The Complexity of Reusing and Modifying Rulebases*

Anthony J. Bonner

University of Toronto
Department of Computer Science
Toronto, Ontario M5S 1A4
Canada

bonner@db.toronto.edu

Abstract

This paper develops a method for reusing and modifying deductive databases. Such methods are needed when new rulebased applications differ only slightly from existing ones or when an application is to be incrementally updated. In order to facilitate reuse, we extend deductive databases by the concept of *predicate substitution*. In this way, during query evaluation, not only variables, but also predicates can be substituted. This paper continues our earlier work on predicate substitution in two directions: (i) We extend the concept to a wider class of rulebase modifications, and (ii) we establish tight bounds on the data complexity of Datalog augmented with substitution, showing it to be EXPTIME-complete. Predicate substitution thus increases the power of Datalog to express database queries. The paper presents a proof theory and model theory for the language, including a fixpoint semantics.

1 Introduction

This paper develops a method for reusing and modifying deductive databases. The need for such techniques occurs when new rulebased applications differ only slightly from existing ones or when an application is to be incrementally updated. Such techniques are particularly important when reprogramming is expensive or unreliable. Unfortunately, these issues have not yet received significant attention in the deductive database literature. This paper addresses attempts to address them, focussing on a particular class of modifications that we call *predicate substitution*. The idea is to reuse and modify a set of rules by substituting one predicate symbol for another. Augmented with this capability, deductive databases can better exploit the polymorphism that exists in many database operations.

Transitive closure is a simple example of such an operation. It is polymorphic in that every binary relation has a transitive closure. Nevertheless, in many deductive database systems, one cannot write a rulebase that computes the transitive closure of an arbitrary binary relation. Instead, one can only write rules that compute the transitive closure of a particular relation. The following rules, for example, compute the transitive closure of the relation R:

$$\begin{array}{l} TR(x,y) \leftarrow R(x,y). \\ TR(x,z) \leftarrow R(x,y),\, TR(y,z). \end{array} \qquad (1)$$

To compute the transitive closure of another relation, say S, one must write a new set of rules, defining a new predicate TS in terms of S. In this way, a user is forced to rewrite rulebases if he wants to apply them to different sets of data.

Predicate substitution carries out this rewriting automatically. For instance, to compute the transitive closure of S, the user specifies that S be substituted for R in the two rules above. Two new rules, equivalent to the following, are then generated:

$$\begin{array}{l} TS(x,y) \leftarrow S(x,y). \\ TS(x,z) \leftarrow S(x,y),\, TS(y,z). \end{array} \qquad (2)$$

If necessary, new predicates such as TS are also generated. In this way, predicate substitution allows a user to specify new rulebases by reusing and modifying old ones.

We believe that predicate substitution is a principle that can be applied to logic-based languages generally in order to facilitate reuse and modification. In this paper, however, we focus on Horn logics and on Datalog in particular. That is, we develop a logic programming language that combines logical inference with predicate substitution. The rules in this logic, called *rules*

*This paper appears in the *Proceedings of the ACM Symposium on the Principles of Database Systems (PODS)*, pages 316–330. Conference held June 2–4 1992 at San Diego, CA. Invited to a special issue of *Journal of Computer and System Sciences (JCSS)*.

with substitution, allow a user to select other rules for reuse and modification and to combine the modified rules into larger rulebases. Rulebases defined in this way can themselves be reused and modified, so that the process of constructing rulebases is closed under predicate substitution.

In an earlier paper, we developed a preliminary version of this logic and established a lower bound of NP on its data complexity [3]. In the present paper, we continue this development in two ways: (*i*) we extend the logic to include a wider class of rulebase modifications, and (*ii*) we establish tight bounds on the data complexity of both the preliminary and extended logics. In particular, we show that both logics are data complete for EXPTIME.[1] Thus, while we have increased the expressibility of the logic, we have not paid a price for this in terms of complexity.

Predicate substitution is an attempt to augment a logic with the *minimal* amount of power needed to reuse and modify programs. This distinguishes predicate substitution from other systems presented in the literature, especially those in which function symbols are used to represent higher-order functions. This is the case, for instance, in the HiLog system of Chen, Kifer and Warren [5], and in the meta-programming methodology of Kwok and Sergot [7]. Although these systems can apply a logic program to different predicates, their use of function symbols gives these systems a great deal of power, so that inference is undecidable. In contrast, inference in Datalog with predicate substitution is decidable, and in fact, it is in EXPTIME.

There are also philosophical and practical differences between predicate substitution and the use of higher-order functions. Although, higher-order functions allow a program to be applied to different predicates, they require that a programmer *anticipate* the reuse and modification of a program *before* writing it, by using special higher-order variables in a program's definition. In contrast, predicate substitution allows a program to be reused even though the programmer originally wrote it with a specific set of predicates in mind. To achieve this, predicate substitution *modifies* programs before reusing them. We present a novel semantics reflecting this capability.

1.1 Extending the Logic

Our earlier work on predicate substitution addressed only a limited class of rulebase modifications [3]. In particular, it required that a predicate substitute only for predicates of *equal* arity. This is the case in the example of transitive closure above, where one binary predicate substituted for another. In contrast, the extended logic developed in this paper allows a predicate of one arity to substitute for predicates of *different* arity.

For example, suppose that $E(x, y, g)$ means that graph g has an edge from node x to node y. We would like to reuse our transitive closure program to compute the transitive closure of every graph g. This is not possible in the preliminary logic, because E is ternary whereas the transitive-closure program TR requires a binary predicate. However, this "mismatch" presents no problem for the extended logic. In effect, the user specifies that for each value of g, the predicate $R(x, y)$ should be replaced by $E(x, y, g)$. Two new rules, equivalent to the following, are then generated automatically:

$$\begin{array}{l} TE(x,y,g) \leftarrow E(x,y,g). \\ TE(x,z,g) \leftarrow E(x,y,g),\, TE(y,z,g). \end{array} \tag{3}$$

Here, $TE(x, y, g)$ means that (x, y) is in the transitive closure of graph g. Again, if necessary, new predicates such as TE are also generated. In this way, we extend the scope of predicate substitution to a wider and more useful class of rulebase modifications.

We develop these ideas in several ways. First, we define predicate substitution precisely in terms of a copy-and-substitute mechanism. This mechanism formalizes our intuition that predicate substitution specifies how to "edit" a set of rules to produce a new set of rules with different predicate symbols. Although this mechanism might seem procedural, we show how to formulate it as a declarative language with a logical semantics. In doing so, we develop both a proof theory and a model theory for the logic, including a fixpoint semantics, and we show that the proof theory is sound and complete for the model theory.

In the model theory, rulebases *without* substitution are the models for rulebases *with* substitution. This semantics corresponds to the intuition that rules with substitution are written by a user who has rules without substitution "in mind." This intuition is similar to the view taken of ordinary Datalog rules, in which the user has an "intended model" in mind (the least model of the rules), which he could have written if he had enough space and enough time. In this way, just as Datalog provides a concise notation for describing large *databases*, Datalog with substitution provides a concise notation for describing large *rulebases*. Hence, it can be argued that in going from Datalog to Datalog-with-substitution, we achieve an upgrade similar to that in going from relational algebra to Datalog.

By combining recursion and substitution, it is possible to reuse and modify rule sets infinitely many times. In [3], we used this idea to prove a qualified theorem: A

[1] In this paper, the terms EXPTIME and "exponential time" refer to DTIME[$2^{n^{O(1)}}$].

finite set of rules with substitution can specify an *infinite* set of rules without substitution, assuming that P $\neq$ NP. In the present paper, we eliminate the qualification and show that the theorem is unequivocally true.

Predicate substitution thus provides an extremely concise notation for specifying sets of Datalog rules. Fortunately, it is not necessary to materialize all these rules, since our inference system works directly on the rule specifications, not on the rules themselves. This is similar to inference in Datalog, where a query can be evaluated without materializing the entire least-fixpoint of the rulebase.

1.2 Complexity Results

Since predicate substitution provides a short-hand notation for describing rule sets, it might appear that it is a merely syntactic device, one that increases the convenience of a logic, but not its expressive power. This conclusion would be incorrect however, since predicate substitution can actually increase the power of a logic. In evidence of this, we show that substitution increases the power of Datalog: Whereas the data complexity of Datalog is complete for PTIME, the data complexity of Datalog augmented with substitution is complete for EXPTIME. When a Datalog rulebase with substitution is EXPTIME-complete, it represents an *infinite* Datalog rulebase *without* substitution. Furthermore, this infinite Datalog rulebase is not equivalent to any finite one. The proof of EXPTIME completeness relies on the soundness-and-completeness of our semantics. In particular, the lower complexity bound follows from the proof theory, and the upper complexity bound follows from the model theory.

It is conceivable that an upper complexity bound could be established directly from the proof theory, without invoking the model theory. This would appear to be a difficult task, though, since the proof system is able to create an unlimited number of new predicate symbols during inference. Furthermore, the arity of these invented predicates is unbounded. On the face of it, then, one might guess that inference with predicate substitution is undecidable. This guess would be incorrect however. The model theory makes this clear, since models are constructed from a *fixed* and *finite* set of predicate and constant symbols. By itself, however, the model theory does not give us a mechanical way of answering queries, since we might have to examine infinitely many models. The fixpoint theory solves this problem. This theory shows that a rulebase with substitution has a least model, containing all the information necessary to answer database queries. Furthermore, the fixpoint theory shows how to construct this model in a bottom-up way by a simple iterative procedure, much as the least model of a Datalog program is constructed. We show that this mechanism runs in exponential time (exponential in the size of the database).

To prove that EXPTIME is also a lower bound, we use predicate substitution to encode the computations of alternating PSPACE machines (APSPACE machines). The lower bound then follows, since APSPACE = EXPTIME [4]. Because the encoding is rather unusual, we briefly describe it here. There are three stages. First, we represent the initial configuration of the machine as a database predicate, in a fairly standard way. Second, we write a small set of Datalog rules that infer whether the machine accepts its input after *exactly one* computation step. Finally, using predicate substitution and recursion, we repeatedly reuse and modify these rules, to generate new Datalog rules. In this way, for each integer k, we generate a set of Datalog rules that infer whether the machine accepts its input after *exactly* k computation steps. In effect, we use substitution to "compose" the original rule-set k times. The result is an *infinite* set of Datalog rules that infers whether the machine accepts its input for *some* k.[2] Because this set is infinite, it is not subject to the normal PTIME bound on the complexity of *finite* Datalog programs. For this reason, the rule set is able to simulate any alternating PSPACE machine.

2 Examples

This section gives several examples of how predicate substitution can be used. Each example uses Horn logic with substitution to exploit the polymorphic nature of certain database operations in a way that Horn logic alone cannot.

In these examples, the expression $[P/Q]$ is a substitution operator that replaces the predicate symbol P by the predicate symbol Q. This operator is applied to other predicates in order to reuse and modify their definitions. For example, the expression "$A(x)[P/Q]$" intuitively means, "Substitute Q for P in the definition of A". Here, the "definition of A" is the set of rules defining A (e.g., the logic program for A, or the Datalog program for A).

The expression $A(x)[P/Q]$ is called a *predicate with substitution*. It is treated as an ordinary predicate of first-order logic except that its definition is not given by the user but is derived automatically by reusing and modifying the definition of $A(x)$. The examples of this section illustrate this idea.

The first example illustrates basic predicate substitution, in which a predicate substitutes for predicates of

[2]Because classical logic is compact, the normal proof methods, such as resolution, remain sound and complete even for infinite rule sets.

equal arity. The remaining examples illustrate full predicate substitution, in which a predicate substitutes for predicates of *different* arity.

Example 1. Let R be the following rulebase, which defines the transitive closure A of a base relation A_0:

$$A(x,y) \leftarrow A_0(x,y).$$
$$A(x,z) \leftarrow A_0(x,y), A(y,z).$$

Thus $R, DB \vdash A(x,y)$ iff (x,y) is in the transitive closure of A_0, for any database DB. To compute the transitive closure of some other relation B_0, we can rewrite these rules replacing A_0 by B_0. The predicate with substitution $A(x,y)[A_0/B_0]$ does this automatically. That is $R, DB \vdash A(x,y)[A_0/B_0]$ iff (x,y) is in the transitive closure of B_0.

This predicate with substitution can also be used in rule bodies to define other predicates. In this way, the following rule defines B to be the transitive closure of B_0:

$$B(x,y) \leftarrow A(x,y)[A_0/B_0]$$

This rule is called a *rule with substitution*. Intuitively, it means that the Horn rules defining B are the same as those defining A with A_0 replaced by B_0. Thus, the new rules use the data in B_0 instead of the data in A_0. If we add this rule to the rulebase R, then $R, DB \vdash B(x,y)$ iff (x,y) is in the transitive closure of B_0. In this way, predicate substitution exploits the polymorphic nature of transitive closure. □

The substitution $[P/Q]$ requires that P and Q be predicate symbols of equal arity. More generally, we allow the arity of Q to be *greater* than the arity of P. These generalized substitutions are represented by expressions of the form $[P/Q](y_1...y_m)$ where $y_1...y_m$ denote the extra arguments of Q that do not appear in P.

Example 2. Suppose the database DB contains a ternary predicate E_0 representing a set of graphs. If g is the name of a graph, then the formula $E_0(x,y,g)$ means that graph g has an edge between nodes x and y. Thus, the number of graphs is not fixed, but depends on the database. To compute the transitive closure of a graph, we would like to reuse and modify the rulebase R from Example 1. The predicate with substitution $A(x,y)[A_0/E_0](g)$ does this automatically. That is, $R, DB \vdash A(x,y)[A_0/E_0](g)$ iff (x,y) is in the transitive closure of graph g.

This predicate with substitution can be used in rule bodies to define other predicates. In this way, the following rule defines a ternary predicate E:

$$E(x,y,g) \leftarrow A(x,y)[A_0/E_0](g)$$

This is a rule with substitution. Intuitively, it means that the Horn rules defining E are the same as those defining A except that A_0 is replaced by E_0. Thus, for each graph g, the new rules use the data in $E_0(x,y,g)$ instead of the data in $A_0(x,y)$. If we add this rule to the rulebase R , then $R, DB \vdash E(x,y,g)$ iff (x,y) is in the transitive closure of graph g. □

The next example illustrates a more sophisticated application of predicate substitution.

Example 3. A university would like to rank students according to a number of different criteria: For each class, it wants to rank students according to their grade in that class. The university does not have a program that does this, but it does have a program that ranks students according to their grade point average. We would like to reuse and modify this program so that it can rank students according to other criteria, where the exact set of criteria depends on the database of courses.

The university's database DB defines two predicates, Gpa and $Grade$. $Gpa(g,s)$ means that g is the grade point average of student s, and $Grade(g,s,c)$ means that g is the grade of student s in class c. In addition, a rulebase R defines a predicate $Rank(s,n)$, which means that student s has the n^{th} highest gpa. Predicate $Rank$ is defined in terms of predicate Gpa. Given this, the following rule with substitution reuses and modifies the rulebase to define a new predicate $CourseRank(s,c,n)$, which means that student s has the n^{th} highest grade in course c:

$$CourseRank(s,c,n) \leftarrow Rank(s,n)[Gpa/Grade](c)$$

Intuitively, this rule says that the Horn rules defining $CourseRank$ are the same as those defining $Rank$ except that $Gpa(g,s)$ is replaced by $Grade(g,s,c)$. Thus, for each class c, the new rules use the data in $Grade(g,s,c)$ instead of the data in $Gpa(g,s)$, thus ranking the students in class c according to their grade. In this way, predicate substitution exploits the polymorphic nature of ranking. □

The above examples are typical of a class of applications in which a module is to be used repeatedly with different inputs. In such cases, we simply "plug in" different definitions for the base predicates of the module. The need to often reuse and modify existing rule bases occurs in rulebased specifications of the operations of large enterprises, such as big corporations, computer operating systems or various legal systems, such as tax-law, citizenship law etc. When changes are small, it simply does not make sense to rewrite possibly-huge rulebased specifications. It is definitely more attractive to reuse as much as possible of the old definition.

3 Inference with Substitution

This section formalizes the notion of predicate substitution, defines the syntax of predicates and rules with substitution, and describes an inference method. An axiomatization of the method also is given.

Definition 1 (Predicates with Substitution)

- *Every atomic formula is a predicate with substitution.*
- *If α is a predicate with substitution, then so is $\alpha[P/Q](y_1...y_k)$, where $y_1...y_k$ are variables, P is a predicate symbol of arity n, Q is a predicate symbol of arity $n+k$, and $k \geq 0$.*

Thus, if predicate symbols P_i and Q_i have arity i, then $A(x)[P_2/Q_3](y)$ is a predicate with substitution, as are $A(x)[P_2/Q_4](y,z)$ and $B(u,v)[P_2/Q_5](x,y,z)$. Predicate substitutions can be compounded, as in $A(x)[P_1/Q_2](y)[P_3/Q_4](z)$, and of course, variables may be repeated, as in $B(x,y)[P_2/Q_4](y,z)[P_1/Q_3](z,x)$. The expression $[P/Q](x,y)$ is called a *predicate substitution*, or simply a *substitution*. For clarity, when P and Q have equal arity, we write $[P/Q]$ instead of $[P/Q]()$. Expressions of the form $[P/Q]$ are called *basic* predicate substitutions. Basic substitution was introduced in [3]. The rest of this section generalizes basic substitution so that a predicate can substitute for predicates of *different* arity.

In the expression $[P/Q](y_1...y_k)$, the variables $y_1...y_k$ denote the extra arguments of Q that do not appear in P. By default, we take these to be the last k arguments of Q. Thus, intuitively the expression $\alpha[P/Q](y_1...y_k)$ means, "For each tuple $(y_1...y_k)$, substitute $Q(x_1...x_n, y_1...y_k)$ for $P(x_1...x_n)$ in the definition of α." We could have chosen $y_1...y_k$ to be any k arguments of Q. The particular choice does not matter, though, since we can use Horn rules to permute any k arguments into the last k positions.

Predicate substitutions should be distinguished from the more familiar form of substitution in which terms are substituted for variables. For instance, in the expression $\{u/f(x), v/y\}$, the terms $f(x)$ and y are to be substituted for the variables u and v, resp. We call these expressions *term substitutions* to distinguish them from predicate substitutions. Throughout this paper, the symbol θ denotes a predicate substitution and σ denotes a term substitution.

Definition 2 (Rules with Substitution) *A rule with substitution is an expression of the form $\beta \leftarrow \beta_1...\beta_k$ where β and each β_i are predicates with substitution. If β is atomic, then the rule is called a* Horn *rule with substitution.*

Given a set of Horn rules *with* substitution, we transform it into an equivalent set of Horn rules *without* substitution. In fact, the rulebase "without substitution" still contains predicates with substitution, but they are now treated as an ordinary predicates of classical logic. The transformation from rules with substitution to rules without substitution is based on the following principle: In the transformed rulebase, the rules defining $A[P/Q](y)$ must be identical to the rules defining A except that $P(x)$ is replaced everywhere by $Q(x,y)$. Intuitively, the rules defining A are automatically copied and edited to produce rules defining $A[P/Q](y)$.

Example 4. Consider the following rulebase, where B and C are base predicates:

$$A(x) \leftarrow B(x), P(x).$$
$$A(x) \leftarrow C(x,y), P(x), P(y).$$

Then the expression $A(x)[P/Q]$ is treated as a new unary predicate defined by the following rules.

$$A(x)[P/Q] \leftarrow B(x), Q(x).$$
$$A(x)[P/Q] \leftarrow C(x,y), Q(x), Q(y).$$

Notice that the definition of $A(x)[P/Q]$ is identical to the definition of $A(x)$ except that P has been replaced by Q in the premise of every rule. Conceptually, these rules are generated automatically. They are *not* supplied by the user.

Likewise, the expression $A(x)[P/Q](z)$ is treated as a new *binary* predicate defined by the following rules:

$$A(x)[P/Q](z) \leftarrow B(x), Q(x,z).$$
$$A(x)[P/Q](z) \leftarrow C(x,y), Q(x,z), Q(y,z).$$

Notice that the rules defining $A(x)[P/Q](z)$ are identical to the rules defining $A(x)$ except that in every rule, the predicate P has been replaced by Q, a predicate of greater arity. Again, these rules are generated automatically. □

The above example is simple in that there is only one derived predicate and no recursion. The next two examples show how substitution operates in more complex situations. For more examples, see [3] and [2].

Example 5. Consider the following rulebase, where C and D are base predicates:

$$A(x) \leftarrow C(x), B(x).$$
$$B(x) \leftarrow D(x), P(x).$$

Notice that the predicate A is defined not just in terms of base predicates, but also in terms of a defined predicate B. In applying the substitution $[P/Q](y)$ to

A, we also apply it to B. In this way, when reusing a program, we automatically reuse any subprograms that it invokes. The predicates $A(x)[P/Q](y)$ and $B(x)[P/Q](y)$ are defined as follows:

$$A(x)[P/Q](y) \leftarrow C(x), B(x)[P/Q](y).$$
$$B(x)[P/Q](y) \leftarrow D(x), Q(x,y).$$

Notice that the base predicates C and D are unaffected by the substitution $[P/Q](y)$. In general, since base predicates are not defined in terms of other predicates, substitutions for other predicates do not affect them. Finally, since we interpret rules with substitution classically, we can combine the two rules above to give the following inference:

$$A(x)[P/Q](y) \leftarrow C(x), D(x), Q(x,y).$$

□

The idea of copying rules and renaming predicates applies to rulebases with substitution as well as to those without. Furthermore, the idea can be applied recursively. That is, once predicates such as $A(x)[P/Q](y)$ have been defined, they can be used to define more complex predicates such as $A(x)[P/Q](y)[R/S](z)$. In this way, the process of constructing rulebases is closed under substitution.

Example 6. Consider the following rulebase, where $D(x)$ and $Q(x,y)$ are base predicates:

$$A(x,y) \leftarrow B(x)[C/Q](y)$$
$$B(x) \leftarrow C(x), D(x).$$

We thus have the following rules:

$$A(x,y)[D/R](z) \leftarrow B(x)[C/Q](y)[D/R](z)$$
$$B(x)[C/Q](y) \leftarrow Q(x,y), D(x).$$
$$B(x)[C/Q](y)[D/R](z) \leftarrow Q(x,y), R(x,z).$$

Here, the substitution $[C/Q](y)$ replaces $C(x)$ by $Q(x,y)$, and the substitution $[D/R](z)$ replaces $D(x)$ by $R(x,z)$. Since the three rules above are to be treated classically, we can combine them to give the following inferences:

$$A(x,y) \leftarrow Q(x,y), D(x).$$
$$A(x,y)[D/R](z) \leftarrow Q(x,y), R(x,z).$$

□

The above examples illustrate the idea of predicate substitution. The following definition makes the idea precise, generalizing the definition given in [3].

Definition 3 *Let $\mathcal{S}$ be a Horn rulebase with substitution. Then $\mathcal{S}^*$ is the set of rules with substitution constructed from $\mathcal{S}$ as follows:*[3]

1. *$\mathcal{S} \subseteq \mathcal{S}^*$*
2. *If $\beta \leftarrow \beta_1 ... \beta_n$ is in $\mathcal{S}^*$, then $\beta\theta \leftarrow \beta_1\theta ... \beta_n\theta$ is in $\mathcal{S}^*$ for every predicate substitution θ.*
3. *$P(\overline{x})[P/Q](\overline{y}) \leftarrow Q(\overline{x}, \overline{y})$ is in $\mathcal{S}^*$ for every predicate substitution $[P/Q](\overline{y})$.*
4. *$B(\overline{x})[P/Q](\overline{y}) \leftarrow B(\overline{x})$ is in $\mathcal{S}^*$ for every predicate substitution $[P/Q](\overline{y})$, and every base predicate $B \neq P$.*

[3] To be precise, $\mathcal{S}^*$ is the smallest set of rules with substitution closed under these operations.

The second item in the definition of $\mathcal{S}^*$ ensures that for every rule defining β, there is a copy of the rule in $\mathcal{S}^*$ defining $\beta\theta$. The third item carries out the substitution $[P/Q](\overline{y})$, so that every occurrence of $P(\overline{x})[P/Q](\overline{y})$ is replaced by $Q(\overline{x}, \overline{y})$. The fourth item ignores the substitution when it is applied to base predicates not equal to P, so that every occurrence of $B(\overline{x})[P/Q](\overline{y})$ is replaced by $B(\overline{x})$. The rulebase $\mathcal{S}^*$ thus provides a complete definition for all possible predicates with substitution. Since there are infinitely many predicates with substitution, $\mathcal{S}^*$ is itself infinite.

Having performed the substitutions, each rule in $\mathcal{S}^*$ will be treated classically, as a rule of Horn logic. Definition 3 thus shows how to transform rules with substitution into Horn rules. To be useful, however, these Horn rules must be augmented with a database and the Horn rules used to infer new atomic formulas. The following definition formalizes this idea. In this definition, and in the rest of the paper, a database is defined to be a set of atomic formulas.

Definition 4 *Suppose β is a predicate with substitution, $\mathcal{S}$ is a Horn rulebase with substitution, and DB is a database. Then,*[4]

$$\mathcal{S}, DB \vdash \beta \quad \textit{iff} \quad \mathcal{S}^* \cup DB \vdash_c \beta$$

3.1 An Axiomatization

Because $\mathcal{S}^*$ is infinite, its value is largely conceptual, providing a concise definition of inference with substitution. To actually perform inference, though, it is impractical to first generate $\mathcal{S}^*$ and then invoke classical inference procedures. This section offers a more practical alternative by providing a finite axiomatization which is sound and complete. The inference system below can be operated in either a top-down or a bottom-up manner. When operated top-down, it attempts only those substitutions relevant to answering a particular query.

[4] In this definition and in the rest of the paper, the symbol $\vdash_c$ denotes classical inference, where all free variables are universally quantified.

Lemma 1 *Let β be a predicate with substitution, $\mathcal{S}$ a Horn rulebase with substitution, and DB a database. Then $\mathcal{S}, DB \vdash \beta$ iff β can be inferred from the following inference system:*

Axioms:

1. *For every substitution* $[P/Q](\overline{y})$, $P(\overline{x})[P/Q](\overline{y}) \leftarrow Q(\overline{x}, \overline{y})$
2. *For every substitution* $[P/Q](\overline{y})$, *and every base predicate* $B \neq P$, $B(\overline{x})[P/Q](\overline{y}) \leftarrow B(\overline{x})$
3. *Every rule in* $\mathcal{S}$.
4. *Every atomic formula in DB.*

Inference Rules:

1. *For any predicate substitution* θ,
$$\frac{\beta \leftarrow \beta_1 ... \beta_n}{\beta\theta \leftarrow \beta_1\theta ... \beta_n\theta}$$
2. *For any term substitution* σ,
$$\frac{\beta \leftarrow \beta_1 ... \beta_n}{(\beta \leftarrow \beta_1 ... \beta_n)\sigma}$$
3. *Modus Ponens:*
$$\frac{\begin{array}{l}\beta_1 \; ... \; \beta_n \\ \beta \leftarrow \beta_1 ... \beta_n\end{array}}{\beta}$$

Given a predicate β, the first inference rule effectively makes copies of the rules defining β in order to derive rules defining $\beta\theta$. The first two axioms come directly from the definition of $\mathcal{S}^*$. The other axioms and inference rules are familiar from classical Horn logic: the second inference rule makes all instances of a rule available for inference, and the third rule applies rules to atomic facts to infer more atomic facts.

4 Lower Complexity Bound

Predicate substitution increases the power of Datalog. Without substitution, the data-complexity of Datalog is complete for polynomial time. Augmented with substitution, it is complete for EXPTIME. This section establishes the lower complexity bound, showing that Datalog with substitution is EXPTIME-hard. In fact, we prove a stronger result: We show that "basic" predicate substitution is EXPTIME-hard. In basic substitution, a predicate is only allowed to substitute for predicates of *equal* arity. "Full" predicate substitution refers to the general case, in which a predicate can substitute for predicates of *different* arity.

Theorem 2 *The data complexity of Datalog rulebases augmented with basic predicate substitution is EXPTIME-hard.*

Corollary 3 *The data complexity of Datalog rulebases augmented with full predicate substitution is EXPTIME-hard.*

To prove Theorem 2, we use Datalog with basic substitution to encode the computations of alternating PSPACE machines (APSPACE machines). The EXPTIME result follows since APSPACE = EXPTIME [4]. For clarity, we break the proof into two parts. First, to illustrate the techniques involved, we encode the computations of *deterministic* PSPACE machines. Then, we use the techniques to encode the computations of *alternating* PSPACE machines.

4.1 Encoding PSPACE Machines

To state the main result of this section precisely, suppose that M is a PSPACE machine. Without loss of generality, we can assume that M has a single tape. We encode this machine as a database $DB(M)$. Given an input string $\overline{s}$ for the machine, we encode it as another database $DB(\overline{s})$. Finally, we construct a Datalog rulebase with substitution $\mathcal{S}$ that simulates the computations of machine M on input $\overline{s}$. In particular,

$$\begin{array}{c} \mathcal{S},\, DB(M) + DB(\overline{s}) \;\vdash\; ACCEPT \\ \textit{iff } M \textit{ accepts } \overline{s} \end{array} \tag{4}$$

where $ACCEPT$ is a 0-ary predicate. The important points are that the rulebase $\mathcal{S}$ is independent of the input $\overline{s}$, and that the database $DB(\overline{s})$ can be constructed in polynomial time (polynomial in the size of the machine input $\overline{s}$). From these facts, we can conclude that the data complexity of Datalog with substitution is PSPACE-hard. The result follows immediately by letting M be a machine that recognizes a PSPACE-complete language.[5]

The rest of this section describes the construction of the rulebase $\mathcal{S}$ and the databases $DB(M)$ and $DB(\overline{s})$. It is worth pointing out that $\mathcal{S}$ has only six rules: five non-recursive Horn rules, and one recursive rule with substitution. This simple structure, which is enough to simulate any PSPACE machine, is a good illustration of the conciseness of predicate substitution.

4.1.1 Strings and Transformations

To encode the computations of a PSPACE machine, we view computation as a sequence of string transformations, where each string represents a machine configuration. A configuration records the contents of the machine tape and the position and state of the control head.

There are many ways to encode a machine configuration as a string. We describe a standard encoding that

[5] See [6] for examples of such languages.

is convenient for the purpose of this paper. First, we encode the machine tape as a string in the obvious way, so that each tape cell is represented by one string character. Thus, a tape holding the characters $s_1, s_2, ..., s_n$ is represented by the string $s_1 s_2 ... s_n$. Second, for technical reasons, we add a special character \$ to the front and end of the string, to produce the string $\$ s_1 s_2 ... s_n \$$. Third, for each tape character s_i and each control state q_j we treat the ordered pair $\langle s_i, q_j \rangle$ as a character and add it to the string alphabet. We use these compound characters to encode the state and position of the control head. For example, if the control head is in state q_j and is scanning the tape cell at position i, then in the string $\$ s_1 ... s_n \$$, we replace the character s_i by $\langle s_i, q_j \rangle$. The configuration of the machine is therefore encoded by the following string, of length 2:

$$\$ s_1 ... s_{i-1} \langle s_i, q_j \rangle s_{i+1} ... s_n \$$$

Because a PSPACE machine uses only polynomial space, we can pad the input with polynomially-many blanks, so that the tape head never goes outside the input area [6]. We can therefore assume that all machine configurations have the same length. Thus, we can represent the transition function of the machine as a string transformation in which a string $a_1 ... a_n$ is mapped to another string of the same length $b_1 ... b_n$. Furthermore, because computation is a *local* phenomena, each output character b_i depends not on the entire input string, but only on the character a_i and its two immediate neighbours, a_{i-1} and a_{i+1}. The transformation is thus completely specified by a ternary function f. Formally,

$$b_1 = a_1 \tag{5}$$
$$b_n = a_n \tag{6}$$
$$b_j = f(a_{j-1}, a_j, a_{j+1}) \quad \textit{for} \;\; 1 < j < n \tag{7}$$

Thus, the end points of the string do not change (leaving the endpoint markers intact), and the interior points are transformed by the function f. We use the symbol t to denote the entire string transformation. Thus $t(a_1 ... a_n) = b_1 ... b_n$.

When f encodes the transition function of a PSPACE machine, we can use the transformation t to simulate the computations of the machine. That is, if $\overline{a}$ is a string representing the configuration of the machine at some point in a computation, then $t(\overline{a})$ represents the next configuration in the computation, $t^2(\overline{a})$ the next configuration after that, etc.

4.1.2 Data and Rules

In this section, we construct a database and a Datalog rulebase with substitution that implement the string transformations described above. The rulebase itself has two parts. First, given a string $\overline{a}$ encoded in the database, a set of Horn rules derives the transformed string $t(\overline{a})$. Second, a recursive rule with substitution applies the transformation $\overline{t}$ over-and-over again to generate the strings $t^i(\overline{a})$. The atom $ACCEPT$ is inferred iff one of these strings encodes an accepting machine state. In this way, we satisfy condition (4).

We encode the finite control of the machine a database $DB(M)$, in which we store the ternary function f and the set of accepting states. To store f, we introduce a 4-ary predicate F and add the formula $F(a, b, c, d)$ to the database iff $f(a, b, c) = d$. To store the accepting states, we introduce a unary predicate $ACCEPTING$. For each compound character $\langle c, q \rangle$, we add the formula $ACCEPTING(\langle c, q \rangle)$ to the database iff q is an accepting state. (The appearance of any of these compound characters in a machine configuration indicates that the machine has accepted its input.) Note that neither of these predicates depends on the machine's input, so we can spend as much time as we like constructing them.

We encode the machine's initial configuration as a database $DB(\overline{s})$. First, we need a counter to represent string positions. We therefore introduce three base predicates, $FIRST$, $NEXT$ and $LAST$. For strings of length n, we add the following formulas to the database:

$$\begin{array}{l} FIRST(1), \;\; LAST(n), \\ NEXT(1,2), \;\; NEXT(2,3) \; ... \; NEXT(n-1, n). \end{array}$$

Given this counter, we introduce a binary predicate S to represent the characters in a string. The formula $S(c, j)$ means that character c appears at position j in the string. If the initial configuration of the machine is given by the string $\overline{a} = a_1 ... a_n$, then we add the following formulas to the database:

$$S(a_1, 1), \;\; S(a_2, 2), \;\; S(a_3, 3) \; ... \; S(a_n, n).$$

Note that this database can be constructed in polynomial time (polynomial in n).

Having constructed the databases $DB(M)$ and $DB(\overline{s})$, the next task is to construct the rulebase $\mathcal{S}$. This rulebase has two parts: (i) a set of Horn rules that implement the string transformation t, and (ii) a recursive rule with substitution that applies the string transformation over-and-over again until an accepting state is reached.

To implement the string transformation t, we add the three Horn rules below to the rulebase R. These three rules correspond to conditions (5)–(7), respectively.

$$\begin{array}{lcl} S_0(x, j) & \leftarrow & FIRST(j), \; S(x, j). \\ S_0(x, j) & \leftarrow & LAST(j), \; S(x, j). \\ S_0(x, j) & \leftarrow & S(u, i), \; S(v, j), \; S(w, k), \\ & & F(u, v, w, x), NEXT(i, j), NEXT(j, k). \end{array}$$

where i, j, k, and u, v, w, x are variables. The first two rules ensure that the end points of the string do not change. The third rule implements the function f that transforms the interior points of the string. Given a string $\overline{a}$ stored in the predicate S, these rules derive the transformed string $t(\overline{a})$ and store it in the predicate S_0. Thus, if S encodes a machine configuration, then S_0 encodes the next machine configuration.

These Horn rules represent *one* step of a computation. To perform a second step, we apply the predicate substitution $[S/S_0]$ to the rules. This substitution reuses and modifies the Horn rules to produce a new set of Horn rules. The new rules are similar to the old rules in that they encode the transition function of the machine M. However, the new rules have different input and output predicates: They take $S_0(x, j)$ as input and derive a new binary predicate $S_0(x, j)[S/S_0]$ as output. If S represents the first configuration of a computation, then S_0 represents the second, and $S_0(x, j)[S/S_0]$ represents the third. By applying predicate substitution repeatedly, we can generate any number of machine configurations. For example, the following binary predicates represent the second, third, fourth and fifth configurations, respectively:

$$\begin{array}{l} S_0(x,j) \\ S_0(x,j)\,[S/S_0] \\ S_0(x,j)\,[S/S_0]\,[S/S_0] \\ S_0(x,j)\,[S/S_0]\,[S/S_0]\,[S/S_0] \end{array}$$

Besides generating configurations, we must determine whether the machine accepts its input. To this end, we define a configuration to be *accepting* iff it contains an accepting control-state or the next configuration is accepting. A machine thus accepts its input iff the initial configuration is accepting. To test for acceptance, we add the following Horn rule to the rulebase $\mathcal{S}$:

$$ACCEPT_0 \;\leftarrow\; S(x,j),\, ACCEPTING(x).$$

where x and j are variables. This rule tests the machine configuration for a character x that represents an accepting control state at some location j.

Because this rule is written in terms of the predicate S, it applies only to the initial configuration stored in the database. However, we can apply this rule to other configurations by using predicate substitution. For example, the following rules determine whether the first, second, third or fourth configurations are accepting:

$$\begin{array}{l} ACCEPT \leftarrow ACCEPT_0 \\ ACCEPT \leftarrow ACCEPT_0\,[S/S_0] \\ ACCEPT \leftarrow ACCEPT_0\,[S/S_0]\,[S/S_0] \\ ACCEPT \leftarrow ACCEPT_0\,[S/S_0]\,[S/S_0]\,[S/S_0] \end{array} \tag{8}$$

In general, the following rule determines whether the i^{th} configuration is accepting:

$$ACCEPT \;\leftarrow\; ACCEPT_0\,[S/S_0]^{i-1}$$

By using recursion through substitution, we can generate all these rules automatically. The following two rules, which we add to the rulebase $\mathcal{S}$, do precisely this:

$$\begin{array}{lcl} ACCEPT & \leftarrow & ACCEPT\,[S/S_0] \\ ACCEPT & \leftarrow & ACCEPT_0 \end{array}$$

These rules infer the atom $ACCEPT$ iff some configuration in the computation is accepting, that is, iff the machine accepts its input. Intuitively, this recursion can be unfolded to produce an infinite number of non-recursive rules. For example, unfolding the first four levels of recursion produces the four rules (8) above. After unfolding, each of these rules is treated as a Horn rule in classical logic, thus effectively producing an infinite Datalog rulebase.

This completes the construction of the rulebase $\mathcal{S}$. Together with the databases $DB(M)$ and $DB(\overline{s})$, this rulebase satisfies condition (4). Since this condition represents a PSPACE-complete problem, we have proven that the data complexity of predicate substitution is PSPACE-hard.

4.2 EXPTIME Lower Bound

The rulebase constructed in Section 4.1 is relatively simple: It has only one recursive rule, which itself has only one premise. Its power to specify a PSPACE-complete problem comes from its use of recursion through substitution. In this section, we use recursion through substitution in a more sophisticated way to specify more complex problems. In particular, we construct a set of mutually recursive rules in which predicate substitution appears in more than one premise and in more than one rule. During inference, this rulebase simulates the computations of an *alternating* PSPACE machine.

Alternating machines are non-deterministic in that they can have many possible transitions at any point in a computation. Unlike ordinary non-deterministic machines, however, alternating machines may require that *all* transitions be successful, not just one. Evidence suggests that alternation increases the power of a machine. For instance, alternation gives PSPACE machines the power of EXPTIME machines. That is, APSPACE = EXPTIME [4]. Therefore, by encoding the computations of APSPACE machines, we can solve EXPTIME-complete problems. In this way, we complete the proof of Theorem 2.

To state the main result of this section precisely, suppose that M is an alternating PSPACE machine. Without loss of generality, we may assume that M has a single tape. As in Section 4.1, we encode this machine as a database $DB(M)$. Given an input string

$\overline{s}$ for the machine, we encode it as another database $DB(\overline{s})$. Finally, we construct a Datalog rulebase with substitution $\mathcal{S}$ that simulates the computations of M on input $\overline{s}$. That is,

$$\begin{gathered}\mathcal{S},\, DB(M) + DB(\overline{s}) \;\vdash\; ACCEPT \\ \textit{iff } M \textit{ accepts } \overline{s}\end{gathered} \tag{9}$$

where $ACCEPT$ is a 0-ary predicate. The important points are that the rulebase $\mathcal{S}$ is independent of the input $\overline{s}$ and that the database $DB(\overline{s})$ can be constructed in polynomial time (polynomial in the size of $\overline{s}$). From these facts, we can conclude that the data complexity of Datalog with substitution is EXPTIME-hard. This result follows immediately by letting M be a machine that recognizes a EXPTIME-complete language.[6]

The rest of this section describes the construction of $DB(M)$, $DB(\overline{s})$ and $\mathcal{S}$. It is worth noting that the entire rulebase consists of twelve rules: nine non-recursive Horn rules and three mutually-recursive rules with substitution. This structure is enough to simulate any APSPACE machine and achieve EXPTIME-hardness.

4.3 Strings and Transformations

To encode the computations of alternating PSPACE machines, we need to enrich the view of computation developed in Section 4.1. In that section, we viewed computation as a *sequence* of machine configurations, so that each configuration had exactly one successor configuration. In this section, we view computation as a *tree* of configurations, so that each configuration can have many successors. Instead of a transition *function*, the finite control of a machine thus has a transition *relation*. As in Section 4.1, we encode each machine configuration as a string. Once again, because the machines use only polynomial space, we can assume that for a given machine and a given input, all configurations are strings of the same length.

In Section 4.1, we represented the state transitions of a machine as a ternary function f. This approach is not possible here because machine transitions are now specified by a relation, not a function. However, without loss of generality, we can assume that for each tape character, a state of the finite control has exactly two successor states (if any). We can therefore represent the transition relation of a machine as two functions f_1 and f_2.

As before, these functions specify string transformations. Formally, a string $a_1 ... a_n$ has two successors $b^1_1 ... b^1_n$ and $b^2_1 ... b^2_n$ where

$$b^i_1 = a_1 \tag{10}$$

$$b^i_n = a_n \tag{11}$$

$$b^i_j = f_i(a_{j-1}, a_j, a_{j+1}) \quad \textit{for} \;\; 1 < j < n \tag{12}$$

[6] See [4] for examples of such languages.

We use the symbols t_1 and t_2 to denote these two string transformations, so that $t_i(a_1 ... a_n) = b^i_1 ... b^i_n$. If $\overline{a}$ encodes the initial configuration of an alternating PSPACE machine, then we can apply t_1 and t_2 recursively to generate a binary tree of configurations rooted at $\overline{a}$ representing the computation tree of the machine.

4.4 Data and Rules

We implement the strings and transformations described above as a rulebase with substitution. As before, we store the finite control and the initial configuration of the machine in the database, and we implement the transformations t_1 and t_2 as a set of Horn rules. Then, starting from the initial configuration, we use predicate substitution to apply the transformations over-and-over again.

Besides specifying the computation tree of the machine, the rulebase also determines whether the tree represents an accepting computation. For alternating machines, acceptance is conveniently defined in terms of accepting configurations. Whether a configuration is accepting depends on its successor configurations and on the state of the control head. An alternating machine has three kinds of states: *accepting*, *universal* and *existential*. Each configuration has exactly one control state q, and the configuration is accepting iff

- q is an accepting state, or
- q is a universal state, and all successor configurations are accepting, or
- q is an existential state, and at least one successor configurations is accepting.

An alternating machine accepts its input iff its initial configuration is accepting. As in Section 4.1, the initial configuration is stored in the database $DB(\overline{s})$.

The finite control is also stored as a database. However, the situation is more complicated than in Section 4.1, since the finite control now has two transition functions and three kinds of control states. To store the transition functions we use two 4-ary predicates F_1 and F_2, and we add $F_i(a, b, c, d)$ to the database iff $f_i(a, b, c) = d$. To store the control states, we introduce three unary predicates, $ACCEPTING$, $UNIVERSAL$ and $EXISTENTIAL$. For each compound character $\langle c, q\rangle$, we add the formula $EXISTENTIAL(\langle c, q\rangle)$ to the database iff q is an existential state. Likewise for $ACCEPTING$ and $UNIVERSAL$. Together, these formulas make up the database $DB(M)$ in statement (9).

The next task is to construct the rulebase $\mathcal{S}$. As in Section 4.1, this rulebase has two parts: (i) a set of Horn rules that implement the string transformations t_1

and t_2, and (*ii*) rules with substitution that determine whether the the machine accepts its input.

To implement the string transformation t_i, we add the three Horn rules below to the rulebase $\mathcal{S}$. These rules correspond to the three conditions (10)–(12), respectively.

$$\begin{aligned} S_i(x,j) &\leftarrow FIRST(j),\ S(x,j). \\ S_i(x,j) &\leftarrow LAST(j),\ S(x,j). \\ S_i(x,j) &\leftarrow S(u,i),\ S(v,j),\ S(w,k), \\ &\qquad F_i(u,v,w,x), NEXT(i,j), NEXT(j,k). \end{aligned}$$

where i, j, k and all the x's are variables. Given a string $\overline{a}$ stored in the predicate S, these rules derive the transformed string $t_i(\overline{a})$ and store it in the predicate S_i.

The three Horn rules defining S_i represent *one* application of the transformation t_i to the string S encoded in the database. As in Section 4.1, we use predicate substitution to apply these transformations over-and-over again. This time, however, we have two substitutions to apply, $[S/S_1]$ and $[S/S_2]$, and the order in which we apply them determines how the transformations t_1 and t_2 are composed. For example, if the predicate $S(x,j)$ represents the string $\overline{a}$, then the following predicates with substitution represent the strings $t_1(\overline{a})$, $t_2t_1(\overline{a})$, $t_1t_2t_1(\overline{a})$, $t_1t_1t_2t_1(\overline{a})$ and $t_2t_1t_1t_2t_1(\overline{a})$, respectively:

$$\begin{aligned} &S_1(x,j) \\ &S_1(x,j)\,[S/S_2] \\ &S_1(x,j)\,[S/S_2]\,[S/S_1] \\ &S_1(x,j)\,[S/S_2]\,[S/S_1]\,[S/S_1] \\ &S_1(x,j)\,[S/S_2]\,[S/S_1]\,[S/S_1]\,[S/S_2] \end{aligned}$$

Besides generating configurations, the rulebase must determine whether the machine accepts its input. We therefore construct rules that determine whether a configuration is accepting. The first step is to determine what kind of control state a configuration has. The following Horn rules accomplish this:

$$\begin{aligned} &ACCEPT \leftarrow ACCEPTING(x),\ S(x,j). \\ &UNIV \leftarrow UNIVERSAL(x),\ S(x,j). \\ &EXIST \leftarrow EXISTENTIAL(x),\ S(x,j). \end{aligned}$$

where x and j are variables. The predicate $EXIST$ is inferred iff the configuration represented by S contains an existential control state x at some position j. Likewise for the predicates $UNIV$ and $ACCEPT$.

If the configuration has an accepting control state, then we are done. If not, then we can examine its successor configurations using predicate substitution. For example, the predicates $ACCEPT\,[S/S_1]$ and $ACCEPT\,[S/S_2]$ ask whether the two successor configurations are accepting, resp. By using recursion through substitution, we can recursively examine all successor configurations. The following rules, which we add to the rulebase $\mathcal{S}$, do exactly this:

$$\begin{aligned} ACCEPT &\leftarrow UNIV,\ ACCEPT\,[S/S_1], \\ &\qquad\qquad ACCEPT\,[S/S_2]. \\ ACCEPT &\leftarrow EXIST,\ ACCEPT\,[S/S_1] \\ ACCEPT &\leftarrow EXIST,\ ACCEPT\,[S/S_2] \end{aligned}$$

The first rule says that a universal configuration is accepting if both its successors are accepting. The second and third rules say that an existential configuration is accepting if at least one of its successors is accepting.

This completes the construction of the rulebase $\mathcal{S}$. Applied to the database $DB(M) + DB(\overline{s})$, constructed above, this rulebase infers the atom $ACCEPT$ iff the initial configuration of the machine is accepting, that is, iff machine M accepts input $\overline{s}$. This rulebase therefore satisfies condition (9). Since this condition represents an EXPTIME-complete problem, the data complexity of Datalog with predicate substitution is EXPTIME-hard. This completes the proof of Theorem 2.

5 Model Theory

It is conceivable that an upper bound on the complexity of predicate substitution could be obtained directly from the proof theory. This would appear to be a difficult task, however, since the proof system is able to create any number of new predicate symbols during inference. Most of these invented predicates are the result of multiple predicate substitutions, as in $A[P/Q][P/Q][P/Q]$. Furthermore, the arity of invented predicates is unbounded since each substitution can introduce new variables, thus creating predicates of larger-and-larger arity, as in $A(v)[P/Q](x)[P/Q](y)[P/Q](z)$. The model theory developed in this section alleviates this problem, since a model is a set of Horn rules *without* substitution. That is, all models are built from a fixed, finite set of predicate and constant symbols.

Besides providing a convenient device for establishing an upper complexity bound, a model theory for predicate substitution needs to be developed anyway, to complete the logical development of predicate substitution. In [3, 2], a model theory was developed for "basic" substitution, i.e., for the case in which a predicate may only substitute for predicates of *equal* arity. In this section, we extend that theory to the general case, in which a predicate can substitute for predicates of *different* arity. Because of space limitations, however, we present the model theory with a minimum of discussion, and with no proofs, providing an example to illustrate the theory in the general case. We refer the interested reader to [3] and [2] for more detailed developments.

5.1 the Model Theory

This section outlines the model theory for full predicate substitution, i.e., for the case in which a predicate may substitute for predicates of *different* arity. In this theory, a model is a Horn rulebase without substitution. Such models support the idea that a rulebase *with* substitution is a short-hand notation for a (possibly larger) rulebase *without* substitution, which the user might have written had he had the time, the space, and the inclination.

Definition 5 (Interpretations) *An interpretation is a set R of Horn rules without substitution.*

We must define what it means for an interpretation R to be a model of a rule with substitution. That is, we must give a precise meaning to the expression $R \models \alpha \leftarrow \beta_1 ... \beta_n$, where each β_i is a predicate with substitution. Before considering the general case, we focus on a simple rule, $A(x,y) \leftarrow B(x)[P/Q](y)$. Informally, this rule means that the definition of predicate A is the same as that of predicate B with P replaced by Q. For instance, suppose that B is defined by the following two rules:

$$\begin{array}{l} B(x) \leftarrow E(x), P(x). \\ B(x) \leftarrow F(x,v), P(x), P(v). \end{array}$$

Then A is defined by the following two rules:

$$\begin{array}{l} A(x,y) \leftarrow E(x), Q(x,y). \\ A(x,y) \leftarrow F(x,v), Q(x,y), Q(v,y). \end{array}$$

If R is the Horn rulebase consisting of these four rules, then we say that R is a *model* of the rule $A(x,y) \leftarrow B(x)[P/Q](y)$. The underlying principle is that in R, the Horn rules defining B give rise to Horn rules defining A. If more Horn rules defining B are added to R, then more Horn rules defining A must also be added; otherwise, R will not remain a model of $A(x,y) \leftarrow B(x)[P/Q](y)$. The following definition formalizes this idea. In this definition, R^* is the rulebase specified by Definition 3. It provides Horn rules defining every predicate with substitution β. An extensive discussion of this definition may be found in [2].

Definition 6 (Models) *Let R be a set of Horn rules, let α be an atomic formula, and let $\beta_1 ... \beta_n$ be predicates with substitution. Then, R is a model of $\alpha \leftarrow \beta_1 ... \beta_n$ iff for all conjunctions of atomic formulas γ, and all term substitutions σ,*

$$R \models_c \alpha\sigma \leftarrow \gamma \quad \textit{if} \quad R^* \models_c \beta_i\sigma \leftarrow \gamma \quad \textit{for each } i.$$

If $\mathcal{S}$ is a Horn rulebase with substitution and R is a Horn rulebase without substitution, then we say that R is a model of $\mathcal{S}$ iff R is a model of each rule in $\mathcal{S}$. In this case, we write $R \models \mathcal{S}$.

Definition 7 *Let R be a set of Horn rules, β a predicate with substitution, and DB a database. Then $R, DB \models \beta$ iff $R^* \cup DB \models_c \beta$.*

Definition 8 (Entailment) *Suppose $\mathcal{S}$ is a set of rules with substitution, DB is a database for $\mathcal{S}$, and β is a predicate with substitution. Then $\mathcal{S}, DB \models \beta$ iff $R, DB \models \beta$ for every model R of $\mathcal{S}$.*

Proofs of the following two theorems are given in [2].

Theorem 4 (Soundness) *Suppose β is a predicate with substitution, $\mathcal{S}$ is a Horn rulebase with substitution, and DB is a database. If $\mathcal{S}, DB \vdash \beta$ then $\mathcal{S}, DB \models \beta$.*

Theorem 5 (Completeness) *Suppose β is a predicate with substitution, $\mathcal{S}$ is a Horn rulebase with substitution, and DB is a database. If $\mathcal{S}, DB \models \beta$ then $\mathcal{S}, DB \vdash \beta$.*

6 Fixpoint Semantics

The model theory of Section 5 is one step in establishing an upper bound on the complexity of predicate substitution. It allows us to answer database queries by constructing models instead of performing inference. By itself, however, the model theory does not give us a mechanical way of answering queries, since we might have to examine infinitely many models. The fixpoint theory developed in this section solves this problem. This theory shows that a rulebase with substitution has a least model, which contains all the information necessary to answer database queries. Furthermore, the fixpoint theory shows how to construct this model in a bottom-up way by a simple iterative procedure, much as the least model of a Datalog program is constructed.

Besides providing a convenient device for establishing an upper complexity bound, a fixpoint theory for predicate substitution needs to be developed anyway. Such a theory, if properly developed, is an essential part of our goal of developing a logic programming language for reusing and modifying rulebases. In [3], a fixpoint theory was introduced for "basic" predicate substitution, i.e., for the case in which a predicate may only substitute for predicates of *equal* arity. In this section, we extend that theory to the general case, in which a predicate can substitute for predicates of *different* arity. Because of space limitations, however, we present the fixpoint theory with a minimum of discussion, and with no proofs, providing an example to illustrate the theory in the general case. We refer the interested reader to [3] and [2] for more detailed developments.

6.1 Sketch of the Fixpoint Semantics

In the tradition of logic-programming [9, 1], we define a "T-operator" that can be applied to a rulebase in a bottom-up fashion to generate a minimal model (or least fixpoint). In Horn logic, the T-operator takes a *database* as input and returns another database as output. In our logic of predicate substitution, the T-operator takes a Horn *rulebase* as input and returns another Horn rulebase as output. Starting with the empty rulebase, the T-operator is applied over-and-over again until the minimal model is generated. Each application of the operator expands rules with substitution into equivalent Horn rules. If there is recursion through substitution, it may take infinitely many applications to generate all the Horn rules in the minimal model. Because the operator is continuous, however, any particular Horn rule will be generated in a finite number of applications. In this way, the Horn rules in the least model can be enumerated.

The central feature of the fixpoint theory is a lattice whose elements are Horn rulebases. The elements of the lattice are ordered according to classical entailment.

Lemma 6 *The set of all Horn rulebases without substitution is a complete lattice under the following operations:*

1. $R_1 \geq R_2$ *iff* $R_1 \models_c R_2$
2. $\sqcup\mathcal{R} = \cup\mathcal{R}$ *for any set* $\mathcal{R}$ *of Horn rulebases.*
3. $\sqcap\mathcal{R} = \cap_{R\in\mathcal{R}} \{\rho \mid R \models_c \rho\}$ *where each* ρ *is a Horn rule.*

Another central feature of any fixpoint theory is a "T operator," which takes one lattice element as input and returns another lattice element as output.

Definition 9 (T operator) *Let* $\mathcal{S}$ *be a Horn rulebase with substitution, and let* R *be a Horn rulebase (without substitution). Then,* $T_\mathcal{S}$ *is a mapping that takes a Horn rulebase* R *as input and returns another Horn rulebase* $T_\mathcal{S}(R)$ *as output. In particular, the value of* $T_\mathcal{S}(R)$ *is*

$$\cup_\sigma \{\alpha\sigma \leftarrow \gamma \mid R^* \models_c \beta_i\sigma \leftarrow \gamma \text{ for each premise } \beta_i \text{ in some rule } \alpha \leftarrow \beta_1 ... \beta_n \text{ in } \mathcal{S}\}$$

where the union is over all term substitutions σ, *and each* γ *is a conjunction of atomic formulas.*

The following basic result says that the fixpoints of $T_\mathcal{S}$ are exactly the models of $\mathcal{S}$.

Lemma 7 $R \geq T_\mathcal{S}(R)$ *iff* $R \models \mathcal{S}$ *for any Horn rulebase* R.

Intuitively, the operator $T_\mathcal{S}$ attempts to "unfold" the rules with substitution in $\mathcal{S}$ and convert them into Horn rules, just as the operator T_R of Horn logic attempts to "unfold" the Horn rules in R and convert them to atoms. We now state the basic results about T-operators without proof. The main idea is to prove that the operator $T_\mathcal{S}$ is monotonic and continuous, so that the Tarski fixpoint theorem can be invoked [8]. Detailed proofs of these results are presented in [2].

Lemma 8 $T_\mathcal{S}$ *is monotonic: If* $R_1 \geq R_2$ *then* $T_\mathcal{S}(R_1) \geq T_\mathcal{S}(R_2)$.

Corollary 9 *The operator* $T_\mathcal{S}$ *has a least fixpoint* $lfp(T_\mathcal{S})$. *That is,* $lfp(T_\mathcal{S}) \leq R$ *for all fixpoints* R *of* $T_\mathcal{S}$.

Lemma 10 $T_\mathcal{S}$ *is continuous: If* $R_1 \leq R_2 \leq R_3 \ldots$ *then* $T_\mathcal{S}(\sqcup_{j\geq 1} R_j) \equiv \sqcup_{j\geq 1} T_\mathcal{S}(R_j)$.

Corollary 11 $lfp(T_\mathcal{S}) \equiv \sqcup_{j\geq 1} T^j_\mathcal{S}(\{\})$

The least fixpoint $lfp(T_\mathcal{S})$ contains all the information needed to answer queries about a database, as the following result shows.

Corollary 12 *Let* $\mathcal{S}$ *be a Horn rulebase with substitution,* β *be a predicate with substitution, and* DB *be a database. Then*

$$\mathcal{S}, DB \models \beta \quad \text{iff} \quad lfp(T_\mathcal{S}), DB \models \beta$$

Corollary 11 provides a mechanism for computing the least fixpoint of a rulebase, and thus for answering queries. The following example shows how this mechanism gradually "unfolds" rules with substitution into Horn rules.

Example 7. Suppose the rulebase $\mathcal{S}$ consists of the following three rules, where B, D, and each C_i are base predicates:

$$A_1(x) \leftarrow B(x), C_1(x).$$
$$A_1(x) \leftarrow D(x,y), C_1(x), C_1(v).$$
$$A_2(x,y) \leftarrow A_1(x)[C_1/C_2](y)$$
$$A_3(x,y,z) \leftarrow A_2(x,y)[C_2/C_3](z)$$

We assume that predicate C_i has arity i. It then takes three applications of the operator $T_\mathcal{S}$ to generate the minimal model of $\mathcal{S}$. That is, $lfp(T_\mathcal{S}) = T^3_\mathcal{S}(\{\})$. This model consists of six Horn rules, generated as follows, where T^i denotes $T^i_\mathcal{S}(\{\})$:

$$\left.\begin{array}{l} A_1(x) \leftarrow B(x), C_1(x). \\ A_1(x) \leftarrow D(x,v), C_1(x), C_1(v). \end{array}\right\} \in T^1$$

$$\left.\begin{array}{l} A_2(x,y) \leftarrow B(x), C_2(x,y). \\ A_2(x,y) \leftarrow D(x,v), C_2(x,y), C_2(v,y). \end{array}\right\} \in T^2$$

$$\left.\begin{array}{l} A_3(x,y,z) \leftarrow B(x), C_3(x,y,z). \\ A_3(x,y,z) \leftarrow D(x,v), C_3(x,y,z), C_3(v,y,z). \end{array}\right\} \in T^3$$

Notice that definition of the predicate A_{i+1} is the same as the definition of A_i except that C_i has been replaced everywhere by C_{i+1}, a predicate of greater arity. $\Box$

7 Upper Complexity Bound

In this section, we use the theory developed in the previous two sections to show that the data complexity of the full logic of predicate substitution is in EXPTIME. Recall that full predicate substitution allows a predicate to substitute for predicates of *different* arity. In contrast, basic substitution only allows substitution for predicates of *equal* arity. Since full substitution includes basic substitution as a special case, this section establishes the following results.

Theorem 13 *The data complexity of Datalog rulebases augmented with full predicate substitution is in EXPTIME.*

Corollary 14 *The data complexity of Datalog rulebases augmented with basic predicate substitution is in EXPTIME.*

The model theory and fixpoint theory developed in the previous two sections are central to our proof of these upper bounds. In particular, if we combine soundness and completeness (Theorems 4 and 5) with Corollary 12, then we get the following result, upon which our proof is based:

$$\mathcal{S}, DB \vdash \alpha \quad \textit{iff} \quad \textit{lfp}(T_{\mathcal{S}}) \cup DB \models_c \alpha \qquad (13)$$

for any ground atomic formula α, where "$\models_c$" denotes classical entailment. In principle, then, instead of performing inference with substitution from the rulebase $\mathcal{S}$, we can perform classical inference from the least model of $\mathcal{S}$.

This approach requires some qualification, since the least model may be infinite, and thus impossible to materialize. This difficulty is related to the power of the least model: It can answer queries about *any* database. Fortunately, to answer queries about a *particular* database, we can use a smaller model. Specifically, we can use the least model, not of the rulebase $\mathcal{S}$, but of its ground instantiation. This model is guaranteed to be *finite*. In fact, we show that it is a set of ground Datalog rules of at-most exponential size (exponential in the size of the database).

To make these ideas precise, we first give the following definition.

Definition 10 (Ground Instantiations) *Let $\mathcal{C}$ be a finite set of constant symbols, let ρ be a rule with substitution, and let $\mathcal{S}$ be a set of rules with substitution.*

- *The ground instantiation of ρ wrt $\mathcal{C}$ is the set of ground instances of ρ derived by replacing each variable in ρ by a constant in $\mathcal{C}$ in all possible ways. It is denoted $inst_{\mathcal{C}}(\rho)$.*
- *The ground instantiation of $\mathcal{S}$ wrt $\mathcal{C}$ is the union of the ground instantiations wrt $\mathcal{C}$ of each rule in $\mathcal{S}$. It is denoted $inst_{\mathcal{C}}(\mathcal{S})$.*

The following basic result about ground instantiation shows that to answer queries about a particular database DB, we can replace a rulebase by its ground instantiation. Note that the ground instantiation depends on the database and must therefore be recomputed whenever the database changes.

Lemma 15 *Let β be a ground predicate with substitution, let DB be a database, and let $\mathcal{S}$ be a Datalog rulebase with substitution. If the constant symbols in β, DB and $\mathcal{S}$ all appear in the set $\mathcal{C}$, then*

$$\mathcal{S}, DB \vdash \beta \quad \textit{iff} \quad inst_{\mathcal{C}}(\mathcal{S}), DB \vdash \beta$$

We will not actually use $inst_{\mathcal{C}}(\mathcal{S})$ to answer queries. Instead, we shall use its least model. This model is a set of ground Datalog rules of at-most exponential size. In fact, we have the following, more-general results.

Lemma 16 *Let $\mathcal{S}$ be a set of ground rules with substitution whose constant symbols are in $\mathcal{C}$. Then $T^i_{\mathcal{S}}(\{\})$ is a set of ground Datalog rules whose constant symbols are also in $\mathcal{C}$.*

Lemma 17 *Given a finite set of predicate symbols (of fixed arity), and a finite set of constant symbols, the number of ground Horn rules constructible from these symbols is exponential in the number of constants.*

Proof: A ground Horn rule is made from a set of ground atomic formulas. Since the arity of the predicates is fixed, the number of distinct atomic formulas is polynomial in the number of constants. In fact, if k is the maximum arity of any predicate, and if n is the number of constants, then the number of ground atomic formulas is $n^{O(k)}$. Thus, the number of *sets* of ground atomic formulas is $2^{n^{O(k)}}$. Thus, the number of ground Horn rules is exponential in n. $\Box$

Because the least model of $inst_{\mathcal{C}}(\mathcal{S})$ is finite, we can materialize it by repeatedly applying its T-operator. That is, starting with $R = \{\}$, we repeatedly execute the following assignment statement:

$$R \;:=\; T_{inst_{\mathcal{C}}(\mathcal{S})}(R) \tag{14}$$

When R no longer changes, it is the least model of $inst_{\mathcal{C}}(\mathcal{S})$.

It is not hard to bound the complexity of this procedure. First, by Lemma 16, at every stage in the computation, R is a set of ground Datalog rules whose constant symbols are in $\mathcal{C}$. Thus, R always has at-most exponential size. From this, it is not difficult to show that each execution of statement (14) takes at most exponential time. Second, since T-operators are monotonic, each execution of the statement, except the last, adds new rules to R. Thus, since R has at-most exponential size, the statement itself executes at-most exponentially many times before saturation is reached. Hence, we have exponentially-many executions, each taking exponential time. The total execution time of the procedure is therefore bounded above by an exponential (exponential in the size of the database).

Finally, having computed R, we can use it to answer queries about the database DB. That is, for any ground atomic formula α,

$$\mathcal{S}, DB \vdash \alpha \quad \mathit{iff} \quad R \cup DB \models_c \alpha$$

Because R is a set of ground Datalog rules, the time needed to infer α is polynomial in the size of $R \cup DB$, and thus exponential in the size of DB. In this way, any atomic query to the rulebase $\mathcal{S}$ can be answered in exponential time, thus establishing the EXPTIME upper-bound of Theorem 13.

References

[1] K.R. Apt and M.H. Van Emden. Contributions to the Theory of Logic Programming. *Journal of the ACM*, 29(3):841–862, 1982.

[2] A.J. Bonner. *Hypothetical Reasoning in Deductive Databases*. PhD thesis, Department of Computer Science, Rutgers University, New Brunswick, NJ 08903, USA, October 1991. Published as Rutgers Technical Report DCS-TR-283.

[3] A.J. Bonner and T. Imielinski. The Reuse and Modification of Rulebases by Predicate Substitution. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, pages 437–451, Venice, Italy, March 26–30 1990. Springer-Verlag. Published as volume 416 of *Lecture Notes in Computer Science*.

[4] A.K. Chandra, D. Kozen, and L.J. Stockmeyer. Alternation. *Journal of the ACM*, 28:114–133, 1981.

[5] W. Chen, M. Kifer, and D.S. Warren. Hilog: A First-Order Semantics for Higher-Order Logic Programming Constructs. In *Proceedings of the North American Conference on Logic Programming (NACLP)*, pages 1090–1114. MIT Press, 1989.

[6] J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.

[7] C.S. Kwok and M. Sergot. Implicit Definiton of Logic Programs. In *International Conference on Logic Programming (ICLP)*, pages 374–385, 1988.

[8] A. Tarski. A Lattice-Theoretical Fixpoint Theorem and its Applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.

[9] M.H. Van Emden and R.A. Kowalski. The Semantics of Predicate Logic as a Programming Language. *Journal of the ACM*, 23(4):733–742, 1976.