Reusing and Modifying Rulebases by Predicate Substitution

Anthony J. Bonner*
University of Toronto
Department of Computer Science
Toronto, Ontario, Canada M5S 3G4
bonner@db.toronto.edu

Tomasz Imielinski
Rutgers University
Department of Computer Science
New Brunswick, NJ 08903, United States

Abstract

We propose a method for reusing and modifying a deductive database. 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. In order to facilitate reuse, we extend deductive database systems by the concept of *predicate substitution*. In this way, during query evaluation, not only variables, but also predicates can be substituted. We provide a proof theory and a model theory for this language, including a fixpoint semantics. In addition, we show that substitution increases the expressive power of Datalog: not only does its data complexity increase from PTIME to EXPTIME, but substitution also allows large sets of Datalog rules to be succinctly expressed. In fact, finite rulebases with substitution can express infinite rulebases without substitution.

Published in the *Journal of Computer and System Sciences (JCSS)*, 54(1):136–166, February 1997. Special issue on the *Eleventh ACM Symposium on Principles of Database Systems (PODS'92)*.

This and related papers are available at the following web page:

`www.cs.toronto.edu/~bonner/papers.html#reuse`

*Work supported in part by an Operating Grant from the Natural Sciences and Engineering Research Council of Canada and by a Connaught Grant from the University of Toronto.

Contents

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 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:

$$\begin{aligned} &TR(x,y) \leftarrow R(x,y). \\ &TR(x,z) \leftarrow R(x,y),\, TR(y,z). \end{aligned}$$

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:

$$\begin{aligned} &TS(x,y) \leftarrow S(x,y). \\ &TS(x,z) \leftarrow S(x,y),\, TS(y,z). \end{aligned}$$

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. Note that in this example, one binary predicate substitutes for another predicate of *equal* arity. More generally, predicate substitution allows for the substitution of predicates of *different* arity. As we shall see, the result is a more flexible language that permits a greater exploitation of polymorphism.

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 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.

We develop these ideas in several ways. First, Section 2 defines 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. Section 2 develops the proof theory for the logic, Section 3 develops the model theory, and Section 4 develops the fixpoint theory. A main result is that the proof theory is sound and complete for the model theory. Likewise for the fixpoint 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 minimal 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. These ideas are illustrated by the Examples in Sections 3.1 and 4.3.

To support this view, we show that predicate substitution is a concise notation for describing large rulebases. This conciseness is achieved by reusing and modifying rule sets, possibly many times. In fact, by combining recursion and substitution, it is possible to reuse and modify rule sets infinitely many times. Using this idea, we show that a *finite* set of rules with substitution can specify an *infinite* set of rules without substitution. Furthermore, this infinite set is not equivalent to any finite one. 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 in that a query can be evaluated without materializing the entire least fixpoint of the rulebase. The conciseness of predicate substitution is illustrated by the Examples in Section 4.3.

In the tradition of logic programming [18, 1], Section 4 complements the model theory of predicate substitution with a fixpoint theory. It defines a monotonic and continuous "T-operator" that can be applied to a rulebase in a bottom-up fashion, generating the minimal model (or least fixpoint) piece-by-piece. In Horn logic, the T-operator takes a database as input and returns another database as output. In our logic, 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 unfolds rules with substitution into equivalent Horn rules. If a rulebase has recursion through substitution, then its minimal model may contain infinitely many Horn rules, and it may take infinitely many applications of the T-operator to generate them all. However, because the T-operator is continuous, any particular Horn rule will be generated after a finite number of applications. In this way, the Horn rules represented by a rulebase with substitution can be enumerated. These ideas are illustrated by the Examples in Section 4.3.

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 flexibility and 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, Sections 6 and 7 show that substitution increases the power of Datalog (function-free Horn logic). Whereas the data complexity of Datalog is complete for PTIME, Datalog with substitution is complete for EXPTIME. (In this paper, the terms EXPTIME and "exponential time" mean DTIME[$2^{n^{O(1)}}$].) Proof theoretically, this boost in expressive power comes from recursion through substitution, and in particular, from the ability of the logic to simulate the computations of alternating PSPACE machines. Model theoretically, this increased power has an interesting interpretation: when a Datalog rulebase with substitution is EXPTIME-complete, it represents an *infinite* Datalog rulebase *without* substitution. Because this rulebase is infinite, it is not subject to the normal PTIME bound on the complexity of *finite* Datalog programs. In cases like this, predicate substitution provides an extremely concise notation for representing Datalog rulebases.

1.1 Related Work

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 related works in the logic-programming literature, especially those that use higher-order function terms. This is the case, for instance, in the HiLog system of Chen, Kifer and Warren [7], and in the meta-programming methodology of Kwok and Sergot [8]. 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, is in EXPTIME.

There are also philosophical and practical differences between predicate substitution and higher-order functions. Although they allow a program to be applied to different predicates, the explicit use of higher-order functions requires a programmer to *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. It is as if *every* predicate symbol in a program with predicate substitution were a higher-order variable, though guaranteed to be bound to a specific predicate symbol at run time. The proof and model theories developed in this paper reflect this capability.

Predicate substitution is also different from works on polymorphism in functional languages, such as [11, 12]. One obvious difference is that predicate substitution applies to logic programming, not functional programming. A deeper difference is that these works are concerned with static type checking and how to make it polymorphic, whereas our work has nothing to do with type checking, whether static or dynamic. As with most logic programs and deductive databases, the language developed in this paper is untyped. However, any typing discipline developed for first-order logic programs should also be applicable to logic programs with predicate substitution. For instance, [13] adds static polymorphic type checking to Prolog, and many of their techniques could be applied to Prolog augmented with predicate substitution. Work on type checking, polymorphic or otherwise, is thus orthogonal to our work.

There is another difference as well. Any typing discipline—even a polymorphic one—*decreases* the flexibility of an otherwise untyped language. In contrast, predicate substitution *increases* the flexibility of (untyped) logic programs, by adding a limited form of second-order functionality. Intuitively, it allows new predicate symbols to be constructed at run time, much as the (untyped) lambda calculus allows new lambda terms to be constructed at run time. In this sense, predicate substitution adds features to logic programming that are normally found only in functional programming and in higher-order logics.

1.2 Introductory Examples

To better convey the intuition behind predicate substitution, we present several examples and discuss possible applications. 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 can be used in queries or in rules bodies, just as other predicates can. In fact, 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 below illustrate this idea.

The first two examples illustrate the reuse and modification of a sorting module. Such modules can be defined in general logic programming languages, like Prolog. By reusing and modifying the module, we greatly extend its range of application. The first example illustrates basic predicate substitution, in which a predicate substitutes for predicates of *equal* arity. The second example illustrates full predicate substitution, in which a predicate substitutes for predicates of *different* arity. Full substitution uses a generalization of the notation described above.

Example 1.1 (Basic Substitution) Given a rulebase R that sorts people by age, we reuse it, with slight modifications, to sort people by weight. We suppose that the database DB defines the predicates $Person(x)$, $Age(x,y)$ and $Weight(x,w)$, which mean respectively that x is a person, and that x has age y and weight w. We suppose also that the rulebase defines a predicate $SortAge(x,n)$ that sorts people by age. That is,

$$R, DB \vdash SortAge(x,n)$$

iff person x is the n^{th} oldest person in the database. We do not know how the predicate *SortAge* is implemented, nor what algorithm it uses. We know only that the rules defining *SortAge* are written in terms of the base predicates *Person* and *Age*.

To sort people by weight instead of by age, we can rewrite these rules substituting *Weight* for *Age*. The predicate with substitution $SortAge(x,n)[Age/Weight]$ does this automatically. That is,

$$R, DB \vdash SortAge(x,n)[Age/Weight]$$

iff x is the n^{th} heaviest person in the database. This predicate with substitution can be used in rule bodies to define other predicates. In this way, the following rule defines a predicate

SortWeight that sorts people by weight:

$$SortWeight(x, n) \leftarrow SortAge(x, n)[Age/Weight]$$

This rule is called a rule with substitution. Intuitively, it means that the Horn rules defining *SortWeight* are the same as those defining *SortAge* except that *Age* is replaced by *Weight*. In this way, predicate substitution exploits the fact that sorting is polymorphic on the sorted attribute. □

Example 1.1 illustrates the most basic kind of predicate substitution. It is basic in several ways: it only substitutes base predicates, it only performs a single substitution, and it only substitutes predicates of equal arity. Predicate substitution is more general than this. In the substitution $[P/Q]$, the predicate P must be a base predicate, but the predicate Q need not be. Furthermore, substitutions can be applied multiple times, to define predicates like $A[P/Q][R/S]$. Finally, by extending the syntax, one predicate can be replaced another of *greater* arity. Specifically, if the arity of Q is greater than that of P, then to replace P by Q, we use the expression $[P/Q](y_1...y_m)$ where $y_1...y_m$ denote the extra arguments of Q that do not appear in P. The next example illustrates these possibilities.

Example 1.2 (Full Substitution) Continuing the previous example, a university would like to use the sorting program, but in a more general fashion. It would like to sort numerous groups of students, and it would like to sort them by grade, not age.

As before, we suppose that the sorting predicate $SortAge(x, n)$ is written in terms of the base predicates $Person(x)$ and $Age(x, y)$. We also suppose that the university's database defines the predicates $FullTime(s)$ and $Grade(s, g, c)$, which mean respectively that s is a full-time student, and that s received grade g in course c. The university wants to sort the full-time students in each course, and it wants to sort them by their grade in that course. The following rules do exactly this:

$$\begin{aligned} FTStudent(s, c) &\leftarrow FullTime(s), grade(s, g, c). \\ SortGrade(s, c, n) &\leftarrow SortAge(s, n)\,[Person/FTStudent](c)\,[Age/Grade](c) \end{aligned}$$

The first rule defines a predicate $FTStudent(s, c)$ that identifies each full-time student s taking course c. Then, for each course, c, the second rule sorts the full-time students in the course by the grade they received in the course. Thus, $SortGrade(s, c, n)$ is true iff of all the full-time students in course c, student s received the n^{th} highest grade. Conceptually, *SortGrade* is defined by Horn rules that are identical to those defining *SortAge* except that the base predicate *Person* is replaced by the derived predicate *FTStudent*, and *Age* is replaced by *Grade*. □

Numerous variations on Examples 1.1 and 1.2 are possible. It is not hard, for instance, to change the comparator on which the sort is based. To do this, suppose the rules defining *SortAge* use the arithmetic predicate *leq* to compare peoples ages, where $leq(y_1, y_2)$ is true iff y_1 and y_2 are integers and $y_1 \leq y_2$. By using predicate substitution, *leq* can be replaced by different comparison predicates. For instance, the predicate $SortAge(s, n)[leq/geq]$ still sorts people by age, but in reverse order. Likewise, comparators for real numbers or other data types can be substituted for *leq*.

The next example shows an application to scientific databases in which a module for matrix multiplication is reused and modified to compute matrix powers. Like the sorting module above, matrix multiplication modules can be defined in general logic programming languages, like Prolog.

Example 1.3 (Recursive Substitution) A particular scientific application needs to perform simple operations on large matrices. Since the application also needs to interface to a database system, it is decided that the matrices should be stored in the database itself. Each matrix is stored as a ternary relation, $M(i, j, v)$, where v is the ij^{th} element of the matrix represented by M. To multiply matrices, the rulebase defines a predicate, $Mult$, that multiplies matrices M_1 and M_2. That is,

$$R, DB \vdash Mult(i, j, v)$$

iff v is the ij^{th} element in the product of M_1 and M_2. To multiply other matrices, we use predicate substitution. For instance, $Mult(i, j, v)[M_1/A][M_2/B]$ represents the product of matrices A and B. By combining multiplication and substitution in more complex ways, we can define other matrix operations. For instance, the following rules use recursion through substitution to compute the n^{th} power of matrix M:

$$\begin{aligned} Power(i, j, v, 1) &\leftarrow M(i, j, v) \\ Power(i, j, v, n) &\leftarrow Mult(i, j, v)[M_1/M][M_2/Power](n-1) \end{aligned}$$

The first rule defines the first power of M to be M itself. The second rule defines the n^{th} power of M to be M times its $n-1^{st}$ power. Thus, $Power(i, j, v, n)$ is true iff v is the ij^{th} element in the n^{th} power of M. To compute the n^{th} power of other matrices, we use predicate substitution. For instance, $Power(i, j, v, n)[M/A]$ represents the n^{th} power of matrix A. $\square$

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.

In addition, rules with substitution seem well suited to the reuse and modification of expert database systems. Such systems have a variety of applications, such as computer-based medical and legal consultation systems. Kowalski and Sergot, for instance, have encoded the British Nationality Act in Prolog [15], and McCarty has developed systems for reasoning about tax law, especially corporate tax law and estate tax law [10, 14]. As laws are amended, such systems have to be updated. For instance, sections of the income tax act might be amended to treat residents as citizens. In this case, every occurrence of the predicate "citizen" could be replaced by a new predicate meaning "citizen or resident". Likewise, other bodies of law might be amended to replace the predicate "male" by "person".[1] Such updates could be expressed and implemented with predicate substitution.

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

[1] L.T. McCarty. Personal communication.

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.

From the above examples it may appear that adding the principle of substitution may increase the convenience and flexibility of a language but not its expressive power. We show, however, that the expressive power increases as well, and in particular, that predicate substitution increases the expressive power of Datalog (function-free Horn logic). To illustrate this increased power, Section 7 provides a set of rules that use recursion through substitution to solve an EXPTIME-complete problem—a task impossible for Datalog.

2 Proof Theory: Inference with Predicate Substitution

This section develops predicate substitution in terms of the reuse and modification of rulebases. The syntax of predicates and rules with substitution is defined precisely, and an inference system is developed in terms of a copy-and-substitute mechanism. The development is initially informal, leaning on a number of illustrative examples. The mechanism is then defined precisely and an axiomatization is given.

2.1 Syntax

The language of predicate substitution includes three infinite sets: predicate symbols, function symbols, and variables. Each predicate and function symbol has a non-negative arity. A function symbol of arity 0 is called a *constant symbol*, or simply a *constant*. From these symbols, we recursively build two other sets of symbols: *function terms* and *predicate symbols with substitution*. Function terms are defined in the standard way: a variable or a constant symbol is a function term; and if f is a function symbol of arity $k \geq 1$, and $x_1, ..., x_k$ are function terms, then $f(x_1, ..., x_k)$ is also a function term.

Definition 2.1 (Predicate Symbols with Substitution)

- *Every predicate symbol of arity k is a predicate symbol with substitution of arity k.*
- *Suppose P and Q are predicate symbols of arity m and $m + k$, respectively, where $k \geq 0$. If $\mathcal{A}$ is a predicate symbol with substitution of arity n, then $\mathcal{A}[P/Q]$ is a predicate symbol with substitution of arity $n + k$.*

As an example, suppose A is a predicate symbol of arity 1, and P_i and Q_i are predicate symbols of arity i. Then $A[P_2/Q_3]$, $A[P_3/Q_5]$ and $A[P_2/Q_3][P_3/Q_5]$ are predicate symbols with substitution of arity 2, 3 and 4, respectively. Thus, while predicate symbols have an arbitrary arity, predicate symbols with substitution have an arity derived from their structure. This is the only formal difference between them. For convenience, we shall sometimes refer to predicate symbols as "predicate symbols without substitution".

In this paper, an atomic formula is an expression of the form $P(x_1 ... x_k)$ where each x_i is a function term and P is a predicate symbol (without substitution) of arity k. We could easily extend the notion of atomic formulas to include predicate symbols with substitution, but it is convenient to use a different notation and a different terminology for them. For instance, although $A[P_2/Q_3]$ is a predicate symbol with substitution of arity

2, we shall not write $A[P_2/Q_3](x, y)$, nor shall we call it an atomic formula. Instead, we shall write $A(x)[P_2/Q_3](y)$ and call it a predicate with substitution. Formally, however, there is no difference between the two notations. The latter notation is convenient because it allows the atomic formula $A(x)$ to preserve its syntactic identify after a predicate substitution is applied to it. Likewise, we shall write $A(x)[P_2/Q_3](y)[P_3/Q_4](z)$ instead of $A[P_2/Q_3][P_3/Q_4](x, y, z)$. In this way, the formulas $A(x)$ and $A(x)[P_2/Q_3](y)$ both preserve their syntactic identities. The next definition makes this terminology precise.

Definition 2.2 (Predicates with Substitution)

- *Every atomic formula is a predicate with substitution.*
- *Suppose P and Q are predicate symbols of arity m and $m + k$, respectively, where $k \geq 0$. If α is a predicate with substitution, then so is $\alpha[P/Q](y_1...y_k)$, for any function terms $y_1...y_k$.*

The expression $[P/Q](y_1...y_k)$ 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* substitutions. In the substitution $[P/Q](y_1...y_k)$, 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. 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 function 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 $[\theta]$ denotes a predicate substitution and σ denotes a term substitution.

Definition 2.3 (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, and $k \geq 0$. If β is an atomic formula, then the rule is called a* Horn rule with substitution. *A* (Horn) rulebase with substitution *is a set of (Horn) rules with substitution.*

In this paper, the terms "Horn rule" and "rule without substitution" shall refer to classical Horn rules without substitution, *i.e.*, to rules of the form $\alpha_0 \leftarrow \alpha_1...\alpha_k$ where each α_i is an atomic formula.

We shall use rules *with* substitution to specify sets of rules *without* substitution. In the process, we take a database perspective. That is, we consider a rulebased system to consist of two parts: a set of rules, called the *rulebase*; and a set of ground atomic formulas, called the *database*. This dichotomy leads naturally to two sorts of predicates: *base predicates*, whose extention is given explicitly in the database; and *derived predicates*, whose extention is implied by the database together with the rulebase. Only derived predicates may appear in the heads of rules, and such rules are said to *define* the predicate. For example, the rule $A(x) \leftarrow B(x, y)$ defines the predicate A. The set of rules defining a predicate is called its *definition*, or its *intension*. These ideas and terminology are common in the deductive database literature.

It is common to define a database to be a finite set of ground atomic formulas constructed from constant symbols and base predicate symbols. However, none of the semantical results in this paper uses this assumption. Only the complexity results assume the database is finite. Thus, to keep the development general, we define a database to be *any* set of ground atomic formulas. A database may thus be finite or infinite, and may be constructed from base or derived predicate symbols, and constant or function symbols. This means that the extention of a base predicate is specified entirely by the database. However, the extention of a derived predicate may be specified partially by the rulebase and partially by the database. Only the rules are affected by predicate substitution.

2.2 The Copy-and-Substitute Mechanism

We treat predicates with substitution as derived predicates. Thus, the expression $A(x)[P/Q]$ is a derived predicate. However, the user is not allowed to provide rules defining these predicates. Instead, the rules defining $A(x)[P/Q]$ are generated *automatically* from the rules defining $A(x)$. Intuitively, the rules defining $A(x)[P/Q]$ are identical to the rules defining $A(x)$ except that P is replaced everywhere by Q. Thus, the user provides a set of *Horn* rules with substitution, from which a set of *non-Horn* rules with substitution is automatically generated.

To be more precise, we define the intension of $A(x)[P/Q]$ in terms of a copy-and-substitute mechanism. First, we copy all the rules in the rulebase, and in this copy, each predicate $B(x)$ is given a new name $B(x)[P/Q]$.[2] In this way, the original rulebase and its copy can co-exist as a single rulebase without interfering with each other. Second, each occurrence of $P(\overline{x})[P/Q]$ in the premises of these rules is replaced by $Q(\overline{x})$. In this way, substitution is carried out. Finally, for all base predicates $B(\overline{x})$, the predicate $B(\overline{x})[P/Q]$ is replaced by $B(\overline{x})$. Intuitively, a base predicate is not defined by rules, so it is unaffected by predicate substitutions. The result of these three steps is a rulebase that is identical to the original except that P has been replaced everywhere by Q. Also, instead of defining $A(x)$, the new rulebase defines $A(x)[P/Q]$, for every derived predicate A.

The rest of this section develops the copy-and-substitute mechanism in detail. Before defining it formally, we illustrate it through a short series of increasingly complex examples. The first two examples both illustrate basic substitution and full substitution.

Example 2.1 (A Simple Case) Let $\mathcal{S}$ be the following rulebase, where B and C are base predicates:

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

(i) If Q is a unary predicate symbol, then $A(x)[P/Q]$ is a unary predicate with substitution, defined by the following rules:

[2] We could equally well use the notation $B'(x)$ instead of $B(x)[P/Q]$. The latter notation simply provides a systematic way of providing new and distinct predicate names for each substitution.

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

(ii) Likewise, if R is a binary predicate symbol, then $A(x)[P/R](z)$ is a binary predicate with substitution, defined by the following rules:

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

□

Example 2.2 (Intermediate Predicates) Let $\mathcal{S}$ be the following rulebase, where C and D are base predicates:

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

(i) If Q is a unary predicate symbol, then $A(x)[P/Q]$ and $B(x)[P/Q]$ are unary predicates with substitution, defined by the following rules:

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

(ii) Likewise, if R is a binary predicate symbol, then $A(x)[P/R](z)$ and $B(x)[P/R](z)$ are binary predicates with substitution, defined by the following rules:

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

□

The idea of copying rules, renaming predicates and making substitutions applies to Horn rulebases *with* substitution as well as to those without. Furthermore, the idea can be applied recursively; that is, once predicates such as $A(x)[P/Q]$ have been defined, they can be used to define more complex predicates such as $A(x)[P/Q][R/S]$. These two ideas are illustrated in the next example.

Example 2.3 (Substitution in Rule Bodies) Let $\mathcal{S}$ be the following rulebase, where Q and R are base predicates:

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

Applying the substitutions $[R/S]$ and $[P/Q]$ to these rules, we get the following rules:

$$A(x)[R/S] \leftarrow B(x)[P/Q][R/S]$$
$$B(x)[P/Q] \leftarrow Q(x), R(x).$$
$$B(x)[P/Q][R/S] \leftarrow Q(x), S(x).$$

Since these rules are treated classically, we can infer the following two rules:

$$A(x) \leftarrow Q(x), R(x).$$
$$A(x)[R/S] \leftarrow Q(x), S(x).$$

□

The examples above illustrate how predicate substitution works. The following definition makes the idea precise.

Definition 2.4 (Predicate Substitution) *Let $\mathcal{S}$ be a Horn rulebase with substitution. Then the closure of $\mathcal{S}$ under predicate substitution, denoted $\mathcal{S}^\dagger$, is the smallest set of rules with substitution satisfying the following conditions:*

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

Here, $\overline{x}$ and $\overline{y}$ are lists of distinct variables.

The four items in this definition are discussed in the proof of Lemma 2.13.

The set of rules $\mathcal{S}^\dagger$ is infinite and contains an infinite number of *syntactically* distinct predicate symbols. These predicates may be *semantically* distinct as well. As the next example shows, this is possible because substitution is *not* idempotent; that is, in general, $A(x)[B/C][B/C]$ is not the same as $A(x)[B/C]$. Thus, from a finite set of predicate symbols, predicate substitution can create new and semantically distinct predicates whose names have arbitrary length, such as $A(x)[B/C][B/C][B/C]$.

Example 2.4 (Infinite Rule Sets) Suppose $\mathcal{S}$ consists of the following two rules, where B and P are base predicates:

$$\begin{aligned} A(x) &\leftarrow P(x) \\ Q(x) &\leftarrow B(x, x'), P(x') \end{aligned}$$

We use Definition 2.4 to generate rules for three predicates with substitution, $A(x)[P/Q]$, $A(x)[P/Q][P/Q]$ and $A(x)[P/Q][P/Q][P/Q]$. Using classical inference, we expand these rules, to derive three new rules:

$$\begin{aligned} A(x)[P/Q] &\leftarrow Q(x) \\ A(x)[P/Q][P/Q] &\leftarrow B(x, x'), Q(x') \\ A(x)[P/Q][P/Q][P/Q] &\leftarrow B(x, x'), B(x', x''), Q(x'') \end{aligned}$$

These rules are not in $\mathcal{S}^\dagger$, but are logically entailed by it. They show that the three predicates with substitution are not equivalent. More generally, the rules—both implicit and explicit—for $A(x)[P/Q]^i$ are different from the rules for $A(x)[P/Q]^{i+1}$. We therefore have an infinite number of semantically distinct predicates with substitution. □

Because Example 2.4 uses basic substitution, each new predicate has the same arity as the original predicate. When full substitution is used, new predicates can have increasing arity, as in $A(x_1)[P/R](x_2)[P/R](x_3)[P/R](x_4)\ldots$. In this way, predicates of unbounded arity can be generated.

2.3 Inference from Databases

The copy-and-substitute mechanism developed above takes a set of Horn rules with substitution, $\mathcal{S}$, and generates a larger set of rules with substitution, $\mathcal{S}^\dagger$. We treat this set classically, *i.e.*, as a set of classical Horn rules. To be useful, classical Horn rules must be augmented with a database, from which the rules can derive atomic formmulas. Definition 2.5 below formalizes this idea. In this definition, and in the rest of the paper, the symbol $\vdash_c$ denotes inference in classical first-order logic, and all free variables are universally quantified at the top level. As defined in Section 2.1, a database is any set of ground atomic formulas.

Definition 2.5 (Inference) *Suppose β is a ground predicate with substitution, $\mathcal{S}$ is a Horn rulebase with substitution, and DB is a database. Then,*

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

For example, if $DB = \{Q(b), R(b), Q(a), S(a), P(b)\}$ then using rulebase $\mathcal{S}$ from Example 2.3 leads to the following inferences (among others):

$$\begin{array}{ll} \mathcal{S}, DB \vdash B(b)[P/Q] & \qquad \mathcal{S}, DB \vdash A(b) \\ \mathcal{S}, DB \vdash A(a)[P/Q] & \qquad \mathcal{S}, DB \vdash B(b) \end{array}$$

Because $\mathcal{S}^\dagger$ is infinite, its value is largely conceptual, providing a concise definition of inference with substitution. However, to actually perform inference, it is impractical to first generate $\mathcal{S}^\dagger$ and then invoke classical inference procedures. The lemma below offers a more practical alternative by providing a finite axiomatization which is sound and complete. The inference system in the lemma can be operated in either a top-down or a bottom-up manner. When operated top-down, it attempts only those substitutions that are relevant to proving its current goal.

Lemma 2.6 (An Axiomatization) *Let β be a ground 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: *For all predicates with substitution $\beta_0 ... \beta_n$,*

1. *For any predicate substitution $[\theta]$,*
$$\frac{\beta_0 \leftarrow \beta_1 ... \beta_n}{\beta_0[\theta] \leftarrow \beta_1[\theta] ... \beta_n[\theta]}$$
2. *For any term substitution σ,*
$$\frac{\beta_0 \leftarrow \beta_1 ... \beta_n}{(\beta_0 \leftarrow \beta_1 ... \beta_n)\sigma}$$
3. *Modus Ponens:*
$$\frac{\begin{array}{l}\beta_1 \; ... \; \beta_n \\ \beta_0 \leftarrow \beta_1 ... \beta_n\end{array}}{\beta_0}$$

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 $\mathcal{S}^\dagger$. 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.

2.4 Basic Properties of Inference

This section develops some of the basic formal properties of the rule set $\mathcal{S}^\dagger$ defined in Section 2.2. These properties are central to the proofs of soundness and completeness given in Section 3. We first prove a compactness result for entailment with substitution. The remaining development is then divided into two subsections, each leading to a central result. In this section, as in the rest of this paper, the symbol $\vdash_c$ denotes inference in classical first-order logic.

To prove our first result, below, recall that Definition 2.4 provides four items that contribute rules to $\mathcal{S}^\dagger$. Using this definition, each rule in $\mathcal{S}^\dagger$ can be generated as follows: First pick a rule from items 1, 3 or 4; then apply item 2 to it some number of times (possibly zero times). It follows that every rule in $\mathcal{S}^\dagger$ originates either as a rule in $\mathcal{S}$ or as a rule described by item 3 or 4. From this observation, we get the following lemma, which is a prelude to compactness.

Lemma 2.7 *Let $\mathcal{S}$ be a (possibly infinite) set of rules with predicate substitution. If $\rho^1 ... \rho^n$ are rules in $\mathcal{S}^\dagger$, then there is a finite subset $\mathcal{S}_o$ of $\mathcal{S}$ of cardinality at most n such that $\rho^1 ... \rho^n$ are all in $\mathcal{S}_o^\dagger$.*

Proof: Each rule ρ^i originates as a single rule ρ_o^i from item 1, 3 or 4 of Definition 2.4, as observed above. Let $\mathcal{S}_o = \{\rho_o^1, ..., \rho_o^n\} \cap \mathcal{S}$. Then $\mathcal{S}_o$ is a finite subset of $\mathcal{S}$ of cardinality at most n, as required. Furthermore, we claim that each ρ^i is in $\mathcal{S}_o^\dagger$. To see this, consider two cases. If ρ^i originates from item 3 or 4, then $\rho \in \mathcal{R}^\dagger$ for every rulebase $\mathcal{R}$. In particular $\rho^i \in \mathcal{S}_o^\dagger$. On the other hand, if ρ^i originates from item 1, then $\rho_0^i \in \mathcal{S}$, so $\rho_0^i \in \mathcal{S}_0$, hence $\rho^i \in \mathcal{S}_o^\dagger$. □

Corollary 2.8 (Compactness) *Suppose that* $\mathcal{S}^\dagger \vdash_c \rho$ *where* ρ *is a Horn rule with substitution, and* $\mathcal{S}$ *is a (possibly infinite) rulebase with substitution. Then* $\mathcal{S}_o^\dagger \vdash_c \rho$ *for some finite subset* $\mathcal{S}_o$ *of* $\mathcal{S}$.

Proof: Suppose that $\mathcal{S}^\dagger \vdash_c \rho$. Then, by the compactness of classical entailment, there is a finite subset, $\mathcal{R}$, of $\mathcal{S}^\dagger$ such that $\mathcal{R} \vdash_c \rho$. By Lemma 2.7, there is a finite subset $\mathcal{S}_o$ of $\mathcal{S}$ such that $\mathcal{R} \subseteq \mathcal{S}_o^\dagger$. Therefore $\mathcal{S}_o^\dagger \vdash_c \mathcal{R}$ and $\mathcal{R} \vdash_c \rho$, and so $\mathcal{S}_o^\dagger \vdash_c \rho$. $\Box$

Properties I:

This section shows that if $\mathcal{S}_1 \vdash_c \mathcal{S}_2$ then $\mathcal{S}_1^\dagger \vdash_c \mathcal{S}_2^\dagger$, where predicates with substitution are treated as ordinary predicates of classical first-order logic. This result is used throughout this paper and is crucial to the proof of completeness in Section 3.3.

Lemma 2.9 *If* $\mathcal{S}^\dagger \vdash_c \rho$ *then* $\mathcal{S}^\dagger \vdash_c \rho[\theta]$, *for any rule with substitution* ρ *and any set* $\mathcal{S}$ *of Horn rules with substitution.*

Proof: Suppose that $\mathcal{S}^\dagger \vdash_c \rho$. Then there is a proof of ρ from some finite subset, $\{\rho_1, ..., \rho_m\}$, of rules in $\mathcal{S}^\dagger$. By replacing each occurrence of ρ_i in this proof by $\rho_i[\theta]$, we obtain a proof of $\rho[\theta]$ from $\{\rho_1[\theta], ..., \rho_m[\theta]\}$. But each $\rho_i[\theta]$ is in $\mathcal{S}^\dagger$. Hence $\mathcal{S}^\dagger \vdash_c \rho[\theta]$. $\Box$

Example 2.5 Suppose that $\mathcal{S}^\dagger$ contains the following two rules:

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

Then $\mathcal{S}^\dagger$ also contains the following two rules, for any P, Q and z:

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

By resolving the first pair of rules above, we get the following rule, which is entailed by $\mathcal{S}^\dagger$:

$$A(x) \leftarrow C(x,y), D(y). \tag{1}$$

Likewise, by resolving the second pair of rules above, we get the following rule, which is also entailed by $\mathcal{S}^\dagger$:

$$A(x)[P/Q](z) \leftarrow C(x,y)[P/Q](z),\ D(y)[P/Q](z). \tag{2}$$

In the notation of Lemma 2.9, rule (1) is ρ, and rule (2) is $\rho[\theta]$, where $[\theta]$ is $[P/Q](z)$. $\Box$

Lemma 2.10 *If* $\mathcal{S}_1^\dagger \vdash_c \mathcal{S}_2$ *then* $\mathcal{S}_1^\dagger \vdash_c \mathcal{S}_2^\dagger$ *where* $\mathcal{S}_1$ *and* $\mathcal{S}_2$ *are sets of Horn rules with substitution.*

Proof: Treating $\mathcal{S}_1^\dagger$ as a classical first-order rulebase, let $cl(\mathcal{S}_1^\dagger)$ be the set of Horn rules classically entailed by $\mathcal{S}_1^\dagger$. We call $cl(\mathcal{S}_1^\dagger)$ the "Horn closure" of $\mathcal{S}_1^\dagger$. This closure has the following properties:

1. $\mathcal{S}_2 \subseteq cl(\mathcal{S}_1^\dagger)$.
2. If $\beta \leftarrow \beta_1 ... \beta_n$ is in $cl(\mathcal{S}_1^\dagger)$, then $\beta[\theta] \leftarrow \beta_1[\theta] ... \beta_n[\theta]$ is in $cl(\mathcal{S}_1^\dagger)$ for every predicate substitution $[\theta]$.
3. The rule $P(\overline{x})[P/Q](\overline{y}) \leftarrow Q(\overline{x}, \overline{y})$ is in $cl(\mathcal{S}_1^\dagger)$ for every substitution $[P/Q](\overline{y})$.
4. The rule $B(\overline{x})[P/Q](\overline{y}) \leftarrow B(\overline{x}, \overline{y})$ is in $cl(\mathcal{S}_1^\dagger)$ for every substitution $[P/Q](\overline{y})$ and every base predicate $B \neq P$.

The first item follows since $\mathcal{S}_1^\dagger \vdash_c \mathcal{S}_2$, by hypothesis; the second item follows from Lemma 2.9; and the last two items follow from the definition of $\mathcal{S}_1^\dagger$. The set $cl(\mathcal{S}_1^\dagger)$ thus satisfies each of the four conditions in the definition of $\mathcal{S}_2^\dagger$. By definition, however, $\mathcal{S}_2^\dagger$ is the smallest set satisfying these conditions. Thus $\mathcal{S}_2^\dagger \subseteq cl(\mathcal{S}_1^\dagger)$. Therefore, each rule in $\mathcal{S}_2^\dagger$ is classically entailed by $\mathcal{S}_1^\dagger$. Hence $\mathcal{S}_1^\dagger \vdash_c \mathcal{S}_2^\dagger$. □

Corollary 2.11 *If* $\mathcal{S}_1 \vdash_c \mathcal{S}_2$ *then* $\mathcal{S}_1^\dagger \vdash_c \mathcal{S}_2^\dagger$ *where* $\mathcal{S}_1$ *and* $\mathcal{S}_2$ *are Horn rulebases with substitution*

Proof: Since $\mathcal{S}_1 \subseteq \mathcal{S}_1^\dagger$, then $\mathcal{S}_1^\dagger \vdash_c \mathcal{S}_1$. But $\mathcal{S}_1 \vdash_c \mathcal{S}_2$, by hypothesis. So $\mathcal{S}_1^\dagger \vdash_c \mathcal{S}_2$. Hence $\mathcal{S}_1^\dagger \vdash_c \mathcal{S}_2^\dagger$, by Lemma 2.10. □

It follows immediately that if $\mathcal{S}_1$ and $\mathcal{S}_2$ are classically equivalent, then so are $\mathcal{S}_1^\dagger$ and $\mathcal{S}_2^\dagger$. Hence, rulebases with substitution that are classically equivalent express exactly the same database queries. We thus have the following result:

Corollary 2.12 *If* $\mathcal{S}_1$ *is classically equivalent to* $\mathcal{S}_2$*, then for any database DB and any predicate with substitution* β*,*

$$\mathcal{S}_1, DB \vdash \beta \quad \textit{iff} \quad \mathcal{S}_2, DB \vdash \beta$$

Properties II:

This section elucidates the relationship between a predicate with substitution $\beta[\theta]$ and the simpler predicate with substitution β. In particular, it is shown that reasoning about $\beta[\theta]$ can be broken into two phases: (i) ignoring $[\theta]$ and reasoning just about β, and (ii) ignoring β and reasoning just about $[\theta]$. The precise statement of this result (Lemma 2.13) is crucial to the proof of soundness given in Section 3.3.

Lemma 2.13 *Let* $\mathcal{S}$ *be a set of Horn rules with substitution, and let DB be a database. If* $\beta[\theta]$ *is a ground predicate with substitution, then* $\mathcal{S}, DB \vdash \beta[\theta]$ *iff the following two conditions are satisfied for some (possibly empty) set of ground atomic formulas,* $\alpha_1 \ldots \alpha_n$*:*

$$\mathcal{S}^\dagger \quad \vdash_c \quad \beta \leftarrow \alpha_1 \ldots \alpha_n \tag{3}$$

$$\mathcal{S}, DB \quad \vdash \quad \alpha_j[\theta] \quad \textit{for all } 1 \leq j \leq n \tag{4}$$

Proof: The *if* direction is straightforward. First observe the following:

$$\begin{array}{rlll} \mathcal{S}^\dagger & \vdash_c & \beta[\theta] \leftarrow \alpha_1[\theta] \ldots \alpha_n[\theta] & \text{by statement (3) and Lemma 2.9,} \\ \mathcal{S}^\dagger \cup DB & \vdash_c & \alpha_j[\theta] \quad \textit{for } 1 \leq j \leq n & \text{by statement (4) and Definition 2.5.} \end{array}$$

Hence, $\mathcal{S}^\dagger \cup DB \ \vdash_c \ \beta[\theta]$ by classical modus ponens, and therefore $\mathcal{S}, DB \vdash \beta[\theta]$ by Definition 2.5.

To prove the *only if* direction, we need the following observations:

1. In Definition 2.4, there are four items that contribute rules to $\mathcal{S}^\dagger$. The first item contributes rules whose heads are atomic (since each rule in $\mathcal{S}$ has an atomic head). The third and fourth items contribute rules whose heads contain exactly one predicate substitution. Only the second item contributes rules whose heads have more than one substitution, and these rules have the form $\beta[\theta] \leftarrow \beta_1[\theta]...\beta_m[\theta]$, where $\beta \leftarrow \beta_1...\beta_m$ is also a rule in $\mathcal{S}^\dagger$.

2. A rule with substitution may have an empty premise. Using logic-programming parlance, we shall call such a rule a *unit rule*. A unit rule is simply a predicate with substitution. Now suppose that $\mathcal{S}^\dagger$ contains the predicate with substitution $\beta[\theta]$. This can only be due to item 2 of Definition 2.4; so $\mathcal{S}^\dagger$ must also contain β. Extending this idea a bit, if $\beta[\theta]$ is a ground instance of a unit rule in $\mathcal{S}^\dagger$, then so is β.

3. In the fixpoint semantics of logic programming [1, 18], each Horn rulebase, R, has an associated operator, T_R. This operator takes a set of ground atoms, I, as input and returns another set of ground atoms, $T_R(I)$, as output. Starting at the empty set, and applying this operator over-and-over again, one can generate all the ground atoms that are classically entailed by R. That is, $R \vdash_c A$ iff $A \in T_R^k(\{\})$ for some finite k. This property remains true even if R is infinite. This is so because infinite rulebases do not change the essential properties of the operator T_R: it remains monotonic and continuous. The Tarski Fixpoint Theorem [16], which is at the heart of the fixpoint semantics, can thus be applied in the usual way [1, 18].[3]

Keeping these observations in mind, we proceed to prove the *only if* direction of the lemma. The main idea is to treat $\mathcal{S}^\dagger \cup DB$ as a classical Horn rulebase and to exploit its associated operator, $T_{\mathcal{S}^\dagger \cup DB}$. Suppose, then, that $\mathcal{S}, DB \vdash \beta[\theta]$. Then $\mathcal{S}^\dagger \cup DB \vdash_c \beta[\theta]$, by definition; so $\beta[\theta] \in T^k_{\mathcal{S}^\dagger \cup DB}(\{\})$ for some some $k \geq 0$. We show that the lemma is true for all $\beta[\theta] \in T^k_{\mathcal{S}^\dagger \cup DB}(\{\})$. The proof is by induction on k. For brevity, we write T^k instead of $T^k_{\mathcal{S}^\dagger \cup DB}(\{\})$.

Basis: Suppose that $\beta[\theta] \in T^0$. Then $\beta[\theta]$ is a ground instance of a unit rule in $\mathcal{S}^\dagger \cup DB$. This can happen in only two ways:

Case (i): $\beta[\theta] \in DB$. This case is impossible, since DB is a set of atoms whereas $\beta[\theta]$ is not atomic, since it contains the predicate substitution $[\theta]$.

[3] Indeed, the Tarski Fixpoint Theorem is only needed for handling infinities. If everything is finite, as with classical Datalog, then much simpler developments exist.

Case (ii): $\beta[\theta]$ is a ground instance of a unit rule in $\mathcal{S}^\dagger$. In this case, β is also a ground instance of a unit rule in $\mathcal{S}^\dagger$, by observation 2 above. Thus $\mathcal{S}^\dagger \vdash_c \beta$ trivially. Hence condition (3) of the lemma is satisfied with with $n = 0$. Consequently, condition (4) is trivially satisfied, since the set $\{\alpha_1, ..., \alpha_n\}$ is empty.

Induction: Suppose that $\beta[\theta] \in T^{k+1}$. We must show that conditions (3) and (4) of the lemma are both satisfied. We consider two cases, depending on whether or not β is atomic:

Case (i): If β is atomic, then we are done, since

$$\begin{array}{rcl} \mathcal{S}^\dagger & \vdash_c & \beta \leftarrow \beta \\ \textit{and} \quad \mathcal{S}, DB & \vdash & \beta[\theta] \end{array}$$

The first line has the form of condition (3), and the second line has the form of condition (4).

Case (ii): If β is non-atomic, then $\beta[\theta]$ contains more than one predicate substitution. Therefore, by observation 1 above, every rule in $\mathcal{S}^\dagger$ that infers $\beta[\theta]$ is due to item 2 of Definition 2.4.

Since $\beta[\theta] \in T^{k+1}$, there must be a rule in $\mathcal{S}^\dagger$ that derives $\beta[\theta]$ from T^k in one step. By item 2 of Definition 2.4, this rule has a ground instance of the form $\beta[\theta] \leftarrow \beta_1[\theta]...\beta_m[\theta]$, for some $\beta_1...\beta_m$, where $\beta \leftarrow \beta_1...\beta_m$ is also ground instance of a rule in $\mathcal{S}^\dagger$. Furthermore, $\beta_i[\theta] \in T^k$ for each i. Thus, by induction hypothesis, the lemma holds for each $\beta_i[\theta]$. There is therefore a set of ground atomic formulas, $\alpha^i_1...\alpha^i_{n_i}$, that makes both the following statements true:

$$\mathcal{S}^\dagger \quad \vdash_c \quad \beta_i \leftarrow \alpha^i_1 \ldots \alpha^i_{n_i} \tag{5}$$

$$\mathcal{S}, DB \quad \vdash \quad \alpha^i_j[\theta] \quad \textit{for all } 1 \le j \le n_i \tag{6}$$

Moreover, since $\beta \leftarrow \beta_1...\beta_m$ is a ground instance of a rule in $\mathcal{S}^\dagger$, we have $\mathcal{S}^\dagger \vdash_c \beta \leftarrow \beta_1...\beta_m$. Combining this with statement (5) gives the following:

$$\mathcal{S}^\dagger \quad \vdash_c \quad \beta \leftarrow \alpha^1_1...\alpha^1_{n_1}\ \alpha^2_1...\alpha^2_{n_2}\ ...\ \alpha^m_1...\alpha^m_{n_m}$$

This statement has the form of condition (3). In addition, statement (6) has the form of condition (4). Thus, conditions (3) and (4) are both satisfied.

□

3 Model Theory

Section 2 developed the proof theory for Horn rulebases with predicate substitution. This Section develops the model theory. In this theory, rulebases *without* substitution are the models for rulebases *with* substitution. This corresponds to the intuition that rules with substitution are written by a user who has rules without substitution "in mind." Section 3.1 develops the basic model-theoretic notions, Section 3.2 establishes the basic properties of models, and Section 3.3 shows that the proof theory developed in Section 2 is sound and complete for the model theory developed in this section.

3.1 Satisfaction, Models and Entailment

This section develops the basic model-theoretic notions of satisfaction, model and entailment for predicate substitution. The definitions are general in that they apply to *arbitrary* rules with substitution, not just to *Horn* rules with substitution. This generality is an important element in our proof of soundness, since we will need to say what it means to be a model of $\mathcal{S}^\dagger$, not just of $\mathcal{S}$. In addition, we shall give two equivalent definitions of "model". One definition is used to prove soundness of the proof theory, and the other is used to prove completeness (in Sections 3.2 and 3.3).

In our model theory, an interpretation is a classical Horn rulebase. These interpretations can be compared to modal structures [6]. This is possible because a classical rulebase can be thought of semantically as a set of classical models, instead of syntactically as a set of rules. In modal logic, an interpretation has a set of states. The expression $M, s \models \phi$ means that formula ϕ is true at state s of interpretation M. M is said to be a model of ϕ if ϕ is true at every state of M. In our logic, an interpretation is a classical Horn rulebase, which has a set of classical Herbrand models. These models contain all the information about the rulebase, and are analogous to the states of a modal structure. Moreover, like modal states, a classical Herbrand model is essentially a set of ground atomic formulas. We shall use databases to specify these Herbrand models. Given a classical Horn rulebase R, a database DB specifies the minimal Herbrand model of $R + DB$. Following modal terminology, we shall speak of formulas being true at a database state. To extend these ideas to predicates with substitution, we look at the classical Herbrand models of $R^\dagger$, the closure of R. The definitions below make these ideas precise.

Definition 3.1 (Interpretations) *An interpretation is any set of classical Horn rules.*

Definition 3.2 (Satisfaction) *Let R be an interpretation, let DB be a database, and let β and β_i be ground predicates with substitution. Then,*

- $R, DB \models \beta$ *iff* β *is in the minimal Herbrand model of* $R^\dagger \cup DB$,
  (i.e., iff $R^\dagger \cup DB \models_c \beta$, *where* $\models_c$ *denotes entailment in classical first-order logic.)*

- $R, DB \models \beta \leftarrow \beta_1 ... \beta_n$ *iff the following holds:*

$$R, DB \models \beta \quad \text{if} \quad R, DB \models \beta_1 \ \cdots \text{and} \cdots \ R, DB \models \beta_n$$

If $R, DB \models \phi$ then we say that R satisfies ϕ at DB.

Definition 3.3 (Models) *If R is an interpretation, then*

- *R is a model of a ground rule with substitution iff R satisfies the rule at every database;*

- *R is a model of a non-ground rule with substitution iff R is a model of every ground instantiation of the rule;*

- *R is a model of a* set *of rules with substitution iff R is a model of each rule in the set.*

If R is a model of ϕ, then we write $R \models \phi$.

Definition 3.4 (Entailment) *Let $\mathcal{S}$ be a set of Horn rules with predicate substitution, let DB be a database, and let β be a ground predicate with substitution. Then $\mathcal{S}$ entails β at DB, written* $\mathcal{S}, DB \models \beta$, *if and only if* $R, DB \models \beta$ *for every model R of $\mathcal{S}$.*

Example 3.1 Suppose R is the following interpretation, where E, F and P are base predicates.

$$\begin{aligned} &B(x) \leftarrow E(x), P(x).\\ &A(x) \leftarrow E(x), Q(x).\\ &A(x) \leftarrow Q(x), F(x,y), Q(y).\end{aligned}$$

Then $B(x)[P/Q]$ is defined by the following rule:

$$B(x)[P/Q] \leftarrow E(x), Q(x).$$

The body of this rule is identical to the body of one of the rules defining $A(x)$. Thus, any database that makes $B(x)[P/Q]$ true also makes $E(x)$ and $Q(x)$ true; so, it makes $A(x)$ true. R is therefore a model of the rule $A(x) \leftarrow B(x)[P/Q]$. □

An Equivalent Definition:

Section 3.3 uses Definition 3.3 to prove that the proof theory of Section 2 is sound. To prove completeness, however, it is convenient to use a different characterization of models. This characterization is based on Example 3.1 above. The gist of this example is the following observation:

> *Suppose that the following holds for any finite set of atomic formulas, $\alpha_1 ... \alpha_n$, and any function term, x,*
>
> $$\textit{if} \quad R^\dagger \models_c B(x)[P/Q] \leftarrow \alpha_1 ... \alpha_n \quad \textit{then} \quad R^\dagger \models_c A(x) \leftarrow \alpha_1 ... \alpha_n \qquad (7)$$
>
> *Then any database that makes $B(x)[P/Q]$ true will also make $A(x)$ true. Hence, R is a model of* $A(x) \leftarrow B(x)[P/Q]$.

The following theorem generalizes this observation, and replaces the *if* by *if and only if*. It is also the basis of the fixpoint semantics developed in Section 4.

Theorem 3.5 *Let R be an interpretation, and let $\beta \leftarrow \beta_1 ... \beta_m$ be a rule with predicate substitution. Then R is a model of the rule iff for all finite sets of atomic formulas, γ, and all term substitutions σ,*

$$R^\dagger \models_c \beta\sigma \leftarrow \gamma \qquad \textit{if} \qquad R^\dagger \models_c \beta_i\sigma \leftarrow \gamma \quad \textit{for } 1 \le i \le m \qquad (8)$$

This theorem is proved by the following two lemmas.

Lemma 3.6 *Let R be an interpretation. If R satisfies condition (8) in Theorem 3.5, then R is a model of* $\beta \leftarrow \beta_1 ... \beta_m$.

Proof: Let DB be a database, and let σ be a term substitution that makes $\beta \leftarrow \beta_1 ... \beta_m$ ground. By Definitions 3.2 and 3.3, we must show that if $R, DB \models \beta_i\sigma$ for $1 \leq i \leq m$, then $R, DB \models \beta\sigma$.

if	$R, DB \models \beta_i\sigma$	for $1 \leq i \leq m$,
then	$R^\dagger \cup DB \models_c \beta_i\sigma$	by Definition 3.2,
	$R^\dagger \cup \gamma_i \models_c \beta_i\sigma$	for some finite subset γ_i of DB, by compactness,
	$R^\dagger \cup \gamma \models_c \beta_i\sigma$	where $\gamma = \gamma_1 \cup ... \cup \gamma_m$ (note: $\gamma \subseteq DB$),
	$R^\dagger \models_c \beta_i\sigma \leftarrow \gamma$	by the Deduction Theorem,
	$R^\dagger \models_c \beta\sigma \leftarrow \gamma$	by statement (8),
	$R^\dagger \cup \gamma \models_c \beta\sigma$	by the Deduction Theorem,
	$R^\dagger \cup DB \models_c \beta\sigma$	since $\gamma \subseteq DB$,
	$R, DB \models \beta\sigma$	by Definition 3.2.

□

Lemma 3.6 proved the *if* direction of Theorem 3.5. The next lemma proves the *only if* direction. A key step in the proof is a generalization from ground formulas to formulas with variables. This is possible because in a first-order query, universally quantified variables can be replaced by skolem constants, i.e., constant symbols that do not appear elsewhere in the query or in the rulebase. Formally, for any two first-order formulas R (the rulebase) and ϕ (the query), classical logic has the following property:

$$\begin{array}{ll} & R \vdash_c \forall_x \phi \\ \textit{iff} & R \wedge \neg\forall_x \phi \ \text{ is unsatisfiable,} \\ \textit{iff} & R \wedge \exists_x \neg\phi \ \text{ is unsatisfiable,} \\ \textit{iff} & R \wedge \neg\phi\{x/c\} \ \text{ is unsatisfiable, where } c \text{ is a skolem constant,} \\ \textit{iff} & R \vdash_c \phi\{x/c\} \ \text{ where } c \text{ does not appear in } R \text{ or } \phi. \end{array} \tag{9}$$

Note that c is a skolem *constant*, not a skolem function, since both R and $\forall_x \phi$ are closed. (As in the rest of this paper, any apparently free variables are actually universally quantified.) Intuitively, since c does not appear in R or ϕ, it is not given special treatment by any of the rules in R. The constant c thus represents an arbitrary value for x.

Lemma 3.7 *If R is a model of $\beta \leftarrow \beta_1 ... \beta_m$, then R satisfies condition (8) in Theorem 3.5.*

Proof: Let γ be a finite set of atomic formulas, and let σ be a term substitution. We must show that if $R^\dagger \models_c \beta_i\sigma \leftarrow \gamma$ for $1 \leq i \leq m$, then $R^\dagger \models_c \beta\sigma \leftarrow \gamma$. To do this, let $\overline{x}$ be a list of all the variables in γ, $\beta\sigma$, and each $\beta_i\sigma$; and let $\overline{c}$ be a list (of the same length) of distinct skolem constants, that is, constants that do not appear in R, γ, $\beta\sigma$, or any $\beta_i\sigma$. Note that the term substitution $\{\overline{x}/\overline{c}\}$ makes each of these formulas (except R) ground. Keeping this in mind, we have the following:

$$
\begin{array}{lll}
\textit{if} & R^\dagger \models_c \beta_i\sigma \leftarrow \gamma & \text{for } 1 \leq i \leq m, \\
\textit{then} & R^\dagger \models_c \forall\overline{x}(\beta_i\sigma \leftarrow \gamma) & \text{since the variables in } \overline{x} \text{ are universally quantified,} \\
& R^\dagger \models_c (\beta_i\sigma \leftarrow \gamma)\{\overline{x}/\overline{c}\} & \text{by observation (9),} \\
& R^\dagger \models_c \beta_i\sigma\{\overline{x}/\overline{c}\} \leftarrow \gamma\{\overline{x}/\overline{c}\} & \\
& R^\dagger \cup \gamma\{\overline{x}/\overline{c}\} \models_c \beta_i\sigma\{\overline{x}/\overline{c}\} & \text{by the Deduction Theorem,} \\
& R, \gamma\{\overline{x}/\overline{c}\} \models \beta_i\sigma\{\overline{x}/\overline{c}\} & \text{by Definition 3.2, treating } \gamma\{\overline{x}/\overline{c}\} \text{ as a database,} \\
& R, \gamma\{\overline{x}/\overline{c}\} \models \beta\sigma\{\overline{x}/\overline{c}\} & \text{by Definition 3.3, since } R \models \beta \leftarrow \beta_1 ... \beta_m, \\
& R^\dagger \cup \gamma\{\overline{x}/\overline{c}\} \models_c \beta\sigma\{\overline{x}/\overline{c}\} & \text{by Definition 3.2,} \\
& R^\dagger \models_c \beta\sigma\{\overline{x}/\overline{c}\} \leftarrow \gamma\{\overline{x}/\overline{c}\} & \text{by the Deduction Theorem,} \\
& R^\dagger \models_c (\beta\sigma \leftarrow \gamma)\{\overline{x}/\overline{c}\} & \\
& R^\dagger \models_c \forall\overline{x}(\beta\sigma \leftarrow \gamma) & \text{by observation (9),} \\
& R^\dagger \models_c \beta\sigma \leftarrow \gamma & \text{where the variables in } \overline{x} \text{ are universally quantified.}
\end{array}
$$

□

Example 3.2 (Models) In the three examples below, E, F and P are base predicates.

Intermediate Predicates: The interpretation below is a model of $A(x) \leftarrow B(x)[P/Q]$.

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

Recursion: The interpretation below is a model of $A(x) \leftarrow A(x)[P/Q]$. A recursive rule can thus have a non-recursive model.

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

Function Symbols: Interpretation R_1 below is a model of $A(z) \leftarrow B(z)[P/Q]$, but interpretation R_2 is not. Here, f is a function symbol.

$$
R_1 = \left\{ \begin{array}{l} B(f(x)) \leftarrow E(f(x),y), P(y). \\ A(x) \leftarrow E(x,y), Q(y) \end{array} \right\} \qquad R_2 = \left\{ \begin{array}{l} B(x) \leftarrow E(x,y), P(y). \\ A(f(x)) \leftarrow E(f(x),y), Q(y) \end{array} \right\}
$$

□

It is worth noting that satisfaction is not monotonic in R. As the next example shows, if $R_1 \subseteq R_2$, then it is not necessarily the case that if $R_1 \models \alpha \leftarrow \beta$ then $R_2 \models \alpha \leftarrow \beta$. This should not be surprising, since satisfaction is not generally monotonic in logical systems. For instance, in classical first-order logic, if s_1 and s_2 are Herbrand structures and $s_1 \subseteq s_2$, then it is not necessarily true that if $s_1 \models_c \psi$ then $s_2 \models_c \psi$ for an arbitrary first-order sentence ψ.

Example 3.3 (Non-Monotonicity of Satisfaction) Interpretation R_1 below is a model of the rule $A(x) \leftarrow B(x)[P/Q]$, but the larger interpretation R_2 is not.

$$R_1 = \left\{ \begin{array}{l} B(x) \leftarrow E(x), P(x). \\ A(x) \leftarrow E(x), Q(x). \end{array} \right\} \qquad R_2 = \left\{ \begin{array}{l} B(x) \leftarrow F(x,y), P(x), P(y). \\ B(x) \leftarrow E(x), P(x). \\ A(x) \leftarrow E(x), Q(x). \end{array} \right\}$$

□

3.2 Basic Properties

This section develops some of the basic formal properties of models and entailment. Lemma 3.8 is a straightforward consequence of either characterization of models—Definition 3.3 or Theorem 3.5. Lemma 3.9 is less straightforward and relies on Definition 3.3. Corollary 3.10 is an immediate consequence of Lemma 3.9 and is crucial to the proof of soundness in Section 3.3.

Lemma 3.8 *Let R be an interpretation, and β and β_i be ground predicates with substitution.*

1. *If* $R \models \beta \leftarrow \beta_1 ... \beta_m$ *and* $R, DB \models \beta_i$ *for each* i, *then* $R, DB \models \beta$.
2. *If* $R \models \beta_1 \leftarrow \beta_2$ *and* $R \models \beta_2 \leftarrow \beta_3$ *then* $R \models \beta_1 \leftarrow \beta_3$.
3. *If* $R \models \beta \leftarrow \beta_1 ... \beta_n$ *then* $R \models (\beta \leftarrow \beta_1 ... \beta_n)\sigma$ *for any term substitution* σ.
4. *If* $R^\dagger \models_c \beta \leftarrow \beta_1 ... \beta_n$ *then* $R \models \beta \leftarrow \beta_1 ... \beta_n$
5. $R \models R^\dagger$
6. $R \models R$

Lemma 3.9 *Let R be an interpretation.*

1. *If* $R \models \beta \leftarrow \beta_1 ... \beta_m$ *then* $R \models \beta[\theta] \leftarrow \beta_1[\theta] ... \beta_m[\theta]$, *for every predicate substitution* $[\theta]$.
2. $R \models P(\overline{x})[P/Q](\overline{y}) \leftarrow Q(\overline{x}, \overline{y})$ *for every predicate substitution* $[P/Q](\overline{y})$.
3. $R \models B(\overline{x})[P/Q](\overline{y}) \leftarrow B(\overline{x})$ *for every predicate substitution* $[P/Q](\overline{y})$ *and every base predicate symbol* $B \neq P$.

Proof: R is a model of $R^\dagger$, by Lemma 3.8. Thus R is a model of the rules $P(\overline{x})[P/Q](\overline{y}) \leftarrow Q(\overline{x}, \overline{y})$ and $B(\overline{x})[P/Q](\overline{y}) \leftarrow B(\overline{x})$, since they are in $R^\dagger$, by Definition 2.4. This proves items *2* and *3*. To prove item *1*, assume that R is a model of the rule $\beta \leftarrow \beta_1 \ldots \beta_m$. We first prove the ground case, *i.e.*, when β, β_i and $[\theta]$ are all ground. With this in mind, let DB be a database, and suppose that $R, DB \models \beta_i[\theta]$ for each $1 \leq i \leq m$.

We must show that $R, DB \models \beta[\theta]$. To prove this, we first apply Lemma 2.13 to $\beta_i[\theta]$ for each i. By this lemma, there is a set of ground atomic formulas, $\alpha^i_1 ... \alpha^i_{n_i}$, such that

$$R^\dagger \ \vdash_c \ \beta_i \leftarrow \alpha^i_1 ... \alpha^i_{n_i} \tag{10}$$
$$\textit{and} \quad R, DB \ \vdash \ \alpha^i_j[\theta] \quad \textit{for each } i \textit{ and } j \tag{11}$$

Therefore,

$R^\dagger \cup \{\alpha^i_1 ... \alpha^i_{n_i}\} \ \models_c \ \beta_i$	by (10) and the Deduction Theorem,
$R^\dagger \cup \{\alpha^1_1 ... \alpha^1_{n_1} \alpha^2_1 ... \alpha^2_{n_2} ... \alpha^1_m ... \alpha^m_{n_m}\} \ \models_c \ \beta_i$	by monotonicity,
$R, \{\alpha^1_1 ... \alpha^1_{n_1} \alpha^2_1 ... \alpha^2_{n_2} ... \alpha^1_m ... \alpha^m_{n_m}\} \ \models \ \beta_i$	by Definition 3.2,
$R, \{\alpha^1_1 ... \alpha^1_{n_1} \alpha^2_1 ... \alpha^2_{n_2} ... \alpha^1_m ... \alpha^m_{n_m}\} \ \models \ \beta$	by Definition 3.3, since $R \models \beta \leftarrow \beta_1 ... \beta_m$,
$R^\dagger \cup \{\alpha^1_1 ... \alpha^1_{n_1} \alpha^2_1 ... \alpha^2_{n_2} ... \alpha^1_m ... \alpha^m_{n_m}\} \ \models_c \ \beta$	by Definition 3.2,
$R^\dagger \ \models_c \ \beta \leftarrow \alpha^1_1 ... \alpha^1_{n_1} \alpha^2_1 ... \alpha^2_{n_2} ... \alpha^1_m ... \alpha^m_{n_m}$	by the Deduction Theorem,
$R, DB \ \models \ \beta[\theta]$	by Lemma 2.13 and Statement (11).

This proves item 1 for the ground case. It therefore proves item 1 for all ground instantiations of *any* rule and *any* predicate substitution. This in turn proves item 1 for all rules and all predicate substitutions, by Definition 3.3. $\Box$

Items *1*, *2* and *3* in Lemma 3.9 correspond respectively to items *2*, *3* and *4* in Definition 2.4. This gives the following result, upon which soundness is based.

Corollary 3.10 *Let R be an interpretation, and $\mathcal{S}$ be a Horn rulebase with substitution. If $R \models \mathcal{S}$ then $R \models \mathcal{S}^\dagger$.*

Proof: Let $cl(R)$ be the set of rules with substitution modelled by R, including both Horn and non-Horn rules. Thus $cl(R) = \{\rho \mid R \models \rho\}$. By assumption, $\mathcal{S} \subseteq cl(R)$, so $cl(R)$ satisfies item *1* in Definition 2.4, and by Lemma 3.9, $cl(R)$ satisfies the other three items in Definition 2.4. But $\mathcal{S}^\dagger$ is defined to be the smallest set satisfying all four of these items. Hence $\mathcal{S}^\dagger \subseteq cl(R)$, that is, $R \models \mathcal{S}^\dagger$. $\Box$

3.3 Soundness and Completeness

This section shows that the proof theory of Section 2 is sound and complete with respect to the model theory of Section 3.1. Given the results of Sections 2.4, 3.1 and 3.2, the proofs are now relatively straightforward. Throughout this section, $\mathcal{S}$ is a Horn rulebase with substitution, and DB is a database.

Theorem 3.11 (Soundness and Completeness) *Suppose β is a ground predicate with substitution, $\mathcal{S}$ is a Horn rulebase with substitution, and DB is a database. Then,*

$$\mathcal{S}, DB \models \beta \quad \textit{iff} \quad \mathcal{S}, DB \vdash \beta$$

Completeness: We first prove the *only if* direction of Theorem 3.11. To do this, we define a rulebase, $R_{\mathcal{S}}$, called the *canonical model* of $\mathcal{S}$. This model, defined proof-theoretically, provides the necessary link between the proof theory and model theory. Completeness follows from the basic properties of this model.

Definition 3.12 *If $\mathcal{S}$ is a set of Horn rules with substitution, then $R_{\mathcal{S}}$ is the set of Horn rules that are classically entailed by $\mathcal{S}^{\dagger}$. That is,*

$$R_{\mathcal{S}} = \{\alpha \leftarrow \alpha_1 ... \alpha_n \ \mid \ \mathcal{S}^{\dagger} \models_c \alpha \leftarrow \alpha_1 ... \alpha_n\}$$

where α and each α_i are atomic formulas.

Lemma 3.13 $\mathcal{S}^{\dagger} \models_c R_{\mathcal{S}}^{\dagger}$

Proof: Follows immediately from Lemma 2.10, since $\mathcal{S}^{\dagger} \models_c R_{\mathcal{S}}$. □

Lemma 3.14 *$R_{\mathcal{S}}$ is a model of $\mathcal{S}$.*

Proof: The proof is based on the characterization of models given in Theorem 3.5. Let γ be a finite set of atomic formulas, and σ be a term substitution. If $\alpha \leftarrow \beta_1 ... \beta_n$ is a rule in $\mathcal{S}$, then it is also in $\mathcal{S}^{\dagger}$, so $\mathcal{S}^{\dagger} \models_c (\alpha \leftarrow \beta_1 ... \beta_n)\sigma$. Keeping this in mind, we have the following:

if	$R_{\mathcal{S}}^{\dagger} \models_c \beta_i\sigma \leftarrow \gamma$	for each i,
then	$\mathcal{S}^{\dagger} \models_c \beta_i\sigma \leftarrow \gamma$	by Lemma 3.13,
	$\mathcal{S}^{\dagger} \models_c \alpha\sigma \leftarrow \gamma$	since $\mathcal{S}^{\dagger} \models_c (\alpha \leftarrow \beta_1 ... \beta_n)\sigma$
	$\alpha\sigma \leftarrow \gamma \in R_{\mathcal{S}}$	by Definition 3.12,
	$R_{\mathcal{S}}^{\dagger} \models_c \alpha\sigma \leftarrow \gamma$	since $R_{\mathcal{S}} \subseteq R_{\mathcal{S}}^{\dagger}$.

Thus $R_{\mathcal{S}} \models \alpha \leftarrow \beta$ by Theorem 3.5. $R_{\mathcal{S}}$ is therefore a model of each rule in $\mathcal{S}$, so it is a model of $\mathcal{S}$. □

The next result shows that inference with substitution is complete with respect to satisfaction in the canonical model.

Lemma 3.15 *If $R_{\mathcal{S}}, DB \models \beta$ then $\mathcal{S}, DB \vdash \beta$, where DB is a database, and β is a ground predicate with substitution.*

Proof:

if	$R_{\mathcal{S}}, DB \models \beta$	
then	$R_{\mathcal{S}}^{\dagger} \cup DB \models_c \beta$	by Definition 3.2,
	$\mathcal{S}^{\dagger} \cup DB \models_c \beta$	by Lemma 3.13,
	$\mathcal{S}^{\dagger} \cup DB \vdash_c \beta$	by classical completeness,
	$\mathcal{S}, DB \vdash \beta$	by Definition 2.5.

□

Completeness follows immediately. By Definition 3.4, if $\mathcal{S}, DB \models \beta$ then $R, DB \models \beta$ for any model R of $\mathcal{S}$. In particular, $R_{\mathcal{S}}, DB \models \beta$ by Lemma 3.14. Hence $\mathcal{S}, DB \vdash \beta$ by Lemma 3.15.

Soundness: We now prove the *if* direction of Theorem 3.11. To do this, recall that if $\mathcal{S}, DB \vdash \beta$ then $\mathcal{S}^{\dagger} \cup DB \vdash_c \beta$, by Definition 2.5. In using classical inference, the predicates with substitution in $\mathcal{S}^{\dagger}$ are treated as ordinary predicates of first-order classical logic. Thus, for purposes of inference, $\mathcal{S}^{\dagger}$ is an (infinite) classical Horn rulebase. β can therefore be derived by the following inference system for Horn rulebases:[4]

Axioms: α is an axiom, for each atomic formula α in DB.

Inference Rules:

Let $\beta \leftarrow \beta_1 ... \beta_m$ be a ground instance of a rule in $\mathcal{S}^{\dagger}$.

If β_i is derivable for each i, then β is also derivable.

To prove the soundness of inference, it is sufficient to prove the following lemma, which states that the axioms and inference rules of the above system are sound.

Lemma 3.16

1. $\mathcal{S}, DB \models \alpha$ *for each atomic formula* α *in* DB.
2. *Let* $\beta \leftarrow \beta_1 ... \beta_m$ *be a ground instance of a rule in* $\mathcal{S}^{\dagger}$.
   If $\mathcal{S}, DB \models \beta_i$ *for each* i, *then* $\mathcal{S}, DB \models \beta$.

Proof: The first item is trivially true, by Definitions 3.4 and 3.2. To prove the second item, let $\beta \leftarrow \beta_1 ... \beta_n$ be a ground instance of a rule in $\mathcal{S}^{\dagger}$, and suppose that $\mathcal{S}, DB \models \beta_i$ for each i. Then for each model R of $\mathcal{S}$, we have $R, DB \models \beta_i$ for each i, by Definition 3.4. Furthermore, R is also a model of $\beta \leftarrow \beta_1 ... \beta_m$, by Corollary 3.10. Thus $R, DB \models \beta$ by item *1* of Lemma 3.8. Since this is true for any model R of $\mathcal{S}$, it follows that $\mathcal{S}, DB \models \beta$. $\Box$

4 Fixpoint Semantics

This section develops the fixpoint semantics of Horn rules with predicate substitution, following the tradition of classical logic programming [1, 18]. A main result is that a Horn rulebase with substitution has a unique minimal model. As in classical Horn logic, the minimal model can be viewed as the user's "intended model", a model that he could have written if he had the time, space and inclination. Another result is that the minimal model can be constructed in a bottom-up, iterative fashion by the repeated application of a monotonic and continuous "T operator." As in classical Horn logic, the T-operator is defined on a lattice of interpretations. However, while the T-operator of classical logic essentially maps databases to databases, the T-operator developed here maps Horn rulebases to Horn rulebases. In effect, predicate substitution elevates Horn rules to the meta-level: Whereas they are normally a specification language, they are now the objects being specified.

Given a Horn rulebase with substitution, $\mathcal{S}$, each application of the T-operator "unfolds" the rules in $\mathcal{S}$ into classical Horn rules. As the operator is applied again and again, more and more unfolding occurs. In this way, the specification of a Horn rulebase is gradually

[4]Because classical logic is compact, we can reason with $\mathcal{S}^{\dagger}$ as if it were finite.

unfolded into the Horn rulebase itself. If $\mathcal{S}$ is non-recursive, then the unfolding process is similar to macro expansion, terminating after a finite number of steps; but if $\mathcal{S}$ is recursive, then the unfolding process may go on indefinitely. However, because the T-operator is continuous, the process converges to the minimal model, so that each Horn rule in the minimal model is generated after a finite (though possibly unbounded) number of steps.

Section 4.1 develops the lattice of interpretations, and shows that it is complete. Section 4.2 defines the T-operator for this lattice, and shows that it is monotonic and continuous, and thus has a least fixpoint. Finally, Section 4.3 illustrates the bottom-up computation of least fixpoints using the T-operator, including the generation of infinite fixpoints.

Example 4.1 (Minimal Models) Suppose that $\mathcal{S}_1$ and $\mathcal{S}_2$ are the following Horn rulebases with substitution, where E, F and P are base predicates:

$$\mathcal{S}_1 = \left\{ \begin{array}{l} B(x) \leftarrow E(x), P(x). \\ A(x) \leftarrow B(x)[P/Q] \end{array} \right\} \qquad \mathcal{S}_2 = \left\{ \begin{array}{l} B(x) \leftarrow E(x), P(x). \\ B(x) \leftarrow F(x,y), P(x), P(y). \\ A(x) \leftarrow B(x)[P/Q] \end{array} \right\}$$

Then each of the following Horn rulebases is a model of $\mathcal{S}_1$, but only R_3 is a model of $\mathcal{S}_2$. In addition, R_1 is the minimal model of $\mathcal{S}_1$, and R_3 is the minimal model of $\mathcal{S}_2$.

$$R_1 = \left\{ \begin{array}{l} B(x) \leftarrow E(x), P(x). \\ A(x) \leftarrow E(x), Q(x). \end{array} \right\} \qquad R_2 = \left\{ \begin{array}{l} B(x) \leftarrow E(x), P(x). \\ A(x) \leftarrow E(x), Q(x). \\ A(x) \leftarrow F(x,y), Q(x), Q(y). \end{array} \right\}$$

$$R_3 = \left\{ \begin{array}{l} B(x) \leftarrow E(x), P(x). \\ B(x) \leftarrow F(x,y), P(x), P(y). \\ A(x) \leftarrow E(x), Q(x). \\ A(x) \leftarrow F(x,y), Q(x), Q(y). \end{array} \right\}$$

In each model, the rules defining $B(x)$ are reused and modified to produce rules defining $A(x)$. In the minimal model, these are the *only* rules defining A. In addition, classical Horn rules in $\mathcal{S}_i$ are the only rules defining B in the minimal model. The minimal model thus contains no superfluous rules. $\Box$

4.1 The Lattice of Horn Rulebases

A central feature of any fixpoint semantics is a lattice of interpretations. In the case of predicate substitution, each lattice element is an equivalence class of classical Horn rulebases (finite or infinite). Two Horn rulebases are in the same equivalence class if and only if they are classically equivalent. For simplicity, we shall speak as if lattice elements were individual Horn rulebases. In effect, we use individual members of an equivalence class to denote the entire class. We shall therefore say that the least element of the lattice is the empty rulebase, and the greatest element is the set of all Horn rules. This understanding simplifies the presentation that follows.

Lemma 4.1 *The set of all classical Horn rulebases 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 classical Horn rulebases.*

3. *$\sqcap\mathcal{R} = \cap_{R\in\mathcal{R}} \{\rho \mid R \models_c \rho\}$ where each ρ is a classical Horn rule.*

Proof: We prove each item in turn.

1. Since classical entailment is transitive and reflexive, it is a partial order.

2. We must show that $\sqcup\mathcal{R}$ is the least upper bound of the elements in $\mathcal{R}$ under the partial order $\geq$. Clearly it is an upper bound. To show that it is minimal, suppose that R_0 is an arbitrary upper bound, that is, $R_0 \geq R$ for every R in $\mathcal{R}$. Then $R_0 \models_c R$ for every R in $\mathcal{R}$. Thus $R_0 \models_c \cup\mathcal{R}$, so $R_0 \geq \sqcup\mathcal{R}$. Hence $\sqcup\mathcal{R}$ is minimal.

3. We must show that $\sqcap\mathcal{R}$ is the greatest lower bound of the elements in $\mathcal{R}$. Clearly it is a lower bound. To show that it is maximal, suppose that R_0 is an arbitrary lower bound. Then,

$$
\begin{array}{ll}
R \geq R_0 & \text{for every } R \text{ in } \mathcal{R} \\
R \models_c R_0 & \text{for every } R \text{ in } \mathcal{R} \\
R_0 \subseteq \{\rho \mid R \models_c \rho\} & \text{for every } R \text{ in } \mathcal{R} \\
R_0 \subseteq \cap_{R\in\mathcal{R}} \{\rho \mid R \models_c \rho\} & \\
R_0 \subseteq \sqcap\mathcal{R} & \\
\sqcap\mathcal{R} \models_c R_0 & \\
\sqcap\mathcal{R} \geq R_0 &
\end{array}
$$

   Thus, $\sqcap\mathcal{R}$ is the greatest lower bound.

□

In general, $R_1 \sqcap R_2 \neq R_1 \cap R_2$. For instance,

$$
\begin{array}{llll}
\textit{if} & R_1 = \{A \leftarrow P,\ \ P \leftarrow B\} & \textit{and} & R_2 = \{A \leftarrow Q,\ \ Q \leftarrow B\} \\
\textit{then} & R_1 \sqcap R_2 \equiv \{A \leftarrow B\} & \textit{and} & R_1 \cap R_2 = \{\}.
\end{array}
$$

The following basic result shows that rulebases higher in the lattice satisfy more formulas.

Lemma 4.2 *If $R_2 \geq R_1$ and $R_1, DB \models \beta$ then $R_2, DB \models \beta$, for any database DB, and any ground predicate with substitution β.*

Proof: Suppose that $R_2 \geq R_1$. Then $R_2 \models_c R_1$, so $R_2^\dagger \models_c R_1^\dagger$ by Corollary 2.11. Hence,

$$
\begin{array}{lll}
\textit{if} & R_1, DB \models \beta & \\
\textit{then} & R_1^\dagger \cup DB \models_c \beta & \text{by definition} \\
 & R_2^\dagger \cup DB \models_c \beta & \text{since } R_2^\dagger \models_c R_1^\dagger \\
 & R_2, DB \models \beta & \text{by definition}
\end{array}
$$

□

4.2 The T operator

Another central feature of any fixpoint semantics is a "T operator," which takes one lattice element as input and returns another lattice element as output. The T operator provides a way of "moving" from point to point within the lattice. Each Horn rulebase with substitution $\mathcal{S}$, has its own operator, $T_{\mathcal{S}}$. An important property of this operator is that the fixpoints of $T_{\mathcal{S}}$ are exactly the models of $\mathcal{S}$, thus providing a link between the fixpoint theory and the model theory.

$T_{\mathcal{S}}$ maps each classical Horn rulebase R to another classical Horn rulebase $T_{\mathcal{S}}(R)$. In the simplest case, if $\mathcal{S}$ contains the Horn rule $A \leftarrow B$, and if R contains the rule $B \leftarrow C$, then $T_{\mathcal{S}}(R)$ contains the rule $A \leftarrow C$. Of course, $\mathcal{S}$ may also contain rules with substitution, such as $A \leftarrow B[P/Q]$. In this case, $R^\dagger$ provides the rules defining $B[P/Q]$. If $R^\dagger \models^c B[P/Q] \leftarrow C$, then $T_{\mathcal{S}}(R)$ contains the rule $A \leftarrow C$. Notice that we ask about the rules *entailed* by $R^\dagger$, and not just about the rules *in* $R^\dagger$. This allows the rule premises in $R^\dagger$ to be expanded until they are atomic. The next definition formalizes these considerations. It should be compared to the characterization of models given in Theorem 3.5. Much of the development in this section depends on that theorem.

Definition 4.3 (T Operator) *If $\mathcal{S}$ is a Horn rulebase with substitution, then $T_{\mathcal{S}}$ is a mapping from classical Horn rulebases to classical Horn rulebases. In particular, for each classical Horn rulebase, R,*

$$\begin{array}{rcll} T_{\mathcal{S}}(R) & = & \cup_\sigma \; \{\alpha\sigma \leftarrow \gamma & | \;\; \textit{for some rule } \; \alpha \leftarrow \beta_1 ... \beta_n \; \textit{ in } \mathcal{S}, \\ & & & \;\;\; R^\dagger \models_c \beta_i\sigma \leftarrow \gamma \;\; \textit{for all } 1 \leq i \leq n \; \} \end{array}$$

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

Intuitively, the operator $T_{\mathcal{S}}$ "unfolds" the rules with substitution in $\mathcal{S}$ and converts them into classical Horn rules, just as the operator T_R of classical Horn logic "unfolds" the Horn rules in R and converts them into atoms. The following basic result shows that the operator $T_{\mathcal{S}}$ maps equivalent rulebases onto equivalent rulebases. (This property is essential since each lattice element is really an equivalence class of Horn rulebases.)

Lemma 4.4 *If $R_1 \equiv R_2$ then $T_{\mathcal{S}}(R_1) \equiv T_{\mathcal{S}}(R_2)$.*

Proof: In fact, we shall prove that $T_{\mathcal{S}}(R_1) = T_{\mathcal{S}}(R_2)$. If $R_1 \equiv R_2$, then R_1 and R_2 are classically equivalent, and so by Corollary 2.11, $R_1^\dagger$ and $R_2^\dagger$ are classically equivalent too. Thus, referring to Definition 4.3, $R_1^\dagger$ classically entails $\beta_i\sigma \leftarrow \gamma$ iff $R_2^\dagger$ does. Thus $T_{\mathcal{S}}(R_1)$ contains $\alpha\sigma \leftarrow \gamma$ iff $T_{\mathcal{S}}(R_2)$ does. Hence $T_{\mathcal{S}}(R_1) = T_{\mathcal{S}}(R_2)$. $\Box$

A *fixpoint* of $T_{\mathcal{S}}$ is a classical Horn rulebase R such that $R \geq T_{\mathcal{S}}(R)$. The following result says that the fixpoints of $T_{\mathcal{S}}$ are exactly the models of $\mathcal{S}$.

Lemma 4.5 $R \models \mathcal{S}$ *iff* $R \geq T_{\mathcal{S}}(R)$

Proof: In the *if* direction, suppose that $R \models_c T_{\mathcal{S}}(R)$. We must show that $R \models \mathcal{S}$. To prove this, suppose that $\alpha \leftarrow \beta_1 ... \beta_n$ is a rule in $\mathcal{S}$. Then, for any term substitution σ, and any finite set of atomic formulas, γ,

if	$R^\dagger \models_c \beta_i\sigma \leftarrow \gamma$	for each i
then	$\alpha\sigma \leftarrow \gamma \ \in \ T_{\mathcal{S}}(R)$	by Definition 4.3
	$R \models_c \alpha\sigma \leftarrow \gamma$	since $R \models_c T_{\mathcal{S}}(R)$
	$R^\dagger \models_c \alpha\sigma \leftarrow \gamma$	since $R \subseteq R^\dagger$, by Definition 2.4.

Thus $R \models \alpha \leftarrow \beta_1 ... \beta_n$ by Theorem 3.5. This is true for any rule in $\mathcal{S}$. Hence $R \models \mathcal{S}$.

In the *only if* direction, suppose that $R \models \mathcal{S}$. We must show that $R \models_c T_{\mathcal{S}}(R)$. Suppose that $\alpha\sigma \leftarrow \gamma$ is a rule in $T_{\mathcal{S}}(R)$. Then, for some rule $\alpha \leftarrow \beta_1 ... \beta_n$ in $\mathcal{S}$,

$$R^\dagger \models_c \beta_i\sigma \leftarrow \gamma$$

for each i, by Definition 4.3. Furthermore, $R \models \mathcal{S}$ by hypothesis, so

$$R \models \alpha \leftarrow \beta_1 ... \beta_n$$

Combining these two statements, we get $R^\dagger \models_c \alpha\sigma \leftarrow \gamma$ by Theorem 3.5. Thus, $R \models_c \alpha\sigma \leftarrow \gamma$ by Definition 2.4, since $\alpha\sigma \leftarrow \gamma$ is a classical Horn rule, and R is a classical Horn rulebase. Therefore R classically entails every rule in $T_{\mathcal{S}}(R)$. Hence $R \models_c T_{\mathcal{S}}(R)$. $\Box$

Monotonicity:

We now show that the operator $T_{\mathcal{S}}$ is *monotonic*: If the input to the operator increases, then so does the output. Because it is monotonic, the operator has a least fixpoint $lfp(T_{\mathcal{S}})$, which is also the unique minimal model of the rulebase $\mathcal{S}$. We show that the minimal model is canonical in that the formulas satisfied in this model are exactly the formulas entailed by $\mathcal{S}$.

Lemma 4.6 (Monotonicity) *If* $R_1 \geq R_2$ *then* $T_{\mathcal{S}}(R_1) \geq T_{\mathcal{S}}(R_2)$.

Proof: Suppose that $R_1 \models_c R_2$. We show that $T_{\mathcal{S}}(R_2) \subseteq T_{\mathcal{S}}(R_1)$. First note that $R_1^\dagger \models_c R_2^\dagger$ by Corollary 2.11. Thus, referring to Definition 4.3,

if	$\alpha\sigma \leftarrow \gamma \ \in \ T_{\mathcal{S}}(R_2)$	
then	$R_2^\dagger \models_c \beta_i\sigma \leftarrow \gamma$	for some rule $\alpha \leftarrow \beta_1 ... \beta_n$ in $\mathcal{S}$, by Definition 4.3,
	$R_1^\dagger \models_c \beta_i\sigma \leftarrow \gamma$	since $R_1^\dagger \models_c R_2^\dagger$,
	$\alpha\sigma \leftarrow \gamma \ \in \ T_{\mathcal{S}}(R_1)$	by Definition 4.3, since $\alpha \leftarrow \beta_1 ... \beta_n$ is in $\mathcal{S}$.

Thus every rule in $T_{\mathcal{S}}(R_2)$ is also in $T_{\mathcal{S}}(R_1)$. $\Box$

Since $T_{\mathcal{S}}$ is a monotonic operator on a complete lattice, it has a least fixpoint, as stated in the following lemma. This result is due to Tarski [16].

Theorem 4.7 (Least Fixpoint) *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 $T_{\mathcal{S}}$.*

As the following corollary shows, the least fixpoint of $T_{\mathcal{S}}$ is a canonical model of $\mathcal{S}$ in that the database queries satisfied in this model are exactly the queries entailed by $\mathcal{S}$.

Corollary 4.8 *Let $\mathcal{S}$ be a Horn rulebase with substitution, β be a ground predicate with substitution, and DB be a database. Then*

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

Proof: In the *only if* direction, suppose that $\mathcal{S}, DB \models \beta$. Then $R, DB \models \beta$ for all models R of $\mathcal{S}$. In particular $\mathit{lfp}(T_{\mathcal{S}}), DB \models \beta$ since $\mathit{lfp}(T_{\mathcal{S}})$ is a model of $\mathcal{S}$. In the *if* direction, suppose that $\mathit{lfp}(T_{\mathcal{S}}), DB \models \beta$. Then,

$R, DB \models \beta$	for all $R \geq \mathit{lfp}(T_{\mathcal{S}})$, by Lemma 4.2
$R, DB \models \beta$	for all fixpoints, R, of $T_{\mathcal{S}}$, since $R \geq \mathit{lfp}(T_{\mathcal{S}})$
$R, DB \models \beta$	for all models, R, of $\mathcal{S}$, since a model is a fixpoint
$\mathcal{S}, DB \models \beta$	by Definition 3.4.

□

Continuity:

We now show that the operator $T_{\mathcal{S}}$ is *continuous.* Intuitively, this means that if a sequence of input values converges to R, then the sequence of output values converges to $T_{\mathcal{S}}(R)$. Because the operator is continuous, its least fixpoint $\mathit{lfp}(T_{\mathcal{S}})$ can be constructed by an iterative, bottom-up procedure. This procedure starts at the bottom of the lattice, and climbs higher and higher up the lattice, converging on the least fixpoint. Continuity is proved by the first two lemmas below.

Lemma 4.9 *Suppose that ρ is a classical Horn rule, and that $R_1 \leq R_2 \leq R_3 ...$ is a (possibly infinite) non-decreasing sequence of classical Horn rulebases. Then,*

$$\textit{if} \quad (\sqcup_j R_j)^\dagger \models_c \rho \quad \textit{then} \quad R_k^\dagger \models_c \rho \quad \textit{for some } k.$$

Proof: Suppose that $(\sqcup_j R_j)^\dagger \models_c \rho$. Then by Corollary 2.8, there is a finite subset, S, of $\sqcup_j R_j$ such that

$$S^\dagger \models_c \rho \tag{12}$$

Since $S \subseteq \cup_j R_j$, each rule in S is in R_j for some j. Since S is finite, these j's have a maximum value, k. Hence $S \subseteq \cup_{j \leq k} R_j$. Therefore

$$\cup_{j \leq k} R_j \ \models_c \ S$$

But $R_k \models_c R_j$ for $j \leq k$, by hypothesis. Thus $R_k \models_c \cup_{j \leq k} R_j$, so $R_k \models_c S$. Thus $R_k^\dagger \models_c S^\dagger$ by Corollary 2.11, so $R_k^\dagger \models_c \rho$ by (12). □

Lemma 4.10 (Continuity) *Suppose that $R_1 \leq R_2 \leq R_3$ ... is a (possibly infinite) non-decreasing sequence of classical Horn rulebases. Then $T_{\mathcal{S}}(\sqcup_j R_j) \ \equiv \ \sqcup_j T_{\mathcal{S}}(R_j)$.*

Proof: First note that $\sqcup_j R_j \geq R_k$ for all k. Thus $T_{\mathcal{S}}(\sqcup_j R_j) \geq T_{\mathcal{S}}(R_k)$ for all k, by monotonicity. Hence $T_{\mathcal{S}}(\sqcup_j R_j) \geq \sqcup_k T_{\mathcal{S}}(R_k)$. To prove the inequality in the other direction, suppose that $\alpha\sigma \leftarrow \gamma$ is a rule in $T_{\mathcal{S}}(\sqcup_j R_j)$. Then, by Definition 4.3, there is a rule $\alpha \leftarrow \beta_1 ... \beta_n$ in $\mathcal{S}$ such that $(\sqcup_j R_j)^\dagger \models_c \beta_i\sigma \leftarrow \gamma$ for each i. Therefore, by Lemma 4.9, for each i there is an integer, k_i, such that

$$R_{k_i}^\dagger \models_c \beta_i\sigma \leftarrow \gamma \tag{13}$$

Let k be the maximum of the k_i's. Then $R_{k_i} \leq R_k$ by hypothesis, so $R_k \models_c R_{k_i}$. Thus $R_k^\dagger \models R_{k_i}^\dagger$ by Corollary 2.11. Hence $R_k^\dagger \models_c \beta_i\sigma \leftarrow \gamma$ for each i, by (13). But $\alpha \leftarrow \beta_1 ... \beta_n$ is a rule in $\mathcal{S}$. Thus $\alpha\sigma \leftarrow \gamma$ is in $T_{\mathcal{S}}(R_k)$ by Definition 4.3. Hence $\alpha\sigma \leftarrow \gamma$ is in $\cup_j T_{\mathcal{S}}(R_j)$, that is, in $\sqcup_j T_{\mathcal{S}}(R_j)$.

We have thus shown that every rule in $T_{\mathcal{S}}(\sqcup_j R_j)$ is also in $\sqcup_j T_{\mathcal{S}}(R_j)$. Thus $T_{\mathcal{S}}(\sqcup_j R_j) \subseteq \sqcup_j T_{\mathcal{S}}(R_j)$, and so $T_{\mathcal{S}}(\sqcup_j R_j) \leq \sqcup_j T_{\mathcal{S}}(R_j)$. □

The following standard definitions define $T_{\mathcal{S}}^*(R)$, which is the result of repeatedly applying the operator $T_{\mathcal{S}}$ to R.

Definition 4.11 *Let $\mathcal{S}$ be a Horn rulebase with substitution, and let R be a classical Horn rulebase. Then,*

- $T_{\mathcal{S}}^0(R) \;=\; R$
- $T_{\mathcal{S}}^{i+1}(R) \;=\; T_{\mathcal{S}}[T_{\mathcal{S}}^i(R)]$
- $T_{\mathcal{S}}^*(R) \;=\; \sqcup_{i\geq 0}\, T_{\mathcal{S}}^i(R)$

Starting from the empty rulebase $\{\}$, the operator $T_{\mathcal{S}}$ produces a sequence of larger and larger Horn rulebases $T_{\mathcal{S}}^0(\{\})$, $T_{\mathcal{S}}^1(\{\})$, $T_{\mathcal{S}}^2(\{\})$ ... The main result of this section is that this sequence converges to the least fixpoint of $T_{\mathcal{S}}$, as stated in the next lemma. This result follows from the monotonicity and continuity of the operator $T_{\mathcal{S}}$, as originally shown by Tarski [16].

Theorem 4.12 $T_{\mathcal{S}}^*(\{\}) \;\equiv\; \mathit{lfp}(T_{\mathcal{S}})$

To start a least fixpoint computation, we must compute $T_{\mathcal{S}}(\{\})$. This is a relatively easy computation. As the next lemma shows, $T_{\mathcal{S}}(\{\})$ extracts the classical Horn rules from $\mathcal{S}$. It may also convert some rules *with* substitution into rules *without* substitution, but these rules are unlikely to be in $\mathcal{S}$ in the first place. For instance, the rule $A(x) \leftarrow P(x)[P/Q]$ is unlikely to be in $\mathcal{S}$, since the programmer would probably have used the rule $A(x) \leftarrow Q(x)$ instead. However, if this rule were in $\mathcal{S}$, then the rule $A(x) \leftarrow Q(x)$ would be in $T_{\mathcal{S}}(\{\})$. i.e., $T_{\mathcal{S}}$ would "unfold" the given rule into a Horn rule.

Lemma 4.13 *The classical Horn rules in $\mathcal{S}$ are also in $T_{\mathcal{S}}(R)$, for any R.*

Proof: Let $\alpha \leftarrow \beta_1 ... \beta_n$ be a classical Horn rule in $\mathcal{S}$. Then each β_i is an atomic formula. Thus, $\beta_1 ... \beta_n$ can play the role of γ in Definition 4.3. To do this, first note that $\beta_i \leftarrow \beta_1 ... \beta_n$ is a classical tautology, for each i, and so $R^\dagger \models_c \beta_i \leftarrow \beta_1 ... \beta_n$. Thus, from Definition 4.3, it follows that $\alpha \leftarrow \beta_1 ... \beta_n$ is in $T_{\mathcal{S}}(R)$, by using $\gamma = \beta_1 ... \beta_n$ and using the empty substitution for σ. □

4.3 Examples

As shown in the previous section, the minimal model of a Horn rulebase with substitution, $\mathcal{S}$, can be generated by applying the operator $T_{\mathcal{S}}$ over and over again. With each application of the operator, substitution rules in $\mathcal{S}$ are expanded into classical Horn rules. The first application of $T_{\mathcal{S}}$ simply extracts the classical Horn rules in $\mathcal{S}$. The second application combines these classical Horn rules with substitution rules in $\mathcal{S}$ to generate new classical Horn rules. The process then repeats: Each application of the operator generates new classical Horn rules, which are then combined with substitution rules in $\mathcal{S}$ to produce still more classical Horn rules. If at any point, no new Horn rules are generated, then the minimal model has been reached. The examples of this section illustrate the step-by-step expansion of substitution rules into classical Horn rules. The last two examples, 4.4 and 4.5, show two ways in which a *finite* set of Horn rules with substitution can be expanded into an *infinite* set of classical Horn rules. Example 4.4 uses function symbols to generate an infinite set of rules, each of fixed length. Example 4.5 uses no function symbols, but generates longer and longer rules.

Example 4.2 Suppose $\mathcal{S}$ consists of the following three rules:

$$\begin{array}{l} A(x) \leftarrow B(x), C(x). \\ A(x) \leftarrow D(x,y), C(x), C(y). \\ A'(x) \leftarrow A(x)[C/C'] \end{array}$$

where B, C and D are base predicates. It then takes two applications of the operator $T_{\mathcal{S}}$ to generate the minimal model of $\mathcal{S}$. That is, $lfp(T_{\mathcal{S}}) = T_{\mathcal{S}}^2(\{\})$. This model consists of four Horn rules, generated as follows:

$$\left.\begin{array}{l} A(x) \leftarrow B(x), C(x). \\ A(x) \leftarrow D(x,y), C(x), C(y). \end{array}\right\} \in T_{\mathcal{S}}^1(\{\})$$
$$\left.\begin{array}{l} A'(x) \leftarrow B(x), C'(x). \\ A'(x) \leftarrow D(x,y), C'(x), C'(y). \end{array}\right\} \in T_{\mathcal{S}}^2(\{\})$$

□

Typically, not all the substitution rules in $\mathcal{S}$ will be applicable during each application of the operator $T_{\mathcal{S}}$. At first, only a few substitution rules may be applicable. However, as more and more classical Horn rules are generated, more and more substitution rules become available for conversion into classical Horn rules. This is illustrated by the following propositional example.

Example 4.3 Suppose $\mathcal{S}$ consists of the following $n+1$ rules:

$$\begin{array}{l} A_1 \leftarrow B_1, B_2, B_3 ... B_n. \\ A_2 \leftarrow A_1[B_1/C_1] \\ A_3 \leftarrow A_2[B_2/C_2] \\ \quad \cdots \\ A_{n+1} \leftarrow A_n[B_n/C_n] \end{array}$$

where each B_i and C_i is a base predicate. It then takes $n+1$ applications of the operator $T_{\mathcal{S}}$ to generate the minimal model of $\mathcal{S}$. That is, $\mathit{lfp}(T_{\mathcal{S}}) = T_{\mathcal{S}}^{n+1}(\{\})$. This model consists of $n+1$ classical Horn rules, generated as follows:

$$
\begin{array}{rcl}
A_1 \leftarrow B_1, B_2, B_3 ... B_n. & \in & T_{\mathcal{S}}^1(\{\}) \\
A_2 \leftarrow C_1, B_2, B_3 ... B_n. & \in & T_{\mathcal{S}}^2(\{\}) \\
A_3 \leftarrow C_1, C_2, B_3 ... B_n. & \in & T_{\mathcal{S}}^3(\{\}) \\
\cdots & & \\
A_{n+1} \leftarrow C_1, C_2, C_3 ... C_n. & \in & T_{\mathcal{S}}^{n+1}(\{\})
\end{array}
$$

□

In general, when there is recursion through substitution, the minimal model of $\mathcal{S}$ will be infinite, and generating this model will require an infinite number of iterations of the operator $T_{\mathcal{S}}$.

Example 4.4 Suppose $\mathcal{S}$ consists of the following three rules:

$$
\begin{array}{l}
A(x) \leftarrow B(x,a), B(x,b), B(x,c). \\
A(x) \leftarrow A(x)[B/B'] \\
B'(x,y) \leftarrow B(x,fy)
\end{array}
$$

where B is a base predicate, and f is a unary function symbol. Note that the second rule involves recursion through substitution. This recursive rule has the following consequence, for any integer $i \geq 1$, and any function terms u, v and w,

$$
\begin{array}{llcl}
\mathit{if} & T_{\mathcal{S}}^i(\{\}) & \models_c & A(x) \leftarrow B(x,u), B(x,v), B(x,w). \\
\mathit{then} & T_{\mathcal{S}}^{i+1}(\{\}) & \models_c & A(x) \leftarrow B'(x,u), B'(x,v), B'(x,w). \\
\mathit{and} & T_{\mathcal{S}}^{i+1}(\{\}) & \models_c & B'(x,y) \leftarrow B(x,fy) \\
\mathit{hence} & T_{\mathcal{S}}^{i+1}(\{\}) & \models_c & A(x) \leftarrow B(x,fu), B(x,fv), B(x,fw).
\end{array}
$$

Thus, since the rule $A(x) \leftarrow B(x,a), B(x,b), B(x,c)$ is in $\mathcal{S}$,

$$
\begin{array}{rcl}
T_{\mathcal{S}}^1(\{\}) & \models_c & A(x) \leftarrow B(x,a), B(x,b), B(x,c). \\
T_{\mathcal{S}}^2(\{\}) & \models_c & A(x) \leftarrow B(x,fa), B(x,fb), B(x,fc). \\
T_{\mathcal{S}}^3(\{\}) & \models_c & A(x) \leftarrow B(x,f^2a), B(x,f^2b), B(x,f^2c). \\
\cdots & & \\
T_{\mathcal{S}}^{i+1}(\{\}) & \models_c & A(x) \leftarrow B(x,f^ia), B(x,f^ib), B(x,f^ic). \\
\cdots & &
\end{array}
$$

Thus, an infinite set of classical Horn rules is built up one rule at a time by reusing and modifying the seed rule $A(x) \leftarrow B(x,a), B(x,b), B(x,c)$ an infinite number of times. □

Example 4.4 shows how a finite rulebase with substitution can represent an infinite rulebase without substitution. In this case, the use of function symbols was crucial. In contrast, the next example shows that an infinite minimal model is possible even *without* function symbols. Intuitively, infiniteness comes about because the *length* of the rules in the minimal model is unbounded: each application of the T-operator generates longer and longer rules. In this respect, Datalog with substitution is different from classical Datalog, for which the minimal model is always finite.

Example 4.5 Suppose $\mathcal{S}$ consists of the following three rules:

$$\begin{array}{l} A(x) \leftarrow P(x) \\ A(x) \leftarrow A(x)[P/Q] \\ Q(x) \leftarrow B(x,y), P(y). \end{array}$$

where B and P are base predicates. Note that the second rule involves recursion through substitution. This recursive rule has the following consequence, for any integer $i \geq 1$, and any sequence of variables $x_1, x_2 ... x_j, x_{j+1}$,

$$\begin{array}{lrll} \textit{if} & T^i_{\mathcal{S}}(\{\}) & \models_c & A(x_1) \leftarrow B(x_1,x_2) ... B(x_{j-1},x_j), P(x_j). \\ \textit{then} & T^{i+1}_{\mathcal{S}}(\{\}) & \models_c & A(x_1) \leftarrow B(x_1,x_2) ... B(x_{j-1},x_j), Q(x_j). \\ \textit{and} & T^{i+1}_{\mathcal{S}}(\{\}) & \models_c & Q(x) \leftarrow B(x,y), P(y). \\ \textit{hence} & T^{i+1}_{\mathcal{S}}(\{\}) & \models_c & A(x_1) \leftarrow B(x_1,x_2) ... B(x_{j-1},x_j), B(x_j,x_{j+1}), P(x_{j+1}). \end{array}$$

Thus, since the rule $A(x) \leftarrow P(x)$ is in $\mathcal{S}$,

$$\begin{array}{lll} T^1_{\mathcal{S}}(\{\}) & \models_c & A(x_1) \leftarrow P(x_1) \\ T^2_{\mathcal{S}}(\{\}) & \models_c & A(x_1) \leftarrow B(x_1,x_2), P(x_2). \\ T^3_{\mathcal{S}}(\{\}) & \models_c & A(x_1) \leftarrow B(x_1,x_2), B(x_2,x_3), P(x_3). \\ \quad\cdots & & \\ T^i_{\mathcal{S}}(\{\}) & \models_c & A(x_1) \leftarrow B(x_1,x_2), B(x_2,x_3) ... B(x_{i-1},x_i), P(x_i). \\ \quad\cdots & & \end{array}$$

Thus, the rules in $T^i_{\mathcal{S}}(\{\})$ test the database for "B chains" of length i. Each application of the operator $T_{\mathcal{S}}$ takes a rule that tests for chains of length i and transforms it into a rule that tests for chains of length $i+1$. In this way, an infinite set of classical Horn rules is built up, one rule at a time, by reusing and modifying the seed rule $A(x) \leftarrow P(x)$ an infinite number of times. □

5 Datalog with Substitution

So far, the development in this paper has been general in that rules and databases could be infinite and could contain arbitrary function terms. In the rest of the paper, we focus on a special case: the finite, function-free Horn case. This is the case of most importance for database applications. Thus, from now on, databases are finite, and all function terms are either constant symbols or variables. In Database parlance, this is called the *Datalog* restriction. We shall use the word "Datalog" as a synonym for "function-free Horn," and we shall refer to classical Datalog rules and to Datalog rules with substitution.

This section develops a specialized fixpoint semantics for Datalog rules with substitution. Unlike the semantics developed earlier, the value of this semantics is not conceptual, but computational: it is a technical device needed to establish the upper complexity bound in Section 6. The model theory and fixpoint theory developed in the previous two sections are the starting point for our proof of this upper bound. This section puts that semantics into a form that is computationally tractable, and in which the minimal model is always finite.

If we combine Corollary 4.8 and Theorem 4.12, then we get the following result:

$$\mathcal{S}, DB \models \beta \quad \textit{iff} \quad T^*_{\mathcal{S}}(\{\}), DB \models \beta \tag{14}$$

Thus, instead of using the rulebase with substitution, $\mathcal{S}$, to answer queries, we can use its minimal model, $T^*_{\mathcal{S}}(\{\})$. This model is a set of classical Horn rules. In principle, we can construct this set by starting with the empty set, $\{\}$, and applying $T_{\mathcal{S}}$ to it over and over again until saturation is reached, i.e., until there are no more changes. Unfortunately, the minimal model may be infinite and thus impossible to materialize, even in the function-free case, as Example 4.5 showed.

To overcome this difficulty, this section develops a ground version of the fixpoint theory. To answer queries about a *particular* database, we shall not use the minimal model of $\mathcal{S}$. Instead, we shall use its ground instantiation. Section 6 will show that this instantiation is a finite set of ground Datalog rules of exponential size. To construct these rules, we do not use $T_{\mathcal{S}}$, but a simpler operator that can be thought of as the "ground instantiation" of $T_{\mathcal{S}}$. Although the least fixpoint of this new operator is finite, it is conceptually different from the least fixpoint of $T_{\mathcal{S}}$. Both fixpoints are sets of classical Horn rules. However, the least fixpoint of $T_{\mathcal{S}}$ is the intended model of $\mathcal{S}$, that is, a set of rules that the user had in mind. In contrast, the least fixpoint of the new operator changes each time the database changes, and thus does not behave like a user-specified rulebase. However, because this least fixpoint is finite, it provides a means of answering queries.

The development is divided into three parts. First, Section 5.1 shows that for a given database, a rulebase with substitution can be replaced by its ground instantiation. Then, Section 5.2 shows that the minimal model can be replaced by its ground instantiation. Finally, Section 5.3 shows that a simplified version of the T-operator will compute the instantiated minimal model. First, however, we define precisely what we mean by a ground instantiation, a notion used throughout this section.

Definition 5.1 (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 $inst_{\mathcal{C}}(\mathcal{S})$.*

5.1 Reduction to Ground Rulebases

In classical logic, to answer a query about a given database, we can replace a Datalog rulebase by its ground instantiation. This subsection shows that Datalog rulebases with substitution enjoy the same property.

To formalize the classical case, let R be a set of Datalog rules, let DB be a database, and let α be a ground atomic formula. Finally, let $\mathcal{C}$ be a set of constant symbols including all constants that appear in R, DB and α. Then,

$$R \cup DB \vdash_c \alpha \quad \textit{iff} \quad inst_{\mathcal{C}}(R) \cup DB \vdash_c \alpha \tag{15}$$

Of course, the instantiation of R depends on the database and may need to be recomputed whenever the database changes.

To extend this result to rulebases with predicate substitution, we must first look more closely at $\mathcal{S}^\dagger$. It turns out that in generating $\mathcal{S}^\dagger$ from $\mathcal{S}$, we can assume that predicate substitutions have a certain form. To see this, look at item *2* of Definition 2.4. This item generates rules of the form $(\beta \leftarrow \beta_1 ... \beta_n)[\theta]$. Let $[P/Q](t_1 ... t_k)$ be the predicate substitution $[\theta]$. Each t_i is a term, *i.e.*, variable or a constant. However, no information is lost by assuming that each t_i is a variable. Some rules are lost, but they are simple instantiations of rules that remain. Likewise, we can assume that each t_i is a new variable, one that does not appear in the rule $\beta \leftarrow \beta_1 ... \beta_n$. Rules that don't satisfy this assumption are specializations of rules that do. The same is true of the predicate substitutions introduced by items *3* and *4* in Definition 2.4. We can therefore assume that each predicate substitution introduced in Definition 2.4 contains only new variables and no constants. We can thus replace $\mathcal{S}^\dagger$ by a rule set that is smaller than that given by definition 2.4, but which is logically equivalent to it. One consequence is that $\mathcal{S}^\dagger$ now contains only those constants that appear in $\mathcal{S}$. In addition, the following result is straightforward.

Lemma 5.2 *Let $\mathcal{S}$ be a set of Datalog rules with substitution, and let $\mathcal{C}$ be a set of constant symbols. Then*

$$inst_{\mathcal{C}}[(inst_{\mathcal{C}}(\mathcal{S}))^\dagger] = inst_{\mathcal{C}}(\mathcal{S}^\dagger)$$

Proof: This lemma says that $\mathcal{S}^\dagger$ and $(inst_{\mathcal{C}}(\mathcal{S}))^\dagger$ have identical instantiations. This is a straightforward consequence of Definitions 5.1 and 2.4. For instance, in Definition 2.4, the rules contributed by items *1*, *3* and *4* have the same instantiations whether we are defining $\mathcal{S}^\dagger$ or $(inst_{\mathcal{C}}(\mathcal{S}))^\dagger$. Item *2* is more subtle. Here, the lemma intuitively says that to instantiate the rule $(\beta \leftarrow \beta_1 ... \beta_n)[\theta]$, we can first instantiate the variables in $\beta \leftarrow \beta_1 ... \beta_n$, and then instantiate the variables in $[\theta]$. This is possible since by the above discussion, we can assume that the two sets of variables are disjoint. $\Box$

Lemma 5.3 *Let $\mathcal{S}$ be a set of Datalog rules with substitution, let DB be a database, and let β be a ground predicate with substitution. In addition, let $\mathcal{C}$ be a set of constant symbols including all constants that appear in $\mathcal{S}$, DB and β. Then,*

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

Proof: By the discussion preceding Lemma 5.2, we can assume that $\mathcal{S}$ and $\mathcal{S}^\dagger$ both contain the same constant symbols. Thus, the constants in $\mathcal{S}^\dagger$ are all in $\mathcal{C}$. Likewise for $(inst_{\mathcal{C}}(\mathcal{S}))^\dagger$. Keeping this in mind,

	$\mathcal{S}, DB \vdash \beta$	
iff	$\mathcal{S}^\dagger \cup DB \vdash_c \beta$	by Definition 2.5,
iff	$inst_{\mathcal{C}}(\mathcal{S}^\dagger) \cup DB \vdash_c \beta$	by statement (15), using $R = \mathcal{S}^\dagger$,
iff	$inst_{\mathcal{C}}[(inst_{\mathcal{C}}(\mathcal{S}))^\dagger] \cup DB \vdash_c \beta$	by Lemma 5.2,
iff	$(inst_{\mathcal{C}}(\mathcal{S}))^\dagger \cup DB \vdash_c \beta$	by statement (15), using $R = (inst_{\mathcal{C}}(\mathcal{S}))^\dagger$,
iff	$inst_{\mathcal{C}}(\mathcal{S}), DB \vdash \beta$	by Definition 2.5.

□

Lemma 5.3 implies that we can use $inst_{\mathcal{C}}(\mathcal{S})$ instead of $\mathcal{S}$ to answer queries. Thus, without loss of generality, we shall assume in the rest of this section that $\mathcal{S}$ is ground. This assumption allows us to simplify the operator $T_{\mathcal{S}}$. For instance, the term substitution, σ, in Definition 4.3 now has no effect, since α and β_i are ground. We can therefore remove σ from the definition of $T_{\mathcal{S}}$, rewriting the definition as follows when $\mathcal{S}$ is ground:

$$\begin{aligned} T_{\mathcal{S}}(R) \quad = \quad \{\alpha \leftarrow \gamma \quad | \quad & \textit{for some rule } \alpha \leftarrow \beta_1 ... \beta_m \textit{ in } \mathcal{S}, \\ & R^{\dagger} \models_c \beta_i \leftarrow \gamma \ \ \textit{for all } 1 \leq i \leq m\ \} \end{aligned} \tag{16}$$

Note that although α and each β_i are ground (because they come from $\mathcal{S}$), γ is not necessarily ground. Thus, a Datalog rulebase with substitution that is ground will have a minimal model that is non-ground. The following subsections deal with this issue.

5.2 A Ground Version of the Minimal Model

Statement (14) says that instead of using a rulebase with substitution to answer queries, we can use its minimal model. This subsection shows that we do not even need that. Instead, we only need the ground instantiation of the minimal model. Section 6 will show that this instantiation is a finite set of ground Datalog rules of exponential size. Unlike the minimal model, though, the rules in the ground instantiation depend on the database, and change as the database changes.

Even if a Datalog rulebase with substitution is ground and finite, its minimal model may be non-ground and infinite. This is because there are infinitely many values for γ in statement (16) above. γ is a finite set of atomic formulas, and there are infinitely many such sets. In fact, there are infinitely many atomic formulas, constructed from an infinite set of constants and variables. These constants and variables are introduced into $T_{\mathcal{S}}(R)$ through γ in statement (16), and thence into the minimal model, $T^*_{\mathcal{S}}(\{\})$. Fortunately, most of the constants are unnecessary and can be replaced by variables, as Lemma 5.4 and Corollary 5.5 show below. Corollary 5.6 then shows that the variables can be replaced by constants, taken from a finite set.

Lemma 5.4 *Let $\mathcal{S}$ be a set of ground Datalog rules with predicate substitution, and let R be a set of classical Datalog rules. Then, without affecting logical equivalence, we can assume that every constant symbol in $T_{\mathcal{S}}(R)$ is in $\mathcal{S}$ or in R.*

Proof: Let c be a "new" constant, that is, a constant not in $\mathcal{S}$ or R. Also, let $\alpha \leftarrow \gamma$ be a rule that does not contain any new constants, but which does contain a variable, x. Then by substituting c for x, we get a new rule, $(\alpha \leftarrow \gamma)\{x/c\}$, which *does* contain a new constant. All rules containing new constants can be derived in this way, i.e., as specializations of rules that do not contain new constants. We shall show that if a rule like $(\alpha \leftarrow \gamma)\{x/c\}$ is in $T_{\mathcal{S}}(R)$, then so is $\alpha \leftarrow \gamma$. Thus, the former rule can be removed from $T_{\mathcal{S}}(R)$ without affecting logical equivalence.

Using the notation of statement (16), let $\alpha \leftarrow \beta_1 ... \beta_m$ be an arbitrary rule in $\mathcal{S}$. This rule gives rise to a set of rules in $T_{\mathcal{S}}(R)$ of the form $\alpha \leftarrow \gamma$, where γ is a finite set of atomic formulas. Consider a γ that has no new constants, but which has a variable, x. Observe that if c is a new constant, then c is not in β_i or γ. Hence,

	$R^\dagger \models_c \beta_i \leftarrow \gamma$	
iff	$R^\dagger \models_c \forall_x(\beta_i \leftarrow \gamma)$	since x is universally quantified, implicitly,
iff	$R^\dagger \models_c (\beta_i \leftarrow \gamma)\{x/c\}$	by observation (9) in Section 3.1.
iff	$R^\dagger \models_c \beta_i \leftarrow (\gamma\{x/c\})$	since β_i is ground.

Consequently,

	$\alpha \leftarrow \gamma \in T_{\mathcal{S}}(R)$	
iff	$\alpha \leftarrow (\gamma\{x/c\}) \in T_{\mathcal{S}}(R)$	by (16),
iff	$(\alpha \leftarrow \gamma)\{x/c\} \in T_{\mathcal{S}}(R)$	since α is ground.

But the last rule, $(\alpha \leftarrow \gamma)\{x/c\}$, is a mere specialization of the first, $\alpha \leftarrow \gamma$. Thus, we can remove it from $T_{\mathcal{S}}(R)$ without affecting logical equivalence. Likewise we can remove *any* rule from $T_{\mathcal{S}}(R)$ that contains constants not in $\mathcal{S}$ or R. $\Box$

Corollary 5.5 *Let $\mathcal{S}$ be a set of ground Datalog rules with substitution. Then, without affecting logical equivalence, we can assume that the constants in $T^i_{\mathcal{S}}(\{\})$ or in $T^*_{\mathcal{S}}(\{\})$ are all in $\mathcal{S}$.*

Proof: For $T^i_{\mathcal{S}}(\{\})$, the result follows by Lemma 5.4 and a simple induction on i. The result for $T^*_{\mathcal{S}}(\{\})$ then follows immediately from Definition 4.11. $\Box$

Corollary 5.6 *Let $\mathcal{S}$ be a set of ground Datalog rules with substitution, let DB be a database, and let β be a ground predicate with substitution. In addition, let $\mathcal{C}$ be a set of constant symbols including all constants that appear in $\mathcal{S}$, DB and β. Then,*

$$\mathcal{S}, DB \models \beta \quad \textit{iff} \quad inst_{\mathcal{C}}[T^*_{\mathcal{S}}(\{\})], DB \models \beta$$

Proof:

	$\mathcal{S}, DB \models \beta$	
iff	$T^*_{\mathcal{S}}(\{\}), DB \models \beta$	by statement (14),
iff	$(T^*_{\mathcal{S}}(\{\}))^\dagger \cup DB \vdash_c \beta$	by Definition 3.2, using $T^*_{\mathcal{S}}(\{\})$ in place of R,
iff	$T^*_{\mathcal{S}}(\{\}), DB \vdash \beta$	by Definition 2.5, using $T^*_{\mathcal{S}}(\{\})$ in place of $\mathcal{S}$,
iff	$inst_{\mathcal{C}}(T^*_{\mathcal{S}}(\{\})), DB \vdash \beta$	by Lemma 5.3, using $T^*_{\mathcal{S}}(\{\})$ in place of $\mathcal{S}$,
iff	$[inst_{\mathcal{C}}(T^*_{\mathcal{S}}(\{\}))]^\dagger \cup DB \vdash_c \beta$	by Definition 2.5, using $inst_{\mathcal{C}}(T^*_{\mathcal{S}}(\{\}))$ in place of $\mathcal{S}$,
iff	$inst_{\mathcal{C}}(T^*_{\mathcal{S}}(\{\})), DB \models \beta$	by Definition 3.2, using $inst_{\mathcal{C}}(T^*_{\mathcal{S}}(\{\}))$ in place of R.

$\Box$

Observe that the proof of Corollary 5.6 uses turnstyles with three different meanings: entailment, satisfaction, and inference. In general, a single turnstyle, $\vdash$, denotes logical inference; a double turnstyle, $\models$, involving $\mathcal{S}$ denotes logical entailment; and a double turnstyle involving $T^*_{\mathcal{S}}(\{\})$ denotes logical satisfaction.

5.3 A Ground Version of the T-operator

The previous subsection showed that we can answer queries by using the ground instantiation of the minimal model, instead of the minimal model itself. However, we still need to construct this instantiated model, without constructing the entire model first. This subsection solves this problem by developing a new T-operator, $T_{\mathcal{S},\mathcal{C}}$, that can be thought of as a ground version of the original T-operator, $T_{\mathcal{S}}$. The least fixpoint of the new operator is precisely the ground instantiation of the least fixpoint of the original operator. We can therefore use the new operator to construct the instantiated fixpoint directly. Unlike the original operator, the new operator has the property that if its input is ground and finite, then so is its output. The new operator can thus be used in computations.

Definition 5.7 *If $\mathcal{S}$ is a set of ground Datalog rules with substitution, and $\mathcal{C}$ is a set of constant symbols, then $T_{\mathcal{S},\mathcal{C}}$ is a mapping from Datalog rulebases to Datalog rulebases. In particular, for each Datalog rulebase, R,*

$$\begin{aligned} T_{\mathcal{S},\mathcal{C}}(R) \quad = \quad \{\alpha \leftarrow \gamma \quad | \quad & \textit{for some rule } \alpha \leftarrow \beta_1 ... \beta_m \textit{ in } \mathcal{S}, \\ & R^\dagger \models_c \beta_i \leftarrow \gamma \quad \textit{for all } 1 \leq i \leq m \ \} \end{aligned}$$

where each γ is a finite set of ground atomic formulas whose constants are in $\mathcal{C}$.

It is the restriction that γ be ground that distinguishes $T_{\mathcal{S},\mathcal{C}}$ from $T_{\mathcal{S}}$. The following lemma states a basic relationship between these two operators. It motivates our description of $T_{\mathcal{S},\mathcal{C}}$ as a "ground version" of $T_{\mathcal{S}}$.

Lemma 5.8 *Let $\mathcal{S}$ be a set of ground Datalog rules with substitution, and let R be a set of classical Datalog rules (not necessarily ground). If $\mathcal{C}$ contains all the constants in $\mathcal{S}$ and R, then*

$$T_{\mathcal{S},\mathcal{C}}(R) \ = \ inst_{\mathcal{C}}[T_{\mathcal{S}}(R)]$$

Proof: Clearly $T_{\mathcal{S},\mathcal{C}}(R)$ is ground and $T_{\mathcal{S},\mathcal{C}}(R) \subseteq T_{\mathcal{S}}(R)$. Thus $T_{\mathcal{S},\mathcal{C}}(R) = inst_{\mathcal{C}}[T_{\mathcal{S},\mathcal{C}}(R)] \subseteq inst_{\mathcal{C}}[T_{\mathcal{S}}(R)]$. To prove the reverse containment, first suppose that $\alpha \leftarrow \gamma$ is a rule in $T_{\mathcal{S}}(R)$. Then by Lemma 5.4, we can assume that γ only contains constants in $\mathcal{S}$ or R, *i.e.*, constants in $\mathcal{C}$. Thus, if $\alpha \leftarrow \gamma$ is a rule in $inst_{\mathcal{C}}[T_{\mathcal{S}}(R)]$, then γ is a finite set of ground atomic formulas whose constants are in $\mathcal{C}$. It therefore satisfies the restrictions given in Definition 5.7. Thus $\alpha \leftarrow \gamma$ is a rule in $T_{\mathcal{S},\mathcal{C}}(R)$. Hence $inst_{\mathcal{C}}[T_{\mathcal{S}}(R)] \subseteq T_{\mathcal{S},\mathcal{C}}(R)$. $\Box$

Observe that under the assumptions of Lemma 5.8, $T_{\mathcal{S},\mathcal{C}}$ has the following closure property: if R is a set of ground Datalog rules whose constants are in $\mathcal{C}$, then so is $T_{\mathcal{S},\mathcal{C}}(R)$. Using this property, we show in a series of lemmas that $T_{\mathcal{S},\mathcal{C}}$ can be used to compute the ground instantiation of the minimal model of $\mathcal{S}$, and to answer queries.

Lemma 5.9 *Let $\mathcal{S}$ be a set of ground Datalog rules with substitution, let R be a set of classical Datalog rules (not necessarily ground), and let $\mathcal{C}$ be a set of constant symbols including all constants that appear in $\mathcal{S}$ or R. Then,*

$$T_{\mathcal{S},\mathcal{C}}(R) = T_{\mathcal{S},\mathcal{C}}[inst_{\mathcal{C}}(R)]$$

Proof: Using the notation of Definition 5.7, let β_i be a ground predicate with substitution whose constants are in $\mathcal{C}$, and let γ be a finite set of ground atomic formulas whose constants are in $\mathcal{C}$. Then,

$$
\begin{array}{lll}
 & R^\dagger \models_c \beta_i \leftarrow \gamma & \\
\textit{iff} & R^\dagger \cup \gamma \vdash_c \beta_i & \text{by the Deduction Theorem,} \\
\textit{iff} & R, \gamma \vdash \beta_i & \text{by Definition 2.5, treating } \gamma \text{ as a database,} \\
\textit{iff} & inst_{\mathcal{C}}(R), \gamma \vdash \beta_i & \text{by Lemma 5.3, using } R \text{ in place of } \mathcal{S}, \\
\textit{iff} & [inst_{\mathcal{C}}(R)]^\dagger \cup \gamma \vdash_c \beta_i & \text{by Definition 2.5,} \\
\textit{iff} & [inst_{\mathcal{C}}(R)]^\dagger \models_c \beta_i \leftarrow \gamma & \text{by the Deduction Theorem.}
\end{array}
$$

Hence, $T_{\mathcal{S},\mathcal{C}}(R) = T_{\mathcal{S},\mathcal{C}}[inst_{\mathcal{C}}(R)]$, by Definition 5.7. □

We can now generalize Lemma 5.8.

Lemma 5.10 *Let $\mathcal{S}$ be a set of ground Datalog rules with substitution, and let $\mathcal{C}$ be a set of constant symbols, including all constants in $\mathcal{S}$. Then,*

$$
\begin{aligned}
T^i_{\mathcal{S},\mathcal{C}}(\{\}) &= inst_{\mathcal{C}}[T^i_{\mathcal{S}}(\{\})] && (17) \\
T^*_{\mathcal{S},\mathcal{C}}(\{\}) &= inst_{\mathcal{C}}[T^*_{\mathcal{S}}(\{\})] && (18)
\end{aligned}
$$

Proof: We prove statement (17) by induction. The basis step is easy:

$$T^0_{\mathcal{S},\mathcal{C}}(\{\}) = \{\} = inst_{\mathcal{C}}[\{\}] = inst_{\mathcal{C}}[T^0(\{\})]$$

The inductive step follows from Lemma 5.9:

$$
\begin{array}{rcll}
T^{i+1}_{\mathcal{S},\mathcal{C}}(\{\}) & = & T_{\mathcal{S},\mathcal{C}}[T^i_{\mathcal{S},\mathcal{C}}(\{\})] & \text{by definition,} \\
 & = & T_{\mathcal{S},\mathcal{C}}[inst_{\mathcal{C}}(T^i_{\mathcal{S}}(\{\}))] & \text{by inductive hypothesis,} \\
 & = & T_{\mathcal{S},\mathcal{C}}[T^i_{\mathcal{S}}(\{\})] & \text{by Lemma 5.9, with } R = T^i_{\mathcal{S}}(\{\}), \\
 & = & inst_{\mathcal{C}}[T_{\mathcal{S}}[T^i_{\mathcal{S}}(\{\})]] & \text{by Lemma 5.8, with } R = T^i_{\mathcal{S}}(\{\}), \\
 & = & inst_{\mathcal{C}}[T^{i+1}_{\mathcal{S}}(\{\})] & \text{by definition.}
\end{array}
$$

Statement (18) is now a straightforward corollary:

$$
\begin{array}{rcll}
inst_{\mathcal{C}}[T^*_{\mathcal{S}}(\{\})] & = & inst_{\mathcal{C}}[\cup_i T^i_{\mathcal{S}}(\{\})] & \text{by definition,} \\
 & = & \cup_i \, inst_{\mathcal{C}}[T^i_{\mathcal{S}}(\{\})] & \\
 & = & \cup_i T^i_{\mathcal{S},\mathcal{C}}(\{\}) & \text{by statement (17),} \\
 & = & T^*_{\mathcal{S},\mathcal{C}}(\{\}) & \text{by definition.}
\end{array}
$$

□

Like $T_{\mathcal{S}}$, it is not hard to show that $T_{\mathcal{S},\mathcal{C}}$ is monotonic and continuous. Thus, it has a least fixpoint, and this fixpoint is precisely $T^*_{\mathcal{S},\mathcal{C}}(\{\})$. It follows from Lemma 5.10 that the least fixpoint of $T_{\mathcal{S},\mathcal{C}}$ is just the ground instantiation of the least fixpoint of $T_{\mathcal{S}}$. Using this idea, the following theorem shows that for given a database, the least fixpoint of $T^*_{\mathcal{S},\mathcal{C}}$ can be used to answer queries about $\mathcal{S}$. This theorem is the main result of this section, and the starting point for the next section.

Theorem 5.11 *Let $\mathcal{S}$ be a set of ground Datalog rules with substitution, let DB be a database, and let β be a ground predicate with substitution. In addition, let $\mathcal{C}$ be a set of constant symbols including all constants that appear in $\mathcal{S}$, DB or β. Then,*

$$\mathcal{S}, DB \models \beta \quad \textit{iff} \quad T^*_{\mathcal{S},\mathcal{C}}(\{\}), DB \models \beta$$

Proof: $\mathcal{S}, DB \models \beta$ iff $inst_{\mathcal{C}}[T^*_{\mathcal{S}}(\{\})], DB \models \beta$, by Corollary 5.6,
iff $T^*_{\mathcal{S},\mathcal{C}}(\{\}), DB \models \beta$, by Lemma 5.10 □

6 Upper Complexity Bounds

This section shows that the data complexity of Datalog with predicate substitution is bounded above by EXPTIME. The proof relies on the simplified fixpoint theory developed in Section 5. The next section will show that EXPTIME is also a lower complexity bound. Datalog with predicate substitution is therefore EXPTIME-complete. For the purpose of this paper, EXPTIME = DTIME[$2^{n^{O(1)}}$].

Theorem 6.1 *The data complexity of Datalog with predicate substitution is in EXPTIME.*

To prove Theorem 6.1, suppose that $\mathcal{S}$ is a finite set of Datalog rules with substitution, DB is a database, and β is a ground predicate with substitution. Let $\mathcal{C}$ be any finite set of constant symbols, including all the constants in $\mathcal{S}$, DB and β. We show that for each $\mathcal{S}$, there is a procedure that takes DB and $\mathcal{C}$ as input, and determines whether $\mathcal{S}, DB \models \beta$ in $2^{n^{O(1)}}$ time, where n is the size of $\mathcal{C}$.

By Lemma 5.3, we can use $inst_{\mathcal{C}}(\mathcal{S})$ instead instead of $\mathcal{S}$. This ground instantiation can be computed in polynomial time. To see this, let q be the maximum number of distinct variables in any rule in $\mathcal{S}$. Each rule in $\mathcal{S}$ then has $O(n^q)$ ground instantiations. Thus, $inst_{\mathcal{C}}(\mathcal{S})$ has size $|\mathcal{S}|\, O(n^q)$, or simply $O(n^q)$. Thus, without loss of generality, we shall assume throughout this section that $\mathcal{S}$ is ground. The operator $T_{\mathcal{S},\mathcal{C}}$ is therefore well defined. Our proof strategy is to show that the least fixpoint of this operator can be computed in $2^{n^{O(1)}}$ time. According to Definition 5.7, the main task in applying this operator to a Datalog rulebase is to infer expressions of the form $R^\dagger \vdash_c \beta_j \leftarrow \gamma$. We therefore spend some time studying the complexity of such inferences.

Because we are dealing with data complexity, we can assume that our language contains only finitely many predicate symbols, each of a fixed arity [4, 19]. In this section, m will usually denote the number of distinct predicate symbols, and k will usually denote an upper bound on their arity.

6.1 Inference from $R^\dagger$

In this section, R is a finite set of classical Datalog rules, and we need to infer the logical consequences of $R^\dagger$. The main problem is that $R^\dagger$ is infinite. However, this subsection shows that we need only use a specific finite subset of $R^\dagger$. The key idea is the *degree* of a predicate with substitution.

Definition 6.2 (Degree of Substitution) *If α is an atomic formula, and $[\theta_1]...[\theta_i]$ are predicate substitutions, then $\alpha[\theta_1]...[\theta_i]$ is a predicate with substitution of* degree i.

Thus, the atom A has degree 0, the predicate $A[P/Q]$ has degree 1, the predicate $A[P/Q][P/Q]$ has degree 2, etc.

In general, a rulebase with predicate substitution, $\mathcal{S}$, may infer predicates of degree i from predicates of degree *greater* than i. For example, the rule $A \leftarrow B[P/Q]$ infers a predicate of degree 0 from a predicate of degree 1. Moreover, this rule gives rise to rules like $A[R/S] \leftarrow B[P/Q][R/S]$ in $\mathcal{S}^\dagger$, which infers a predicate of degree 1 from a predicate of degree 2. This is not true for Horn rules, however, as the following lemma shows. This lemma is an immediate consequence of Definition 2.4.

Lemma 6.3 *Let R be a set of classical Datalog rules. Then, in each rule of $R^\dagger$, the degree of each predicate in the body does not exceed the degree of the head predicate.*

$R^\dagger$ is an infinite rule-set, because it defines predicates of unbounded degree, such as $A[P/Q]$, $A[P/Q][P/Q]$, $A[P/Q][P/Q][P/Q]$, etc. To infer predicates of degree i, however, we need only use a finite subset of $R^\dagger$, as the following definition and the subsequent lemma show.

Definition 6.4 *Let $\mathcal{S}$ be a set of Datalog rules with predicate substitution. The restriction of $\mathcal{S}^\dagger$ to degree i, written $\mathcal{S}^\dagger|_i$, is the set of rules in $\mathcal{S}^\dagger$ whose heads are predicates of degree at most i.*

Lemma 6.5 *Let R be a set of classical Datalog rules, let DB be a database, and let β be a ground predicate with substitution of degree at most i. Then $R^\dagger \cup DB \vdash_c \beta$ iff $R^\dagger|_i \cup DB \vdash_c \beta$.*

Proof: In this lemma, we are treating each predicate with substitution as an ordinary predicate of classical logic, and we are treating $R^\dagger$ and $R^\dagger|_i$ as classical Datalog rulebases. The *if* direction is therefore trivial since $R^\dagger|_i \subseteq R^\dagger$. To prove the *only if* direction, we use the classical Horn T-operator[5] of the rulebase $R^\dagger \cup DB$. For brevity, we write T^N instead of $T^N_{\mathcal{R}^\dagger \cup DB}(\{\})$. Recall that $R^\dagger \cup DB \vdash_c \beta$ iff $\beta \in T^N$ for some N. We show that if $\beta \in T^N$ then $R^\dagger|_i \cup DB + \vdash_c \beta$. The proof is by induction on N. In this induction, $\mathcal{C}$ is any set of constant symbols containing all the constants in $R^\dagger$, DB and β.

Basis: Suppose that $\beta \in T^0$. Then β is a ground instance of a unit rule in $R^\dagger \cup DB$. Thus, either $\beta \in DB$ or $\beta \in inst_\mathcal{C}(R^\dagger)$. In the latter case, β is also in $inst_\mathcal{C}(R^\dagger|_i)$, since β has degree at most i. In either case, $\beta \in inst_\mathcal{C}(R^\dagger|_i) \cup DB$, and so $R^\dagger|_i \cup DB \vdash_c \beta$.

Induction: Suppose the result is true for all ground predicates in T^N of degree at most i. Suppose also that $\beta \in T^{N+1}$. Then there is a rule in $inst_\mathcal{C}(R^\dagger)$ that derives β from T^N in one step. This rule has the form $\beta \leftarrow \beta_1...\beta_m$, where each $\beta_j \in T^N$. Since β has degree at most i, so does each β_j, by Lemma 6.3. Thus by induction hypothesis, $R^\dagger|_i \cup DB \vdash_c \beta_j$ for $1 \leq j \leq m$. Thus $inst_\mathcal{C}(R^\dagger|_i) \cup DB \vdash_c \beta_j$, by statement (15) in

[5] not to be confused with the T-operator developed in Section 4 for predicate substitution.

Section 5. Since the rule $\beta \leftarrow \beta_1 ... \beta_m$ is in $inst_{\mathcal{C}}(R^{\dagger})$, it is also in $inst_{\mathcal{C}}(R^{\dagger}|_i)$, since the head predicate, β, has degree at most i. We thus have the following:

$$\begin{array}{ll} & inst_{\mathcal{C}}(R^{\dagger}|_i) \cup DB \ \vdash_c \ \beta_j \qquad \text{for } 1 \leq j \leq m \\ \textit{and} & \beta \leftarrow \beta_1 ... \beta_m \ \in \ inst_{\mathcal{C}}(R^{\dagger}|_i) \\ \textit{hence} & inst_{\mathcal{C}}(R^{\dagger}|_i) \cup DB \ \vdash_c \ \beta \end{array}$$

□

6.2 The Complexity of Inference from $R^{\dagger}$

This section shows that classical inference from $R^{\dagger}$ can be done in exponential time (exponential in the size of the data domain). The key steps are showing that the ground instantiation of $R^{\dagger}|_i$ has exponential size, and that it can be constructed in exponential time.

Lemma 6.6 *Let R be a finite set of classical Datalog rules, and let $\mathcal{C}$ be a finite set of constant symbols, including all constants in R. Then, there are at most $a2^{(bn^{ik})}$ rules in $inst_{\mathcal{C}}(R^{\dagger}|_{i-1})$, where n is the cardinality of $\mathcal{C}$, k is an upper bound on the arity of predicate symbols, and a and b are constants independent of $\mathcal{C}$ and R.*

Proof: Let m be the number of predicate symbols in our language. We bound the size of several sets, as follows:

- Each atomic formula in $inst_{\mathcal{C}}(R^{\dagger}|_{i-1})$ is ground and has the form $P(x_1 ... x_p)$, where $p \leq k$. There are m choices for P and at most n choices for each of the x's, for a total of at most mn^p atomic formulas, or at most mn^k.

- According to Definition 2.2, a predicate substitution has the form $[P/Q](y_1 ... y_p)$, where p is the difference in the arities of Q and P. Thus p is no bigger than the maximum arity of any predicate symbol, *i.e.*, $p \leq k$. To bound the number ground predicate substitutions, note that there are m choices for P, m choices for Q, and at most n choices for each of the y's, for a total of at most $m^2 n^p$ choices, or at most $m^2 n^k$.

- Each predicate with substitution in $inst_{\mathcal{C}}(R^{\dagger}|_{i-1})$ is ground and has the form $\alpha[\theta_1][\theta_2]...[\theta_{i-1}]$, where α is a ground atomic formula. As shown above, there are at most mn^k choices for α, and at most $m^2 n^k$ choices for each of the predicate substitutions, $[\theta_j]$, for a total of at most $mn^k (m^2 n^k)^{i-1}$ choices, that is, $m^{(2i-1)} n^{ik}$. Thus, there are at most an^{ik} ground predicates with substitution, where $a = m^{(2i-1)}$.

- Each rule in $inst_{\mathcal{C}}(R^{\dagger}|_{i-1})$ consists of a head and a body. The head is a ground predicate with substitution, of which there are at most an^{ik} choices. The body is a *set* of such predicates, of which there are at most $2^{(an^{ik})}$ choices. Thus, the number of rules in $inst_{\mathcal{C}}(R^{\dagger}|_{i-1})$ is at most:

$$an^{ik} 2^{(an^{ik})} \ \leq \ a2^{n^{ik}} 2^{(an^{ik})} \ = \ a2^{[(a+1)n^{ik}]} \ = \ a2^{(bn^{ik})}$$

where $b = a + 1$ and $a = m^{(2i-1)}$. Note that a and b are independent of R and $\mathcal{C}$.

□

Corollary 6.7 *Let R be a finite set of ground Datalog rules. Then there are at most $a2^{(bn^k)}$ rules in R, where n is the number of distinct constant symbols in R, k is an upper bound on the arity of predicate symbols, and a and b are constants independent of R.*

Proof: Because R is Horn, every rule in R has degree 0, so $R = R^\dagger|_0$. Moreover, since R is ground, $R = inst_{\mathcal{C}}(R) = inst_{\mathcal{C}}(R^\dagger|_0)$, where $\mathcal{C}$ is the set of constant symbols in R. Thus, by Lemma 6.6, there are at most $a2^{(bn^k)}$ rules in R. □

Lemma 6.8 *Let R be a finite set of classical Datalog rules, and let $\mathcal{C}$ be a finite set of constant symbols, including all constants in R. Then $inst_{\mathcal{C}}(R^\dagger|_{i-1})$ can be constructed in $2^{O(n^{ik})}$ time, where n is the cardinality of $\mathcal{C}$, and k is an upper bound on the arity of predicate symbols.*

Proof: Referring to Definition 2.4, $inst_{\mathcal{C}}(R^\dagger|_{i-1})$ can be constructed by the following procedure:

1. $R_0 := inst_{\mathcal{C}}(R)$
2. For j from 1 to $i-1$ do $R_j := \{\}$
3. Add the following rules to R_1:
   - $P(\overline{a})[P/Q](\overline{b}) \leftarrow Q(\overline{a}, \overline{b})$ for every ground predicate substitution $[P/Q](\overline{b})$, and every ground atomic formula $P(\overline{a})$.
   - $B(\overline{a})[P/Q](\overline{b}) \leftarrow B(\overline{a})$, for every ground predicate substitution $[P/Q](\overline{b})$, and every ground atomic formula $B(\overline{a})$, where B is a base predicate and $B \neq P$.
4. For j from 1 to $i-1$ do the following:

   Add the rule $\beta[\theta] \leftarrow \beta_1[\theta]...\beta_n[\theta]$ to R_j, for each rule $\beta \leftarrow \beta_1...\beta_n$ in R_{j-1}, and each ground predicate substitution $[\theta]$.[6]
5. Return $R_0 \cup R_1 \cup ... \cup R_{i-1}$.

To see that this procedure is correct, observe that because R is Horn, every rule in R has degree 0, so $R = R^\dagger|_0$. Thus $R_0 = inst_{\mathcal{C}}(R^\dagger|_0) = inst_{\mathcal{C}}(R^\dagger)|_0$, so R_0 consists of those rules in $inst_{\mathcal{C}}(R^\dagger)$ of degree *exactly* 0. Likewise, for $j > 0$, R_j consists of those rules in $inst_{\mathcal{C}}(R^\dagger)$ of degree *exactly* j. The procedure thus returns all rules in $inst_{\mathcal{C}}(R^\dagger)$ of degree at most $i-1$. *i.e.*, The procedure returns $inst_{\mathcal{C}}(R^\dagger)|_{i-1} = inst_{\mathcal{C}}(R^\dagger|_{i-1})$.

To bound the running time of the procedure, observe that the procedure constructs each rule in $inst_{\mathcal{C}}(R^\dagger|_{i-1})$ exactly once. By Lemma 6.6, $inst_{\mathcal{C}}(R^\dagger|_{i-1})$ contains at most $a2^{bn^{ik}}$ rules. The procedure thus runs in $2^{O(n^{ik})}$ time. □

[6] $[\theta]$ denotes *any* ground predicate substitution, *i.e.*, substitutions of the form $[P/Q](b_1...b_k)$, as well as $[P/Q]$.

Lemma 6.9 *Let R be a finite set of classical Datalog rules, let DB be a database, let β be a ground predicate with substitution of degree at most $i-1$, and let $\mathcal{C}$ be a finite set of constant symbols, including all constants in R, DB and β. Then, there is a procedure that takes R, DB, β and $\mathcal{C}$ as input, and determines whether $R^\dagger \cup DB \vdash_c \beta$. Furthermore, this procedure runs in $2^{O(n^{ik})}$ time, where n is the cardinality of $\mathcal{C}$, and k is an upper bound on the arity of predicate symbols.*

Proof: The main problem is that $R^\dagger$ is infinite and contains variables. However, by Lemma 6.5, we can use $R^\dagger|_{i-1}$ instead. Moreover, by the discussion preceding Lemma 5.2, we can assume that every constant in $R^\dagger$ is in R, and thus in $\mathcal{C}$. The same is therefore true of $R^\dagger|_{i-1}$. Thus,

$$\begin{array}{lll} & R^\dagger \cup DB \vdash_c \beta & \\ \textit{iff} & R^\dagger|_{i-1} \cup DB \vdash_c \beta & \text{by Lemma 6.5,} \\ \textit{iff} & inst_{\mathcal{C}}(R^\dagger|_{i-1}) \cup DB \vdash_c \beta & \text{by statement (15) in Section 5.} \end{array}$$

We shall focus on the last of these three equivalent statements. First note that $inst_{\mathcal{C}}(R^\dagger|_{i-1})$ can be constructed in $2^{O(n^{ik})}$ time, by Lemma 6.8. Second, if we treat each ground predicate with substitution as a propositional atom, then inference becomes propositional. In general, inference from propositional Horn rulebases takes time that is polynomial in the size of the rulebase. That is, there is a procedure that takes a set of propositional Horn rules as input, and returns the set of propositional atoms entailed by the rules as output. Furthermore, this procedure runs in time that is polynomial in the size of the rule-set,[7] where size is defined as the number of rules in the rulebase plus the number of distinct atoms. We can easily bound this size. The proof of Lemma 6.6 shows that DB has at most mn^k ground atomic formulas, and that $inst_{\mathcal{C}}(R|_{i-1})$ has at most $a2^{bn^{ik}}$ rules, made from at most an^{ik} ground predicates with substitution. Here, a, b and m are constants that are independent of the inputs, R, DB, β and $\mathcal{C}$. Thus, the size of the rule-set $inst_{\mathcal{C}}(R|_{i-1}) \cup DB$ is at most $mn^k + an^{ik} + a2^{bn^{ik}}$, or simply $2^{O(n^{ik})}$, since $i \geq 1$ and $k \geq 0$. The inference procedure runs in time that is polynomial in this size, *i.e.*, in time $[2^{O(n^{ik})}]^{O(1)} = 2^{[O(n^{ik})O(1)]} = 2^{O(n^{ik})}$. □

6.3 The Complexity of Inference from $\mathcal{S}^\dagger$

This subsection uses the above results to prove Theorem 6.1. As in the rest of this section, we assume that $\mathcal{S}$ is a finite set of Datalog rules with substitution, that DB is a database, and that β is a ground predicate with substitution. In addition, $\mathcal{C}$ is a finite set of constant symbols, including all the constants in $\mathcal{S}$, DB and β. We show that for each $\mathcal{S}$, there is a procedure that takes DB and $\mathcal{C}$ as input, and determines whether $\mathcal{S}, DB \models \beta$ in $2^{n^{O(1)}}$ time, where n is the size of $\mathcal{C}$. As mentioned in the discussion following Theorem 6.1, we shall assume without loss of generality that $\mathcal{S}$ is ground. The operator $T_{\mathcal{S},\mathcal{C}}$ is therefore well defined. The main problem is to show that $T^*_{\mathcal{S},\mathcal{C}}(\{\})$ can be computed in exponential time. For convenience, we use T^m as an abbreviation for $T^m_{\mathcal{S},\mathcal{C}}(\{\})$.

We first show that each application of $T_{\mathcal{S},\mathcal{C}}$ takes exponential time. To see this, let k be the maximum arity of any predicate symbol in our language, and let $i-1$ be the maximum degree of any predicate with substitution in any rule in $\mathcal{S}$. From Definition 5.7,

[7] Ullman, for instance, describes an algorithm due to Bernstein that takes linear time [17, 2].

the main task in computing $T_{\mathcal{S},\mathcal{C}}(R)$ is inferring expressions of the form $R^\dagger \vdash_c \beta_j \leftarrow \gamma$. By the Deduction Theorem of classical logic, $R^\dagger \vdash_c \beta_j \leftarrow \gamma$ iff $R^\dagger \cup \gamma \vdash_c \beta_j$, since γ is ground; and by Lemma 6.9, this latter inference takes $2^{O(n^{ik})}$ time. It follows in a straightforward way that $T_{\mathcal{S},\mathcal{C}}(R)$ can be computed in $2^{n^{O(1)}}$ time. This is true for any finite set of classical Datalog rules, R, whose constant symbols are all in $\mathcal{C}$. In particular, it is true for $R = T^m$. Thus T^{m+1} can be computed from T^m in $2^{n^{O(1)}}$ time.

Starting from the empty set, we repeatedly apply the T-operator until saturation, that is, until $T^m = T^{m+1}$, at which point $T^m = T^*$. Because the operator is monotonic, $T^m \subseteq T^{m+1}$. Thus, at every step except the last, T^{m+1} has at least one more rule than T^m. But T^m is a finite set of ground Datalog rules whose constants are in $\mathcal{C}$. Thus by Corollary 6.7, it contains at most $a2^{(bn^k)}$ rules. Saturation will thus be reached after at most $a2^{(bn^k)}$ steps. Computing T^* thus takes $2^{n^{O(1)}}$ steps, each taking $2^{n^{O(1)}}$ time, so the entire computation takes $2^{n^{O(1)}}2^{n^{O(1)}} = 2^{n^{O(1)}}$ time.

Finally, having computed T^*, we determine whether $T^*, DB \models \beta$. By Definition 3.2, this means determining whether $(T^*)^\dagger \cup DB \vdash_c \beta$. By Lemma 6.9, this takes $2^{n^{O(1)}}$ time.

Thus, all the computations needed to determine whether $T^*, DB \models \beta$ can be completed in $2^{n^{O(1)}}$ time. By Theorem 5.11, this is the time needed to determine whether $\mathcal{S}, DB \models \beta$. This completes the proof of Theorem 6.1.

7 Lower Complexity Bounds

This section uses the proof theory of Section 2 to show that the data complexity of Datalog with predicate substitution is EXPTIME-hard. Together with the EXPTIME upper-bound of Section 6, it proves that Datalog with predicate substitution is EXPTIME-complete. Predicate substitution therefore increases the computational power of Datalog, which is only PTIME-complete. A concrete illustration of this power is given in [3], where Datalog with predicate substitution is used to solve the propositional satisfiability problem.

EXPTIME completeness holds *both* for Datalog with full substitution *and* for Datalog with basic substitution. Recall that full predicate substitution allows substitution for predicates of *different* arity, while basic substitution only allows substitution for predicates of *equal* arity. Thus, although full substitution provides a more flexible language, it does *not* increase the data complexity. The increased flexibility thus comes for free. The following theorem, which is the main result of this section, establishes an EXPTIME lower-bound for Datalog with basic substitution. The same lower bound therefore holds for Datalog with full substitution.

Theorem 7.1 *The data complexity of Datalog with basic predicate substitution is EXPTIME-hard.*

To prove Theorem 7.1, we use Datalog with basic substitution to encode the computations of an alternating Turing machine [5]. Alternating machines are a generalization of non-deterministic machines. Like non-deterministic machines, they can have many possible transitions at each point in a computation; but they may require that *all* transitions be successful, not just one. By encoding an arbitrary alternating PSPACE machine, we show that the data complexity of Datalog with basic substitution is complete for APSPACE. Theorem 7.1 follows since APSPACE = EXPTIME [5].

To state the main result precisely, let M be an alternating PSPACE machine. Without loss of generality, we 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 M on input $\overline{s}$. Formally,

$$\mathcal{S},\, DB(M) + DB(\overline{s}) \;\vdash\; ACCEPT \quad \textit{iff} \quad M \textit{ accepts } \overline{s} \tag{19}$$

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 this, we conclude that the data complexity of Datalog with basic substitution is EXPTIME-hard. This result follows immediately by letting M be a machine that recognizes a EXPTIME-complete language. Examples of such languages are given in [5].

Section 7.2 describes the construction of $DB(M)$, $DB(\overline{s})$ and $\mathcal{S}$. First, however, Section 7.1 describes the view of computation upon which the constructions are based.

7.1 Computation as Transformation

To encode the computations of a Turing 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. We represent a configuration as a string in a standard way, and we represent the transition table of the machine as a (non-deterministic) string transformation. We choose this representation because it is easy to implement with Datalog rules and basic substitution.

There are many ways to encode a machine configuration as a string. We describe a standard encoding that is convenient for the purpose of this section. 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 each end of the string, to give 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 compound 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 thus encoded by the following string:

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

Since our machine uses polynomial space, we can assume that all configurations are strings of the same length. That is, there is an integer k such that for inputs of length n, the machine uses at most n^k tape cells. Adding the two special characters, each configuration has $n^k + 2$ characters. The important points are that the length of a configuration does not change during a computation, and that the initial configuration can be constructed in polynomial time.

Deterministic Computation:

To simplify the discussion, we first consider deterministic Turing machines. Each step of a deterministic computation transforms one machine configuration into another. We represent this transformation as a mapping from strings to strings. This transformation maps each character in a string onto another character. Moreover, because computation is *local*, each character in the output string depends not on the entire input string, but only on a few of its characters.

Formally, the string transformation is specified by a function f that takes three characters x, y and z as input, and returns a single character $f(x, y, z)$ as output. Here, x, y and z denote any three consecutive characters in a string. For instance, if a string contains the sequence abc, and if $f(a, b, c) = d$, then the string is transformed by replacing this occurrence of b by d. Formally, the string $a_1...a_n$ is mapped onto the string $b_1...b_n$ where[8]

$$
\begin{aligned}
b_1 &= a_1 \\
b_n &= a_n \\
b_j &= f(a_{j-1}, a_j, a_{j+1}) \quad \textit{for} \;\; 1 < j < n
\end{aligned}
$$

Thus, the end points of the string do not change (leaving the two special characters invariant), and the interior points are transformed by the function f. We use the symbol t to denote the entire string transformation. Thus $t(a_1...a_n) = b_1...b_n$. In general, t takes a string $\overline{a}$ as input, and returns a string $t(\overline{a})$ of the same length as output.

The ternary function f can encode the transition function of any one-tape Turing machine. For example, suppose the transition function includes the following instruction: *If the control head is in state q_1 and is reading the character a, then write the character b, move one square to the left, and go into state q_2.* In this case, any string of the form "$...w\, x\, \langle a, q_1\rangle\, y\, z...$" is mapped onto the string "$...w\, \langle x, q_2\rangle\, b\, y\, z...$". The ternary function f thus satisfies the following equations:

$$
\begin{aligned}
f(w, x, \langle a, q_1\rangle) &= \langle x, q_2\rangle \\
f(x, \langle a, q_1\rangle, y) &= b \\
f(\langle a, q_1\rangle, y, z) &= y
\end{aligned}
$$

for any characters w, x, y and z. Of course, during any step of a computation, most characters in a configuration do not change. Thus $f(x, y, z) = y$ as long as x, y and z represent tape characters, not control states. This is an instance of the frame problem of Artificial Intelligence [9]. This problem does not concern us here however, since it is handled by the function f. Note in particular that f is finite and depends only on the machine M, not on the input $\overline{s}$. We can therefore spend as much time as we like constructing f.

When f encodes the transition function of a deterministic PSPACE machine, we can use the corresponding string transformation, t, to simulate the machine's computation. This transformation maps one configuration of the machine onto the next. Thus, if $\overline{a}$ is a string representing the configuration of the machine at some point in a computation, then $t(\overline{a})$

[8] ¡Here, and in the rest of this section, the symbol n shall denote the length of a machine configuration, not the length of the machine's input.

represents the next configuration, $t^2(\overline{a})$ the next configuration after that, etc. If $\overline{a}$ represents the initial configuration, then we can simulate the entire computation by generating one configuration after another. In this case, the machine accepts its input iff the finite control is in an accepting state for some configuration $t^i(\overline{a})$.

Alternating Computation:

In alternating computations, each machine configuration can have many successors. We therefore need to view computation not as a *sequence* of configurations, but as a *tree*. Without loss of generality, we can assume the tree is binary; so each configuration has at most two successors. We can therefore represent the transition table of an alternating machine as two functions f_1 and f_2.

As in the deterministic case, these functions specify string transformations. Formally, a string $a_1...a_n$ has two successors $b_1^1...b_n^1$ and $b_1^2...b_n^2$ where

$$b_1^i = a_1 \tag{20}$$
$$b_n^i = a_n \tag{21}$$
$$b_j^i = f_i(a_{j-1}, a_j, a_{j+1}) \quad \textit{for} \quad 1 < j < n \tag{22}$$

As before, the end points of the string do not change, and the interior points are transformed by the functions f_1 and f_2. We use the symbols t_1 and t_2 to denote these two string transformations, where $t_i(a_1...a_n) = b_1^i...b_n^i$. In general, t_i takes a string $\overline{a}$ as input, and returns a string $t_i(\overline{a})$ of the same length as output. 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. For example, $t_1(\overline{a})$ and $t_2(\overline{a})$ represent the two immediate successors of $\overline{a}$. Likewise, $t_1t_i(\overline{a})$ and $t_2t_i(\overline{a})$ represent the two immediate successors of $t_i(\overline{a})$.

7.2 Encoding an Alternating PSPACE Machine

We implement the strings and transformations described above as a Datalog rulebase with basic substitution. 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 Datalog rules. Starting from the initial configuration, we use predicate substitution to apply the transformations over-and-over again. The entire rulebase consists of twelve rules: nine non-recursive Datalog rules, and three mutually-recursive rules with substitution. This structure is enough to simulate any APSPACE machine and achieve EXPTIME-hardness.

The Machine:

Here we specify the database $DB(M)$ in statement (19). This database encodes the machine's control states and the two ternary functions f_1 and f_2. To encode the two functions, we introduce two 4-ary predicates, F_1 and F_2. We add the formula $F_i(a,b,c,d)$ to the database iff $f_i(a,b,c) = d$. These formulas encode the transition table of the machine.

In addition, we must be able determine whether a computation is accepting. For alternating machines, acceptance is defined in terms of accepting configurations. Whether a configuration is accepting depends on its successor configurations and on the state of the finite control. An alternating machine has three kinds of state: *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 configuration is accepting.

An alternating machine accepts its input iff its initial configuration is accepting. To encode the control states, we use 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*.

The Initial Configuration:

Here we specify the database $DB(\overline{s})$ in statement (19). This database encodes the machine's initial configuration. First, we need a counter to represent string positions. For this, we introduce three base predicates, *FIRST*, *NEXT* and *LAST*. For strings of length n, we add the following formulas to the database:

$$FIRST(1), \quad NEXT(1,2), \quad NEXT(2,3) \; ... \; NEXT(n-1,n), \quad LAST(n).$$

Next, 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:

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

Recall that the machine input $\overline{s}$ is encoded in the initial configuration $\overline{a}$.

The Computation:

Our final task is to encode the computation of an alternating Turing machine as a Datalog rulebase with predicate substitution, $\mathcal{S}$. This rulebase has two parts: (i) a set of Datalog rules that implement the string transformations t_1 and t_2, and (ii) rules with substitution that apply these transformations recursively, and that determine whether the the machine accepts its input. These rules are described below.

String Transformations: To implement the string transformation t_i, we add the Datalog rules below to rulebase $\mathcal{S}$. These three rules correspond to the three conditions (20)–(22), respectively. 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.

$$\begin{aligned} S_i(x,j) &\leftarrow FIRST(j),\ S(x,j). \\ S_i(x,j) &\leftarrow LAST(j),\ S(x,j). \\ S_i(x_4,j_2) &\leftarrow F_i(x_1,x_2,x_3,x_4),\ NEXT(j_1,j_2),\ NEXT(j_2,j_3), \\ &\quad\ S(x_1,j_1),\ S(x_2,j_2),\ S(x_3,j_3). \end{aligned}$$

Here, all the j's and x's are variables. The first two rules ensure that the end points of the string do not change. The third rule implements the function f_i that transforms the interior points of the string. We thus add six Datalog rules to the rulebase $\mathcal{S}$—three for S_1 and three for S_2.

The Datalog rules defining S_i represent a single application of the transformation t_i to the string S encoded in the database. To apply the transformation more than once, we apply the predicate substitution $[S/S_i]$ to the rulebase. This substitution reuses and modifies the rules to produce another set of rules representing another application of t_i. The order in which the two substitutions $[S/S_1]$ and $[S/S_2]$ are applied determines the order in which 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 $t_1(\overline{a})$, $t_1t_2(\overline{a})$, $t_1t_2t_1(\overline{a})$ and $t_1t_2t_1t_1(\overline{a})$, respectively:

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

In general, the predicate with substitution $S_{i_1}(x,j)\,[S/S_{i_2}]\,[S/S_{i_3}]...[S/S_{i_k}]$ represents the string $t_{i_1}t_{i_2}t_{i_3}...t_{i_k}(\overline{a})$.

Accepting Configurations: Besides generating configurations, the rulebase must determine whether the machine accepts its input. The first step is to determine what kind of control state a configuration has. The following Datalog 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 if and only if the configuration represented by S contains an existential control state. Likewise for the predicates $UNIV$ and $ACCEPT$. With these three rules, the rulebase $\mathcal{S}$ now has a total of nine Datalog rules.

If the initial configuration has an accepting control state, then we are done. If not, then we can examine its successor configurations by using predicate substitution. In the simplest case, the predicates $ACCEPT\,[S/S_1]$ and $ACCEPT\,[S/S_2]$ ask whether the two immediate successor configurations have an accepting control state. By using multiple substitutions, we can examine more distant successors. For example, if the predicate $S(x,j)$ represents the configuration $\overline{a}$, then the successor configurations $t_1(\overline{a})$, $t_1t_2(\overline{a})$, $t_1t_2t_1(\overline{a})$ and $t_1t_2t_1t_1(\overline{a})$ have an accepting control state if and only if the following predicates with substitution are true, respectively:

$$\begin{array}{l} ACCEPT\,[S/S_1] \\ ACCEPT\,[S/S_1]\,[S/S_2] \\ ACCEPT\,[S/S_1]\,[S/S_2]\,[S/S_1] \\ ACCEPT\,[S/S_1]\,[S/S_2]\,[S/S_1]\,[S/S_1] \end{array}$$

In general, the configuration $t_{i_1}t_{i_2}...t_{i_k}(\overline{a})$ has an accepting control state if and only if $ACCEPT\,[S/S_{i_1}]\,[S/S_{i_2}]...[S/S_{i_k}]$ is true.

Even if a configuration does not have an accepting state, it may still be an accepting configuration, depending on its successors. We can specify this dependency by using recursion through substitution. The following rules, which we add to the rulebase $\mathcal{S}$, do exactly this:

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

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.

These three rules determine if a configuration, $\overline{a}$, is accepting by examining its two immediate successors, $t_1(\overline{a})$ and $t_2(\overline{a})$. However, we still need rules to determine whether these two successors are accepting. The copy-and-substitute mechanism automatically generates such rules. For instance, by applying the substitution $[S/S_i]$, it generates the following three rules:

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

These rules determine if the configuration $t_i(\overline{a})$ is accepting by examining its two immediate successors, $t_1t_i(\overline{a})$ and $t_2t_i(\overline{a})$. As before, more rules may be needed to determine whether the two successors are accepting; and the copy-and-substitute mechanism generates them. This time, by applying the multiple substitution $[S/S_j]\,[S/S_i]$, it generates the following three rules:

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

These rules determine whether the configuration $t_jt_i(\overline{a})$ is accepting. In this way, the copy-and-substitute mechanism generates rules of acceptance for *every* machine configuration. These rules all belong to $\mathcal{S}^{\dagger}$, the closure of rulebase $\mathcal{S}$.

$\mathcal{S}$ thus consists of nine Datalog rules and three rules with substitution. Applied to the database $DB(M) + DB(\overline{s})$ constructed above, this rulebase infers the atom $ACCEPT$ if and only if the initial configuration is accepting, that is, if and only if machine M accepts input $\overline{s}$. This rulebase therefore satisfies condition (19), thus completing the proof of Theorem 7.1.

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] P.A. Bernstein. Synthesizing third normal form relations from functional dependencies. *ACM Transactions on Database Systems (TODS)*, 1(4):277–298, 1976.

[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 and D. Harel. Structure and Complexity of Relational Queries. *Journal of Computer and System Sciences (JCSS)*, 25(1):99–128, 1982.

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

[6] B.F. Chellas. *Modal Logic: an Introduction.* Cambridge University Press, 1980.

[7] 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.

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

[9] J.M. McCarthy and P.J. Hayes. Some philosophical problems from the standpoint of artificial intelligence. In B. Meltzer and D. Michie, editors, *Machine Intelligence*, volume 4, pages 463–502. Edinburgh University Press, 1969. Reprinted in *Readings in Artificial Intelligence*, 1981, Tioga Publ. Co.

[10] L.T. McCarty. A Language for Legal Discourse. I. Basic Features. In *Proceedings of the Second International Conference on Artificial Intelligence and Law*, pages 180–189. ACM Press, June 1989.

[11] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences (JCSS)*, 17:348–375, 1978.

[12] A. Mycroft. Polymorphic type schemes and recursive definitions. In M. Paul and B. Robinet, editors, *Proceedings of the International Symposium on Programming*, number 167 in Lecture Notes in Computer Science, pages 217–228. Springer-Verlag, April 1984.

[13] A. Mycroft and R.A. O'Keefe. A polymorphic type system for Prolog. *Artificial Intelligence*, 23(3):295–307, 1984.

[14] D.A. Schlobohm and L.T. McCarty. EPS-II: Estate planning with prototypes. In *Proceedings of the Second International Conference on Artificial Intelligence and Law*, pages 1–10. ACM Press, June 1989.

[15] M.J. Sergot, F. Sadri, R.A. Kowalski, F. Kriwaczek, P. Hammond, and H.T. Cory. The British Nationality Act as a logic program. *Communications of the ACM*, 29(5), 1986.

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

[17] J.D. Ullman. *Principles of Database Systems.* Computer Science Press, Rockville, Maryland, 1982.

[18] 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.

[19] M. Vardi. The Complexity of Relational Query Languages. In *Proceedings of the ACM Symposium on Theory of Computing (STOC)*, pages 137–146, 1982.