Glue-Nail: A Deductive Database System

Geoffrey Phipps Marcia A. Derr*
Kenneth A. Ross
Department of Computer Science, Stanford University, Stanford CA 94305 †
{phipps, mad, kar}@cs.stanford.edu

13th March, 1991

Abstract

Glue is a procedural language for deductive databases. It is designed to complement the purely declarative NAIL! language, firstly by performing system functions impossible to write in NAIL!, and secondly by allowing the procedural specification of algorithms for critical code sections. The two languages together are sufficient to write a complete application. Glue was designed to be as close to NAIL! as possible, hence minimizing the impedance mismatch problem. In this paper we concentrate on Glue. Pseudo-higher order programming is used in both languages, in the style of HiLog [1]. In particular Glue-Nail can handle set valued attributes (non-1NF schemas) in a clean and efficient manner. NAIL! code is compiled into Glue code, simplifying the system design. An experimental implementation has been written, a more efficient version is under design.

1 Introduction

The Glue language grew out of our experiences of designing and implementing the first NAIL! system [3], and of using commercial database systems.

From a software engineering point of view, declarative logic based languages offer many advantages over traditional relational databases, primarily due to their simplicity and high-level approach (for example, see the introductions of [4] and [10]). Relational database systems free the programmer from worrying about the physical data representation and access methods. Deductive database systems do the same for views and recursion. One currently unsolved problem with declarative languages is how to integrate them into a full system without becoming tangled in the impedance mismatch problem (described below). This is a problem which has plagued traditional database systems. Glue-Nail is designed to solve it.

There is no precise definition of "declarative", although most people would agree that in a declarative language, the programmer states *what* is desired, not *how* to do it. Hence declarative languages significantly reduce the amount of code that a programmer must write for a given application. In addition, the code that the programmer must write is more strongly focused on the actual application, rather than on the technical details of a particular solution algorithm. This focusing effect should reduce the number of mismatches between the application specification and the programmer's implementation.

Unfortunately there are certain operations which implicitly have a notion of state, for example Input/Output (I/O) and EDB[1] updates. These operations have side effects which cannot be easily captured in logic.[2] Any real application language must be able to perform I/O and EDB update operations; they cannot be ignored. Computations involving side effects cannot be written declaratively because the programmer must be able to specify the order in which the side effects occur. In other words the programmer must give the intermediate steps in the computation. Procedural languages are well suited to describing such computations, declarative languages are not.

We feel that the increase in programming efficiency provided by a declarative language is very important, and so we have preserved the declarative nature of NAIL!. Hence we need another (procedural) language

*Also AT&T Bell Laboratories, Murray Hill, NJ 07974

†This work supported by AFOSR-88-0266, NSF-87-12791, and a gift of Mitsubishi Electric.

[1]The Extensional Data Base (EDB) stores tuples; in a relational system it would just be called the "database".

[2]One "solution" is to carry state variables around in the logical rules, but the programmer must ensure that the variables are strung together in the correct order, which is equivalent to specifying the order of computation.

to complement NAIL!. This language is Glue.

Embedding a query language in a procedural language is common in databases, for example, embedding SQL in C. Unfortunately we then usually run into the *impedance mismatch* problem (for an example description, see [4]). This is the name for the collection of problems that arise when we interface two dissimilar languages. It has no formal academic definition, but it is a serious problem in real programming systems. The problems include differing type systems, set oriented versus tuple oriented computation, differing data lifetime, wildly differing syntax, and an inability to carry optimizations across the interface. For example, SQL uses the relation as the basic data type. C uses single valued variables, so C is a tuple oriented language. When SQL is merged with C the concept of a cursor has to be introduced so that C can iterate (or recurse) over all the elements in the SQL relation. SQL has efficient set oriented algorithms for dealing with relations, but C does not. An even more serious efficiency problem is that a two language system has two separate optimizers. Each individual SQL query is optimized independently, without any reference to other nearby SQL queries.

To avoid these problems, Glue was designed to resemble NAIL! as closely as possible. Both languages have tuples and relations as their basic data objects, "all-solution" computation, similar syntaxes, and identical type systems. For example, in Glue a subgoal can be a NAIL! predicate, or an EDB relation or a Glue procedure. The syntax and behavior is the same in all three cases. In each case the subgoal returns a set of tuples.

We expect that an application programmer will write the declarative, query oriented sections of the application in NAIL!, using Glue mostly for the interface and for EDB modifying code. Sometimes it might be useful to use Glue for a particularly speed-critical query, for which an especially efficient special purpose algorithm is known. Such a practice is analogous to writing speed critical sections of a C program in assembler; there is an increase in speed, but at the expense of clarity.

The remainder of the paper is organized as follows. Section 2 discusses the basic data and predicate types. Section 3 is a tutorial introduction to the basic Glue assignment statement. Section 4 describes Glue procedures. In Section 5 we describe Glue-Nail's set system. We also discuss higher-order programming in general. Section 6 briefly describes Glue's module system. Section 8 briefly describes three other deductive database systems (LDL, CORAL, and Aditi), and compares their approaches with Glue-Nail. In Section 9 we describe the current experimental implementation. Section 10 discusses some known problems with Glue-Nail, and what we are planning to do with Glue-Nail.

2 Predicates

There are four kinds of predicates in Glue-Nail:

EDB Relations: Ground tuples (facts) are stored in relations in the Extensional Data Base (EDB). The EDB is equivalent to the "database" of a traditional relational system.

Local Relations: Glue procedures can have local relations; in a sense these are temporary EDB relations with restricted lexical scope.

NAIL! Rules: These define Intensional Database (IDB) tuples, the appropriate parts of which are computed on demand using the current value of the EDB.

Glue Procedures: Glue procedures also belong to the IDB, in that they define tuples which are only computed on demand. Unlike NAIL! predicates, Glue procedures can also use EDB updates and input (I/O) in their computations.

Predicates do not have duplicates.[3]

In Prolog a subgoal P in rule R can unify with either a fact or a predicate; there is no syntactic or behavioral difference discernible within rule R. Either P is true immediately or it can be derived, it makes no difference[4] to rule R. This usage equivalence of EDB and IDB is also true of languages like NAIL!; it is one of their great advantages over traditional embedded relational systems. Glue-Nail also has this advantage; a subgoal in Glue or NAIL! can reference an EDB relation, a NAIL! predicate, or a Glue procedure, and the syntax and semantics are identical in all three cases. The meaning is always: use the current value. For an EDB relation this value comes directly from the database. For a NAIL! predicate it is derived from the current state of the EDB. For a Glue procedure it is computed from the current EDB, and perhaps from input.

An attribute of a tuple can be either an atom (a number or a string), or a compound term. In Glue there is no difference between atoms and strings. In Prolog-style languages, the two data types are distinct, and a programmer is forever converting atoms into strings and vice versa. Strings are first class data types in Glue, and the language has built-in operators (concatenation, length, and substring) to aid in their use. Strings are central to databases, so they must be well supported.

Relations may contain only completely ground tuples. Hence Glue only needs to use matching when comparing subgoals against a relation, rather than use

[3]Or rather, the system must remove them if the programmer's code would behave differently in the presence of duplicates.

[4]Ignoring nonterminating derivations.

full unification. This restriction is also very important for the code optimizer, because it allows the system to know at compile time when a variable in an assignment statement becomes bound. Such knowledge is useful in many optimizations. If binding time analysis could not be performed at compile time, then it would have to be done at run time.

3 Assignment Statements

3.1 Basic Elements

The basic element of Glue is the assignment statement. Here is an example:

```
r(X,Y)+= s(X,W) & t(f(W,X),Y).
```

The effect of executing this statement is that the tuple `(X,Y)` is to added to relation `r` if there is a tuple `(X,W)` in relation `s`, and a tuple `(f(W,X),Y)` in relation `t`. All such `(X,Y)` tuples are to added to relation `r`.

Glue assignment statements are not logical rules, they are operational directives. They do not define tuples, they command their creation (or destruction or modification).

In their basic form Glue assignment statements have a single head term, and a conjunction[5] of subgoals in the body. The body is executed and produces a relation of tuples over the variables in the body. This relation is then used to modify the head relation. The subgoals and the head term may have compound terms as their arguments. Although Glue assignment statements may look a lot like Prolog rules, the control flow is completely different. The Prolog control strategy is "tuple at a time" with backtracking. Glue's strategy is "all solutions" with no backtracking. In Prolog, the binding for a variable is a single term. In Glue, the binding is a set of terms.

For the purposes of side effects and aggregations, the order of evaluation of Glue subgoals is fixed and is from left to right. Each subgoal is completely solved before the next subgoal is processed. We will refer to side effecting or aggregating subgoals as *fixed* subgoals. A fixed subgoal is either an EDB updating subgoal, a `group_by`, an aggregator (see Section 3.3), or a call to a Glue procedure which is known to be fixed. A Glue procedure is fixed if it contains a fixed subgoal. The predefined I/O procedures are all fixed. A Glue system is free to reorder the non-fixed subgoals, although procedures must still have their input arguments bound, and subgoals cannot be moved past an aggregator.

There are four assignment operators in Glue:

`:=` Clearing assignment. The head relation is overwritten by the result of the body.

`+=` Insertion assignment. The tuples from the body are added to the head relation.

`-=` Deletion assignment. The tuples from the body are removed from the head relation.

`+=[`$\vec{Z}$`]` Modify assignment, meant to be used as "update by key." Analogous to `UPDATE` in SQL. The key is the variables in the vector $\vec{Z}$.

For example, suppose the unary relation `row` contains the integers 1 to N, and that the ternary relation `matrix` contains (row, column, value) triples. Then the code:

```
matrix(X,X, 1.0):= row(X).
matrix(X,Y, 0.0)+= row(X) & row(Y) & X != Y.
```

would create an identity matrix of size N in relation `matrix`.

3.2 The Supplementary Relation Model

To explain the semantics of assignment statements it is useful to employ the supplementary relation model. The supplementary relations of an assignment statement hold the bindings for the variables. If there are n subgoals in an assignment statement, then there are $n+1$ supplementary relations, named sup_0 to sup_n. The ith supplementary relation sup_i has as its attributes all the variables occurring in the first i subgoals. Note that the zeroth supplementary relation, sup_0, is a relation of arity zero. It contains a single tuple, the null tuple ϵ. The body of an assignment statement of the form:

$b_1(\vec{B_1})$ & ...... & $b_n(\vec{B_n})$.

can be rewritten using supplementary relations as:

$sup_0() := true.$

$sup_1(\vec{S_1}) := b_1(\vec{B_1}).$

$sup_2(\vec{S_2}) := sup_1(\vec{S_1})\& \ b_2(\vec{B_2}).$

$\ldots$

$sup_n(\vec{S_n}) := sup_{n-1}(\vec{S}_{n-1})\& \ b_n(\vec{B_n}).$

The attributes of sup_i are the union of the variables in $\vec{S}_{i-1}$ with the variables in $\vec{B_i}$, i.e. $\vec{S_i} = vars(\vec{S}_{i-1}) \bigcup vars(\vec{B_i})$. Note that no variables are ever dropped from one supplementary relation to the next. However, variables that are not used further on in the assignment statement can be projected out, unless there are aggregators later in the assignment statement. For example, the supplementary relations of the code:

```
h(X,W):= a(X,A,B) & b(A,C) & c(B,C,W).
```

are:

[5] The body may contain control operators other than conjunction, but we will not discuss them in this paper.

```
sup_1(X,A,B):= a(X,A,B).
sup_2(X,A,B,C):= sup_1(X,A,B) & b(A,C).
sup_3(X,A,B,C,W):= sup_2(X,A,B,C) & c(B,C,W).
```

These supplementary relations need not actually exist in the implementation, but they are very useful when thinking about the meaning of an assignment statement. They emphasize the fact that the set (relation) of bound variables has tuples of bindings as its value.

Execution of an assignment statement is from left to right, each supplementary relation being (conceptually) completed before the next is begun. Each subgoal is completely solved before executing the next subgoal. Execution of an assignment statement stops whenever a supplementary relation is empty.

3.3 Aggregation

It often happens that we want to find the "aggregate" value of a set of tuples, for example the minimum value of a particular attribute. In the version of Glue described so far, the value of a tuple in a supplementary relation is independent of all the other tuples. This is not true for statements containing aggregate operators. Here the values of tuples typically do depend on each other.

The aggregate operators (aggregators) available in Glue are: `min`, `max`, `mean`, `sum`, `product`, `arbitrary`, `std_dev` (standard deviation), and `count`. These operators take a single bound term as an argument, and return a single value. The operator `arbitrary` returns a single arbitrary value from the binding set of the argument term, the other operators have their usual meanings. A simple example:

```
max_temp( MaxT ):=
        temperature( T ) & MaxT = max(T).
```

The `max` operator computes the maximum `T` over all the bindings it has for `T` at that point in the statement. For example, if the value of `temperature` were { (10), (35) }, then `max` would operate over sup_1 = { (10), (35) }, `MaxT` would be bound to 35, and $sup_2(T, MaxT)$ would be { (10,35), (35,35) }.

To explain the semantics of the aggregate operators it is easiest to refer to the supplementary relation model. If the jth subgoal is an aggregate operator, then it operates over the tuples in the $(j-1)$th supplementary relation. If the argument term is T, the aggregator looks at the T value for each tuple in the supplementary relation, rather than at each tuple in the relation formed by projecting the supplementary relation onto the variables of T (i.e. $\pi_{vars(T)} sup_{j-1}$). Choosing the second method would delete meaningful duplicates. For example, suppose we were computing the average temperature of a set of readings taken at various locations. If two temperature readings were identical, then that temperature reading would only appear only once if we projected the supplementary relation onto the temperature column. The temperature reading appears twice (as it should) if we look at each tuple in the supplementary relation.

Note that the variable resulting from the aggregation is in the jth supplementary relation. Here we are free to equate it to other bound variables (perform a "join"), so as to select particularly interesting tuples. For example, suppose we want to find the coldest city.[6] We want the names of the city, not the actual minimum temperature. The following code would provide the desired answer.

```
coldest_city( Name ):=
      daily_temp( Name, T ) &
      MinT = min(T) & T = MinT.
```

The third subgoal joins the `T` and `MinT` columns, hence the only tuples left in the supplementary relation after this subgoal are those with minimal temperatures. For example:

sup_1

Name	T
San Francisco	12
Madang	36
Copenhagen	-2

sup_2

Name	T	MinT
San Francisco	12	-2
Madang	36	-2
Copenhagen	-2	-2

sup_3

Name	T	MinT
Copenhagen	-2	-2

In actual fact the third subgoal is not really necessary. We could perform the restriction immediately by combining the second and third subgoals, as in:

```
coldest_cities( Name ):=
    daily_temp( Name, T ) & T = min(T).
```

3.3.1 Group_by

By default, aggregation operators use the entire supplementary set in their computations. There are often occasions when we want to partition the supplementary relation's tuples into a number of *groups*, and calculate aggregates over each group. For example, the supplementary relation might contain course-student-grade triples, and we might want to calculate the average grade in each course. In Glue we write this as:

[6] or cities, in the case of a tie.

```
course_average( C, Average ):=
    course_student_grade(C,S,G) &
    group_by(C) & Average = mean(G).
```

The effect of the second subgoal `group_by(C)` is to partition the supplementary set into groups, all the tuples in a group having the same `C` value. The groups are maximal, in that no two groups can have the same `C` value. All subsequent aggregate operators then operate over each of these groups independently.

Group_by statements cascade; that is, if a group_by subgoal has split the supplementary relations into n different groups, then the next group_by subgoal will operate on each of these n groups separately, perhaps splitting them into smaller groups.

4 Glue Procedures

We will explain the structure of Glue procedures using the following example.

```
procedure tc_e (X:Y)
  rels connected(X,Y);

  connected(X,Y):= in(X) & e(X,Y)
  repeat
    connected(X,Y)+= connected(X,Z) & e(Z,Y).
  until unchanged( connected(_,_) );
  return(X:Y):= connected(X,Y).
end
```

The name of this procedure is `tc_e`. The procedure's arity is (1:1), meaning that it produces binary tuples, given one bound input argument. Whenever `tc_e` is used as a subgoal, the first argument must be bound. Informally, this procedure calculates the nodes `Y` reachable from `X` via edge relation `e`. More correctly, given a set of unary tuples (sole attribute `X`), the procedure `tc_e` extends these tuples to be a set of binary tuples (X,Y) such that `Y` is reachable from `X` via edge relation `e`. All Glue procedures declare a subset of their formal arguments to be bound when the procedure is called. This binding restriction is the only restriction on the use of a Glue procedure as a subgoal, otherwise they are identical in their use to NAIL! predicates or EDB relations.

The procedure has one local relation, `connected`, of arity two. Procedures may be called recursively. Each invocation of a procedure has its own copies of its local relations. Declarations of local relations "hide" the declarations of other predicates with which they unify.

There is a repeat-until loop, the termination condition being `unchanged(connected(_,_))`. The built-in predicate `unchanged`(P) is true if predicate P has not changed since the last time that particular `unchanged` statement was executed.[7] The predicate `unchanged`(P) is always false the first time it is executed.

All procedures have two special relations, `in` and `return`. The relation `in` holds the input tuples to the procedure. The relation `in` has an arity equal to the bound arity of the procedure, i.e. the arity to the left of the colon in the procedure definition (in this case it has an arity of one). The relation `return` is used to hold the output tuples for the procedure. Assigning to this relation also has the effect of exiting the procedure. The `return` relation has the same arity as the procedure. An assignment statement that assigns to the `return` relation has an implicit `in` subgoal as its first subgoal. The arguments of the `in` subgoal are the same as the arguments to the left of the colon in the return head, for example:

```
return(X:Y):= in(X) & connected(X,Y).
```

The implicit `in` relation has a natural meaning, it restricts the return relation to be only those tuples which extend the input relation.

When a Glue procedure is used as a subgoal it is called once on all of the bindings for its input arguments, rather than being called many times, once for each binding for its input arguments.

5 Sets and Meta-programming

5.1 Sets

Set valued attributes are useful; they give a language more practical expressiveness, and can lead to more space efficient relations. Accordingly we wanted Glue and NAIL! to have sets. In both LDL and CORAL sets and relations are different things, whereas in logical terms they are both just sets of tuples. An LDL or CORAL rule with a set-generating operator needs to be read differently from a standard LDL or CORAL rule. Rules often produce a set of sets when what one really wants is the union of the sets. These sets of sets then have to be explicitly flattened. The only type of set equality available is set unification, which can be expensive.

Glue-Nail borrows the second order syntax scheme of HiLog [1]. In this scheme a set valued attribute contains the name of a predicate (i.e. the name of a set), rather than the value (members) of a set. Sets are therefore just normal predicates. In addition, compound terms can have arbitrary terms as their functors, rather than being limited to atoms as in normal logic based languages. Hence subgoals may have variables for their

[7]The semantics of unchanged are under review, and may be changed slightly.

predicate names. In particular we can store the name of a predicate in a tuple, then extract it using a variable and use that variable as a subgoal name. For example:

```
dept_employees( toy, E_set ) &
E_set( Emp_name ) & ...
```

The second attribute of `dept_employees` relation is a set valued attribute, it holds the name of the predicate which holds the employees in the toy department.

Although the syntax is second order, the semantics is first order. Predicate variables can only range over *predicate names*, not over all *predicate extensions (values)*. This distinction is important, because the set of predicate names is always finite, whereas the set of possible predicates is infinite. The scoping rules of Glue's modules and procedures give the compiler a list of the predicates which a subgoal variable could possibly match, so much of the predicate selection analysis can be done at compile time.

Here is an example set definition in Glue:

```
class_info( ID, Ins, Room,
                   tas(ID), students(ID)):=
    class_instructor( ID, Ins ) &
    class_room( ID, Room ).
tas(ID)(Grad_student):=
    class_subject( ID, Subject) &
    failed_exam( Grad_student, Subject ).
students(ID)(S):=
      attends( S, ID ).
```

The predicate `class_info` contains information about a class: its identifying code, instructor, set of TA's (Teaching Assistants), and set of students. The predicate `tas(ID)` defines the TA's for a course, notably those graduate students who failed the graduate qualifying exam in the course's subject area. Observe that the name of this predicate is a compound term. The predicate `students(ID)` contains the names of the students who are taking the course `ID`. The predicates `class_instructor`, `class_room`, `class_subject`, `failed_exam`, and `attends` are defined elsewhere. Here is an example EDB:

```
class_instructor( cs99, smith ).
class_room( cs99, mjh460a ).
class_subject( cs99, databases ).
failed_exam( jones, databases ).
attends( wilson, cs99 ).
attends( green, cs99 ).
```

It implies the following IDB tuples:

```
class_info( cs99, smith, mjh460a,
          tas(cs99), students(cs99)).
tas(cs99)( jones ).
students(cs99)( wilson ).
students(cs99)( green ).
```

A typical use of the `class_info` predicate might be:

```
class_info(C,I,R,T,S) & T(TA) & S(Student)
```

There is no automatic need for set unification in Glue-Nail; if two set valued attributes contain the same predicate name, then the two sets are identical. Hence much of the time a simple string-string matching suffices to determine equality. Of course, there will be times when the programmer needs to test whether two differently named sets have the same members. Here is a small procedure which compares two sets `S` and `T`.

```
proc set_eq( S, T: )
rels different(S,T);
  different(S,T):= in(S,T) & S(X) & !T(X).
  different(S,T)+= in(S,T) & T(X) & !S(X).
  return(S,T:):= !different(S,T).
end
```

5.2 Meta-programming

The HiLog system also allows the writing of parameterized predicates; for example the following NAIL! code defines the transitive closure of an arbitrary edge relation `E`:

```
tc(E,X,X).
tc(E,X,Z):- tc(E,X,Y) & E(Y,Z).
```

Our example in Section 4 could have taken `e` as a formal argument, thus allowing us to write one universal transitive closure predicate.

6 Modules

Both logic programming and deductive database languages have had problems "programming in the large," partly due to their lack of large scale code organization structures. Hence, in common with several other languages, Glue-Nail has a module system. Modules are purely a compile time concept, they do not have any run time semantics. Besides offering the usual advantages of separate compilation, modularity etc; modules give the Glue compiler valuable information concerning which predicates are visible at any point in a program. This information can be used to perform much of the predicate dereferencing at compile time, work which would otherwise have to be done at run time. This is especially true for subgoals which use predicate variables.

Modules have:

- a name,

- a list of imported EDB predicates,
- a list of imported IDB predicates,
- a list of exported IDB predicates, and
- IDB predicate code, both for Glue procedures and NAIL! rules.

Notice that a module can contain both Glue procedures and NAIL rules, thus allowing the programmer to group predicates by function, rather than by type.

7 A Larger Example

Space precludes us from including a large example, but Figure 1 gives some interface code lifted from a micro-CAD system, other examples may be found in [5]. We show a complete module, although this code was originally (and more sensibly) part of a larger module.

The procedure `select` allows the user to use the mouse to select a graphical element. The procedure first finds all elements within some tolerance of the user's mouse point. It then presents the elements to the user one at a time, in increasing order of increasing distance from the mouse point. The procedure returns the key of the selected element, if any.

Tha NAIL! rule `graphic_search` is user to find the elements within the given tolerance of the mouse point.

```
module example;
export select(:Key);
from windows import event( :Type, Data );
from graphics import
  highlight( Key: ), dehighlight( Key: );
edb element(Key, Origin, P1, P2, DS ),
    tolerance(T);

proc select( :Key )
rels
  possible(Key, D), try(Key), confirmed(Key);
  possible( Key, D ):=
        event( mouse, p(X,Y) ) &
        graphic_search( p(X,Y), Key, D ).
  repeat
    try(Key):=
      possible( Key, D ) &
      D= min(D) &
      It= arbitrary(Key) &
      --possible( It, D).
    confirmed(K):=
      try(K) &
      highlight(K) &
      write( 'This one?' ) &
      event( keyboard, KeyBuffer ) &
      dehighlight( K ) &
      KeyBuffer = 'y'.
  until {confirmed(K) | empty(possible(K)) };
  return(Et,Ed:Key):= confirmed( Key ).
end

graphic_search( p(X,Y), Key, Dist ):-
  element( Key, _, p(Xmin, Ymin), _,_ ) &
  tolerance( T ) &
  (X-Xmin)*(X-Xmin) + (Y-Ymin)*(Y-Ymin) < T.
end
```

Figure 1: Cad Example

8 Comparison to Some Other Systems

There are several more systems than we mention here, space prevents us from mentioning them all.

It could be said that Aditi has started with the relational engine (the *back end*), CORAL with the query language (the *front end*), with Glue holding the middle ground. NAIL! has already covered the front end.

8.1 LDL

LDL ([2], [4]) does not have a separate procedural language, it can itself perform I/O and EDB updates. As in Glue, update and I/O subgoals are fixed in a rule and cannot be moved. Rules containing updates are not allowed to fail. There is a `forever` meta-predicate. This predicate iteratively executes some rule body if that rule body is forever true (i.e. will never fail). The `forever` predicate is specifically designed to be used in rules with updates, so that the update will never fail.

Sets in LDL use extensional semantics, so that a set-valued attribute has the elements of a set as its value. Aggregation operators only operate over sets. The set grouping operator can only be understood if the usual tuple-based reading of a rule is abandoned. By "tuple-based," we mean that a tuple is true for a rule irrespective of all the other tuples which may or may not be true for that rule. With the set-grouping operator, one must implicitly gather up all the solutions of a rule, then form them into sets.

Sets (and therefore aggregations) must be stratified.

LDL does not have any meta-programming features. LDL has a module system. LDL modules have effects at run time as well as at compile time.

LDL uses stratified negation, although it could probably be extended to modular stratification.

LDL is compiled into C, and it has a foreign language interface to C. The LDL implementation has progressed much further than the implementations of either Glue-Nail or CORAL.

Our experience of writing LDL programs is that the procedural parts of the program (updates, and sets to a lesser extent) tend to dominate the programmer's thinking, hence negating the theoretically declarative nature of the language.

8.2 CORAL

The query language of CORAL [8] is very similar to that of LDL, however it uses the same two language approach as Glue-Nail. CORAL has chosen to use the existing object-oriented language C++ as the procedural language. The idea here is that the flexible type system of C++ will allow the easy creation of relation and tuple types in C++, reducing the impedance mismatch problem to a tolerable level. Although we have conducted no formal experiments, we suspect that C++ will present more of an impedance problem than Glue. C++ is built on C, and so has inherited the strong philosophies of C; philosophies which are radically different to those present in a logic-based query language. Using two completely separate languages also makes optimization very difficult. Glue-Nail aims to avoid these problems.

CORAL has a powerful and complicated module system. CORAL modules have run time semantics, Glue modules are purely compile time. Module import lists can be bound at run time, allowing a form of meta programming. In Glue-Nail we use the higher-order system of HiLog, there is no separate system for meta-programming.

CORAL allows variables in the EDB, partly to allow the use of the Magic Templates query compilation algorithm [7]. Hence a database lookup in CORAL requires unification, not just matching. Searching the database for a tuple match is the fundamental operation of any deductive database system. It remains to be seen whether the extra power provided by magic templates justifies the increased cost of a database lookup.

CORAL has the same set and aggregation scheme as LDL.

Like LDL, CORAL uses stratified negation.

CORAL has evaluable I/O predicates. They are given a logical semantics, but the semantics relies on state variables to ensure that the predicates are executed in the correct sequence. These extra variables carry no useful information, they merely exist to force a certain procedural reading.

8.3 Aditi

The Aditi project has concentrated on building an industrial strength back end for a deductive database language. Aditi-Prolog is the query language; it is pure Prolog with extensions for type and mode declarations, quantification and aggregate operations. At present Aditi-Prolog can be used interactively, although there are plans to embed Aditi-Prolog queries in Nu-Prolog or C.

Glue-Nail and Aditi have so far concentrated on different issues in deductive databases, so comparisons cannot yet be made.

9 Current Implementation

An experimental implementation has been written. Only the parser is written in C; the real meat of the compiler is written in Prolog. The compiler produces Prolog code for a small virtual machine, also written in Prolog. The virtual machine uses the Prolog database to store all relations. The parser is 3600 lines of C, the rest of the compiler is 4500 lines of Sicstus Prolog. The system compiles about two statements per Mips-second in compiled Sicstus Prolog on an IBM PC/RT.

The compiler is not a naive implementation; the aim has been to do as much as possible at compile time. For example, using the scope rules, in Glue it is possible at compile time to determine which predicate classes (i.e. EDB, IDB, Glue procedure, or reference[8]) a statically unbound name, such as `X`, could refer to at run time. A naive system would wait until `X` becomes bound at run time, and then check it against the four possible cases. The current compiler will have already eliminated those choices which were seen to be impossible at compile time. Procedure calls are expensive, so it is very important to identify at compile time those subgoals which cannot possibly be procedure calls.

We have used a pipelined (nested join) execution strategy for the implementation, this being forced on us

[8] Not discussed in this paper.

by Prolog's tuple at a time strategy. The experimental implementation has revealed a number of bottlenecks in a pipeline design. The main problem found was that certain language features force pipeline termination and the materialization of a supplementary relation. Breaking the pipeline and materializing the supplementary relation incurs some computational overhead, reduces the join order flexibility, may use extra space, and costs an extra load and store for each tuple. We can eliminate duplicates at this point, which is also expensive, but so far has proven to be cost effective. For various reasons (perhaps related to the particular application programs that we have run), the Glue assignment statements that we have examined have produced a large number of duplicates, so removing duplicates early has always been advantageous. However, in the worst case pipeline breakage is a loss. Breaks are required whenever a Glue procedure is called. We have to project the supplementary relation onto the input arguments, and call the Glue procedure once on all the input arguments. Breaks can also be required if we have an update operation in the body,[9] or an aggregator.

Two undergraduate students are writing medium sized test applications in Glue. Their experiences have helped in the development of the optimizer algorithms, in identifying problematic areas of the language design, and in debugging the compiler. More undergraduates will be writing senior projects in Glue-Nail.

10 Known Problems and Future Work

Glue is intended to be a complete application language, but in order to do so it probably needs a foreign language interface capability. Many applications use windowing systems, typically with a C interface. It is not reasonable to ask the programmer to write an entire windowing system in Glue, so we must provide some way of interfacing to languages such as C. It would also be difficult to write a windowing scheme in Glue, because Glue has such a simple type (and hence I/O) system. A window system might require talking to a device in terms of bitmaps or bytes, and Glue has no easy way of doing this.

We have written some non-trivial programs in Glue, but we plan to write several more so as to evaluate the system. In the process of designing Glue we wrote several small and one medium sized (400 lines) micro-CAD program. It would be useful to take a subset of an existing CAD program (or some other application), rewrite it in Glue-Nail and then compare the two implementations.

[9] A language feature which is not discussed in this paper.

It became clear when implementing the first version of NAIL! that it is a mistake to build a deductive database system on top of an existing relational database system. In a traditional relation database there are few relations, they live for a long time, and they usually have large numbers of tuples. These things are not true for deductive databases, where a query or program execution might produce hundreds of small, very short lived temporary relations. Such relations do not need the level of protection that a relational database provides, and in fact the system wastes much of its time performing such tasks. All the usual impedance mismatch problems occur, in particular the front end and the back end cannot intergrate their optimization strategies. Hence we need our own back end.

Work is in progress on designing an efficient relational back end for Glue. The kinds of applications we envision for Glue are single-user on small-to-medium sized databases. Thus, the back end will ignore concurrency issues and will manage relations in main memory as much as possible, storing EDB relations on disk between runs. The back end will be tailored to properties of deductive databases programs. For example, it will implement a "uniondiff" operator ([9]) in order to support compiled recursive NAIL! queries. Because Glue programs create and update many relations at run-time, queries involving those relations are difficult to optimize at compile-time. However, optimizing every statement each time it is executed is would be too expensive. Furthermore, for some queries, performing optimization may be more expensive than executing the query. Therefore, the back end will employ adaptive optimization techniques that select appropriate storage structures and access methods at run-time based on changing properties of the database and patterns of access. For example, an index could be created for a relation after the cumulative cost of selection by scanning the relation reaches the cost of creating the index.

We are currently building the NAIL! to Glue compiler. We may need to tune Glue so as to evaluate NAIL! queries as efficiently as possible.

11 Conclusions

Much work has been done on declarative query languages for deductive database systems — on the forms of such languages, and the algorithms to implement them. Current systems are addressing the problem of turning them into full application languages. The Glue-Nail system has taken the approach of providing two tightly knit languages, one declarative and one procedural. We feel that this approach is sound.

There are also many questions involved involved in the design of an efficient relational back end. Some headway has been made in reducing the cost of higher-order programming by compile time analysis, but much more work remains to be done.

We claim that Glue has effectively dealt with the impedance mismatch problem. Here follows a list of the main problems and Glue-Nail's solutions to them.

Separate optimization: NAIL! code is compiled into Glue procedures; the Glue optimizer runs over all the code.

Tuple oriented versus Set oriented: Both NAIL! and Glue are relation oriented, so the interface does not require buffering, back-tracking, or iterating schemes. The programmer does not have to constantly flip mental models.

Types: Identical systems,[10] both languages allow function symbols and use HiLog terms.

Syntax: Similar but not identical. The two types of code can occur in the same module, allowing code to be grouped according to function, rather than language.

Data Lifetimes: Explicit. Permanent data is stored in the EDB, Glue procedures and NAIL! rules both compute their values from the current state of the EDB.

A full description of Glue is available in [6].

12 Acknowledgements

The majority of the design of Glue is due to Geoffrey Phipps. Ken Ross provided the basic design of the assignment statement with respect to relations, and its coupling with the repeat loop. The implementation of the parser and compiler was done by Geoffrey Phipps. Marcia Derr is designing and implementing the relational back end. Compilation strategies for NAIL! are being investigated by Ashish Gupta, Geoffrey Phipps and Ken Ross. David Chen and Kathleen Fisher have written application programs in Glue. Besides the three authors, the following people also contributed to discussions concerning Glue's design: Hakan Jakobsson, Inderpal Mumick, Yehoshua Sagiv, and Jeffrey Ullman.

References

[1] Weidong Chen, Michael Kifer, and David S. Warren. HiLog: A First-Order Semantics for Higher-Order Logic Programming Constructs. In *Proceedings 2nd Int Workshop on Database Programming Languages*, 1989.

[2] Danette Chimenti and Ruben Gamboa. The SALAD Cookbook: A User/Programmer's Guide. Technical Report ACT-ST-346-89, Microelectronics and Computer Technology Corporation, 1989.

[3] Katherine Morris, Jeffrey Ullman, and Allen van Gelder. Design Overview of the NAIL! System. In *Proceedings 3rd Int Conference on Logic Programming*, pages 554–568, New York, 1986. Springer-Verlag.

[4] Shamim Naqvi and Shalom Tsur. *A Logical Language for Data and Knowledge Bases.* Computer Science Press, New York, 1989.

[5] Geoffrey Phipps. Glue - A Deductive Database Programming Language. In Jan Chomicki, editor, *Proceedings of the NACLP'90 Workshop on Deductive Databases.* Kansas State University Technical Report TR-CS-90-14, 1990.

[6] Geoffrey Phipps. The Glue Manual, Version 1.0. Technical Report STAN-CS-91-1353, Department of Computer Science, Stanford University, 1990.

[7] Raghu Ramakrishnan. Magic Templates: A Spellbinding Approach to Logic Programs. In *Proceedings Fifth International Conference on Logic Programming*, 1988.

[8] Raghu Ramakrishnan, Per Bothner, Divesh Srivastava, and S. Sudarshan. CORAL - A Database Programming Language. In Jan Chomicki, editor, *Proceedings of the NACLP'90 Workshop on Deductive Databases.* Kansas State University Technical Report TR-CS-90-14, 1990.

[9] Jayen Vaghani, Kotagiri Ramamohanarao, David B. Kemp, Zoltan Somogyi, and Peter J. Stuckey. The Aditi Deductive Database System. In Jan Chomicki, editor, *Proceedings of the NACLP'90 Workshop on Deductive Databases.* Kansas State University Technical Report TR-CS-90-14, 1990.

[10] Carlo Zaniolo. Deductive Databases - Theory Meets Practice. In *Proceedings 2nd International Conference on Extending Database Technology*, 1990.

[10] To be honest it could be said that neither language really *has* a type system.