor a wrapper associated with it. This can happen, for example, if the server providing the wrapper is temporarily unavailable.

(a) If there is already a goal node g in G with $g.Atom$ slot of the form $\neg A_1$, we add j to $g.Parents$ and set $j.Subgoal$ to g and $j.SubgoalIndex$ to 0.
(b) Otherwise, if there is no goal node g in G with $g.Atom$ of the form $\neg A_1$, we create a new goal node g' so that
    i. $g'.Literal$ is $\neg A_1$,
    ii. $g'.Parents$ is $\{j\}$,
    iii. $g'.Index$ is $T + 1$,
    iv. $g'.Claims$ is the empty sequence,
    v. $g'.Status$ is *Open*.

    All the other fields of g are left empty, and the global time parameter T in the state is incremented by one.

2. **Resolve:** For a goal node g in G with $g.Status = Open$, and $g.Literal = \neg A$, and each clause $B :- Q$ in R, we add a new clause node j to J, where $j.Clause = K = \sigma(B :- Q)$ with $\sigma = mgu(A, B) \neq \perp$, $j.Goal = g$, and $j.SubgoalIndex = 0$, with all the other fields empty. We also set $g.Status$ to *Resolved*.
3. **External:** For a goal node g in G with $g.Status = Open$, and $g.Literal = \neg A$, and for each clause K returned by $E(A)$, where $g.Literal = \neg A$,
we add a new clause node j to J, where $j.Clause = K$, $j.Goal = g$, and $j.SubgoalIndex = 0$, with all the other fields empty. We also set $g.Status$ to *Resolved*.
4. **Propagate:** For some goal node g and for some clause node j' in $g.Parents$, where $j'.Clause$ is $B :- A, Q$ and $j'.SubgoalIndex$ is smaller than the length of $g.Claims$,
let $j = g.Claims[j'.SubgoalIndex]$ with $j.Clause$ of the form A', we create a clause node j'' with
    (a) $j''.Clause$ set to $\sigma(B :- Q)$ where $A' = \sigma(A)$,
    (b) $j''.Goal$ set to $j'.Goal$, and
    (c) $j''.SubgoalIndex$ set to 0.

    Also, add j'' to $j'.Subclause$, set $j''.Subclause$ to the empty set, and increment $j'.SubgoalIndex$ by one.
5. **Claim:** For a clause j where $j.Claim$ is of the form B, we add j to the end of $j.Goal.Claims$ unless B is already present as $j'.Claim$ for some j' in $j.Goal.Claims$. We assume that this step is done immediately after a **Propagate**, **Resolve**, or **External** step whenever a claim is generated.

The abstract machine is initialized with a single initial goal node g with $g.Literal = \neg A$ with $g.Index = 1$ and with $T = 1$. The evaluation is terminated when no rule is applicable. In the next subsection, we augment the abstract machine with support for detecting termination. The key modifications in the machine defined above are

1. The clauses in J are no longer maintained as a set. This is to simplify the termination check.
2. The **Propagate** step does not scan all the claims but is instead triggered by the addition of a claim to the goal.

5.3 Abstract Machine with Termination Check

The abstract machine with the inference steps **Backchain**, **Resolve**, **External**, **Propagate**, and **Claim** is a refinement of the abstract inference system in Subsection 5.1. However, it lacks a way of checking that a subgoal g has been fully evaluated, i.e., no further claims can be added to $g.Claims$. There is a simple but impractical way to do this that is already implicit in the abstract inference system: if the computation is stuck so that no further inference steps can be applied, then the computation has terminated. This only works if we are evaluating a single query in a sequential setting. However, ETB is a distributed system where the Datalog engine is evaluating many queries simultaneously and these computations could be sharing subgoals. Some of these subgoals might be fully evaluated even while new queries are being added and other parts of the computation have only be partially completed. A global termination check will not work in this context. We still need a termination check so that completed subgoals can be garbage collected.

Checking termination is not straightforward since the evaluation graph consisting of goal and clause nodes can contain cycles. Swift and Sagonas [10] interleave the evaluation with a check for strongly connected components (SCCs) to identify the fully evaluated nodes. We present a more fine-grained method for checking termination that can be run alongside the normal evaluation. For this purpose, we augment the state of the goals g with

1. A map $g.T$ from goals to sets of clauses such that $g.T(h)$ is nonempty only when h is an immediate subgoal of the goal g, and $g.T(h)$ is the set of clauses $\{j | j.Goal = g \wedge j.Subgoal = h\}$.
2. A partial map $g.D$ from goals to a number so that $g.D(h)$ is defined only when $h.Index < g.Index$ and h is not closed. The entry $g.D(h)$ is the number of claims from h that have been fully propagated in the derivation rooted at g. This means that every sub-derivation of g has propagated at least k claims from h for $k = g.D(h)$. The partial map $g.D$ contains the unclosed subgoals of g at the point when the **Close** rule (defined below) is applied.
3. A slot $g.Unclosed$ which, when defined, is the maximal index of an unclosed subgoal h of g such that $h.Index < g.Index$. In particular, $g.Unclosed$ is defined when g is closed, and it is the maximal index of a goal h such that $g.D$ is defined.

We modify the **Backchain** step of the abstract machine so that whenever it is applied to a clause j to set $j.Subgoal$ to h, we also add j to $g.T(h)$, where $g = j.Goal$.

Define $min(i_1, i_2)$ for two possibly undefined numeric values i_1 and i_2 as

1. *undefined*, if both i_1 and i_2 are undefined
2. i_1, if i_2 is undefined or $i_1 \leq i_2$, and
3. i_2 if i_1 is undefined or $i_2 < i_1$.

For a set of indices I, $min(i, I)$ is i if I is empty, or it is the minimal index in $\{i\} \cup I$. If I is a nonempty set of indices, then $min(I)$ is the minimal index in I.

Close: The **Close** rule performs a step in the termination check computation. We say that g is *closed* if $g.Status$ is $Closed$ or $Completed$. When $g.Status = Closed$, then the only way a new claim can be added to g is if it is the result of adding a new claim to some subgoal h of g such that $h.Index < g.Index$. When $g.Status = Completed$, then no further claims can be added to g.

The **Close** rule is applied to a goal g where

1. $g.Status \in \{Resolved, Closed\}$, and for all $j \in g.Children$, $j.Clause$ is a claim or $j.Subgoal$ has been set. This ensures that the immediate children of g are either claims or have generated subgoals. Note that the **Backchain** rule registers a subgoal in $g.T$ as soon as it is generated.
2. For all h we check that either $g.T(h)$ is empty, $h.Index \leq g.Index$, or $h.Status$ is $Closed$ and $h.Unclosed \leq g.Index$ when $h.Unclosed$ is defined. In the latter two cases, we also check that for all j in $g.T(h)$, $j.SubgoalIndex = |h.Claims|$ and for all $j' \in j.Subclause$, either $j'.Clause$ is a claim or $j'.Subgoal$ has been set. This check ensures that we have a complete set of subgoals that have propagated all their claims, and the resulting clauses have also generated their subgoals (if any).

When this check is valid for a goal g, we compute the value of $g.D(h)$ for h such that $h.Index < g.Index$. We first compute for any h such that $h.Index \leq g.Index$, the set

$$\tau(g)(h) = \{h'.D(h) | h' \text{ is closed}, g.T(h') \text{ is nonempty}, h'.D(h) \text{ is defined}\}.$$

If $\tau(g)(g)$ is either empty or $min(\tau(g)(g)) = |g.Claims|$, then we mark $g.Status$ as $Closed$ and then set $g.D(h)$ as below for unclosed h such that $h.Index < g.Index$. If $g.T(h)$ is nonempty, $g.D(h)$ is set to $min(|h.Claims|, \tau(g)(h))$. Otherwise, we set $g.D(h)$ to $min(\tau(g)(h))$. In any remaining case, $g.D(h)$ is undefined. Note that because of the way that τ is computed, $g.D(h)$ is defined only when h is not closed and $h.Index < g.Index$.

Once $g.D$ is set, we can recompute $g.Unclosed$ as the maximal unclosed h such that $g.D(h)$ is defined. If $g.D$ is everywhere undefined, then $g.Unclosed$ is also undefined. The information that g is closed needs to be propagated to any goal node h such that $h.Unclosed = g$, and this happens when the **Close** rule is applied to h.

Complete: The rule **Complete** marks nodes as completed. If for some g, $g.Status$ is $Closed$ then $g.Status$ can be set to $Completed$ if either

1. $g.D(h)$ is everywhere undefined, or
2. For some goal h such that g is an immediate subgoal of h, $h.Status = Completed$. Recall that g is an immediate subgoal of h when for some j in $g.Parents$, $j.Goal = h$.

5.4 An Example

We illustrate the abstract machine on a simple example using the program in Figure 2 consisting of clauses C_1 through C_7.

C_1	$black(a,b)$
C_2	$white(b,c)$
C_3	$white(b,a)$
C_4	$blackpath(X,Y) :- \quad black(X,Y)$
C_5	$blackpath(X,Y) :- black(X,Z), whitepath(Z,Y)$
C_6	$whitepath(X,Y) :- \quad white(X,Y)$
C_7	$whitepath(X,Y) :- white(X,Z), blackpath(Z,Y)$

Fig. 2. An Example Datalog Program

The derivation is summarized in the goal table and the clause table in Figures 3 and 4, respectively. The **Backchain** rule is implicit in the *Parent* column and the **Claim** rule is implicit in the *Claims* column of the goal table. The derivation steps for **Resolve** and **Propagate** are marked in the clause table. are marked in the

Goal	*Literal*	*Parents*	*Claims*	*Children*	*Status*
G_1	$\neg blackpath(a,Y)$	J_{13}	J_4, J_{17}, J_{18}	J_1, J_2	*Resolved*
G_2	$\neg black(a,Z)$	J_1, J_2	J_3	J_3	*Resolved*
G_3	$\neg whitepath(b,Y)$	J_5	$J_{10}, J_{11}, J_{19}, J_{20}$	J_{12}, J_{13}	*Resolved*
G_4	$\neg white(b,Z)$	J_6, J_7	J_8, J_9	J_8, J_9	*Resolved*
G_5	$\neg blackpath(c,Y)$	J_{12}		J_{15}, J_{16}	*Resolved*
G_6	$\neg black(c,Z)$	J_{16}			*Resolved*

Fig. 3. The Goal nodes

We can now look at the termination process. The map $G_6.T$ is everywhere empty since it has no immediate subgoals. We can therefore mark it as *Closed* with $G_5.Unclosed$ undefined, and then mark G_6 as *Completed* since $G_6.D$ is also everywhere undefined.

The map $G_5.T$ is only defined at G_6 and $G_5.T(G_6) = \{J_{15}\}$. The preconditions of the **Close** rule hold for G_65 since $G_6.Status$ is *Closed*, $G_6.Unclosed$ is undefined, and $J_{15}.Subgoals$ is empty. G_5 can therefore be marked as *Closed* and *Completed*, and $G_5.D$ is everywhere undefined, and $G_5.Unclosed$ is also undefined.

The map $G_4.T$ is also everywhere empty since it has no subgoals, and it can also be marked as *Closed* and *Completed* with $G_4.D$ everywhere undefined.

The map $G_3.T$ is nonempty on G_4, G_5, and G_1 so that $G_3.T(G_1) = \{J_{13}\}$, $G_3.T(G_4) = \{J_7\}$, $G_3.T(G_5) = \{J_{12}\}$. Since both G_4 and G_5 are closed, we set $G_3.D(G_1) = 3$, leave $G_3.D$ undefined on other arguments, and mark G_3 as *Closed*.

$Node$	$Clause$	$Derivation$
J_1	$blackpath(a, Y) :- black(a, Y)$	**Resolve**(G_1, C_4)
J_2	$blackpath(a, Y) :- black(a, Z), whitepath(Z, Y)$	**Resolve**(G_1, C_5)
J_3	$black(a, b)$	**Resolve**(G_2, C_1)
J_4	$blackpath(a, b)$	**Propagate**(J_3, J_1)
J_5	$blackpath(a, Y) :- whitepath(b, Y)$	**Propagate**(J_3, J_2)
J_6	$whitepath(b, Y) :- white(b, Y)$	**Resolve**(G_3, C_6)
J_7	$whitepath(b, Y) :- white(b, Z), blackpath(Z, Y)$	**Resolve**(G_3, C_7)
J_8	$white(b, c)$	**Resolve**(G_4, C_2)
J_9	$white(b, a)$	**Resolve**(G_4, C_3)
J_{10}	$whitepath(b, c)$	**Propagate**(J_8, J_6)
J_{11}	$whitepath(b, a)$	**Propagate**(J_9, J_6)
J_{12}	$whitepath(b, Y) :- blackpath(c, Y)$	**Propagate**(J_8, J_7)
J_{13}	$whitepath(b, Y) :- blackpath(a, Y)$	**Propagate**(J_9, J_7)
J_{14}	$whitepath(b, b)$	**Propagate**(J_4, J_{13})
J_{15}	$blackpath(c, Y) :- black(c, Y)$	**Resolve**(G_5, C_4)
J_{16}	$blackpath(c, Y) :- black(c, Z), whitepath(Z, Y)$	**Resolve**(G_5, C_5)
J_{17}	$blackpath(a, c)$	**Propagate**(J_{10}, J_5)
J_{18}	$blackpath(a, a)$	**Propagate**(J_{11}, J_5)
J_{19}	$whitepath(b, c)$	**Propagate**(J_{17}, J_{13})
J_{20}	$whitepath(b, a)$	**Propagate**(J_{18}, J_{13})

Fig. 4. The Clause nodes

The goal G_2 has no immediate subgoals and can be marked as *Completed*. The goal G_1 has subgoals G_2 and G_3 as immediate subgoal so that $G_1.T(G_2) = \{J_2, J_2\}$ and $G_1.T(G_3) = \{J_5\}$. The τ definition for G_1 has $\tau(G_1)(G_1) = 3$, and since $\tau(G_1)(G_1) = |G_1.Claims|$, we can mark $G_1.Status$ as *Closed*, and since there are no goals with smaller indices, $G_1.Status$ can also be marked as *Completed*. This is then propagated to G_3, so that every goal node is now marked as completed.

5.5 Correctness

The new abstract machine can be simulated by the abstract inference procedure, but it is not easy to see why the termination check works. The termination check marks a goal node as *Closed* when it has been completely evaluated modulo the goal nodes with smaller indices. Each closed node also tracks its open subgoals in $g.D$ along with the minimal number of claims propagated from these subgoals. The **Close** step ensures that a goal node is closed only when it is current with respect to all its immediate subgoals. These subgoals can add new claims but this has to be initiated by the addition of a claim to an open subgoal. We can then make the following claims.

Theorem 1. *Let g be a goal node with $g.Status = Closed$ and let $Pr(g)(h)$ represent the number of claims propagated from an immediate subgoal h of g in the derivation of g at the point when $g.Status$ was last set to Closed. If a new claim is added to g, then for some subgoal h different from g, $|h.Claims| > Pr(g)(h)$.*

This is because a closed goal node is fully evaluated in terms of propagating claims from its subgoals and applying the **Backchain** rule to the clauses resulting from the propagation. The only way a new claim can be added to g is through a **Propagate** step applied to some subgoal of g other than g.

Theorem 2. *When a goal node g is marked with $g.Status = Closed$, its evaluation is complete modulo the evaluation of the goals h such that $g.D(h)$ is defined.*

This means that no new claims can be added to g unless there are new claims (beyond the number recorded in $g.D(h)$) are added to some goal h such that $g.D(h)$ is defined. We maintain the invariant that if $g.D(h)$ is defined, then $g.Index > h.Index$ and h is not closed. If we look at the subgoal relation in the derivation of g, then the entry $g.D(h)$ is defined for every unclosed subgoal h of g, and $g.D(h)$ is the minimum number of claims that have been propagated in the derivation of g. By Theorem 1, the only way g can add a new claim is if some immediate subgoal propagates a new claim. By induction, the only way that a claim can be propagated to g is if a new claim is added to some unclosed subgoal h of g, i.e., one where $g.D(h)$ is defined. Hence, the theorem.

Theorem 3. *When a goal node g is marked as completed, no further claims can be added for it.*

This is because for such a node, $g.D$ is everywhere undefined, and hence by Theorem 2, it is not waiting on new claims from any other nodes. In fact, such a node can be seen as the root node of a strongly connected component (SCC) in the evaluation graph. If every node in the strongly connected component is closed modulo other completed nodes or other closed nodes in the strongly connected component, then the entire component has been completely evaluated.

Note that the **Close** step can be interleaved with other steps in the evaluation. It would also make sense to run the **Close** computation in rounds by scanning the goals that are not marked as completed from the highest index downwards.

The implementation of the ETB Datalog engine builds an an Application Programming Interface (API) that can be used to implement the goals. The API includes the following operations for adding goal nodes, processing a goal by either resolving it against the rules or through external evaluation, processing a clause node, propagating a new claim, and closing the evaluation.

We have thus defined an abstract machine for evaluating Datalog programs that operates in a distributed setting.

6 Conclusions

The Evidential Tool Bus (ETB) is a framework for defining distributed workflows that construct claims supported by arguments, where some of the subclaims can be established by external services. ETB uses a variant of Datalog as the scripting language for defining workflows and as the metalanguage for representing arguments. The main novelty of ETB Datalog is that it enhances the basic Datalog language with external predicates for defining computations that invoke external services over a distributed network. We have presented a denotational semantics for ETB Datalog and defined an abstract machine that captures the evaluation of programs using both internal and external predicates. This abstract machine is the basis for the implementation of the Datalog engine used in ETB.

The novel contributions of our work include

1. A powerful mechanism for external predicates that incorporates distributed services.
2. A semantics for Datalog extended with external predicates.
3. An abstract machine that works in a distributed setting.
4. A novel termination check for the abstract machine that indicates when the evaluation of a subgoal has been completed.

The semantic treatment of ETB Datalog given here is a step toward a richer language for defining distributed workflows. The semantics we have given works in a distributed setting where new goals can be added, but the evaluation is still sequential. The body of a clause is evaluated in left-to-right order even when there is no dependency. Since we would like to allow the definition of workflows that exploit parallelism, we are working on extending the language to include annotations for parallel evaluation.

References

1. S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, Reading, Massachusetts, 1995.
2. Peter Alvaro, William R. Marczak, Neil Conway, Joseph M. Hellerstein, David Maier, and Russell Sears. *Dedalus: Datalog in time and space*. Springer, 2011.
3. S. Ceri, G. Gottlob, and L. Tanca. What you always wanted to know about Datalog (and never dared to ask). *IEEE Transactions on Knowledge and Data Engineering*, 1(1):146–166, 1989.
4. John G. Cleary, Mark Utting, and Roger Clayton. Datalog as a parallel programming language. Technical Report ISSN 1177-777X, University of Waikato, Department of Computer Science, 2010.
5. Simon Cruanes, Grégoire Hamon, Sam Owre, and Natarajan Shankar. Tool integration with the evidential tool bus. In Roberto Giacobazzi, Josh Berdine, and Isabella Mastroeni, editors, *Verification, Model Checking, and Abstract Interpretation, 14th International Conference, VMCAI 2013, Rome, Italy, January 20-22, 2013. Proceedings*, volume 7737 of *Lecture Notes in Computer Science*, pages 275–294. Springer, 2013.
6. Herve Gallaire and Jack Minker. *Logic and data bases*. Perseus Publishing, 1978.
7. Joseph M Hellerstein. The declarative imperative: experiences and conjectures in distributed logic. *ACM SIGMOD Record*, 39(1):5–19, 2010.
8. Shan Shan Huang, Todd Jeffrey Green, and Boon Thau Loo. Datalog and emerging applications: An interactive tutorial. In Timos K. Sellis, Renée J. Miller, Anastasios Kementsietsidis, and Yannis Velegrakis, editors, *SIGMOD Conference*, pages 1213–1216. ACM, 2011.
9. Boon Thau Loo, Tyson Condie, Minos Garofalakis, David E. Gay, Joseph M. Hellerstein, Petros Maniatis, Raghu Ramakrishnan, Timothy Roscoe, and Ion Stoica. Declarative networking: Language, execution and optimization. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, pages 97–108. ACM, 2006.
10. Konstantinos Sagonas and Terrance Swift. An abstract machine for tabled execution of fixed-order stratified logic programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 20(3):586–634, 1998.
11. Natarajan Shankar. Inference systems for logical algorithms. In R. Ramanujam and Sandeep Sen, editors, *FSTTCS 2005: Foundations of Software Technology and Theoretical Computer Science*, volume 3821 of *Lecture Notes in Computer Science*, pages 60–78. Springer Verlag, 2005.
12. John Whaley, Dzintars Avots, Michael Carbin, and Monica S. Lam. Using Datalog with binary decision diagrams for program analysis. In *Programming Languages and Systems*, pages 97–118. Springer, 2005.