CORAL–A Database Programming Language*

Raghu Ramakrishnan   Per Bothner   Divesh Srivastava
S. Sudarshan
Computer Sciences Department,
University of Wisconsin-Madison, WI 53706, U.S.A.

1 Introduction

CORAL[1] is a database programming language being developed at University of Wisconsin-Madison. We present an outline of the language design, which is currently far from complete. However, this outline should give the reader a feel for the language and the goals of the project.

We hope that CORAL will be suitable for a variety of applications, including genetic databases, path computations such as bill-of-materials, expert systems for financial, legal and other domains, natural language processing, and planning and search problems.

The first prototype version will be a main-memory implementation. We will subsequently implement a version, using Exodus or Postgres, that supports persistence and large disk-resident relations.

1.1 The CORAL Paradigm

A CORAL program can be viewed as a collection of *modules*. Each module is declared to be in one of two classes at compile time: *imperative*, or *declarative*. The design enables a CORAL user to write programs in a given style essentially independently of the semantics and constructs associated with the other style. A CORAL program has a main program (typically an imperative module) and a default (declarative) module that is searched for predicate definitions unless another module is explicitly specified.

For declarative modules, CORAL evaluation is guaranteed to be sound; it is complete whenever: (1) all "calls" to external modules are sound, complete and safely evaluable, (2) all "calls" to negative (and groupby) subgoals are safely evaluable, and (3) the "depends-on" graph for modules is acyclic.

In the following sections, we describe the various kinds of modules and how they interact. We conclude by identifying additional goals and directions for further research.

*This work was supported in part by a David and Lucile Packard Foundation Fellowship in Science and Engineering, an IBM Faculty Development Award and NSF grant IRI-8804319. The authors' e-mail addresses are {raghu,bothner,divesh,sudarsha}@cs.wisc.edu.

[1] CORAL stands for "COmbining Relations And Logic".

module *Sort* (*less*).
/* *less* is a relation-valued parameter which can be used in this module */
export *qsort* (bf).
/* other modules can query $qsort^{bf}$ */
import *Listroutines*.
/* predicates exported by *Listroutines* can be used here */

$r1 : qsort([\,], [\,]).$
$r2 : qsort([H \mid T], Y) \quad :- \ split(H, T, X1, X2), qsort(X1, Y1), qsort(X2, Y2), append(Y1, [H \mid Y2], Y).$
$r3 : split(I, [\,], [\,], [\,]).$
$r4 : split(I, [H \mid T], X1, [H \mid X2]) :- \ less(I, H), split(I, T, X1, X2).$
$r5 : split(I, [H \mid T], [H \mid X1], X2) :- \ \textbf{not}\ less(I, H), split(I, T, X1, X2).$

end module.

module *Listroutines*.
export *append* (bbf, bfb, fbb).

$r6 : append([\,], L, L).$
$r7 : append([H \mid T], L, [H \mid L1]) \quad :- \ append(T, L, L1).$

end module.

Figure 1: Generic Sort Module

2 Declarative Modules

Declarative programming is supported by allowing the user to organize sets of rules and facts into declarative modules. We introduce the syntax of declarative modules using a program to quicksort a list; this program is given in Figure 1 and the various features of the program are explained below. (The rule names are optional and are given here for convenience in referring to the rules in the discussion.)

2.1 Exporting and Importing Predicates

The module definition permits a subset of the defined predicates to be named as exported predicates, and other modules can pose queries over these predicates. The query forms permitted for each exported predicate are also indicated in the **export** declaration. For instance, the *Sort* module exports the *qsort* predicate (the only permissible query form has the first argument bound) and the *Listroutines* module in the above program exports the *append* predicate (in three permissible query forms—each corresponding to one of the three arguments being free and the other two bound).

Similarly, predicates that are defined and exported by other modules can be used within a module by importing the modules in which they are defined. If module M imports module N, M is said to *depend-on* N. In the above program, *append* is used within the *Sort* module by importing the *Listroutines* module in which it is defined.

However, if a predicate exported by two modules $N1$ and $N2$ has the same name p, and both modules are imported by a module M, this could lead to conflicts. One way of avoiding this conflict is to explicitly use $N1.p$ or $N2.p$ in rules in M.

Consider a module M that imports module N and has a reference to a predicate p exported by N. During the evaluation of M, N may be invoked several times; while applying a rule in M containing an occurrence $p()$, the partially instantiated literal $p()$ is treated as a goal and the module N is invoked with this goal before the rule application is completed. The result of an invocation is the complete set of answers to the goals that invoked the module; this set is then available to the invoking module. In order to be able to obtain the complete set of answers to this goal without invoking module M recursively, we require that the depends-on graph for modules be acyclic[2]. To satisfy this requirement, we require that any set of mutually recursive predicates must be defined within a single module; this results in an acyclic depends-on graph for modules.

We also provide the *macro definition* feature that does not impose such a restriction; this feature is different from the notion of a module and is described in some detail in Section 3.

2.2 Parametrized Modules and Parametrized Predicates

We allow modules to have relation-valued and module-valued parameters. Such a facility allows us to write generic code, for example, and provides limited second order capability. For instance, the *Sort* module of Figure 1 takes the comparison predicate *less* as a relation-valued parameter; this allows us to write a generic Quick Sort routine—one can sort any type of list by using the appropriate comparison relation as the actual parameter.

The semantics of allowing module-valued parameters to a module M is that predicates exported by argument modules can be used by M (in a manner similar to predicates in imported modules). For instance, the formal parameter to the *Sort* module could be replaced by a module-valued parameter *Comp*; if there was a *IntComparators* module that defined *geq* (greater than or equal to) in addition to *less* for integers, *IntComparators* could be used as an actual parameter to *Sort* during a "call," and $geq(I, H)$ could be used in the body of rule $r5$ instead of **not** $less(I, H)$. Note that this differs from importing *IntComparators* in that it also allows us to write generic routines.

Module-valued parameters are also used to provide database support in CORAL. Databases containing base relations can be viewed as declarative modules defining the base predicates and such modules can be specified as module-valued parameters to other modules that use these base predicates.

A module-valued argument N to a module M is similar to an imported module in M in that M is said to depend on N. An acyclicity requirement must also be satisfied by this depends-on relation, though (unlike imported modules) we may need run-time checks to ensure this. These run-time checks may be avoided by imposing further syntactic restrictions that ensure the acyclicity requirement at compile time itself.

We also allow predicates to have module and predicate valued attributes. However we impose restrictions to disallow quantification over such attributes, and impose acyclicity restrictions similar to those for module arguments. The use of such predicates can be seen in the matrix multiplication example of Figure 2 where the first attribute of *dim* is predicate valued.

[2]The restriction that calls across modules be non-recursive is related to completeness issues in our (bottom-up) evaluation strategy.

2.3 Rules

Rules in declarative modules are essentially Horn clauses with the following extensions.

Generalized Terms: We allow the use of uninterpreted functor symbols, and variables in the construction of terms. We do not require rules to be range restricted; hence, facts are not necessarily ground.

Negation: We allow the use of stratified negation. Thus, if the definition of predicate p depends negatively upon q, the definition of q is not allowed to depend upon p. The keyword **not** is used as a prefix to indicate a negated body literal. For instance, in the *Sort* module of Figure 1, we use the literal **not** $less(I, H)$; this use is stratified due to the acyclicity requirements imposed on modules.

Generating Sets and Multisets: We allow stratified multiset generation, with a few restrictions on the operations permitted since we wish to avoid performing the costly (multi)set unification operation. Multiset-valued arguments of a predicate can be generated by using the operator "<>" in the head of a rule. We can obtain a set from a multiset set using the prefix *set* operator. Among the operations permitted on sets and multisets are testing for membership (the infix operator **in** is used for this purpose), union, intersection and difference.

Aggregates: We allow a limited degree of higher-order manipulation through constructs such as aggregation, similar to constructs in LDL. The current list of aggregate operators includes $count, min, max, sum, product$ and $average$. The use of grouping and aggregation in CORAL can be seen from Figure 4 where

$$numofemps(M, count(set(< E >))) :- worksfor(E, M).$$

results in one tuple per manager with the second argument as the number of distinct employees working under him.

Named Attributes: We allow named as well as positional syntax for for predicate argument positions. For instance, if we had the declaration

set-relation $\{person\ ename, person\ mname, int\ sal\}$ emp;

then $emp(X, Y, \_)$ is equivalent to $emp(mname : Y, ename : X)$.

2.4 Additional Features of Declarative Modules

Declarative modules have other features which have not been illustrated in the Generic Sort example of Figure 1. We briefly describe these features in this section.

Multiset, Set and Irrset Predicates: We support a view of relations as multisets (in which the cardinality of a tuple is equal to the number of derivations for it), sets (where each distinct tuple appears exactly once) or irrsets (in which no tuple subsumes another). Again syntactic restrictions are imposed to provide a consistent semantics. For example, a set/multiset relation is not allowed to be mutually recursive with an irrset relation, when generalized tuples are permitted.

Metaprogramming: We plan to explore the use of Prolog-style metaprogramming features such as **var, nonvar, number, functor, integer**, etc. in declarative modules. For instance, we could have a rule:

$$p(X) :- q(X,Y), \mathbf{integer}(Y).$$

The semantics of this is that the metapredicate **integer** (Y) succeeds only for $q(X,Y)$ facts with an integer-valued second argument; metapredicates do not further bind their arguments. For other q facts the metapredicate fails. This could also be used for code optimization purposes.

Input and Output: We also provide evaluable predicates for reading and writing files, where files are treated as lists of *dataitems*. $readf(filein, dataitem1, fileout)$ must be called with its first argument bound and the other two free; $dataitem1$ is the first *dataitem* in $filein$ and $fileout$ is the list obtained after $dataitem1$ has been removed from the head of $filein$. $writef(filein,$ $dataitem2, fileout)$ must be called with its first two arguments bound and the other free; $fileout$ is the list obtained by appending $dataitem2$ to the end of $filein$; this provides the facility of append only files. Reading and writing files in this fashion provides a consistent logical semantics in that evaluating these predicates multiple times with the same value in the bound argument(s) yields the same values for the free arguments.

2.5 Evaluation of Declarative Modules

We plan to use a variety of optimization techniques for evaluating declarative modules; we outline these in this section without describing them in any detail. The optimization techniques include: rewritings such as Magic Templates, Existential query optimization, and Factoring; program analyses such as inferring functional dependencies, duplicate (subsumption) freedom, and argument types of predicates; and bottom-up evaluation optimizations such as semi-naive evaluation, rule and predicate orderings, and discarding derived facts during the evaluation.

3 Macro Definitions

We allow the use of macro definitions as illustrated in Figure 2. Macros are expanded inline at their point of occurrence (with name substitutions depending on the arguments).

Consider representing a matrix $A_{i,j}$ as a predicate $a(i,j,A_{i,j})$. Let the two matrices $A1$ and $C1$ be defined in terms of each other, using mutually recursive predicates $a1$ and $c1$, as shown in Figure 2. The first row of $c1$ can be computed using the first row of $a1$ (and the first column of $a2$). Since the values of the elements in the i^{th} row of the matrix $a1$ depend on the values of the elements in the previous row of $c1$, $a1$ and $c1$ are mutually recursive; hence, *matrixmult* cannot be defined as a module that takes (via either input parameters or imported modules) a and b and returns c (via an exported predicate), due to the acyclicity requirements imposed on modules. Macros thus provide a convenient mechanism for avoiding repetitive programming (the alternative would have been to repeat essentially the same code for defining both $c1$ and $c2$ in $Main$).

Predicate names that occur in the **local** declaration (*temp* and *tempsum* in this case) are treated as "local" predicate names and unique names are generated for these predicates at expansion time, to prevent name clashes between different expansions of the same macro definition.

macro $matrixmult(a, b, c)$.
local $temp, tempsum$.
/* these predicates are local to the macro; unique names need to be generated on expansion */

$temp(I, J, K, Val) \quad :- \ dim(a, Ra, Ca), dim(b, Ca, Cb), K$ **in** $\{1..Ca\}, a(I, K, Va), b(K, J, Vb), Val = Va * Vb.$
$tempsum(I, J, 0, 0).$
$tempsum(I, J, K, Val) :- \ K > 0, tempsum(I, J, K-1, Val1), temp(I, J, K, Val2), Val = Val1 + Val2.$
$c(I, J, Vc) \quad :- \ dim(a, \_, Ca), tempsum(I, J, Ca, Vc).$
$dim(c, Ra, Cb) \quad :- \ dim(a, Ra, Ca), dim(b, Ca, Cb).$
/* the first attribute of *dim* is predicate-valued */

end macro.

module $Main$.

$a1(1, J, Va) \quad :- \ J \geq 1, dim(a1, Ra1, Ca1), J \leq Ca1, b(J, Va).$
$a1(I, J, Va) \quad :- \ dim(a1, Ra1, Ca1), dim(c1, Rc1, Ca1), Rc1 \geq Ra1 - 1, c1(I-1, J, Va).$
$matrixmult(a1, a2, c1).$
$matrixmult(b1, b2, c2).$
/* the *matrixmult* macro is expanded inline at these two occurrences */

end module.

Figure 2: Matrix Multiplication

```
imperative-module Update_Sals (emp).
set-relation {person ename, person mname, int sal}  emp;
/* type of parameter to module */
{
/* Declarations local to module follow */
set-relation {person ename, int num}  numofemps;
int N;
tuple-of emp T;

    [    /* Begin inline block */
    set-relation {person ename, person mname}  worksfor;   /* local to block */

     worksfor(E, M)                       :- emp(E, M, _).
     worksfor(E, M)                       :- worksfor(E, E1), emp(E1, M, _).
     numofemps(M, count(set(< E >)))      :- worksfor(E, M).

    ]    /* End inline block */

for-each-tuple T in emp do
    if ((T.ename != "john") && (numofemps(T.ename, N)))   T.sal+ = T.sal * 0.10 * N;

}
```

Figure 3: Updating Employee Salaries

4 Imperative Modules

An imperative module is a program in an imperative language (currently, C++) augmented by adding a layer of new types and constructs. We introduce some features of imperative modules using an example program[3] that updates the salary of a person depending on the number of employees the person has (transitively). The various features of this program are explained below.

4.1 Types and Constructs

Two new types described in the above program are relations and tuples. As in C++, the type of a variable must be declared before the variable is used; the scope of variables of these new types is determined by the static scoping rules of C++. We also describe several constructs of the language.

Relations: There are three subtypes of type **relation**: **set-relation**, **multiset-relation** and **irrset-relation**, as we mentioned in the discussion of declarative modules. Types of relation variables can be declared as shown in the above program. The **relation** type is implemented as an ADT with the following operations:

Iterator: The general form of the iterator construct is:

for-each-tuple T [= `<tuple>`] **in** R **do** `<stmt>` [**until** `<condition>`]

[3]The syntax used for describing the program is still preliminary and has been used here for expository purposes.

T is a variable of type **tuple** (described in the following subsection). The relation R can be a defined relation (view), and the optional pattern `<tuple>` is used to restrict the subset that must be evaluated. The statement `<stmt>` is executed for each (generated) tuple in R that unifies with `<tuple>`. The **until** clause provides a way to break from the loop.

Updates: The update operators can only be applied to a database (EDB) relation or the relation to which a variable of type **relation** is instantiated. However, we cannot update derived relations (for instance, relations defined using (non-trivial) rules in declarative modules)[4].

Tuples to which a tuple variable is instantiated can be updated or deleted within iterator loops. For instance, in the above program employees are given salary increases by updating the tuple variable T that loops over the *emp* relation (using the C++ style $T.sal+ = \ldots$). A tuple can be deleted from the database by using **delete** T in an iterator loop. We also allow inserting and deleting multiple tuples into relations; the syntax and semantics for these are still under consideration.

Tuple Variables: We allow *tuple* variables for notational convenience. Tuple variables can be instantiated to argument vectors, but not to atoms. The terms in each field of an instantiated tuple variable T can be referred to by positional or name notation; for instance, we use $T.ename$ in the above program though we could have also used $T.1$ instead.

Query as a Boolean Condition: It is possible to evaluate a query as a condition. For instance, in Figure 4 $numofemps(T.ename, N)$ is a (partially instantiated) literal that is evaluated as a goal. If the goal is successful, the variables (N in our example) are instantiated by an answer substitution, and the condition evaluates to true. (If there are many answer substitutions, one is chosen non-deterministically.)

Calls to Modules: Imperative modules can be called as subroutines, by using the module name with an appropriate argument list. For instance, another imperative module (say, an accounting routine) could call $Update\_Sals$ with the employee relation as an argument to give salary raises. Any of the exported predicates of declarative modules in the program can be also called: $S = M.p(\ldots)$ results in the set of answers to the call being assigned to the (relation) variable S.

Inline Modules: As a syntactic convenience, a declarative module can also be defined within an imperative module. Such a module is delimited by [...], is not named and has no external interface definition. It is treated as a block in the imperative module for purposes of variable scoping, and behaves as an atomic statement that uses a set of logical rules to compute and assign sets of tuples to relational variables declared in the imperative module.

Logical Variables: In addition to relation and tuple variables, we also allow *logical variables*; these allow us to provide pattern-matching capabilities. A logical variable is initially free, and progressively gets bound (by unifying with first-order terms) to more instantiated terms until it is explicitly unbound (reset to free).

Input and Output: We provide additional I/O features to augment the low-level (C++) I/O facilities; these allow us to conveniently read/write files as relations.

5 Future Directions

Currently, we have implemented several features of CORAL declarative modules, but have not yet implemented imperative modules.

[4]Note that the set of updatable relation names can be identified at compile time.

We wish to investigate several issues using CORAL as a vehicle. Of particular interest is support for combining declarative programming with other styles of programming, and this has been discussed in the preceding sections. We intend to study many other problems in this setting, and we touch upon them below.

Controlled Modules: We would like to use controlled modules to explore Kowalski's paradigm of Algorithm = Logic + Control. Such modules can can contain anything declarative modules can and can include some additional specification of control.

The control constructs we would like to consider include the primitive *choice* that, like Prolog's *cut*, allows us to prune the computation. However, it can be understood declaratively. Consider a literal $p(\ldots)$ in a rule. Let X and Y denote (disjoint) subsets of the set of argument positions of p. Then $< p(\ldots), X \rightarrow Y >$ is a *choice literal*. The meaning is that for every assignment of terms to the positions in X, only one assignment of terms to the positions in Y need be considered. This information may be used by the implementation to avoid generating further solutions. Note that this semantics is different from the semantics given to *choice* in LDL.

We would also like to allow the user to specify a desired semantics by providing an ordering over program rules in the form of regular expressions over such rules.

Prolog Modules: We would also like to explore the possibility of providing some standard dialect of Prolog as a Prolog module in CORAL. A goal to such a module would then be executed by invoking the Prolog system.

Support for Richer Data Types and Abstractions: Such support would enhance the declarative paradigm, but presents several technical challenges.

Constraints: We wish to investigate support for constraints over domains such as the rationals, as in systems like CHIP and CLP(R), in the context of bottom-up evaluation.

Parallel Evaluation of Declarative Modules: We have not discussed evaluation techniques in this overview, but the approach we take is based upon program transformation — using Magic Templates and additional optimizations — followed by fixpoint evaluation. This offers much promise for parallelization.

The Role of Object-Orientation: Can object-orientation have a place within within the declarative paradigm, or vice-versa? One possibility is suggested by the embedding of declarative facilities in CORAL; the user might be able to manipulate objects in an object-oriented style in imperative modules, and as values (using an object identifier as a "handle") within declarative modules.

Debugging and Program Development Environment: Supporting a good program development environment in the context of such a rich and diverse set of constructs presents a challenge. In particular, the declarative subset, which is extensively optimized, must be treated in such a way that the programmer is not required to think operationally; at the same time, he must have enough details of the execution visible to identify the source of an error.

6 Acknowledgements

While we do not provide a list of references in this informal presentation, we would like to acknowledge our debt to SQL, Starburst, LDL, NAIL!, and various implementations of Prolog.