Specification and Implementation of Programs for Updating Incomplete Information Databases
(Preliminary Report)

Stephen J. Hegner
Department of Computer Science and Electrical Engineering
Votey Building
University of Vermont
Burlington, VT 05405
hegner@uvm
..!(decvax,ihnp4)!dartvax!uvm-gen!hegner

Abstract

The problem of updating incomplete information databases is viewed as a programming problem. From this point of view, formal denotational semantics are developed for two applicative programming languages, **BLU** and **HLU**. **BLU** is a very simple language with only five primitives, and is designed primarily as a tool for the implementation of higher-level languages. The semantics of **BLU** are formally developed at two levels, possible worlds and clausal, and the latter is shown to be a correct implementation of the former. **HLU** is a "user level" update language. It is defined entirely in terms of **BLU**, and so immediately inherits its semantic definition from that language. This demonstrates a level of completeness for **BLU** as a level of primitives for update language implementation. The necessity of a particular **BLU** primitive, *masking,* suggests that there is a high degree of inherent complexity in updating logical databases.

0. Introduction

Database systems may be viewed as consisting of two components. A *database schema* specifies the general structure of admissible data, and remains constant. *Database instances*, on the other hand, record the actual state of the world at a given point in time, and changes upon update. In the case of complete information, there is exactly one instance associated with the system at any given point, whereas in the incomplete information case, there is a collection of alternative instances, or *possible worlds.*

In the complete information case, the representation of the system state is a usually a direct one (such as a set of relations in the relational case), although indirect representation is also possible, as in the *negation as failure,* or *closed world* clausal representation [3].

In the case of incomplete information, on the other hand, direct representation is impractical, due to the potential size of the set of possible worlds. Therefore, a method of indirect representation must be employed. These include *template* methods [12], as well as the use of logic [16]. Approaches which combine these two philosophies have also been suggested [11]. The key point is that, *regardless of the method of representation, the foundations rest in possible world semantics.*

It is our thesis that for the purposes of *updating* an incomplete information database, similar principles should apply. Fundamental semantics should be at the possible worlds level, while the representation and manipulation mechanism needs to be at an indirect level to be practicable.

In this work, we present the foundations for understanding the process of updating incomplete information databases. The basic idea is to regard updates to databases as specified in an update programming language. To such a language, we assign an *instance semantics* which describes how its programs behave at the level of possible worlds. Any other implementation, using an indirect form of possible worlds representation, must respect this instance semantics.

We actually develop two update programming languages, **HLU** and **BLU**. **HLU** (for **H**igh-level **L**anguage for **U**pdates) is our "user level" language for expressing updates. It has only two basic *sorts* or primitive data types, **`<possible-worlds>`** and **`<masks>.`** Its syntax is summarized by the following set of productions.

```
<HLU-program>  →
  (assert <possible-worlds>) |
  (clear <mask>) |
  (insert <possible-worlds>) |
  (delete <possible-worlds>) |
  (modify <possible-worlds> <possible-worlds>) |
  (where <possible-worlds> <HLU-program>) |
  (where <possible-worlds>
         <HLU-program> <HLU-program>)
```

HLU may be implemented at various levels, including instances, templates, and logic. The formal semantics is presented in Section 3; here we give an informal sketch for motivational purposes. but independently of any particular implementation. It is always assumed that there is a particular extant collection of possible worlds, denoted by S, which is the current state of the database system. For any representation W of **`<possible-worlds>`**, let **pw**(W) denote the actual collection of possible worlds represented. Each **HLU** program modifies the current state. The program **`(assert W)`** modifies the database state S to one in which the the only possible worlds are those common to S and **pw**(W). It monotonically increases the information in the state S, by reducing the membership in the collection of possible worlds of S. The program **`(mask M)`** modifies the database state to be a *view* of its previous state, by masking out all information of a certain nature specified by the mask. For example, in the clause world, the program **`(mask {A,B})`** would remove from S all information regarding the truth values of A and B. The program **`(insert W)`** generalizes the notion of insertion into a complete information database. The programs **`(delete W)`** and **`(modify W V)`** similarly generalize the notions of deletion from and modification of complete information databases in a manner which will be made precise later. The control program **`(where W P Q)`** splits S into two parts,

$S \cap \mathbf{pw}(W)$ and $S \setminus \mathbf{pw}(W)$. The program P is then run on the first set of worlds and the program Q on the second, and the results are then combined. The program **`(where W P)`** is equivalent to **`(where W P I)`**, where I is the identity program.

We do not implement **HLU** directly. Rather, the semantics of **HLU** is expressed *formally* as programs in a more basic language, which we have named **BLU** (for **B**asic **L**anguage for **U**pdates. Despite the fact that **BLU** has only five very elementary primitive operations, it is more than powerful enough to support the implementation of **HLU**. Indeed, it is a major claim of this work that the **BLU** primitives are precisely those needed for update language implementation. Underlying this claim is the *mask-assert paradigm,* which states that all updates are founded upon the composition of two operations; a *masking* which projects out certain information, and so constitutes a *decrease* in information content of the database, followed by an *assertion* which restricts the set of possible worlds, and so constitutes an *increase* in information content.

Two implementations of **BLU** are provided. First, we define the possible worlds instance semantics. Then, we provide an *algorithmic* clause-level implementation, based upon the resolution method of clausal inference. Because **HLU** is defined in terms of **BLU**, these immediately provide implementations of **HLU** also.

The core of our presentation is based initially upon propositional logic. This enables us to concentrate on core issues, without becoming bogged down in the details of a full relational framework. Nonetheless, even if "grounding techniques" are employed to convert a finite relational framework to an equivalent propositional one, both conceptual and practical problems remain. These issues are briefly addressed at the end of the paper.

Remark This paper is in the form of an extended abstract. Due to space limitations, proofs are generally omitted, and topics are often treated in a terse and/or somewhat informal fashion. It is anticipated that more formal and elaborated versions of these results will appear elsewhere.

1. Foundations of Propositional Database Systems

Since database systems founded upon propositional logic underly many of the developments in this work, it is essential that we begin with a firm understanding of precisely what is meant by a database system and related concepts within a propositional framework. Such is the purpose of this section.

1.1 Propositional Logic

Familiarity with propositional logic, as discussed in, *e.g.,* [6], is assumed. The primary purpose of this section is to establish a notational base.

A *propositional logic* is a pair **L**=(**P**,**C**), with **P** a set of proposition names (denoted $\mathbf{Prop}(\mathbf{L})$) and the nonlogical symbols $\mathbf{C} = \{\wedge, \vee, \neg, \Rightarrow, \Leftrightarrow, (,)\}$. In this work, **P** is generally taken to be finite, and is usually taken to be a sequence of symbols named by a single letter indexed by an initial segment of natural numbers; *e.g.,* $\mathbf{P} = \{A_1, A_2, \ldots, A_n\}$. These indices give **P** an implicit order. In this work, the set of nonlogical symbols is always the same as given above. Thus, as an abuse of notation, we always identify the propositional logic with its set of propositional symbols.

A *structure* for **L** is a function $s : \mathbf{A} \to \{0,1\}$ and may be represented naturally as an n-tuple over $\{0,1\}$, with the i-th entry the value of $s(A_i)$. The set of all structures for **L** is denoted $\mathbf{Struct}[\mathbf{L}]$.

$\mathbf{WF}[\mathbf{L}]$ denotes the set of *well-formed formulas* (*wff's*) for **L**. $\bar{s} : \mathbf{WF}[\mathbf{L}] \to \{0,1\}$ denotes the natural extension of the structure s to $\mathbf{WF}[\mathbf{L}]$. For $\Phi \subseteq \mathbf{WF}[\mathbf{L}]$, $\mathbf{Prop}[\Phi]$ = $\{ A \in \mathbf{L} \mid A$ occurs in some formula $\phi \in \Phi \}$, while $\mathbf{Mod}[\Phi]$ = $\{ s \in \mathbf{Struct}[\mathbf{L}] \mid \bar{s}(\phi) = 1$ for each $\phi \in \Phi \}$. Conversely, for $S \subseteq \mathbf{Struct}[\mathbf{L}]$, $\mathbf{Sat}[S]$ = $\{ \phi \in \mathbf{WF}[\mathbf{L}] \mid \bar{s}(\phi) = 1$ for all $s \in S \}$. $\mathbf{Dep}[S]$ defi nes the *dependency set* of S; it consists of all proposition letter which occur in every Φ for which $\mathbf{Mod}[\Phi] = S$. Finally, the *theory* of Φ is $\mathbf{Th}[\Phi] = \{ \phi \mid \Phi \models \{\phi\} \}$.

We also assume familiarity with resolution and the associated language of clauses, as described in [2]. $\mathbf{Lit}[\mathbf{L}]$ denotes the set of all literals over **L**, while $\mathbf{CF}[\mathbf{L}]$ denotes the set of all clauses over **L**. $\mathbf{Lit}[\phi]$ denotes just the set of literals occurring in the clause ϕ. The *length* of a clause ϕ is the number of distinct literals occurring in that clause, while the length of a set Φ of clauses is the sum of the lengths of its constituents. The notation $\mathbf{Length}[\Phi]$ is used to denote the length of the set of clauses Φ. $\square$ is reserved to denote the empty clause, as is **0**; the two are entirely synonymous. **1** denotes a tautological clause which is "always true". $\mathbf{Resolvent}(\phi_1, \phi_2, A)$ is the resolvent, with respect to atom A, of the clauses ϕ_1 and ϕ_2, if it exists.

1.2 Database Schemata and Instances

In the logical approach to relational database systems, as described in, for example, [20] or [8], a database schema **E** is given by a fi nite set of *relation names* **R**, a fi nite set of *constant names* **K**, and a set of typing and domain closure constraints $\mathbf{TC}$. A *ground fact* is just a formula of the form $R(a_1, a_2, ..., a_n)$, in which $R \in \mathbf{R}$, each $a_i \in \mathbf{K}$, and which satisfi es all typing constraints. (For now, just think of a typing and domain closure constraint as rules stating precisely which constant names may occur in which positions in an elementary fact, and that these are the only elementary facts possible.) Since both **R** and **K** are taken to be fi nite, so too is the collection of elementary facts. The *grounding* of **E** yields a propositional schema **D**, whose atom names are precisely the elementary facts of **E**.

Upon grounding, the *state* of **E** may be completely represented by an interpretation $s : \mathbf{Prop}[\mathbf{D}] \to \{0,1\}$, with $s(R(a_1, a_2, ..., a_k)) = 1$ if and only if the tuple $(a_1, a_2, ..., a_k)$ is "in" the relation R in the state s.

The viability of the grounding technique is at the root of the justifi cation for using a propositional logic to study updates to incomplete information databases, and is invoked explicitly in [22] and at least implicitly in [7]. We shall have more to say on this issue in Section 5; for now we proceed to the formal development of the propositional framework.

1.2.1 Definition (a) A (propositional) *database schema* is a pair **D** = $(\mathbf{Prop}[\mathbf{D}], \mathbf{Con}[\mathbf{D}])$, in which $\mathbf{Prop}[\mathbf{D}]$ is a propositional logic and $\mathbf{Con}[\mathbf{D}] \subseteq \mathbf{WF}[\mathbf{Prop}[\mathbf{D}]]$, is the set of *integrity constraints.* In general, we may use any notation given for a propositional logic; so, for example, $\mathbf{WF}[\mathbf{D}]$ denotes the set of well-formed formulas over the propositional logic associated with **D**.

(b) A *database* for $\mathbf{D}$ is any $s \in \mathbf{Struct}[\mathbf{Prop}[\mathbf{D}]]$; s is *legal* if $s \in \mathbf{Mod}[\mathbf{Con}[\mathbf{D}]]$ also. We write $\mathbf{DB}[\mathbf{D}]$ for the set of all databases for $\mathbf{D}$, and $\mathbf{LDB}[\mathbf{D}]$ for just the legal ones.

1.2.2 Definition An *incomplete information database* for $\mathbf{D}$ is a subset of $\mathbf{DB}[\mathbf{D}]$; $\mathbf{IDB}[\mathbf{D}]$ denotes the set of all incomplete information databases for $\mathbf{D}$. Similarly, a *legal* incomplete information database is a subset of $\mathbf{LDB}[\mathbf{D}]$, and we write $\mathbf{ILDB}[\mathbf{D}]$ to denote the set of all legal incomplete information databases for $\mathbf{D}$. Each member s of $S \in \mathbf{ILDB}[\mathbf{D}]$ is called a *possible world* of S.

1.2.3 Notational Convention Throughout the rest of this paper, the symbols $\mathbf{D}$ and $\mathbf{D}_i$, for i an integer, shall denote arbitrary propositional database schemata, without the need to explicitly designate them as such.

Crucial to the entire theory presented here is the way in which the complete information case extends to the incomplete information case. This is formalized at the level of database instances by the following identifi cation maps.

1.2.4 Definition Let $\mathbf{D}$ be a database schema. The *natural identification maps*

$$\mathbf{DB}[\mathbf{D}] \to \mathbf{IDB}[\mathbf{D}]$$
$$\mathbf{LDB}[\mathbf{D}] \to \mathbf{ILDB}[\mathbf{D}]$$

are those which send each element to its corresponding singleton; *i.e.,* $S \mapsto \{S\}$.

1.3 Deterministic Database Morphisms

On the logical level, a database morphism is an interpretation between theories. The idea of regarding a database morphism as such was fi rst explicitly proposed by Jacobs [13,14], although it has been implicit in the defi nition of queries at least since the early work of Codd [4]. It is also the basis for an extensive study of database decomposition [9], to which the reader is referred for more motivation and discussion.

1.3.1 Definition A *(deterministic) morphism* $f : \mathbf{D}_1 \to \mathbf{D}_2$ is an assignment $\mathbf{Prop}[\mathbf{D}_2] \to \mathbf{WF}[\mathbf{D}_1]$. (Note the direction!) f extends naturally to $\bar{f} : \mathbf{WF}[\mathbf{D}_2] \to \mathbf{WF}[\mathbf{D}_1]$ by substituting $f(A_i)$ for each occurrence of A_i. If $f : \mathbf{D}_1 \to \mathbf{D}_2$ and $g : \mathbf{D}_2 \to \mathbf{D}_3$ are morphisms, the composition $g \circ f : \mathbf{D}_1 \to \mathbf{D}_3$ is defi ned by

$$\mathbf{Prop}[\mathbf{D}_3] \xrightarrow{g} \mathbf{WF}[\mathbf{D}_2] \xrightarrow{\bar{f}} \mathbf{WF}[\mathbf{D}_1]$$

Defi ne $f' : \mathbf{DB}[\mathbf{D}_1] \to \mathbf{DB}[\mathbf{D}_2]$ by $s \mapsto (A_i \mapsto \bar{s}(f(A_i)))$. f' is extended to operate on incomplete information databases $\mathbf{IDB}[\mathbf{D}] \to \mathbf{IDB}[\mathbf{D}]$ by the rule $S \mapsto \bigcup\{f(s) \mid s \in S\}$. As a slight abuse of notation, we use the symbol f' to denote both of these structure mappings.

1.3.2 Fact Let $f : \mathbf{D}_1 \to \mathbf{D}_2$ and $g : \mathbf{D}_2 \to \mathbf{D}_3$ be database morphisms. Then $(g \circ f)' = g' \circ f'$. □

The morphism $f : \mathbf{D}_1 \to \mathbf{D}_2$ is *correct* if either of the equivalent conditions of the second part of 1.3.3 is met. It is easy to see that the composition of correct morphisms is itself correct.

In a complete relational database, there is an implicit *closed world assumption* which states that tuples which are not presented are assumed to designated false statements. A request to insert a tuple t means that whatever knowledge currently exists regarding the truth value of the information represented by t should be replaced with the fact that it is true. The truth values of other tuples are not affected. In a deletion, whatever knowledge is currently present regarding that tuple is replaced by the knowledge that the information represented by the tuple is now false. Modification is slightly more complex. Here we wish to change a a tuple t to a tuple u, provided that t is present. If t is absent, we do nothing. Thus, the truth value of the information associated with t becomes false regardless, while the truth value of the information associated with u becomes true if either it were true before, or else the truth value of t is true. Relative to the propositional framework, this is all formalized as follows.

1.3.3 Definition Let **D** be a database schema, and let $A_i, A_j \in \mathbf{Prop}[\mathbf{D}]$.

(a) $\text{insert}[A_i] : \mathbf{D} \to \mathbf{D}$ is given by

$$A_k \mapsto \begin{cases} \mathbf{1} & (k = i) \\ A_k & (k \neq i). \end{cases}$$

(b) $\text{delete}[A_i] : \mathbf{D} \to \mathbf{D}$ is given by

$$A_k \mapsto \begin{cases} \mathbf{0} & (k = i) \\ A_k & (k \neq i). \end{cases}$$

(c) $\text{modify}[A_i, A_j] : \mathbf{D} \to \mathbf{D}$ is given by

$$A_k \mapsto \begin{cases} \mathbf{0} & (k = i) \\ A_i \vee A_j & (k = j) \\ A_k & (k \neq i, j). \end{cases}$$

Note that the definition of f’ on incomplete information databases immediately tells us how to interpret these update operations on such databases; namely, update each possible world individually. It should also be noted that the above update operations are not necessarily correct. In the deterministic case, the updated database is computed, and then checked for compliance with the integrity constraints. If those constraints are not satisfied, the update is rejected. In the incomplete information case, it is possible to interpret the update somewhat differently. We update each possible world individually, and then those which are not legal are eliminated. In either case, the process of enforcing integrity constraints is not immediately representable as a morphism operation. For this reason, we shall, unless otherwise mentioned, ignore integrity constraints in the basic handling of updates.

It is convenient to extend the definition of insertion to include not just atom names, but literals as well, with $\text{insert}[\neg A_k]$ defined as $\text{delete}[A_k]$. Then, the insertion operation may be extended to sets of consistent literals by just inserting them all. The modify operation has a similar extension; in $\text{modify}[\Phi_1, \Phi_2]$, if each literal in Φ_1 is true, we delete the literals of Φ_1 and then insert the literals of Φ_2. The formal definitions follow.

1.3.4 Definition Let **D** be a database schema, and let Φ_1 and Φ_2 be consistent sets of literals over $\mathbf{Prop}[\mathbf{D}]$.

(a) $\text{insert}[P] : \mathbf{D} \to \mathbf{D}$ is given by

$$A_k \mapsto \begin{cases} \mathbf{1} & (A_k \in \Phi_1) \\ \mathbf{0} & (\neg A_k \in \Phi_1) \\ A_k & (A_k, \neg A_k \in \Phi_1) \end{cases}$$

(b) $\mathrm{modify}[\Phi_1, \Phi_2] : \mathbf{D} \to \mathbf{D}$ is given by

$$A_k \mapsto \begin{cases} \mathbf{1} & (\bigwedge\Phi_1 \equiv \mathbf{1} \text{ and } ((A_k \in \Phi_1 \backslash \Phi_2) \text{ or } (\neg A_k \in \Phi_2))) \\ \mathbf{0} & (\bigwedge\Phi_1 \equiv \mathbf{1} \text{ and } ((\neg A_k \in \Phi_1 \backslash \Phi_2) \text{ or } (A_k \in \Phi_2))) \\ A_k & (\bigwedge\Phi_1 \equiv \mathbf{0}) \end{cases}$$

1.4 Nondeterministic Database Morphisms

Incomplete information may arise in a database system in two distinct ways. First, the database itself may not represent complete information, but rather a set of possible alternatives. We have already examined this type of incompleteness in the previous two subsections. Second, a database mapping, such as an update, may itself be incompletely specified (such as in a request to insert $A_1 \vee A_2$). To represent the latter type of incompleteness, we introduce the concept of a nondeterministic morphism.

1.4.1 Definition (a) A *nondeterministic morphism* of database schemata $F : \mathbf{D}_1 \circ\!\!\to \mathbf{D}_2$ is a set of deterministic morphisms from $\mathbf{D}_1$ to $\mathbf{D}_2$. Thus, each $f \in F$ is a function $f : \mathbf{Prop}[\mathbf{D}_2] \to \mathbf{WF}[\mathbf{D}_1]$. We shall always use the "$\circ\!\!\to$" arrow to denote nondeterministic morphisms, while the ordinary "$\to$" will be used to denote deterministic morphisms, so no confusion can result.
(b) If $F : \mathbf{D}_1 \circ\!\!\to \mathbf{D}_2$ and $G : \mathbf{D}_2 \circ\!\!\to \mathbf{D}_3$, then $G \circ F : \mathbf{D}_1 \circ\!\!\to \mathbf{D}_3$ is defined to be $\{ g \circ f \mid f \in F \text{ and } g \in G \}$.
(c) For $F : \mathbf{D}_1 \circ\!\!\to \mathbf{D}_2$, define the extension $F' : \mathbf{DB}[\mathbf{D}_1] \to \mathbf{IDB}[\mathbf{D}_2]$ by $s \mapsto \{ f'(s) \mid f \in F \}$. Define $\bar{F} : \mathbf{IDB}[\mathbf{D}_1] \to \mathbf{IDB}[\mathbf{D}_2]$ by $S \mapsto \bigcup \{ F'(s) \mid s \in S \}$.

1.4.2 Fact Let $F : \mathbf{D}_1 \circ\!\!\to \mathbf{D}_2$ and $G : \mathbf{D}_2 \circ\!\!\to \mathbf{D}_3$. Then $(G \circ F)' = (G' \circ F')$. □

It is essential that the definitions of nondeterministically specified updates (such as the insertion of $A_1 \vee A_2$) be extensions of the deterministic cases. In other words, a request to nondetermistically insert A_1 should be identical in action to a request to deterministically perform the update. The following definition formalizes the obvious embedding.

1.4.3 Definition Let $f : \mathbf{D}_1 \to \mathbf{D}_2$ be a deterministic database morphism. The *corresponding nondeterministic morphism* is $\{f\}$.

We now turn to the issue how to interpret a nondeterministic update request such as "insert[$A_1 \vee A_2$]". The idea is to regard such a request as a nondeterministic morphism, each of whose components is a deterministic insertion request (to a possibly incomplete information database). Thus, to extend the operation of insertion to nondeterministically specified updates, it is necessary to express it as a nondeterministic morphism, each of whose components is a deterministic insertion request. The deletion and modification operations extend in a similar fashion. The formalization is contained in the following.

1.4.4 Definition Let $\Phi \subseteq \mathbf{WF}[\mathbf{D}]$.
(a) The *literal base* of Φ, denoted $\mathbf{LB}[\Phi]$, is the set

$\{ \Psi \subseteq \mathbf{Lit}[\mathbf{D}] \mid \Psi$ is consistent and $\Psi \models \Phi \}$.
(b) For $l \in \mathbf{Lit}(\mathbf{D})$, l is *irrelevant* if for every $\Psi \in \mathbf{LB}(\Phi)$, $l \in \Psi$ implies that $\Psi \setminus \{l\}$, $\Psi \setminus \{\neg l\} \in \mathbf{LB}(\Phi)$ also. Ψ is *minimal* if it contains no irrelevant elements.
(c) $\Psi \in \mathbf{LB}[\Phi]$ is *complete* if it is minimal, and, for any other $\Lambda \in \mathbf{LB}[\Phi]$, $\Psi \subseteq \Lambda$ implies that $\Psi = \Lambda$.
(c) $\mathbf{Inset}[\Phi] = \{ \Phi \in \mathbf{LB}[\Phi] \mid \Psi$ is complete$\}$. $\mathbf{Inset}[\Phi]$ is called the *literal insertion set* for Φ.

1.4.5 Definition Let $\Phi \subseteq \mathbf{WF}[\mathbf{D}]$.
(a) Define the nondeterministic morphism $\mathrm{insert}[\Phi] : \mathbf{D} \circ\!\!\rightarrow \mathbf{D}$ as $\{ \mathrm{insert}[\Psi] : \mathbf{D} \rightarrow \mathbf{D} \mid \Psi \in \mathbf{Inset}[\Phi] \}$.
(b) Define the nondeterministic morphism $\mathrm{delete}[\Phi] : \mathbf{D} \circ\!\!\rightarrow \mathbf{D}$ as $\mathrm{insert}[\Psi] : \mathbf{D} \rightarrow \mathbf{D}$, with $\Psi = \{ \neg(\bigwedge\phi) \mid \phi \in \Phi_1 \}$.
(c) Define the nondeterministic morphism $\mathrm{modify}[\Phi_1, \Phi_2] : \mathbf{D} \circ\!\!\rightarrow \mathbf{D}$ as $\{ \mathrm{modify}[\Psi_1, \Psi_2] : \mathbf{D} \rightarrow \mathbf{D} \mid \Psi_1 \in \mathbf{Inset}[\Phi_1]$ and $\Psi_2 \in \mathbf{Inset}[\Phi_2] \}$.

1.4.6 Discussion It is important to understand these concepts intuitively. As a concrete example, let $\Phi = \{A_1 \vee A_2\}$. The literal base of Φ consists of sets of literals, with each set sufficiently rich to semantically entail Φ. A minimal such set contains no literals which are totally irrelevant to the truth value of Φ. Thus, $\{A_1, \neg A_2, A_3\}$ is in the literal base of Φ, but is not minimal, since A_3 is irrelevant. A complete set contains enough literals to span all truth values which need to be known. Thus, $\{A_1\}$, while minimal, is not complete for Φ. In fact, $\mathbf{Inset}(\Phi) = \{\{A_1, A_2\}, \{A_1, \neg A_2\}, \{\neg A_1, A_2\}\}$. This defines precisely the updates which must be performed to implement the insertion of $A_1 \vee A_2$. Each possible world is replace by three new worlds, one for each of the three deterministic updates $\mathrm{insert}[\{A_1, A_2\}]$, $\mathrm{insert}[\{\neg A_1, A_2\}]$, and $\mathrm{insert}[\{A_1, \neg A_2\}]$. Note that these three correspond precisely to the three possible ways in which truth values can be assigned to A_1 and A_2 such that $A_1 \vee A_2$ is true.

1.4.7 Remark The update semantics which we have just described is very similar to that proposed by Wilkins in [22], although we have arrived at it in quite a different manner. However, there is one very significant difference. Her approach is somewhat syntactic, in that a request of the form $\mathrm{insert}[\{A_1 \vee \neg A_1\}]$ is treated nontrivially. In our framework, such an update request would result in the identity update, since the empty set is complete for $\{A_1 \vee \neg A_1\}$. However, in her approach, the update request would result in what is equivalent to performing each of the four deterministic updates $\mathrm{insert}[\{A_1, A_2\}]$, $\mathrm{insert}[\{\neg A_1, A_2\}]$, $\mathrm{insert}[\{A_1, \neg A_2\}]$, and $\mathrm{insert}[\{\neg A_1, \neg A_2\}]$, which amounts to masking all information regarding A_1. We prefer our definition, because the update only depends upon the semantics of formulas, and not upon the representation. Nonetheless, the action of masking all information about one or more proposition letters is important in its own right, and will be examined more closely in the next subsection.

1.5 Masks and Congruences

Whenever we have a database morphism $f : \mathbf{D}_1 \rightarrow \mathbf{D}_2$ and two states $s_1, s_2 \in \mathbf{DB}(\mathbf{D}_1)$ for which $r = f'(s_1) = f'(s_2)$, the state r does not contain enough information to recover its preimage; some information is *masked.* All we know is that the preimage is a member of $f'^{-1}(r)$; a member of an equivalence class of states of $\mathbf{DB}(\mathbf{D}_1)$. If f is an update operation, it is critical to identify the information which it masks.

1.5.1 Definition Let $F : \mathbf{D} \circ\!\!\rightarrow \mathbf{D}_1$. Define $\mathbf{Congruence}[F]$ to be the equivalence relation on $\mathbf{D}$ defined by $\{(s_1, s_2) \mid f(s_1) = f(s_2) \text{ for all } f \in F\}$. This equivalence relation is called the *mask congruence* of F. A *mask* is any such equivalence relation.

There is a particularly important class of mask congruences, which is defined as follows.

1.5.2 Definition A *symbolwise nondeterministic morphism* $F : \mathbf{D}_1 \circ\!\!\rightarrow \mathbf{D}_2$ is an assignment $F : \mathbf{Prop}[\mathbf{D}_2] \rightarrow \mathbf{2}^{\mathbf{WF}[\mathbf{D}_1]}$. The *corresponding nondeterministic morphism* is given by $\{ f \mid f(A) \in F(A) \text{ for all } A \in \mathbf{Prop}[\mathbf{D}_2] \}$.

1.5.3 Definition Let $P \subseteq \mathbf{Prop}[\mathbf{D}]$.
(a) Define the nondeterministic morphism $\mathrm{mask}[P] : \mathbf{D} \circ\!\!\rightarrow \mathbf{D}$ symbolwise by

$$A_k \mapsto \begin{cases} \{\mathbf{0}, \mathbf{1}\} & (A_k \in P) \\ A_k & (A_k \notin P) \end{cases}$$

(b) Let $P \subseteq \mathbf{Prop}[\mathbf{D}]$. Define the *simple mask* for P to be the congruence induced by the morphism $\mathrm{mask}[P] : \mathbf{D} \rightarrow \mathbf{D}$. This mask is denoted by $\mathbf{s}\!-\!-\mathbf{mask}[P]$. $\mathbf{s\text{-}mask}[\mathbf{D}]$ denotes the collection of all simple masks over $\mathbf{D}$.

The following result is crucial. It says that an insertion masks precisely those proposition letters upon which the inserted formula depends.

1.5.4 Theorem Let $\Phi \subseteq \mathbf{WF}(\mathbf{D})$. Then $\mathbf{Congruence}(\mathrm{insert}[\Phi])) = \mathbf{s}\!-\!-\mathbf{mask}[\mathbf{Prop}[\mathbf{Inset}[\Phi]]]$. □

2. Specification of the Programming Language BLU

In the previous section, we gave precise definitions for basic update operations to incomplete information databases. However, they were just definitions; little was presented which indicated just how one might compute the results of update requests. In this section, we develop a simple applicative programming language **BLU**. The primary purpose of this language is as a tool for the specification and implementation of higher level update languages, rather than as an end in itself.

2.1 The Syntax of BLU

The syntactic specification of **BLU** is specified as an *algebraic signature.* Due to space limitations, we must be brief and somewhat informal. The reader is referred to the excellent references [5] and [17] for a much more detailed development of the relevant issues.

2.1.1 Definition (a) The algebraic signature (or syntax of) **BLU** is defined as follows.
(i) There are two *sorts,* which represents fundamental data types. **S** denotes the sort of *states,* and **M** denotes the sort of masks.
(ii) There are five *operation* symbols. Together with their arities, they are given below.

$$\begin{aligned}
\texttt{assert} &: \mathbf{S} \times \mathbf{S} \to \mathbf{S} \\
\texttt{combine} &: \mathbf{S} \times \mathbf{S} \to \mathbf{S} \\
\texttt{complement} &: \mathbf{S} \to \mathbf{S} \\
\texttt{mask} &: \mathbf{S} \times \mathbf{M} \to \mathbf{S} \\
\texttt{genmask} &: \mathbf{S} \to \mathbf{M}
\end{aligned}$$

(b) There are two countable families of variables, one for each sort. $\mathbf{Var}[\mathbf{S}] = \{\texttt{s0}, \texttt{s1}, \texttt{s2}, \cdots\}$; $\mathbf{Var}[\mathbf{M}] = \{\texttt{m0}, \texttt{m1}, \texttt{m2}, \cdots\}$;

(c) Terms are built up in the standard way. However, we use a Lisp-like list formalism, rather than the more conventional mathematical formalism.

(i) Each **`si`** $\in \mathbf{Var}[\mathbf{S}]$ is an **S**-term, and each **`mi`** $\in \mathbf{Var}[\mathbf{M}]$ is an **M**-term.

(ii) If s_0 and s_1 are **S**-terms, and $\mathbf{m}$ is an **M**-term, then **`(assert`** s_0 s_1**`), (combine`** s_0 s_1**`), (complement`** s_0**`),`** and **`(mask`** s_0**`)`** are **S**-terms, and **`(genmask`** $\mathbf{m}$**`)`** is an **M**-term.

Think of terms for **BLU** as just s-expressions which have the right sorts for their arguments. For example, the following is an **S**-term for **BLU**.

```
(combine (assert (s1
                 (mask (genmask s1)
                       (assert s2 so))))
         (assert (complement s2) s0))
```

2.1.2 Definition A **BLU** program is an expression of the form

`(lambda <varlist> <S-term>)`

subject to the following conditions.

(i) **`<varlist>`** is a list of variables starting with **`s0`**, and containing precisely the variables which occur in the succeeding **S**-term.

(ii) **`<S-term>`** is an **S**-term which contains the variable **`s0`**.

Thus, **BLU** programs are syntactically similar to the lambda forms of Lips and Scheme [19]. The convention of requiring that **`s0`** be present is one of convenience; we will always assume that s0 identifies the program state. This is done to facilitate the definition of the form of **HLU** programs given in the introduction, in which the system state is implicit. **`s0`** will always denote the system state in their implementation.

2.1.3 Example and Discussion The following is a simple **BLU** program

```
(lambda (s0 s1 s2)
        (combine
          (assert s1
                  ((mask (genmask s1)
                         (assert s2 so)))
          (assert (complement s2) s0))))
```

Ultimately (in **HLU**) we will have need for variables which can take on programs as values; that is, we will need to give programs first-class citizenship. Therefore, we use the Scheme formalism [19] **`define`** for the assignment of a program value to a variable, as in

```
(define insert
     (lambda (s0 s1 s2)
        (combine
         (assert s1
                 ((mask (genmask s1)
                        (assert s2 so)))
         (assert (complement s2) s0)))))
```

This example names the program of the previous example **`insert`** by assigning the variable of the same name to have the program definition as its value.

2.2 Fundamental Denotational Semantics of BLU

In assigning a denotational semantics to a programming language, we assume the existence of a set S of underlying states, and we seek a systematic way of assigning a function $S \to S$ to each program. In general programming languages, the real challenge lies in addressing looping and recursion; sophisticated methods of dealing with limits must be employed [17]. However, in **BLU**, there are no looping or recursive constructs, so the formalities of specifying the semantics are quite straightforward.

In this section, we give a simple denotational semantics for **BLU** at the level of structures; the state set S will be $\mathbf{IDB}[\mathbf{D}]$ for our reference database schema **D**. On a more formal level, while the syntax of **BLU** is defined using an algebraic signature, the semantics is defined by actual algebras for this signature. As in the previous subsection, here we present a somewhat informal sketch.

2.2.1 Definition An implementation **A** of **BLU** consists of the following.
(i) The designation of two sets **A**[**S**] and **B**[**S**] which are the *concrete domains* for the sorts.
(ii) The assignment of functions of the appropriate arities to each function symbol. For example, to **`genmask`** we assign a function $\mathbf{A}[\texttt{genmask}] : \mathbf{A}[\mathbf{S}] \times \mathbf{A}[\mathbf{M}] \to \mathbf{A}[\mathbf{S}]$.

Running a **BLU** program in an implementation **A** just amounts to binding appropriate concrete domain values to the argument list of the lambda expression and then "evaluating the term." Although this process may be given a formal definition, we shall not do so here, but rather rely on the reader's intuition of that process.

We now turn to specifying the actual instance-level semantics for **BLU**. In the following, it is assumed that there is a reference database schema **D** upon which the constructions are based.

2.2.2 Definition The **BLU** implementation $\mathbf{BLU} - \mathbf{I}$ is defined as follows.
(a) Sorts:
(i) $\mathbf{BLU} - \mathbf{I}[\mathbf{S}] = \mathbf{IDB}[\mathbf{D}]$.
(ii) $\mathbf{BLU} - \mathbf{I}[\mathbf{M}] = \textbf{s-mask}[\mathbf{D}]$.

(b) Operators:
(i) **`combine`** : $(X, Y) \mapsto X \cup Y$
(ii) **`assert`** : $(X, Y) \mapsto X \cap Y$
(iii) **`complement`** : $X \mapsto \mathbf{ILDB}[\mathbf{D}] \backslash X$
(iv) **`mask`** : $(R, X) \mapsto \{ y \mid (\exists x \in X) R(x, y) \}$

(v) **genmask** : X $\mapsto$ **s – –mask**[**Dep**[X]]

Observe first of all that the the three operations **combine, assert,** and **complement** are precisely those which make **IDB**[**D**] into a Boolean algebra under the usual set-theoretic operations. **mask** performs, at the level of instances, precisely the masking operation described in 1.5. **genmask** generates the mask corresponding to the set of all proposition letters upon which the set of possible worlds depends.

The remarkable fact is the simplicity of this collection of operations. We are only allowed the usual set theoretic manipulations, plus the operations of generating and applying masks, and yet we claim that this is a complete set of primitives for the implementation of update programs for incomplete information databases.

2.3 Fundamental Clausal Semantics for BLU

The instance-level semantics for **BLU** described in the previous section provides us with the fundamental definition of how **BLU** programs should behave. However, direct implementation of **BLU** as a manipulator of sets of possible worlds would be inefficient, if not impossible, for any reasonably size language. Therefore, we need to identify a means of representing and manipulating such states at a higher level, and emulate the implementation **BLU – –I** at that level. In this subsection, we present an implementation **BLU – –C** of **BLU** at the level of clauses which is an emulation of **BLU – –I**. A key feature of the definition of **BLU – –C** is that its operations are not specified merely as abstract operations, but rather as resolution-based algorithms operating on sets of clauses. Thus, it is a relatively straightforward task to actually implement **BLU – –C**.

We begin by sketching what it means for one implementation of a **BLU** to be an emulation of another. Basically, we want to represent each state of **BLU – –I** with one or more states of **BLU – –C** in such a way that performing operations in **BLU – –C** and then examining the corresponding state in **BLU – –I** is exactly the same as mapping the arguments of the computation down to **BLU – –I** and performing the computation there.

2.3.1 Definition Let **A** and **B** be implementations of **BLU**. Formally, an *emulation* e of **B** by **A** is a surjective morphism of the defining algebras. This means that it is given by a pair of surjective functions e[**S**] : **A**[**S**] $\to$ **B**[**S**] and e[**M**] : **A**[**M**] $\to$ **B**[**M**] which respect the operations of **BLU**. For example, for **mask** we require that e[**S**]((**A**[**mask**] **s m**)) = (**B**[**mask**] e[**S**](**s**) e[**M**](**m**))

2.3.2 Definition (a) The **BLU** implementation **BLU – –C** is defined as follows.
(a) Sorts:
(i) **BLU – –C**[**S**] = $2^{\mathbf{CF}[\mathbf{D}]}$
(ii) **BLU – –C**[**M**] = $2^{\mathbf{Prop}[\mathbf{D}]}$

(b) Operators:
(i) **combine, assert,** and **complement** are defined by Algorithm 2.3.3.
(ii) **mask** is defined by Algorithm 2.3.5.
(v) **genmask** is defined by Algorithm 2.3.8.
(b) The *canonical emulation* $e_{\mathbf{CI}}$ of **BLU – –I** by **BLU – –C** is defined as follows.
(i) $e_{\mathbf{CI}}$[**S**] : $\Phi \mapsto$ **Mod**[Φ]
(ii) $e_{\mathbf{CI}}$[**M**] : P $\mapsto$ **s – –mask**[P].

The algorithms for the computation of the three Boolean-algebraic functions **combine**, **assert** and **complement** are quite straightforward. We next present them, in an Ada-like syntax, together with a statement of their complexity.

2.3.3 Algorithms

```
function BLU–⊣[assert] (Φ1, Φ2: CF[D])
         returns CF[D] is
  begin
    return Φ1 ∪ Φ2;
  end;

function BLU–⊣[combine] (Φ1, Φ2: CF[D])
         returns CF[D] is
  begin
    return {φ1∨φ2 | φ1∈Φ1 and φ2∈Φ2};
  end;

function BLU–⊣[complement] (Φ1: CF[D])
         returns CF[D] is
  begin
    Ψ ← {□};
    C(Φ1,Ψ);
    return Ψ;
  end;

procedure C ( Γ: in CF[D], Δ: CF[D] in out )
          is
-- support procedure for C[complement].
  begin
    if Γ = ∅
      then return;
      else
        γ ← any element of Γ;
        Γ ← Γ\{γ};
        for each δ∈Δ loop
          Δ ← Δ\{δ};
          for each λ∈Lit[{γ}] loop
            Δ ← Δ ∪ {δ∨¬λ};
          end loop;
        end loop;
        C(Γ,Δ);
    end if;
  end;
```

2.3.4 Theorem (a) The algorithms defined in 2.3.3 are correct, in the sense that they respect the emulation defined in 2.3.2(b).
(b) Their worst case time and space complexities are as follows.
(i) **BLU**$-\dashv$**[assert]**: $\Theta(\mathbf{Length}[\Phi_1] + \mathbf{Length}[\Phi_2])$.
(ii) **BLU**$-\dashv$**[combine]**: $\Theta(\mathbf{Length}[\Phi_1] \times \mathbf{Length}[\Phi_2])$.

(iii) **BLU – –I**[**complement**]: $\Theta\left(\varepsilon^{\mathbf{Length}(\Phi_1)}\right)$, where $\varepsilon = e^{1/e}$.

(c) The bounds specified in (b) are in fact problem complexity bounds; no algorithms of lower worst case asymptotic complexity are possible. □

Rather than discuss the implications of algorithm and problem complexity on a case-by-case basis as they are presented, we defer the discussion until Section 4, where they will be considered within a more global context.

We now turn to the implementation of **`mask`** in **BLU – –C**. It is quite a bit less straightforward than the implementation of the three Boolean algebra operations, and involves the use of two auxiliary algorithms. **`rclosure`** simply closes up the set of clauses Φ under resolution with the proposition letters in P. **`drop`** eliminates all clauses which involve the proposition letters in its argument. Thus, we can compute a mask by repeating each of these steps on each letter to be masked. In effect, the **`rclosure`** step ensures that, when we discard those clauses involving the masked letters, there are enough others around to completely describe the constraints on those which are left. It would trivially be sufficient to close up Φ_1 under total resolution; what is somewhat remarkable is that it suffices to close it up under just those proposition letters which are to be masked.

2.3.5 Algorithms

```
function rclosure (Φ: CF[D], P: s-mask[D])
         return CF[D] is
  begin
    Γ ← Φ1;
    for each A ∈ P loop
      Γ+ ← all γ ∈ Γ with A ∈ Lit({γ})
      Γ- ← all γ ∈ Γ with ¬A ∈ Lit({γ})
      for each γ+ ∈ Γ+ loop;
        for each γ- ∈ Γ- loop;
          Γ ← Γ ∪ resolvent(γ+, γ-, A);
        end loop;
      end loop;
    end loop;
    return Γ;
  end;
```

```
function drop (Φ: CF[D], P: s-mask[D])
        return CF[D] is
  begin
    Ψ ← ∅;
    for each φ ∈ Φ loop
      if Lit[Prop[{φ}]] ∩ P = ∅
        then
          Ψ ← Ψ ∪ φ;
        end if;
    end loop;
    return Ψ;
  end;

function BLU−−C[mask] (Φ: CF[D], P: s-mask[D])
        return CF[D] is
  begin
    Ψ ← ∅;
    for each A∈P loop
      Ψ ← (Drop {A} (Rclosure Φ1 {A}));
    end loop;
    return Ψ;
  end;
```

2.3.6 Theorem (a) The algorithm **BLU−−I**[mask] defined in 2.3.5 is correct, in the sense that it respects the emulation of 2.3.2(b).

(b) The worst-case time and space complexity is bounded by $\mathbf{O}(\mathbf{Length}[\Phi]^{2^{\mathbf{Card}[P]}})$, where **Card** denotes cardinality. Furthermore, as long as $\mathbf{Card}(P) \ll \mathbf{Card}[\mathbf{Prop}[\mathbf{D}]]$ and $\mathbf{Length}[\Phi] \ll \mathbf{Maxclause}[\Phi]$, where **Maxclause** denotes the maximum length of a set of consistent clauses over P and "≪" means "sufficiently smaller than", this lower bound may be realized. (The precise characterization of these conditions is too complex to develop here.) □

Thus, the computation of a mask for a set of clauses is inherently a very hard problem, in the worst case. This is not surprising. Essentially, computing a mask is computing a projection on a schema. In fact, there is a relational schema with only one relational symbol of only five arguments, and constrained by only three functional dependencies, with a projection onto four of its columns which is not finitely axiomatizable in first-order logic [10]. If we ground such a schema and use finite domain closure, we get a very large number of dependencies in the view, relative to the base schema. In short, a fast algorithm for computing mask implies a fast algorithm for solving the implied constraint problem for views [14], and that is simply not possible.

The final operation which we need to implement at the clause level is **`genmask`**. In testing the dependency of Φ upon A, the basic idea is to take two copies of Φ, assign A to be true in one and false in the other, and then look for truth assignments on the other letters which yield a difference. We need a few auxiliary definitions.

Definition 2.3.7 Let P be a set of propositions, and Φ a set of clauses.

(a) $\mathbf{CLS}[\Phi]$ denotes the set of all consistent subsets of $\mathbf{Lit}[\Phi]$ which contain either A or else $\neg$A for each $A \in \mathbf{Prop}[\Phi]$.

(b) For $A \in \mathbf{Prop}[\Phi]$, $\mathbf{Ldiff}[A,\Phi]$ denotes the set of all pairs $(L_1,L_2) \in \mathbf{CLS}[\Phi] \times \mathbf{CLS}[\Phi]$ which differ only in that $A \in L_1$ and $\neg A \in L_2$.

2.3.8 Algorithm

```
function unitres (Φ: CF[D], L: CLS[Φ])
         returns Boolean is
-- Unit resolution computation.
  begin
    Ψ ← Φ;
    for each l ∈ L loop
      for each φ ∈ Φ loop
        if φ = ψ∨¬l
          then Ψ ← Ψ\{φ}∪{ψ};
        end if;
      end loop;
    end loop;
  return Ψ;
  end;

function BLU−⊣[genmask] (Φ: CF[D])
         returns P: s-mask[D] is
  begin
    X ← ∅;
    for each A ∈ Prop[Φ] loop
      for each (L1,L2) ∈ Ldiff[A,Φ] loop
        if unitres(Φ,L1) ≠ unitres(Φ,L2)
          then
            X ← X∪{A};
            exit loop;
        end if;
      end loop;
    end loop;
  return X;
  end;
```

2.3.9 Theorem (a) The algorithm **BLU**−⊣[genmask] defined in 2.3.7 is correct, in the sense that it respects the emulation of 2.3.2(b).
(b) The worst-case time complexity is $\Theta(2^{\mathbf{Card}[\mathrm{Prop}[\Phi]]} \cdot \mathbf{Length}[\Phi] \cdot \mathbf{Card}[\mathrm{Prop}[\Phi]]^2)$. where **Card** denotes cardinality.
(c) The problem of deciding whether a set of clauses depends upon a particular proposition letter is **NP**-complete. □.

3. Specification of the Programming Language HLU

In this section, we demonstrate the utility of **BLU** by defining the semantics of the language **HLU**, which was informally described in the introduction, entirely in terms of **BLU**. The development of **HLU** is divided into two parts. In 3.1, we present a formal semantic description of **simple-HLU**, which is a subset of full **HLU** which contains all of the constructs except the two involving the **`where`** construct. These are implemented directly in **BLU**. In 3.2, we provide the definitions of the two **`where`** constructs, using a form of macro expansion.

3.1 Direct Semantics for Simple-HLU

3.1.1 Definition The algebraic signature **simple-HLU** is defined as follows.
(a) The sorts are the same as for **BLU**, namely $\{\mathbf{S},\mathbf{M}\}$.
(b) There are five operation symbols. Together with their arities, they are given below.

`assert` : $\mathbf{S} \times \mathbf{S} \rightarrow \mathbf{S}$
`clear` : $\mathbf{S} \times \mathbf{M} \rightarrow \mathbf{S}$
`insert` : $\mathbf{S} \times \mathbf{S} \rightarrow \mathbf{S}$
`delete` : $\mathbf{S} \times \mathbf{S} \rightarrow \mathbf{S}$
`modify` : $\mathbf{S} \times \mathbf{S} \rightarrow \mathbf{S}$

These five operation symbols correspond to the first five operations listed for **HLU** in the introduction. The arities seem different because in the "user's syntax" of **HLU**, the system state is hidden. In each of the five cases above, the first argument corresponds to the system state. Thus, the "user level" **HLU** program **`(insert X),`** using the syntax described in the introduction, is more properly represented as **`(insert s0 X).`**

We now turn to expressing the semantics in terms of **BLU**. The process is very simple. We use the **`define`** convention outlined in 2.1.3 to express each **simple-HLU** program as a **BLU** program.

3.1.2 Definition The **BLU**-based semantics for **simple-HLU** is given as follows.

```
(define HLU-assert
   (lambda (s0 s1) (assert so s1)))
(define HLU-clear
   (lambda (s0 s1) (mask s0 s1)))
(define HLU-insert
   (lambda (s0 s1)
    (assert (mask s0 (genmask s1)) s1)))
(define HLU-delete
   (lambda (s0 s1)
         (assert (mask s0 (genmask s1))
                 (complement s1))))
(define HLU-modify
   (lambda (s0 s1 s2)
    (combine
     (assert (assert
              (mask
               (assert (mask (assert s0 s1)
                             (genmask s1))
                       (complement s1)))
               (genmask s2))
             s2)
     (assert s0 (complement s1)))))
```

Note how **insert, delete,** and **modify** all conform to our "mask and assert paradigm." In **insert** and **delete,** the mask corresponding to the insertion state is generated, applied to the system state, and the insertion state is then asserted upon the system state. To assert formal compliance with the definitions of 1.4.5, we need to define the clause-level implementation.

3.1.3 Definition We define **simple-HLU – –C** and **simple-HLU – –S** as the **BLU – –I** and **BLU – –C** based implementations of **simple-HLU**, respectively.

As long as we understand the ideas of lambda expression evaluation, there is nothing further to explain regarding this definition. All of the work was done in the definitions of the implementations of **BLU**. The formal statement of correctness is as follows.

3.1.4 Theorem The definitions of **HLU-insert**, **HLU-delete**, and **HLU-modify**, implemented in **simple-HLU**, are logically equivalent to those defined in 1.4.5.□

3.1.5 Example Consider the **simple-HLU – –C** update program **(insert**($\{A_1 \vee A_2\}$) with system state $\Phi = \{\neg A_1 \vee A_3, A_1 \vee A_4, A_4 \vee A_5, \neg A_1 \vee \neg A_2 \vee \neg A_5\}$. This translates to the following **BLU** program.

(assert (mask Φ **'**$\{A_1, A_2\}$**)) '**$\{A_1 \vee A_2\}$ **)).**

First we compute **(genmask '**$\{A_1 \vee A_2\}$**)** to be $\{A_1, A_2\}$. Next, **(mask** Φ **'**$\{A_1, A_2\}$**)** = $\{A_4 \vee A_5, A_3 \vee A_4\}$. Finally, **(assert '**$\{A_4 \vee A_5, A_3 \vee A_4\}$ **'**$\{A_1 \vee A_2\}$**)** = $\{A_1 \vee A_2, A_4 \vee A_5, A_3 \vee A_4\}$.

3.2 Direct Semantics for full HLU

3.2.1 Definition The algebraic signature **HLU** is defined as follows.
(a) There are three sorts. In addition to the two sorts **S** and **M**, there is an additional sort **P**, which represents the abstract data type of **BLU** programs.
(b) The operator names consist of those of **simple-HLU**, together with the two listed below.

$\texttt{where1} : \mathbf{S} \times \mathbf{P} \rightarrow \mathbf{S}$
$\texttt{where2} : \mathbf{S} \times \mathbf{P} \times \mathbf{P} \rightarrow \mathbf{S}$

`where1` and **`where2`** represent the "where" construction of **HLU** with one and two program arguments, respectively. To handle this construction, we define them as *macros* which force the expansion of these program arguments. We borrow both the name **`syntax`** and the semantics from the TI Implementation of Scheme [21]. We also assume that the reader is familiar with Lisp/Scheme quasi-quote syntax as well as the definition of the primitives **`cdr`** and **`cons`**; refer to [23] for an explanation. The actual macro semantics is very easy; the key point is that a call to this macro is to return a **BLU** program *as its value.* The first name in the list following the **`syntax`** is the name of the expanded macro; the rest of the elements in the list are the formal arguments. The following list is the body, which is expanded at the call. The only technical "problem" is argument naming; the returned function must have a formal argument list free of name collisions; in a call of the form **`(where s p1 p2)`** we must ensure that the formal parameter lists of p1 and p2 do not have collisions with one another or with s. To address this, we define a few simple support functions.

3.2.2 Definitions (a) Let $\Lambda = (\lambda_1 \cdots \lambda_m)$ be a list of atoms names, and let σ be a string. Define $\texttt{(atomappend}\ \sigma\ \Lambda\texttt{)} = (\lambda_1{\cdot}\sigma \cdots \lambda_m{\cdot}\sigma)$.
(b) Let **`foo`** be any **BLU** program. **`(arglist 'foo)`** returns the list of formal arguments for **`foo.`**

3.2.3 Definition The **BLU**-based semantics for **HLU** is defined as follows.
(a) The semantics of operations in **simple-HLU** is exactly as given in 3.1.2.
(b) The semantics of **`where1`** is defined as follows.

```
(syntax (where1 s0 s1 p0)
  `(lambda ,(append
             ´(s0 s1)
             (atomappend ".0"
                (cdr (arglist p0))))
        (combine
          (,p0 ,(cons ´(assert s0 s1)
                   ,(atomappend ".0"
                      (cdr (arglist p0)))))
         (assert s0 (complement s1)))))
```

(c) The semantics of **`where2`** is defined as follows.

```
(syntax (where2 s0 s1 p0 p1)
  `(lambda
      ,(append
        ´(s0 s1)
          (atomappend ".0"
               (cdr (arglist p0)))
          (atomappend ".1"
               (cdr (arglist p1))))
       (combine
          (,p0 ,(cons
                 ´(assert s0 s1)
                 ,(atomappend ".0"
                     (cdr (arglist p0)))))
         (,p1 ,(cons ´(assert s0 s1)
                ,(atomappend ".1"
                   (cdr (arglist p0))))))))
```

In the expansion of **(where2 s0 p1 p2)**, the first argument of **p1** and of **p2** remains as **s0** (recall the convention defined in 2.1.2). However, the rest of the arguments of **p1** have the string **".1"** appended to them, and the rest of the arguments of **p2** have **".2"** appended to them. This ensures that there are no formal argument naming collisions. The case of **where1** is similar. Let us examine an example at the clause level.

3.2.5 Example Let the system state Φ be as in 3.1.4, and consider the program

(where $'\{A_5\}$ **(insert** $'\{A_1 \vee A_2\}$**)).**

First, let us expand the more general program

(where s1 (insert s2)).

Using 3.2.3 and 3.1.2, we get

```
(lambda (s0 s1 s1.0)
  (combine
    ((lambda (assert (mask s0 (genmask s1))
                      s1))
                     ((assert s0 s1) s1.0))
    (assert s0 (complement s1))))
```

We may use lambda variable substitution to reduce this to the following program.

```
(lambda (s0 s1 s1.0)
   (combine
      (assert (mask (assert s0 s1)
                    (genmask s1.0)) s1.0 ))
      (assert s0 (complement s1))))
```

Now we can perform the actual parameter substitution and evaluation, with **s0** $\leftarrow \Phi$; **s1** $\leftarrow \{A\}$; **s1.1** $\leftarrow \{A_1 \vee A_2\}$. We already know that **(genmask** $'\{A_1 \vee A_2\}$**)** $= \{A_1, A_2\}$, and that **(assert** $\Phi\ '\{A_5\}$**)** $= \Phi \cup \{A_5\}$. Now **(mask** $\Phi \cup \{A_5\}\ \{A_1, A_2\}$**)** is computed as in 3.1.5 to yield $\{A_4 \vee A_5, A_3 \vee A_4, A_5, A_1 \vee A_2\}$. Thus, **(assert** $'\{A_4 \vee A_5, A_3 \vee A_4, A_5\}\ '\{A_1 \vee A_2\}$**)** $= \{A_4 \vee A_5, A_3 \vee A_4, A_5, A_1 \vee A_2\}$. Next, **(complement** $'\{A_5\}$**)** $= \{\neg A_5\}$, and **(assert** $\Phi, '\{\neg A_5\}$**)** $= \Phi \cup \{\neg A_5\}$. Thus, the final result is

(combine $'\{A_4 \vee A_5, A_3 \vee A_4, A_5, A_1 \vee A_2\}$ $'\{\neg A_1 \vee A_3, A_1 \vee A_4, A_4 \vee A_5, \neg A_1 \vee \neg A_2 \vee \neg A_5\}$**).**
We leave it to the reader to expand this into the 16 clauses yielded by Algorithm 2.3.3.

3.3 Comparison to Other Work

3.3.1 The work of Wilkins In [22], Wilkins presents semantics and algorithms for updates of the form **(assert** ϕ**), (where** ϕ **(insert** ω**)), (where** $\phi \wedge$**t (delete t)),** and **(where** $\phi \wedge$**t (modify t w))** are presented, with ϕ and w arbitrary wff's, and **t** a ground fact. (We have altered her syntax slightly to conform with ours.) With the exception noted in 1.4.7, the semantics of her update algorithms are identical to ours. However, the actual algorithms are very different. Specifically, her algorithms introduce new auxiliary proposition letters at each update. In effect, she defers the computation of the mask component via the retention of historical information. Her *update* algorithms are unquestionably faster than ours. In fact, they are linear in the sizes of the database and update formulas. However, the price is repaid when the database is queried. Each update adds at least one new proposition letter. Thus, after a large number of updates, query processing becomes very expensive, since the query solver must constantly eliminate auxiliary symbols from formulas. It would seem that, after a large number of updates, a system based upon her algorithms would have its query evaluation mechanism greatly slowed by the presence of the large number of auxiliary symbols employed. To "clean up" the knowledge base, masking of these auxiliary symbols would be necessary. However masking is inherently a hard problem (see 2.3.6), and so her algorithms would not seem to offer a superior alternative to ours.

3.3.2 The flock approach In [7], an alternative for updating logical databases is proposed. This strategy may be broadly characterized as *minimal change.* For example, in inserting α into the database, rather generating a database independent mask for α, we look for minimal ways to alter the database so that the insertion of α will be consistent. However, this definition of minimality is a purely syntactic one, and so the spirit of the approach differs fundamentally from ours. While it is possible to obtain a semantic version of minimal change, at the expense of a greatly complicated masking function, space limitations preclude presentation in this paper.

3.3.3 A tabular approach Both of the works mentioned above are basically propositional in nature. In [1], Abiteboul and Grahne present a structure-oriented approach to update specification, and implementation using the notion of tables of Imieliński and Lipski [12]. As such, their approach directly uses relations. It is interesting to note that two of their basic update operators are precisely union and intersection, which, at the instance level, are precisely our **combine** and **assert.** Set-theoretic difference is also one of their primitives; it can easily be realized as a combination of intersection and absolute complement. Thus, of their six primitives, three are essentially identical to three of our five **BLU** primitives. Their other three primitives are relation-by-relation versions of these same primitives. At the propositional level, these correspond to possible-world by possible-world logical operations of $\wedge$, $\vee$, and $\not\Rightarrow$. These primitives are also sufficient in power to realize **HLU**, although it appears that they are strictly less powerful than those of **BLU**, in that **genmask** cannot be realized.

A detailed comparison of the two approaches is warranted.

Another paper promoting a relational approach is [15]. It deals more with pragmatics of individual examples, and so differs in emphasis from our work.

4. Towards a Practical Implementation

We briefly examine the practicability of the definitions presented herein, together with some of our future directions.

Notice that there are two ways in which a possible world definition (*qua* clause) may be an argument to an **HLU** program. First, the system state itself is such an argument. Second, any user-supplied update parameter is such an argument also. In general, we would expect the system state to have a much larger and more complex representation than a typical user supplied parameter. Now we may observe from the definition of **HLU** that the **BLU** primitives **`complement`** and **`genmask`** only take user supplied parameters as arguments. Thus, even though these problems have inherently high degrees of complexity for the clausal representation, they will likely be applied only to arguments which are small and simple enough to be manageable. The clausal implementations of **`assert`** and **`combine`** are quite respectable in terms of performance, even though they do take the system state as arguments in several **HLU** definitions.

It seems likely that the bottleneck in any clausal implementation of **HLU** base upon **BLU** is going to be the implementation of **`mask.`** The masking problem is inherently difficult; yet it is essential to an implementation. The question is whether there is another, much efficient implementation of **HLU** which avoids masking entirely. The answer is no. Masking is itself a form of insertion; just as **`(insert`** ´$\{A_1 \vee A_2\}$**`)`** says that three of the four truth values of (A_1, B_1), so too does **`(mask`** $\{A_1, A_2\}$**`)`** say that all four are possible. At any rate, the inherent complexity of inserting $\{A_1 \vee A_2\}$ is no less than masking $\{A_1, A_2\}$.

Thus, it is clear that the worst case in any clausal implementation of **HLU** is going to be intolerably bad. The question remains, however, as to whether there is some other "reasonable" implementation which admits more efficient execution. If so, the representation must be far removed from the clausal one, else we could efficiently reduce the clausal approach to it. Care must be taken in expressing the problem; for example, we might demand that all sets of clauses be fully expanded to include all consequences. Masking then becomes trivial. Of course, other operations then become intolerably slow.

The most promising approach to take, from a practical point of view, would seem to be to look for an incomplete implementation which nonetheless covers many interesting cases. Currently, we are pursuing two avenues in this spirit. The first is to implement a small version of **HLU** in Lisp. The implementation is based substantially upon the **BLU** definition, although a number of correctness-preserving optimizations are employed. The implementation is initially for a propositional logic, although we plan to extend it to the first order situation sketched in the next section in the near future. The purpose of this implementation is to study empirically the bottlenecks in such a system.

The second avenue is to examine in more detail other realizations. Foremost, we are looking at the template model [12], and particularly the work on updates for it

[1]. Although this model is not able to represent all possible worlds, it can represent many important cases arising in practice. A comparison of these two approaches will hopefully shed light on some of the more practical aspects of the problem.

5. Extension to a First-Order Relational Framework

5.1 Problem Statement

Despite the fact that database schemata are not propositionally based, there is ample justification for an initial examination of the update problem on propositional schemata. First, the propositional framework provides a "stripped down" testbed; if the problems cannot be adequately formulated and solved at the propositional level, there is little hope of a more general solution at the relational level. Second, it may be argued that relational databases are finite, and so may be represented logically as a set of ground clauses. Nonetheless, we argue that it is not sufficient, from a practical point of view, to invoke this grounding assumption and limit the investigation to the propositional case.

5.1.1 Motivating example Consider a simple relational schema with a single ternary relational symbol R[NDT], and attributes N = name; D = department; T = telephone. Further consider an update request expressed informally as "Jones has a new telephone number." Implicit in this request is that Jones' new telephone number is *not* known. Assume that the appropriate domain closure axioms are present [20], so we know, in particular, that there is a finite set of constant symbols $\mathbf{T} = \{t_1, t_2, .., t_n\}$ which represent all possible telephone numbers. Then it is possible to express this update request directly in **HLU**. Let JD denote Jones' department, and ϕ be the clause which is the enormous disjunction $\bigvee\{(\text{Jones}, \text{JD}, t) \mid t \in \mathbf{T}\}$. Then the appropriate update request in **HLU** is **`(insert`** $\{\phi\}$**`).`** However, there are at least two problems. First of all, we *must* know Jones' department in order to specify the update, even though it is totally irrelevant, and remains unchanged. Second, in a realistic application, the collection of telephone numbers would indeed be enormous, making the direct expression of this update request all but impossible.

5.2 Solution Sketch

The overall goal of this component of our work is to extend the framework and algorithms already developed to a first-order framework in which updates such as the example given above are easily handled. The key idea is to maintain the same set of possible worlds as the purely propositional case, but to employ representation techniques which admit much more efficient manipulation.

We follow the idea of grounding as described at the beginning of 1.2, with some important extensions. There are two kinds of constant symbols, internal and external. The *external* constant symbols correspond exactly to those of the purely propositional framework. They have unique naming relative to other external constant symbols, and are visible to the user in the query and update languages. The *internal* constant symbols, on the other hand, do not have unique naming enforced, and are not directly visible to the user. They are countably infinite in number, although only a finite number are active at any given time. There is a *modified close world assumption* stating that the external and active internal constant symbols are the only ones known, and, furthermore, that each internal constant symbol is equal to some external constant symbol. Essentially, the modified internal symbols correspond to null values, as described by Reiter in [20].

Also present in the extension is a *Boolean algebra of types.* These correspond to the Boolean categories of McSkimin and Minker[18]. Reiter has proposed a similar framework.[20]. The *constant dictionary* is used to classify each external and active internal constant symbol, and has one entry for each. An entry for an external symbol contains just one component; that which identifies the smallest type to which it belongs. An entry for an internal symbol u contains what McSkimin and Minker call a *Boolean category expression.* It identifies the *underlying type* **ty**(u), together with a list of *inclusion exceptions* **ie**(u) and a list of *exclusion exceptions* **ee**(u). The semantics is that the actual value of u, which is some external symbol, is either of type **ty**(u) or a member of the set **ie**(u), but is not a member of **ee**(u). The lists of inclusion and exclusion exceptions may contain internal as well as external symbols. As a simple example, to represent the fact that Jones has an unknown telephone number, we active an internal symbol u, and designate it to have type $\tau_{\mathbf{telno}}$, the type of all telephone numbers. The fact about Jones would be represented as the single literal R(Jones, JD, u). "Jones" would be an external constant; JD might be internal or external.

To render this representation useful, resolution must be extended to make use of it. This is done by employing a special case of *semantic resolution* developed by McSkimin and Minker[18]. Basically, when resolving R(a, ...) and R(b, ...) on the first argument (for example), we turn to the constant dictionary to determine the *intersection* of the constant values represented. This intersection is effectively the unification.

It is quite possible to use the full Π-σ clause framework of McSkimin and Minker[18] to represent universal quantification as well, although it will add substantially to the complexity of the computations.

To make the extension complete, it is also necessary to augment the query language, so that queries such as that illustrated at the beginning of this section may be formed. The key idea here is to allow variables in the "where" part of **HLU** programs. These variables define an instance-by-instance environment for the action of the where. As a concrete example, here is our example query expressed in this extended language.

```
(where ((Jones = x) (y ∈ τu))
       (insert ((∃ w ∈ τtelno) (R x y w))))
```

Here x is bound to "Jones", while y is bound to the universal type τ_u from the calling environment, on a case-by-case basis. This means that, for every binding of (x, y) satisfying these constraints, perform the insertion specified. (Of course, assuming that Jones has a unique department, there will only be one such binding.) The"existence of w" statement in the insertion is converted to an internal constant, constrained to type τ_{telno}.

There are, of course, many subtleties to this process which space limitations prohibit us from expressing. However, the key point should be evident. It is possible to extend the purely propositional framework described in this paper to a useful subset of relational logic. Since resolution has a direct extension, so too do our algorithms.

References

1. S. Abiteboul and G. Grahne, “Update semantics for incomplete databases,” pp. 1-12 in *Proc. 1985 VLDB Conference*, ().
2. C.-L. Chung and R. C.-T. Lee, *Symbolic Logic and Mechanical Theorem Proving,* Academic Press (1973).
3. K. L. Clark, “Negation as Failure,” pp. 293-322 in *Logic and Data Bases*, ed. H. Gallaire and J. Minker, Plenum Press (1978).
4. E. F. Codd, “Relational completeness of database sublanguages,” pp. 65-98 in *Data Base Systems*, ed. R. Rustin, Prentice-Hall (1972).
5. H. Ehrig and B. Mahr, *Fundamentals of Algebraic Specification 1,* Springer-Verlag (1985).
6. H. B. Enderton, *A Mathematical Introduction to Logic,* Academic Press (1972).
7. R. Fagin, G. M. Kuper, J. D. Ullman, and M. Y. Vardi, “Updating Logical Databases,” pp. 1-18 in *Advances in Computing Research*, ed. P. Kanellakis, JAI Press (1986).
8. H. Gallaire, J. Minker, and J. M. Nicolas, “Logic and databases: a deductive approach,” *ACM Computing Surveys* **16**(2) pp. 153-185 (1984).
9. S. J. Hegner, *Relational Database Decomposition: Logical and Algebraic Foundations,* monograph in preparation, to appear (1987).
10. R. Hull, “Finitely specifi able Implicational Dependency Families,” *JACM* **31**(2) pp. 210-226 (1984).
11. T. Imieliński, “On algebraic query processing in logical databases,” pp. 285-318 in *Advances in Data Base Theory*, ed. H. Gallaire, J. Minker, and J. M. Nicolas, Plenum Press (1984).
12. T. Imieliński and W. Lipski, Jr., “Incomplete information in relational databases,” *JACM* **31** pp. 761-791 (1984).
13. B. E. Jacobs, “On database logic,” *JACM* **29**(2) pp. 310-332 (1982).
14. B. E. Jacobs, A. R. Aronson, and A. C. Klug, “On interpretations of relational languages and solutions to the implied constraint problem,” *ACM TODS* **7**(2) pp. 291-315 (1982).
15. A. Keller and M. W. Wilkins, “Approaches for updating databases incomplete information with nulls,” pp. 332-340 in *Proc. International Conference on Database Engineering*, (24-27 April 1984).
16. H. J. Levesque, “The Logic of Incomplete Knowledge Bases,” pp. 165-186 in *On Conceptual Modelling*, ed. M. L. Brodie, J. Mylopoulos, and J. W. Schmidt, Springer-Verlag (1984).
17. E. G. Manes and M. A. Arbib, *Algebraic Approaches to Program Semantics,* Springer-Verlag (1986).
18. J. R. McSkimin and J. Minker, “A Predicate calculus based semantic network for deductive searching,” pp. 205-238 in *Associative Networks*, ed. N. V. Findler, Academic Press (1979).
19. J. Rees and W. Clinger, “Revised^3 report on the algorithmic language Scheme,” *SIGPLAN Notices* **21**(12) pp. 37-79 (December 1986).

20. R. Reiter, “Towards a Logical Reconstruction of Relational Database Theory,” pp. 191-238 in *On Conceptual Modelling*, ed. M. L. Brodie, J. Mylopoulos, and J. W. Schmidt, Springer-Verlag (1984).

21. Texas Instruments, *TI Scheme Language Reference Manual,* Texas Instruments, Inc. (1985).

22. M. W. Wilkins, “A Model Theoretic Approach to Updating Logical Databases,” Report No. STAN-CS-86-1096,, Department of Computer Science, Stanford University (January 1986).

23. P. H. Winston and B. K. P. Horn, *Lisp, Second edition,* Addison-Wesley (1984).