The Reuse and Modification of Rulebases by Predicate Substitution*

Anthony J. Bonner Tomasz Imielinski

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

March, 1990

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. Substitution increases the expressive power of Datalog. Not only does data complexity increase from $PTIME$ to $EXPTIME$, but substitution also allows large sets of Datalog rules to be succinctly expressed. The paper provides a proof and model theory for this language, including a fixpoint semantics.

*This paper appears in the *Proceedings of the International Conference on Extending Database Technology (EDBT)*, pages 437–451. Lecture Notes in Computer Science, volume 416. Springer Verlag. Conference held March 26–30 1990 in Venice, Italy.

1 Introduction

This paper proposes 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. Unfortunately, these issues have not yet received significant attention in the deductive database literature. This paper attempts to address them.

We focus on a particular class of modifications which we call *predicate substitutions*. The idea is to reuse and modify a set of rules by substituting one predicate symbol for another. To this end, we introduce a modal-like notation $A(x)[P/Q]$ which we call a *predicate with substitution*. Intuitively, this expression means, "replace P by Q in the definition of A". To take the simplest example, suppose that A is defined by the following rules:

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

where B and C are base predicates. Then the expression $A(x)[P/Q]$ is treated a new predicate symbol defined by the following rules.

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

Conceptually, these rules are generated automatically and do not have to be supplied by the user.

Like ordinary predicates, predicates with substitution can be used in the body of rules, as in the rule $B(x) \leftarrow A(x)[P/Q]$. In the case when A is defined by a large number of rules, generating the definition of $A(x)[P/Q]$ by hand could be an expensive and unreliable task. Additionally, it could double the size of the rulebase. In this way, rules with substitution can be viewed as a short-hand notation for describing rulebases without substitution. This conciseness is achieved by reusing and modifying rule sets, possibly many times. In fact, by having recursion through substitution, it is possible to reuse and modify rules an unbounded number of times. Using this idea, we show that a *finite* set of rules with substitution can describe an *infinite* set of rules without.

We believe that substitution is a principle which 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. Several examples are given to illustrate the idea of substitution and its utility. Substitution is then defined precisely and an inference system is developed for Horn rules with substitution.

The issue of semantics is also addressed. First, a semantics is developed in which Horn rulebases *without* substitution are the models for Horn rulebases *with* substitution. This captures, in a model-theoretic way, the view of this paper: that rulebases with substitution are a short-hand notation for rulebases without. The inference system for substitution is shown to be sound and complete with respect to this semantics.

As in the theory of Horn logic [8, 1], we complement the model-theoretic semantics with a fixpoint semantics. We define a monotonic "T-operator" which 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 expands rules with substitution into equivalent Horn rules. If there is recursion through substitution, it may take infinitely many applications to generate all the Horn rules in the minimal model. Because the operator is continuous, however, any particular Horn rule will be generated in a finite number of applications. In this way, the Horn rules represented by a rulebase with substitution can be materialized.

In the function-free case, predicate substitution increases the power of Horn logic. Whereas the complexity of Datalog (i.e., function-free Horn logic) is complete for PTIME, the complexity of Datalog augmented with substitution is complete for EXPTIME. Although a proof of this result is beyond the scope of this paper, we present an example showing that the data complexity is NP-hard. Syntactically, this boost in expressive power comes about from recursion through substitution. Semantically, this increased power has an interesting interpretation: When a Datalog rulebase with substitution is EXPTIME-complete, it represents an *infinite* Datalog rulebase *without* substitution. Furthermore, this infinite Datalog rulebase is not equivalent to any finite one. In such cases, substitution provides an extremely concise notation for representing a set of Datalog rules.

1.1 Applications

In this section, we present an example of how predicate substitution can be used, and we discuss several other possible applications. In the example, a module of rules for solving a graph problem is described. The idea is to reuse and modify this general-purpose module in order to solve certain problems in airline flight reservations.

Let G be a weighted graph in which each edge has an associated cost. Given a path between two nodes, the cost of the path is the sum of the weights on each edge. Let R be a set of rules which for any pair of nodes x and y, computes the minimum cost of all the paths from x to y. The rules R will have two base predicates (i.e., input predicates): $Edge(x, y)$, meaning that there is an edge from x to y in the graph; and $Weight(x, y, w)$, meaning that the edge from x to y has weight w. The output predicate of R is $MinCost(x, y, w)$ where w is the minimum cost of a path from node x to node y.

Suppose now that we wish to use the set of rules R in an airline reservation system. The graph G now represents flight connections in the United States. We would like to reuse the rules R for many different notions of cost. In particular, we would like to know the following three parameters for the connections between Tucson, Arizona and Bismarck, North Dakota: (i) the smallest number of stopovers, (ii) the minimum flying time, and (iii) the minimum price. For this purpose, we can reuse the module R, substituting the following predicates for the predicate $Weight$:

- $W_1(x, y, w)$ where w is 1 if x and y are directly connected, and 0 otherwise. In this case, we want to calculate the minimum number of stopovers.
- $W_2(x, y, w)$ where w is the price of a direct flight from x to y (here we assume that the global cost of the connection is a sum of individual costs). In this case, we want to calculate the minimum price.
- $W_3(x, y, w)$ where w is the flying time for a direct flight from x to y. In this case, we want to calculate the minimum flying time.

Using these three different predicates, we can reuse the minimum-cost module by simply writing the following rules:

$$MinStopovers(x, y, w) \leftarrow MinCost(x, y, w)[Weight/W_1]$$

$$MinPrice(x, y, w) \leftarrow MinCost(x, y, w)[Weight/W_2]$$

$$MinFlyingTime(x, y, w) \leftarrow MinCost(x, y, w)[Weight/W_3]$$

In each case, the rules in the module R are copied and modified so that $Weight$ is replaced by W_i. In this way, a possibly complex and large module

R does not have to be rewritten every time it is reused.

The above example is 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 this way, for instance, a module for matrix multiplication could be reused many times for different types of matrices or even different types of multiplication (integer, real, complex, etc.)

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, and McCarty and Sridharan have developed expert systems for reasoning about contract law and corporate tax law [5]. As laws are amended, such systems have to be updated. For instance, sections of the income tax act could be amended to treat residents as citizens. In this case, every occurrence of "citizen" could be replaced by a new predicate meaning "citizen or resident". Similarly, other bodies of law could be amended to replace the predicate "male" by "person". Such updates could easily be expressed and implemented with predicate substitutions.

The need to often reuse and modify existing rule bases occurs in rulebased specifications of the operations of large enterprises, such as big corporations, computer operating systems or various legal systems, such as tax-law, citizenship law etc. When changes are small, it simply does not make sense to rewrite possibly-huge rulebased specifications. It is definitely more attractive to reuse as much as possible of the old definition.

From the above examples it may appear that adding the principle of substitution does not increase the expressive power of a language. We show later that this is not the case. In fact, the data complexity of Datalog with substitution is complete for EXPTIME. To illustrate this increased power, section 2.2 shows a set of rules which use recursion through predicate substitution to solve an NP-complete problem—a task impossible for Datalog, unless P = NP.

2 Inference with Predicate Substitution

This section formalizes the notion of predicate substitution. In particular, the syntax of predicates and rules with substitution is defined and an inference system is described. The description of inference is initially informal and is given in terms of a copy-and-substitute mechanism. Several examples are presented. Finally, the mechanism is defined precisely and an axiomatization is given.

Definition 1 *Every atomic formula is a predicate with substitution. Furthermore, if α is a predicate with substitution, then so is $\alpha[P/Q]$, where P and Q are predicate symbols of equal arity.*

Expressions of the form $[P/Q]$ shall be called *predicate substitutions*, or simply *substitutions*.

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

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

From the database perspective, a rulebased system consists 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 types of relations: *base relations* and *derived relations*. A base relation is defined by the database, and an derived relation is defined by the rulebase. The division of relations in this way is common in deductive databases. Furthermore, by separating the rulebase from the database in this way, we are able to treat the rulebase as a mapping from base relations to derived relations. Substitution is then a composition of one map with another.

To represent these two types of relations, we use two types of predicate symbols: base and derived. Base predicate symbols are used to construct the atomic formulas in the database but may not appear in the heads of rules. Conversely, derived predicate symbols may appear in the heads of rules but may not appear in the database.

We treat predicates with substitution as derived predicates. For instance, the expression $A(x)[P/Q]$ is treated as just another derived predicate. This predicate has a special meaning however: The rules defining $A(x)[P/Q]$ must be identical to the rules defining $A(x)$ except that P is replaced everywhere by Q.

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 $B(x)[P/Q]$.[1] 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 new predicate $B(\overline{x})[P/Q]$ is replaced by $B(\overline{x})$. In this way, the new rulebase is able to access the database. The result is a rulebase which is identical to the original except that P has been replaced everywhere by Q; and instead of defining $A(x)$, the new rulebase defines $A(x)[P/Q]$, for every derived predicate A.

Example 1. Let $\mathcal{S}$ be the following rulebase, where B and C are base predicates:

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

[1] 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.

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

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

Notice that the definition of $A(x)[P/Q]$ is identical to the definition of $A(x)$ except that P has been replaced by Q in the premise of every rule. Conceptually, these rules are generated automatically; and in particular, they do not have to be supplied by the user. □

The above example is the simplest possible in that there is only one derived predicate and no recursion. The next two examples illustrate the substitute-and-copy mechanism for slightly more complex rulebases.

Example 2. Let $\mathcal{S}$ be the following rulebase, where C and D are base predicates:

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

Then the predicates $A(x)[P/Q]$ and $B(x)[P/Q]$ are defined by the following rules:

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

Since these rules are to be treated classically, we can make the following inference:

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

□

Example 3. Let $\mathcal{S}$ be the following rulebase, where C is a base predicate:

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

The following rules then define the predicate $A(x)[P/Q]$:

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

Since these rules are to be treated classically, we can make the following inferences (among others):

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

□

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 4. Let $\mathcal{S}$ be the following rulebase, where Q and R are base predicates:

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

We then 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 to be treated classically, we can make the following inferences:

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

□

The copy-and-substitute mechanism outlined above describes informally the idea of substitution for Horn rules with substitution. The following definition makes the idea precise.

Definition 3 *Let $\mathcal{S}$ be a Horn rulebase with substitution. Then $\mathcal{S}^*$ is the set of rules with substitution constructed from $\mathcal{S}$ as follows:*[2]

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

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

It is not enough, however, to simply generate Horn rules. To be useful, a rulebase must be augmented with a database and the rules used to infer new atomic formulas. Having generated $\mathcal{S}^*$, we now treat it as a set of Horn rules and apply it to a database using classical inference.

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

Definition 4 *Suppose β is a predicate with substitution, $\mathcal{S}$ is a Horn rulebase with substitution, and DB is a database. Then,*[3]

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

Example 5. Suppose DB consists of the atoms $Q(a)$, $S(a)$, $Q(b)$, $R(b)$, $P(b)$. Applying rulebase $\mathcal{S}$ from example 4 to this database gives the following inferences (among others):

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

□

The set of rules $\mathcal{S}^*$ is infinite and contains an infinite number of distinct predicate symbols. As the next example shows, this is 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, it is possible to create new and distinct predicates whose names have arbitrarily length, such as $A(x)[B/C][B/C][B/C]$. Because of this, $\mathcal{S}^*$ contains an infinite number of distinct predicate symbols and is itself infinite.

Example 6. Suppose $\mathcal{S}$ consists of the following two rules, where B and P are base predicates:

$$A(x) \leftarrow P(x)$$
$$Q(x) \leftarrow B(x,y), P(y)$$

The predicates $A(x)[P/Q]$, $A(x)[P/Q][P/Q]$ and $A(x)[P/Q][P/Q][P/Q]$ are then defined as follows:

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

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

In general, the definition of $A(x)[P/Q]^i$ is different from the definition of $A(x)[P/Q]^{i+1}$, and so they denote distinct predicates. In other words, substitution is not idempotent. □

2.1 An Axiomatization

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

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

Axioms:

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

Inference Rules:

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

Given a predicate β, the first inference rule effectively makes copies of the rules defining β in order to derive rules defining $\beta[\theta]$. The first two axioms come directly from the definition of $\mathcal{S}^*$. The other axioms and inference rules are familiar from classical 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.

2.2 The Power of Substitution

In the function-free case, predicate substitution increases the power of Horn logic. Whereas the complexity of Datalog (i.e., function-free Horn logic) is complete for PTIME, the complexity of Datalog augmented with substitution is complete for EXPTIME. Syntactically, this boost in expressive power comes about from recursion through substitution. Although a proof of this result is beyond the scope of this paper, this section presents an example in which rules with predicate substitution are used to solve an NP-complete problem—a task impossible for Datalog, unless P = NP.

Given a combinational circuit (i.e., a logic network without feedback), it is NP-complete to determine whether the output of the circuit can ever have the value 1.[4] We construct a Datalog program with substitution that generates all possible input values and computes the output value for each one. The program has three parts: (i) a set of Horn rules which evaluates the output of the circuit for a given set of input values, (ii) a set of Horn rules which generates the next input set from the current input set, and (iii) a set of rules with substitution that replaces the current input set with the next input set.

The circuit itself is encoded as a database. There are many ways in which this can be done. For instance, constant symbols can denote wires and predicates can denote circuit connections. Thus, the predicate $Nor(x, y, z)$ could mean that wire x is the output of a nor-gate whose input wires are y and z. Similarly, the values on wires can be represented by the binary predicate $Val(x, y)$, which means that wire x has value y, where y is either 0 or 1.

(i) Given such a representation of the circuit and a set of input values, it is not difficult to write a set of Datalog rules to compute the output value of the circuit. In particular one can define a zero-ary predicate $Test$ which is true iff the output value is 1. For each circuit, the value of $Test$ will depend upon the circuit's input values, that is, upon the predicate $Val(x, y)$. The rules defining $Test$ are independent of the database (i.e., of the circuit).

(ii) If the circuit contains n input lines, then any set of input values can be encoded as an n-bit binary number. Thus, to generate all possible input sets, we can build a binary counter from 0 to $2^n - 1$. Each input line then corresponds to a binary digit. Suppose that u_i is the value of the i^{th} input line. The entire input to the circuit is then represented by the binary number $u_1...u_n$, which is represented by the following atomic formulas:

$$Val(a_1, u_1),\ Val(a_2, u_2)\ ...\ Val(a_n, u_n)$$

where a_i denotes the i^{th} input line. We use the predicate $Val^+(x, y)$ to represent the successor of the binary number represented by Val. In particular, if $v_1...v_n$ is the successor of $u_1...u_n$, then it is represented by the following atomic formulas:

$$Val^+(a_1, v_1),\ Val^+(a_2, v_2)\ ...\ Val^+(a_n, v_n)$$

It is not difficult to write a set of Datalog rules which given the relation Val will generate the relation Val^+. In this way, given a binary number, we can increment it by one.

By using predicate substitution, we can compute larger increments. For instance, the predicate $Val^+[Val/Val^+]$ increments the binary number Val by two. The rules defining this predicate are identical to the rules defining Val^+ except that all the predicates are given new names, and in particular, the predicate Val is replaced by Val^+. Thus, $Val^+[Val/Val^+]$ is computed from Val^+, which in turn is computed from Val. In this way, the Horn rules for computing a single increment are reused and modified to produce rules which compute a double increment.

The substitution $[Val/Val^+]$ can be applied over and over again to produce rules for computing larger and larger increments. Thus, the predicates $Val^+[Val/Val^+]$, and

[4] The satisfiability of propositional formulas can be reduced to this problem in a straightforward way.

$Val^+[Val/Val^+][Val/Val^+]$ represent increments of two and three, resp. Conceptually, each additional application of the substitution $[Val/Val^+]$ generates a new copy of the rules defining Val^+, with the output of one copy being used as the input for the next. Each such copy produces an additional increment of one.

(iii) Given a set of input values, the Horn rules described in (i) infer the atom $Test$ iff the output of the circuit is 1. By using the rules in (ii) and predicate substitution, we can test the output value of the circuit for any subsequent set of input values. For instance, the predicate $Test[Val/Val^+]$ is true iff the output of the circuit is 1 on the *next* set of input values.

If the predicate Val represents the number 0, then we can test the circuit for all input values. For instance, suppose the circuit has four input lines, each of them 0. $Test$ is then true iff the output is 1 on the input 0000. In addition, $Test[Val/Val^+]$ is true iff the output is 1 on the input 0001, and $Test[Val/Val^+][Val/Val^+]$ is true iff the output is 1 on input 0010. In general, the predicate $Test[Val/Val^+]^i$ is true iff the output of the circuit is 1 when the input is i (in binary). In this way, we can test the output of the circuit for any set of consecutive input values. The following four rules, for example, infer the atom Sat iff the output of the circuit is 1 for one of the input sets 0000, 0001, 0010 or 0011.

$$
\begin{aligned}
&Sat \leftarrow Test\\
&Sat \leftarrow Test[Val/Val^+]\\
&Sat \leftarrow Test[Val/Val^+][Val/Val^+]\\
&Sat \leftarrow Test[Val/Val^+][Val/Val^+][Val/Val^+]
\end{aligned}
$$

By using recursion through substitution, it is possible to test the circuit for all possible input sets. The rules below do precisely this. In particular, they infer the atom Sat iff the output of the circuit is 1 for some set of inputs.

$$
\begin{aligned}
&Sat \leftarrow Sat[Val/Val^+]\\
&Sat \leftarrow Test
\end{aligned}
$$

Intuitively, this recursion can be unfolded to produce an infinite set of Datalog rules. For instance, the first four levels of recursion produce the four Datalog rules above plus a Datalog definition of the predicates $Test[Val/Val^+]^i$. After unfolding, each of these predicates is treated as a predicate of classical logic. In this way, the definitions of $Test$ and Val^+ are reused and modified over and over again to produce an infinitely large Datalog rulebase.

Notice that there is no *finite* set of Datalog rules which is equivalent to this rulebase, unless P = NP. This is because the above rules with substitution solve an NP-complete problem, whereas Datalog rules can only solve problems in P. In this way, substitution offers a succinct notation for describing Horn rulebases. In this case, a finite rulebase with substitution describes an infinite Datalog rulebase.

3 Horn Semantics

Given a set of Horn rules $\mathcal{S}$ *without* substitution, it is relatively clear what the rulebase $\mathcal{S}^*$ means. For such rulebases, the copy-and-substitute mechanism of definition 3 produces a set of Horn rules for $A(x)[B/C]$ in terms of the Horn rules for $A(x)$. Intuitively, the definition of $A(x)[P/Q]$ is identical to that of $A(x)$ except that P is replaced everywhere by Q. The meaning is less clear, however, when the rulebase $\mathcal{S}$ contains rules with substitution, especially when there is recursion through substitution. In the rule $A(x) \leftarrow B(x,y), A(y)[P/Q]$, for instance, the definitions of $A(x)$ and of $A(x)[P/Q]$ are circular. What does the expression $A(x)[P/Q]$ mean in such cases? Section 2 provided a proof-theoretic meaning. This section provides a model-theoretic one.

3.1 Models and Entailment

A model in our semantics is a Horn rulebase (without substitution). Not only do such rulebases provide a non-circular definition for predicates with substitution, they also support the idea that a rulebase with substitution is a shorthand notation for a (possibly larger) Horn rulebase, which the user might have written had he had the time, space and inclination.

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

Rules with substitution are intended to be a shorthand notation for a set of Horn rules. Suppose, for instance, that $\mathcal{S}$ contains the rule $A(x) \leftarrow B(x)[P/Q]$. $\mathcal{S}^*$ then determines how this rule can be expanded into Horn rules. In particular, suppose that $\mathcal{S}^*$ contains the following two rules, where f is a unary function symbol:

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

Then the rule $A(x) \leftarrow B(x)[P/Q]$ can be expanded into the following two Horn rules:

$$\begin{array}{l} A(x) \leftarrow C(x,y), D(y). \\ A(fx) \leftarrow E(x) \end{array} \qquad (1)$$

This suggests what it means for a Horn rulebase R to be a model of a rulebase with substitution $\mathcal{S}$. In this example, if R does not contain the two rules (1), then we shall say that R is not a model of $\mathcal{S}$. The following definition formalizes the idea that premises with substitution should be unfolded in the natural way.

Definition 6 *(Satisfaction) Let R be a set of Horn rules, let α be an atomic formula, and let $\beta_1 ... \beta_n$ be predicates with substitution. Then, $R \models \alpha \leftarrow \beta_1 ... \beta_n$ iff for all term substitutions σ, and all strings of atomic formulas γ, $R \models_c \alpha\sigma \leftarrow \gamma$ whenever $R^* \models_c \beta_i\sigma \leftarrow \gamma$ for each $1 \le i \le n$.*

If $\mathcal{S}$ is a rulebase with substitution and R is a Horn rulebase, then we say that R satisfies $\mathcal{S}$ iff R satisfies each rule in $\mathcal{S}$. In this case, we also say that R is a *model* of $\mathcal{S}$.

Note that the definition of satisfaction involves the rulebase R^*. Because R is Horn rulebase without substitution, the intuitive meaning of R^* is provided in a straightforward way by the copy-and-substitute mechanism.

The following example shows models of a rulebase in which recursion occurs through substitution. Note that recursive rules may have non-recursive models.

Example 7. Suppose $\mathcal{S}$ consists of the following two rules, where B and C are base predicates:

$$A(x) \leftarrow B(x)$$
$$A(x) \leftarrow A(x)[B/C]$$

Then both of the following are models of $\mathcal{S}$, where D is a base predicate:

$$R_1 = \begin{cases} A(x) \leftarrow B(x) \\ A(x) \leftarrow C(x) \end{cases}$$

$$R_2 = \begin{cases} A(x) \leftarrow B(x) \\ A(x) \leftarrow C(x) \\ A(x) \leftarrow D(x,y), C(y). \end{cases}$$

To see this, note that in R_1^*, the predicate $A(x)[B/C]$ is defined by the following rule:

$$A(x)[B/C] \leftarrow C(x)$$

and that in R_2^*, it is defined by the following two rules:

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

Thus, in either rulebase, any expansion of the rule $A(x) \leftarrow A(x)[B/C]$ will only produce rules which are already in the rulebase. Thus both rulebases satisfy $A(x) \leftarrow A(x)[B/C]$. Similarly, both rulebases satisfy $A(x) \leftarrow B(x)$. Thus, both R_1 and R_2 are models of $\mathcal{S}$. □

It is worthwhile noting that satisfaction is not monotonic in R. That is, if $R_1 \subseteq R_2$, then in general it is not true 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. In classical logic, for instance, if s_1 and s_2 are first-order 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.

As in section 2, our goal is not to derive rules, but to apply rules to a database to derive atomic formulas. We thus make the following definitions.

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

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

3.2 Soundness and Completeness

This section establishes the soundness and completeness of the inference system of section 2 with respect to the semantics of substitution developed above. That is, we prove the following two theorems:

Theorem 1 *(Soundness) Suppose β is a predicate with substitution, $\mathcal{S}$ is a Horn rulebase with substitution, and DB is a database. If $\mathcal{S}, DB \vdash \beta$ then $\mathcal{S}, DB \models \beta$.*

Theorem 2 *(Completeness) Suppose β is a predicate with substitution, $\mathcal{S}$ is a Horn rulebase with substitution, and DB is a database. If $\mathcal{S}, DB \models \beta$ then $\mathcal{S}, DB \vdash \beta$.*

To prove completeness, we define a rulebase $R_{\mathcal{S}}$ which we call the *canonical model* of $\mathcal{S}$. This model, defined proof-theoretically, provides the necessary link between inference and semantics.

Definition 9 *If $\mathcal{S}$ is a set of Horn rules with substitution, then $R_{\mathcal{S}}$ is the set of Horn rules that can be inferred from $\mathcal{S}$. That is,*

$$R_{\mathcal{S}} = \{\alpha \leftarrow \alpha_1 ... \alpha_n \mid \mathcal{S}^* \vdash_c \alpha \leftarrow \alpha_1 ... \alpha_n\}$$

where each α and each α_i are atomic formulas.

The following lemmas are straightforward.

Lemma 2 *$R_{\mathcal{S}}$ is a model of $\mathcal{S}$. That is, $R_{\mathcal{S}} \models \mathcal{S}$*

Lemma 3 *Let β be a predicate with substitution, $\mathcal{S}$ a Horn rulebase with substitution, and DB a database. If $R_{\mathcal{S}}, DB \models \beta$ then $\mathcal{S}, DB \vdash \beta$.*

Completeness follows immediately. By definition, if $\mathcal{S}, DB \models \beta$ then $R, DB \models \beta$ for any model R of $\mathcal{S}$. Thus $R_{\mathcal{S}}, DB \models \beta$ by lemma 2. Hence $\mathcal{S}, DB \vdash \beta$ by lemma 3.

3.2.1 Soundness

The proof of soundness is more involved. Recall that the inference system of section 2 derives rules as well as atomic formulas. This is a convenience: The derived rules are simply an intermediate result in the derivation of atomic formulas. Many of the derived rules are non-Horn; that is, they have predicates with substitution in the head as well as in the body. Definition 6, however, gives the semantic meaning of $R \models \beta \leftarrow \beta_1 ... \beta_n$ only in the case in which β is atomic. This is the case we are really interested in, since inference has been defined only for rulebases consisting of such rules. However, for the purposes of proving soundness, it is convenient to have a more general definition of satisfaction. The following definition does exactly this. Note that this definition is purely a formal device and is not intended as a semantics for rules with substitution in the head. Also note that in the special case in which the rule head β is atomic, this definition is equivalent to definition 6.

Definition 10 *(Generalized Satisfaction) Let R be a set of Horn rules, and β and $\beta_1 ... \beta_n$ be predicates with substitution. Then, $R \models \beta \leftarrow \beta_1 ... \beta_n$ iff for all term substitutions σ, and all strings of atomic formulas γ, $R^* \models_c \beta\sigma \leftarrow \gamma$ whenever $R^* \models_c \beta_i\sigma \leftarrow \gamma$ for each $1 \leq i \leq n$.*

We also need a more general notion of entailment, one which says what it means to entail a rule. The definition below does exactly this. The lemma following shows that the notion of entailment given in definition 8 can be reduced to this definition in a straightforward way.

Definition 11 *(Generalized Entailment) Suppose β and $\beta_1 ... \beta_n$ are predicates with substitution, and $\mathcal{S}$ is a set of rules with substitution. Then $\mathcal{S} \models \beta \leftarrow \beta_1 ... \beta_n$ iff $R \models \beta \leftarrow \beta_1 ... \beta_n$ for every model R of $\mathcal{S}$.*

Lemma 4 $\mathcal{S}, DB \models \beta$ *iff* $\mathcal{S} \models \beta \leftarrow DB$

We now outline the proof of soundness in a series of lemmas. These lemmas describe some of the fundamental properties of entailment.

Lemma 5 *Suppose β and $\beta_1 ... \beta_n$ are predicates with substitution, and $\mathcal{S}$ is a rulebase with substitution. Then,*

- $\mathcal{S} \models \mathcal{S}$
- *If* $\mathcal{S} \models \beta \leftarrow \beta_1 ... \beta_n$
  then $\mathcal{S} \models \beta[\theta] \leftarrow \beta_1[\theta] ... \beta_n[\theta]$
  for any substitution $[\theta]$.
- $\mathcal{S} \models P(\overline{x})[P/Q] \leftarrow Q(\overline{x})$
  for any substitution $[P/Q]$.
- $\mathcal{S} \models B(\overline{x})[P/Q] \leftarrow B(\overline{x})$
  for any base predicate $B \neq P$.

Each item in this lemma corresponds to an item in the definition of $\mathcal{S}^*$ (definition 3). Thus, each rule in $\mathcal{S}^*$ is entailed by $\mathcal{S}$.

Corollary 1 *If $\mathcal{S}$ is a rulebase with substitution, then* $\mathcal{S} \models \mathcal{S}^*$.

Lemma 6 *Suppose β and $\beta_1 ... \beta_n$ are predicates with substitution, $\mathcal{S}$ is a rulebase with substitution, and DB is a database. Then,*

- $\mathcal{S}, DB \models DB$
- *If* $\mathcal{S} \models \beta \leftarrow \beta_1 ... \beta_n$ *and* $\mathcal{S}, DB \models \beta_i$ *for each i, then* $\mathcal{S}, DB \models \beta$.

This lemma shows that semantically, we can derive formulas from a database DB by applying the rules entailed by $\mathcal{S}$ as if they were Horn rules. In particular, we can apply the rules in $\mathcal{S}^*$ to DB as if they were Horn rules. We thus get the following corollary.

Corollary 2 *If* $\mathcal{S}^* \cup DB \vdash_c \beta$ *then* $\mathcal{S}, DB \models \beta$.

Soundness follows by definition 4.

4 Fixpoint Semantics

As in the theory of Horn logic [8, 1], we complement the model-theoretic semantics with a fixpoint semantics. In particular, we define a monotonic "T-operator" which 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 input and output of the T-operator is a Horn rulebase. Starting with the empty rulebase, the T-operator is applied over-and-over again until the minimal model is generated. Each application of the operator expands rules with substitution into equivalent Horn rules. If there is recursion through substitution, it may take infinitely many applications to generate all the Horn rules in the minimal model. However, because the operator is continuous, any particular Horn rule will be generated in a finite number of applications. In this way, the Horn rules represented by a rulebase with substitution can be materialized.

Formally, the set of Horn rulebases (i.e., the interpretations of rulebases with substitution) is viewed as a lattice, and the T-operator maps one element of this lattice onto another element. The operator is then shown to be monotone and continuous, and the Tarski fixpoint theorem [6] is applied in the usual way. A precise statement of definitions and results is provided. Several examples are given of the application of the T-operator to actual rulebases, illustrating the way in which each application of the operator "unfolds" rules with substitution to produce equivalent Horn rules. When no more unfolding is possible, the least fixpoint has been reached.

Lemma 7 *The set of Horn rulebases (without substitution) forms a complete lattice under the following operations:*

- $R_1 \leq R_2$ *iff* $R_2 \models_c R_1$
- $R_1 \sqcup R_2 \;=\; R_1 \cup R_2$.
- $R_1 \sqcap R_2 \;=\; \{\phi \mid R_1 \models_c \phi \text{ and } R_2 \models_c \phi\}$ *where each ϕ is a Horn rule.*

Note that if R_1 and R_2 are classically equivalent, then $R_1 \leq R_2$ and $R_2 \leq R_1$, so that $R_1 = R_2$ in the lattice. It is reasonable to treat equivalent rulebases as equal since if R_1 and R_2 are equivalent, then for any rulebase with substitution $\mathcal{S}$, R_1 is a model of $\mathcal{S}$ iff R_2 is.

Definition 12 *(T-operator) Let $\mathcal{S}$ be a Horn rulebase with substitution, and let R be a Horn rulebase (without substitution). Then*

$$T_{\mathcal{S}}(R) \;=\; \cup_{\sigma}\{\; \alpha\sigma \leftarrow \gamma \;\mid\; \alpha \leftarrow \beta_1...\beta_n \;\in\; \mathcal{S} \;\; \textit{and} \;\; R \models \beta_i\sigma \leftarrow \gamma \;\; \textit{for each } i\}$$

where each σ is a term substitution, and each γ is a list of atomic formulas.

A fixpoint of $T_{\mathcal{S}}$ is a Horn rulebase R such that $T_{\mathcal{S}}(R) \leq R$. Our first result says that R is a Horn model of $\mathcal{S}$ iff it is a fixpoint of $T_{\mathcal{S}}$. The proof is straightforward.

Lemma 8 $R \models \mathcal{S}$ *iff* $T_{\mathcal{S}}(R) \leq R$

Thus, as in the fixpoint semantics of Horn logic, we can study the models of a rulebase by studying the fixpoint structure of its T-operator.

Lemma 9 *$T_{\mathcal{S}}$ is a monotonic operator. That is, if $R_1 \leq R_2$ then $T_{\mathcal{S}}(R_1) \leq T_{\mathcal{S}}(R_2)$.*

Corollary 3 *$T_{\mathcal{S}}$ has a unique minimal fixpoint $lfp(T_{\mathcal{S}})$.*

Since every fixpoint is a model (and vice-versa), this shows that a rulebase with substitution $\mathcal{S}$ has a unique minimal model. The minimal model contains all the important information about $\mathcal{S}$, as the following corollary shows.

Corollary 4 *Let $\mathcal{S}$ be a Horn rulebase with substitution, β be a predicate with substitution, and DB be a database. Then,*

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

By showing that it is continuous, the T-operator can be used to construct the least fixpoint.

Lemma 10 *$T_{\mathcal{S}}$ is a continuous operator. i.e., If $R_1 \leq R_2 \leq R_3$ ..., then $T_{\mathcal{S}}(\cup_i R_i) = \cup_i \; T_{\mathcal{S}}(R_i)$.*

Theorem 3 $lfp(T_{\mathcal{S}}) = T^*(\{\}) = \cup_i \; T^i_{\mathcal{S}}(\{\})$

This theorem says that the least fixpoint can be constructed in a bottom-up, iterative fashion by starting with the empty rulebase and applying the operator $T_{\mathcal{S}}$ to it over and over again until there are no more changes. In general, this will take infinitely many iterations and $lfp(T_{\mathcal{S}})$ itself will be infinitely large. However, only finitely many iterations are needed to determine whether a particular Horn rule is in $lfp(T_{\mathcal{S}})$.

4.1 Examples

As the operator $T_{\mathcal{S}}$ is applied over-and-over again, the substitution rules in $\mathcal{S}$ are expanded into equivalent Horn rules. The first application of $T_{\mathcal{S}}$ extracts the Horn rules in $\mathcal{S}$. These rules then act as seeds for the generation of new Horn rules. Thus, the second application of $T_{\mathcal{S}}$ combines these Horn rules with the substitution rules in $\mathcal{S}$ to generate new Horn rules. The third application then combines these new Horn rules with the substitution rules to generate still more Horn rules. When no new Horn rules can be generated in this fashion, the least fixpoint has been computed. The examples below illustrate this bit-by-bit conversion of substitution rules into Horn rules.

In general, $T^{i-1}_{\mathcal{S}}(\{\}) \subseteq T^i_{\mathcal{S}}(\{\})$; so for the purpose of the examples below, we define $\Delta T^i_{\mathcal{S}}(\{\}) =$

$T^i_{\mathcal{S}}(\{\}) - T^{i-1}_{\mathcal{S}}(\{\})$. $\Delta T^i_{\mathcal{S}}(\{\})$ is the set of new Horn rules created by the i^{th} application of $T_{\mathcal{S}}$.

Example 8. Suppose the rulebase $\mathcal{S}$ consists of the following three rules, where B, C and D are base predicates:

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

It then takes two applications of the operator $T_{\mathcal{S}}$ to generate the minimal model of $\mathcal{S}$. That is, $lfp(T_{\mathcal{S}}) = T^2_{\mathcal{S}}(\{\})$. This model consists of four Horn rules, generated as follows:

$$\left.\begin{aligned}
&A(x) \leftarrow B(x), C(x).\\
&A(x) \leftarrow D(x,y), C(x), C(y).
\end{aligned}\right\} \in T^1_{\mathcal{S}}(\{\})$$

$$\left.\begin{aligned}
&A'(x) \leftarrow B(x), C'(x).\\
&A'(x) \leftarrow D(x,y), C'(x), C'(y).
\end{aligned}\right\} \in \Delta T^2_{\mathcal{S}}(\{\})$$

Thus, the definition of the predicate A' is the same as the definition of A except that C has been replaced everywhere by C'. $\Box$

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 Horn rules are generated, more and more substitutions become available for conversion into Horn rules. This is illustrated by the following propositional example.

Example 9. Suppose the rulebase $\mathcal{S}$ consists of the following $n+1$ rules, where each B_i is a base predicate:

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

It then takes $n+1$ applications of the operator $T_{\mathcal{S}}$ to generate the minimal model of $\mathcal{S}$. That is, $lfp(T_{\mathcal{S}}) = T^{n+1}_{\mathcal{S}}(\{\})$. This model consists of $n+1$ Horn rules, generated as follows:

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

The $i+1^{st}$ application of $T_{\mathcal{S}}$ effectively makes a copy of the rule defining A_i, substitutes C_i for B_i in the premise, and then uses this as the definition of a new predicate A_{i+1}. $\Box$

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 $T_{\mathcal{S}}$. Unlike the previous examples, the next example cannot be terminated after a finite number of iterations.

Example 10. Suppose the rulebase $\mathcal{S}$ consists of the following three rules:

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

where f is a unary function symbol. Then,

$$\begin{aligned}
&\textit{if} \quad A(x) \leftarrow B(x,y), B(x,z) \in T^i_{\mathcal{S}}(\{\})\\
&\textit{then} \quad A(x) \leftarrow B(x,fy), B(x,fz) \in T^{i+1}_{\mathcal{S}}(\{\})
\end{aligned}$$

Thus,

$$A(x) \leftarrow B(x,a), B(x,b). \;\in\; T^1_{\mathcal{S}}(\{\})$$
$$A(x) \leftarrow B(x,fa), B(x,fb). \;\in\; \Delta T^2_{\mathcal{S}}(\{\})$$
$$A(x) \leftarrow B(x,f^2a), B(x,f^2b). \;\in\; \Delta T^3_{\mathcal{S}}(\{\})$$
$$\ldots$$
$$A(x) \leftarrow B(x,f^ia), B(x,f^ib). \;\in\; \Delta T^i_{\mathcal{S}}(\{\})$$
$$\ldots$$

□

5 Concluding Remarks

We believe that substitution is a principle which can be applied to logic-based languages generally in order to facilitate reuse and modification. In this paper, however, we focused on Horn logics and on Datalog in particular. Several examples were given to illustrate the idea of substitution and its utility.

We established formal proof and model theories for Horn logic with predicate substitution. In particular, we developed (in our opinion) natural semantics in which an ordinary set of substitution-free rules is a model of a set of rules with substitution. This corresponds to the intuition that rules with substitution are written by a user who has rules without substitution "in mind," rules which he might have written if he had enough space. This is similar to the view taken of ordinary Datalog rules, in which the user has an "intended model" in mind (the least fixpoint of the rules), which he would have written if he had enough space. Hence, it can be argued that by moving from Datalog to Datalog-with-substitution, we achieve an upgrade similar to that of going from relational algebra to Datalog.

Substitution adds a great deal of power to Datalog. Whereas the complexity of Datalog is complete for $PTIME$, the complexity of Datalog augmented with substitution is complete for $EXPTIME$. An example was presented showing that its data complexity is NP-hard. Syntactically, this boost in expressive power comes about from recursion through substitution. Semantically, this increased power has an interesting interpretation: When a Datalog rulebase with substitution is $EXPTIME$-complete, it represents an *infinite* Datalog rulebase without substitution. Furthermore, this infinite Datalog rulebase is not equivalent to any finite one. In such cases, substitution provides an extremely concise notation for representing a set of Datalog rules.

We view predicate substitution as a principle which in order to facilitate the reuse and modification of deductive rules, has to be added to logic-based languages. Predicate substitution can be implemented in several existing formalisms: state variables (with unary function symbols), complex objects, or Prolog. However, predicate substitution assures reusability with the minimal possible cost (i.e. with a minimal increase in expressive power). Besides, we see Datalog with predicate substitution as a natural *specification language* facilitating reuse. Indeed, we insist that the concept of predicate substitution is a more natural specification vehicle for reusability than the concepts of state variable or complex objects.

In this paper, we studied the complexity and expressive power of Datalog with substitution; but we plan to investigate the effect of substitution on other languages, such as the various extensions of Datalog with complex objects [4, 7, 2], as well as extensions of Datalog which allow restricted use of function symbols, as in [3].

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] C. Beeri, S. Naqvi, and R. Ramakrishnan. Sets and Negation in a Logic Database Language (LDL). In *Proceedings of the ACM Symposium on the Principles of Database Systems (PODS)*, San Diego, CA, 1987.

[3] J. Chomicki and T. Imielinski. Relational Specification of Infinite Query Answers. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Portland, Oregon, May 1989.

[4] G.M. Kuper. Logic Programming with Sets. In *Proceedings of the ACM Symposium on the Principles of Database Systems (PODS)*, San Diego, CA, 1987.

[5] L.T. McCarty and N.S. Sridharan. The Representation of an Evolving System of Legal Concepts. II. Prototypes and Deformations. In *Proceedings of the Seventh IJCAI*, pages 246–253, 1981.

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

[7] S. Tsur and C. Zaniolo. LDL: A Logic-Based Data-Language. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, Kyoto, Japan, 1986.

[8] 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.