THE CORAL USER MANUAL
A Tutorial Introduction to CORAL*

Raghu Ramakrishnan Praveen Seshadri Divesh Srivastava
S. Sudarshan
Computer Sciences Department,
University of Wisconsin–Madison, WI 53706, U.S.A.

*The authors' e-mail addresses are {raghu,divesh,praveen}@cs.wisc.edu; sudarsha@research.att.com.

Contents

1 Preface **3**

2 Introduction **4**

2.1 A Guide to this Document 4

2.2 Other Documentation 6

3 A First Session **7**

3.1 Getting Started With Some Help 7

3.2 A Note on Modules 8

3.3 Compiling and Executing Programs 9

3.4 Order of Answers 12

3.5 The Number at the Prompt 12

3.6 A Note on Entering Rules 12

3.7 Using Arithmetic Predicates 13

3.8 Debugging in CORAL 14

3.9 A Note for the Lazy 14

3.10 Some Guidelines for Writing Efficient Programs 15

4 Declarative Language Features: Basics **16**

4.1 Facts 16

4.2 Rules 17

4.3 Semantics of a Program 17

4.4 A Note on Evaluation 20

5 Declarative Language Features: Negation **21**

5.1 Non-Floundering Programs 21

5.2 Negation and Recursion 22

5.3 Negation in Pipelined Modules 25

5.4 A Note on Efficiency 25

6 Declarative Language Features: Sets and Multisets 26

6.1 An Important Restriction 26

6.2 Creating Multisets 26

6.3 Multiset Operators 28

6.3.1 The member Multiset Operator 29

6.3.2 Converting Lists to Multisets 32

6.4 Multisets in Pipelined Modules 33

6.5 Relations as Multisets 34

6.6 Multiset Grouping and Stratification 36

6.7 A Note on Efficiency 37

7 Declarative Language Features: Advanced 38

7.1 Non-ground Facts 38

7.2 Head Deletes 41

7.3 Prioritization 44

7.4 Aggregate Selections 45

7.4.1 Aggregate Selections and Single Answer Queries 50

7.5 Monotonic Programs 52

8 Modules in CORAL 55

8.1 A Note on Module Names 55

8.2 Inter-Module Calls . . . 56

8.3 Negation, Grouping and Module Structure . . . 57

8.4 Head Deletes, Prioritization and Module Structure . . . 69

8.5 The Save Module Annotation . . . 69

9 Declarative Language Features: Annotations and Control 72

9.1 Program Transformation . . . 72

9.1.1 Basic Rewriting Techniques . . . 72

9.1.2 Passing Bindings in Conjunction with No Rewriting . . . 74

9.1.3 Factoring . . . 75

9.1.4 Negation and Grouping . . . 77

9.2 Controlling the Mode of Execution . . . 77

9.2.1 Intelligent Backtracking . . . 78

9.2.2 Indexing in CORAL . . . 81

9.2.3 Duplicate Checks . . . 83

9.2.4 Lazy Evaluation . . . 84

10 Declarative Language Features: Pipelined Evaluation 85

10.1 Negation and Grouping in Pipelined Modules . . . 85

10.2 Pipelined Evaluation and Embedded Commands . . . 85

10.2.1 Indexing and the Order of Insertions . . . 87

10.2.2 Converting Multisets to Lists . . . 87

10.3 The Cut Operator . . . 89

10.3.1 Cuts and Single Answers for Materialized Execution . . . 90

10.4 Ordering of Base Facts . . . 91

10.5 Pipelined Evaluation and Exported Query Forms 92

11 Some Useful Built-in Predicates 94

11.1 Arithmetic Built-Ins 94

11.2 Multiset Operators 94

11.3 Metaprogramming in CORAL 95

11.4 Miscellaneous Operators 95

12 CORAL Commands 97

12.1 Executing Commands 97

12.2 Commands Discussed Elsewhere 97

12.3 Consult 97

12.4 Commands That Modify the Execution Defaults 98

12.5 Commands for Debugging 98

12.5.1 Explanations 99

12.5.2 Tracing 100

12.5.3 Profiling 101

12.5.4 Timing Information 101

12.6 Commands that Manipulate Relations and Workspaces 101

13 CORAL and C++ 103

13.1 Adding Relations to C++ 103

13.2 Defining Built-Ins 107

14 Extensibility in CORAL 111

14.1 Arrays 111

15 Programming in CORAL: Some Guidelines **117**

15.1 Order of Literals in a Rule 117

15.2 Modules 118

15.3 Pipelining vs. Materialization 118

15.3.1 Materialization 119

15.4 Using C++ and CORAL Effectively 119

16 Current Status **121**

A The CORAL Installation Guide **124**

B The CORAL–C++ Interface Specification **125**

1 Preface

CORAL[1] is a database programming language based on Horn clause logic developed at the University of Wisconsin–Madison. Source code for CORAL (written in C++) is available by anonymous ftp over the internet. The CORAL project was initiated in 1988-89, under the name Conlog, and a preliminary report was presented at a workshop in the NACLP 89 conference. Preliminary versions of the system have been used by a few groups, but this is the first widely available release. We would welcome any feedback on the system. Comments, bug reports and questions should be mailed to *coral@cs.wisc.edu*.

We would like to acknowledge the contributions of the following people to the CORAL system. Per Bothner, who was largely responsible for the initial implementation of CORAL that served as the basis for subsequent development, was a major early contributor. Joseph Albert worked on some aspects of the set-manipulation code; Tarun Arora implemented several utilities and built-in/library routines in CORAL, and contributed to the explanation package; Tom Ball implemented an early prototype of the Seminaive evaluation system; Lai-chong Chan did the initial implementation of existential query optimization; Sumeer Goyal implemented embedded CORAL constructs in C++; Vish Karra did the initial implementation of pipelining; Robert Netzer did the initial implementation of Magic rewriting; and Bill Roth created test suites, improved some of the I/O routines and implemented the graphical aspects of the explanation package.

This work was supported by a David and Lucile Packard Foundation Fellowship in Science and Engineering, a Presidential Young Investigator Award, with matching grants from Digital Equipment Corporation, Tandem and Xerox, and NSF grant IRI-9011563.

R. Ramakrishnan
P. Seshadri
D. Srivastava
S. Sudarshan

[1] CORAL stands for "COntrol, Relations And Logic".

2 Introduction

This tutorial provides a step-by-step introduction to CORAL through the use of example programs. All programs and data-sets in this tutorial are available as part of the CORAL release, in the directory named *doc/examples*. The file containing the program is named when the program is presented.

CORAL is an interactive system that is invoked by the command `coral` at the UNIX prompt. You will then see the CORAL prompt 1 >, from which you can create, modify and execute programs. CORAL can also be used essentially as a C++ extension. In this mode, as usual there is a main program and, possibly, several subprograms.

There are many ways to look at CORAL. It can be viewed as a (deductive) database query language, a logic programming language, or as an extension of C++ that provides support for creating and manipulating relations. An extensive *declarative* sublanguage is supported, and this can be used to define relations (or views, in relational database terminology). In addition, a layer of constructs is provided to allow easy manipulation of relations — either explicitly stored relations or defined relations — in C++ code.

2.1 A Guide to this Document

The following are some of the notable aspects of the CORAL system. They are addressed in detail in one or more sections of this document.

Declarative Language Features:

A powerful language based on Horn clause rules is supported. For programs containing only pure Horn clauses, the declarative *least fixpoint* or *least Herbrand model* semantics is supported. Intuitively, this means that rules can be understood simply as if-then statements in logic, without regard to the order of evaluation.

Programs with *negation* or *set-grouping* are restricted to be *left-to-right modularly stratified*. For such programs, the *well-founded model* semantics is supported. The intuitive reading of programs as if-then statements is preserved.

Programs can also be evaluated in *pipelined* fashion, very similar to Prolog execution.

Relevant Sections: 4, 5, 6, 7, 8, 10, 15

The Module System:

A CORAL program is a collection of *modules*, and every module can be understood simply as defining one or more exported *relations*, or sets of facts.

Relevant Sections: 8

Control Features:

The evaluation of declarative modules can be refined by the user through high-level annotations that provide hints to the compiler. In particular, the user can optionally control the use of subsumption checks, add indices, influence the order in which inferences are made and when facts are discarded, choose from a variety of optimizing program transformations, specify when a single answer is sufficient, choose pipelined or memoing evaluation strategies, etc.

The user is not required to specify any of this control information since the system makes default decisions in the absence of user-specified annotations. While the range of features available for controlling the evaluation is extensive, efficient programs can be written by keeping in mind some simple points, given a broad understanding of the underlying evaluation techniques. A summary of these points is presented in 15.

Relevant Sections 8, 9, 15

Persistent Relations:

Disk-resident persistent relations are supported using the EXODUS storage manager [CDRS86].

Relevant Sections 12

Multiple Workspaces:

A workspace is a collection of relations, each defined by an explicit collection of facts or by a collection of rules. A user can simultaneously maintain several workspaces and switch between them. All queries are evaluated against the current workspace.

Relevant Sections: 12

I/O:

Both fact-at-a-time and relation-at-a-time I/O are supported. Relations can be read in or written out in relational form (i.e. each tuple as a predicate name followed by arguments) or in tabular form (i.e. each tuple as just a vector of arguments), in sorted order if desired. (More information is available through the *help* command; type *help*(*io*).)

Debugging and Profiling:

The execution of a program can be traced and profiled. There is also an *explanation* facility that allows users to examine the *derivation trees* for generated facts using a graphical menu-driven interface.

Relevant Sections: 12

CORAL–C++ Interface :

CORAL provides an interface to C++. Some constructs are added to C++ to enable the user to manipulate relations easily. This extended C++ can be used to define relations just as in other CORAL modules. However, such code has to be linked with the CORAL system code, and this is a relatively slow process. We therefore recommend that imperative modules be used sparingly. However, imperative modules are very useful in adding new data types to CORAL or in extending the set of built-in functions.

Relevant Sections 13 and 14

Extensibility: CORAL is extensible in many ways: new data types as well as new relation and index implementations can be added.

Relevant Sections 14

2.2 Other Documentation

The reader is referred to "An Overview of CORAL" [RSSS93] for an overview. The overview complements this tutorial introduction by elaborating upon issues introduced and illustrated here, and the two documents are intended to be used in conjunction. In addition, there is an on-line *help* command. See "Implementation of the CORAL Deductive Database System" [RSSS92] for a description of how CORAL is implemented.

3 A First Session

The example programs used in this chapter are quite simple, and are introduced informally. The goal here is to show you how to use the CORAL system, so that you can actually run the programs discussed in later sections.

We will assume that the CORAL system is already installed. Instructions for doing this are given in Appendix A (Installation Guide).

3.1 Getting Started With Some Help

To invoke CORAL, type `coral` at the UNIX prompt (%):

```
%coral
-----------------------------------------------------

Welcome to CORAL.

All commands MUST end with a period .
  Type help. to access help information.

-----------------------------------------------------

1:>
```

The "$n >$" symbol is the CORAL command interface prompt. You can execute several commands now; note that every command is terminated by a ".". The most useful command initially is the `help` command:

```
1:>help.

---------------- CORAL :: General Help Facility ---------------------

All queries must be preceded by a ?

All queries and commands must be followed by a period.
```

```
For more specific help information, type help(TOPIC).


HELP TOPICS


annotations builtins commands consult

c++interface defaults imperative io

language  modules persistence profile

query relations rules timer

trace  shell workspaces


-------------------------- END HELP ---------------------------------
```

To find out about the other commands that you can execute at the CORAL prompt, for example, you can type `help(commands)`. The `help` command lets you find out many things that you need to know to use CORAL, but it does not discuss several language features that are described in this tutorial or the overview [RSSS93]. If you can't find a specific `help` topic that addresses your question, it is likely that you will find the information in this tutorial or its companion document, the overview.

You can execute any UNIX command from the CORAL prompt by typing (the quotes are required):

$n >$ `shell("unix-command").`

3.2 A Note on Modules

Every CORAL program is a collection of one or more *modules* and a collection of *facts*. When a module is compiled, it provides a definition for one or more predicates that it *exports*. Queries on these predicates are answered using the compiled form of these definitions; a predicate exported from one module can be used in another module.

For now, we will assume that there is a single module. Programs with multiple modules are discussed in Section 8.

3.3 Compiling and Executing Programs

Facts can be entered at the command line by typing them in (if the insert_mode flag is ON; the default is OFF, but this can easily be changed using the 'set' command from the prompt).

n >`parent(adam,cain).`
n >`parent(adam,abel).`
n >`parent(eve,cain).`
n >`parent(eve,abel).`
n >`parent(abel,sem).`

Note the terminating period; it is required. To see that these facts have indeed been entered correctly, type

n >`list_rels.`

CORAL responds with:

```
parent/2 :  (base)
```

indicating that there is a relation *parent* whose facts have been explicitly enumerated (i.e., it is a base relation, not defined by rules). To see the actual facts in *parent*, we can pose a *query*:

n >`?parent(X,Y).`

Note the "?" prefix; this indicates a query. Without this prefix, CORAL attempts to insert the fact parent(X,Y) if them "insert_mode" flag is on, and an error message is printed if the flag is off. By default, the insert_mode flag is OFF. (To see the status of all such flags, type *display_defaults*. These flags can be set from the prompt using *set/clear* and *assign* commands; the *help* command can be used to get more information about these commands.

The query ?parent(X,Y) asks for all (X,Y) pairs such that there is a corresponding fact in the *parent* relation, and we get:

```
X=abel, Y=sem.
X=eve, Y=abel.
X=eve, Y=cain.
```

```
X=adam, Y=abel.
X=adam, Y=cain.
(Number of Answers = 5)
```

The above facts can also be entered by putting them in a file and consulting the file. Note that the insert_mode flag does not affect facts when they are consulted from a file; it only applies to facts entered via stdin, which is normally the terminal. (They are in a file named start1.F, if you want to try this. Recall that all example programs are in directory *doc/examples.*)

The easiest way to compile a program is to use a text editor, say vi, to create a file containing the program, and to then `consult` the file. There is a simple program in file start1.P. To see this program, type

n >`shell("more start1.P").`

and the following is displayed:

```
module start_eg1.
export grandparent(ff,bf).

grandparent(X,Z) :- parent(X,Y), parent(Y,Z).

end_module.
```

The line "`module start_eg1.`" marks the beginning of this module; the name of the module is required, but has no significance (at least, in the current version of CORAL). The line "`export grandparent(ff, bf).`" declares that this module provides a definition of the predicate grandparent, and that two *query forms* are optimized. The first optimized query form is *grandparent_ff*, which asks for a listing of all grandparent facts, and the second optimized query form is *grandparent_bf*, which takes a constant, say john, and asks for all values of X such that there is a fact of the form grandparent(john,X). (*f* stands for "free" and *b* stands for "bound".) The line "`end_module.`" marks the end of the module.

The program in file start1.P can be compiled by typing

n >`consult(start1.P).`

Consulting a file results in the compilation of all modules in the file. To see the effect of consulting start1.P, type

n >`list_rels.`

A program is specified by enclosing rules within one or more modules. It is usually stored in a text file whose name is, by convention, terminated by ".P". Consulting the file is virtually equivalent to actually typing the contents of the file at the prompt. When such a file is consulted, the entire compilation process — program transformations, index creation etc. — is invoked. Sets of facts can be in any file, and can be specified both within a module, and independently outside of any module (perhaps in a single file containing both modules and facts). Putting facts within modules results in their being handled rather inefficiently as general program rules; however, this may be appropriate for enforcing an order on the facts in conjunction with pipelined evaluation (see Section 10).

The line "`grandparent(X,Z) :- parent(X,Y), parent(Y,Z).`" defines *grandparent*. This can be read intuitively as "X is the grandparent of Z if X is Y's parent and Y is Z's parent". In relational database terms, this rule is a *join* of the relation *parent* with itself.

Type

n >`?grandparent(U,V).`

to see all the *grandparent* facts. This corresponds to the query form $grandparent\_ff$. Incidentally, the choice of variable names is not important here; you will see the same answers if you type

n >`?grandparent(X,Y).`

On the other hand, there are no answers to the query `?grandparent(X,X)` since there is no one who is their own grandparent.

To find out all grandchildren of "adam", type

n >`?grandparent(adam,X).`

This query is evaluated using the optimized version of the program for the query form $grandparent\_bf$. The answers to this query will be correctly computed even if the query form $grandparent\_bf$ were not exported; only, the execution would be less efficient. The following is equivalent to the above query, and is also evaluated using the optimized program for query form $grandparent\_bf$:

n >`?U=adam, grandparent(U,X).`

3.4 Order of Answers

CORAL generally produces answers in no particular order since relations are often accessed via hash-based indexes. However, CORAL does provide a way to display and print out answers in sorted order using the *sort* command. Type *help*(*sort*). for more information on this command.

3.5 The Number at the Prompt

Perhaps you have noticed that the prompt has a number that incremented by one each time you type in a command. This number is the basis for a simple *history* mechanism, as is found in many other systems. Type *history*. to see a list of the most recent 25 commands along with their numbers. To repeat any of these commands, simply type *history*(*number*).

3.6 A Note on Entering Rules

Whereas facts can either be consulted from a file or typed in at the CORAL prompt, it is important to note that rules should only be within modules that are consulted. If you type in:

n >`gp(X,Z) :- parent(X,Y), parent(Y,Z).`

you will see an error message. (A similar error occurs if the connective $<-$ is used, since it is just alternative notation for :-)

However, CORAL supports various forms of rule-based assignments from the prompt. For example,

n >`gp(X,Z) := parent(X,Y), parent(Y,Z).`

assigns the result of joining the *parent* relations to the *gp* relation. Any existing tuples in *gp* are over-written. If += were used as the rule connective symbol, the new tuples would be added to existing tuples in *gp*, and if -= were used, the new tuples would be deleted from the set of existing tuples. The difference with respect to the :- connective is seen in the following:

n >`p(X,Z) := p(X,Y), p(Y,Z).`

The old set of *p* facts is used in the join, and when the join has been fully computed, the result set of tuples is assigned to p, replacing the old set. In contrast, the use of :- (only possible inside a module) indicates a recursive definition of p; if p is initialized to some set of tuples, the use of this rule could compute a transitive closure of that set of tuples.

3.7 Using Arithmetic Predicates

The next program illustrates arithmetic in CORAL, and is in file start2.P:

```
module start_eg2.
export incr1(bf), incr2(bf).

incr1(X,Y) :- Y = X+1.

incr2(X,X+1).

end_module.
```

The two exported predicates are equivalent. The query `?incr1(2,X)` will succeed with `X=3`, as will the query `?incr2(2,X)`. It is important to note that the first argument *must* be bound. Otherwise, the implementation of plus (+) will raise a run-time error. While it is clear why a query of the form ?incr2(X,Y) causes a problem, you may wonder why ?incr2(X,3) should do so. After all, it seems clear that X=2 is the correct answer in this case. The problem is that CORAL does not attempt to solve constraints; the implementation of plus takes two arguments and produces their sum, but does not go in the other direction, i.e., it does not take the sum and one argument and produce the other argument. Thus, in the presence of arithmetic predicates, the position of body literals takes on a significance that is absent in the purely logical reading of a rule.

A general point to keep in mind with respect to the use of arithmetic, related to the above observations, is that CORAL proceeds left-to-right within a rule, by default. *If the arguments to arithmetic predicates are not sufficiently bound at the point in the rule where they occur, this will lead to a run-time error.* Two simple guidelines that follow from this observation eliminate many potential errors:

1. If an argument position in the head of a rule contains an arithmetic expression, then it should correspond to a "free" (f) argument position. That is, this argument should not be bound in calls to the rule.
2. If an argument position in the body of a rule contains an arithmetic expression, all variables in it should also appear to its left in the rule.

The following is the list of arithmetic and comparison operations supported in CORAL: $/, -, +, *, pow, abs, mod, <, >, <=, >=, =$.

A list of several important builtins, along with acceptable binding patterns, can be seen by typing "help(builtins)" at the CORAL prompt. Many of the CORAL commands are also implemented as builtins; these builtin predicates are stored in the *builtin_rels* workspace, and a comprehensive list can be obtained by typing *list_rels*(*builtin_rels*) at the CORAL prompt.

3.8 Debugging in CORAL

Since many people like to learn about a system by running programs, it seems advisable to present the debugging features of CORAL even at this early point!

CORAL provides three ways to obtain more information about a program execution. The first is a tracing facility. You can find out more about this by typing *help*(*trace*). at the CORAL prompt. This enables you to create a listing of all inferences as they are carried out, either on stderr (the default) or on some file that you choose. You can trace the entire execution, or trace the execution for selected (exported) predicates. This feature is likely to be useful if you are trying to identify the source of a logical error in your program.

The second feature is a profiling facility. You can find out more about this by typing *help*(*profile*). at the prompt. This enables you to collect a lot of information about the numer of inferences, unifications, and other basic operations that are carried out. This feature is likely to be useful if you are trying to find out which parts of a program are taking the most time during execution.

The third feature is an *explanation* facility. The *explain_on* command can be used to turn on "forwarding" of all inferences to a graphical interface for display as *derivation trees* Type *help*(*explain_on*). and *help*(*explain*). at the CORAL prompt for more information.

3.9 A Note for the Lazy

If the long command names get to be a nuisance, you can redefine them. This is discussed in the section on commands (Section 12).

Also, CORAL can be invoked with several options. For example, typing in

```
coral -i filein -o fileout -q -I
```

has the following effect: filein is consulted automatically, output is written to fileout (default being the terminal), CORAL runs in "quiet" mode (suppressing several messages), and without "interactive mode". (By default, CORAL is not quiet, and it runs in interactive mode, wherein the user is prompted after each answer before proceeding with the computation.)

To see a list of the options, type in *coral -x*. ("x" is a non-existent option; all acceptable options are listed in the resulting error message.)

3.10 Some Guidelines for Writing Efficient Programs

At this point, you are probably more concerned with writing simple programs that work rather than writing efficient programs. However, it is appropriate to close this introductory chapter with some remarks about how to write efficient programs.

The CORAL compiler attempts to perform some optimization on its own. However, you should bear in mind that it has some important limitations. While some of these can be fixed by refining it further, our conclusion, after using CORAL extensively, is that a user must have some minimal understanding of how programs are evaluated in order to write efficient programs. While this may seem to be at odds with the claim that CORAL is declarative and CORAL programs can be understood non-operationally, it is nonetheless true. Your *can* write programs that can be understood non-operationally, and often your programs will run just fine. But if your program runs slower than you'd like, you may have to understand the underlying evaluation method, at least in broad terms, in order to make it run efficiently. In CORAL, a program can typically be made to run faster without changing the underlying logic, by means of some high-level re-organization and the use of a "annotations" or hints to the compiler. The issue of efficiency is addressed in many of the subsequent chapters, and Section 15 contains a summary of the points to keep in mind.

4 Declarative Language Features: Basics

In Section "Declarative Language Features: Basics" of [RSSS93], the concepts of constants, variables, terms, facts and rules in CORAL are introduced. We refer the reader to the overview, and to introductory logic programming texts such as [Llo87], for a detailed presentation of logic programs.

The following examples illustrate these concepts informally (and briefly!).

4.1 Facts

Assume that we have executed the command `coral` from the UNIX prompt. At the CORAL prompt $n >$, we can enter facts:

$n >$`employee(john, "Toys for Tots", 3, 35.5).`

This fact could be interpreted as follows: John is an employee in the Toys for Tots department who has been with the company for 3 years and makes 35.5K. The first string is not quoted since it begins with a lower case letter and contains only letters or numerals (indeed, it does not even contain numerals). Argument types are not declared in CORAL, except for persistent relations, which we discuss later. Various built-in predicates, for example arithmetic predicates, expect arguments of a particular type, and such type checks are carried out at run-time.

We can add another fact:

$n >$`employee(joan, "Toys for Tots", 2/3, 30).`

This indicates that Joan has worked for 2/3 years; the expression 2/3 is evaluated and stored as a double-precision floating point number. The following fact states that Adam has worked for the company "since day 1":

$n >$`employee(adam, "Toys for Tots", "since day 1", 30).`

CORAL accepts this fact. However, if this fact were to be used later in a context where the third argument was passed to an arithmetic predicate (to compute seniority, perhaps) there would be a run-time type error.

CORAL supports more complex terms:

$n >$`address(john, residence("Madison",street_add("Oak Lane", 3202), 53606)).`

John lives in Madison, and has a street address with a zip of 53606. We see the use of *functor symbols* used as record constructors. The string "street_add" is used as a functor symbol of arity 2, and "residence" is a functor of arity 3. Notice the nesting of terms in this fact.

n `>address(john, po_box("Madison", 288)).`

John also has a post-office box; in this case, the address is simpler in form.

4.2 Rules

Rules take the form *head :- body.* Informally, a rule is to be read as an assertion that for all assignments of terms to the variables that appear in the rule, the head is true if the body is true. (Thus, a *fact* is just a rule with an empty body.) In the deductive database literature, it is common to distinguish a set of facts as the EDB or *extensional database*, and to refer to the set of rules as the IDB or the *intensional database*. The significance of the distinction is that at compile time, only the IDB is examined; the EDB is viewed as an input.

Consider a rule

$$ancestor(X, Y) \colon -parent(X, Z), ancestor(Z, Y).$$

and suppose that we have the facts $parent(1, 4)$ and $ancestor(4, 5)$. We "unify" $parent(1, 4)$ with the first literal of the rule, by setting $X \leftarrow 1, Z \leftarrow 4$. Now we can further unify $ancestor(4, 5)$ with the second literal of the rule by setting $Y \leftarrow 5$ (Z has already been assigned a value). Since all the literals in the rule have been unified with facts, we can now derive the head fact $ancestor(1, 5)$ (which is obtained from the assignment of values to the head variables).

A note on syntax: Either $: -$ or $< -$ can be used as the separator between the head and the body of a rule.

4.3 Semantics of a Program

A program is a collection of facts and rules. A principal attraction of the logic programming paradigm is that there is a natural meaning associated with a program. As we have seen, each fact and rule can be read as a statement of the form "if <something is true> then <something else is also true>". In the absence of rules with negation and set-grouping, the meaning of a program can be understood essentially by reading each of the rules in the program in this manner, with the further understanding that the only true facts are those that are either part of the program or that follow from a repeated use of program rules. (Rules with negative body literals and/or set-grouping in the head are discussed in Sections 5 and 6. The semantics are a natural extension that preserves the intuition behind the "if-then" reading of program rules.)

CORAL goes much farther towards supporting this simple semantics than other logic programming languages, for example Prolog. For programs with only constants and variables as terms and without negation or set-grouping, this simple semantics is guaranteed. (More precisely, the default evaluation strategy is sound and complete for this class of programs. If the user specifies annotations to modify the execution — this aspect of CORAL is discussed in Section 9 — this guarantee could be compromised.)

It is possible that the set of relevant inferences is infinite in the presence of terms with function symbols; in this case, termination cannot be guaranteed. The following program, in file declba1.P, illustrates this:

```
module declba_eg1a.
export nonterm(f).

%  Does not terminate; infinite number of answers.

nonterm(0).
nonterm(1).
nonterm(f(Y)) :- nonterm(Y).

end_module.
```

At the end of each iteration, the answers generated in that iteration are displayed, and CORAL waits to be prompted before generating further answers. However, the program will not terminate since the set of facts to be generated is infinite.

The following program, also in file declba1.P, illustrates the consequences of top-down goal generation.

```
module declba_eg1b.
export toomanygoals(f).

%  The default CORAL strategy is to use (Supplementary) Magic
%  Sets rewriting. This mimics the generation of goals in Prolog,
%  and leads to non-termination.  The annotation ''@no_magic''
%  would override the default; with this, execution will terminate.
```

```
%  (The two facts are generated, and the third rule
%  does not match any fact.)

toomanygoals(0).
toomanygoals(1).
toomanygoals(Y) :- toomanygoals(f(Y)).

end_module.
```

CORAL uses the Magic Templates rewriting algorithm to "mimic" Prolog. The set of goals generated in a Prolog-style execution are computed in so-called "magic" predicates, which serve as filters on the original rules to restrict the computation to only generate facts that match some goal. In this program, the set of goals is infinite even though the set of answers to the original program is finite. Thus, the fixpoint evaluation of the original program terminates whereas the fixpoint evaluation of the rewritten program does not. The Magic Templates algorithm is thus just a heuristic, and it is sometimes preferable not to use it. The annotation "@no_magic" can be used to suppress the rewriting step. To see the difference, run the above two programs with tracing turned on. (Tracing is a useful debugging facility in CORAL. It is turned on by typing *trace_on().* at the CORAL prompt, and turned off by typing *trace_off().* (Section 12).)

For programs with non-ground terms (other than simple variables), the absence of occur-checks in the current implementation could cause a run-time error because of a failure to detect a cycle in unsuccessful unifications; this is true of most other logic programming systems as well. The following program, in file declba2.P, illustrates this (you may want to hit Ctrl-C to interrupt the execution, since otherwise the system will eventually run out of memory):

```
module declba_eg2.
export cycle(f).

b(X,X).
cycle(Y) :- b(Y,f(Y)).

end_module.
```

The fact $b(X, X)$ can be understood intuitively as " $b(x, x)$ is true for any value x"; non-ground terms are discussed further in Section 7. The rule for *cycle* causes both Y and $f(Y)$ to be unified with X, and we have $Y = f(Y)$, which is a contradiction unless we view this as a cyclic representation of an infinite term. CORAL's treatment of such terms is not consistent with the infinite term viewpoint. Dereferencing such a cyclic structure causes CORAL to get into an infinite loop, leading to a core dump. (Yes, this is not a very graceful way of handling an error!) Note that this problem does not arise if all (given as well as derived) facts are ground, or at least, do not contain arguments with nested variables.

4.4 A Note on Evaluation

CORAL provides several alternative evaluation methods. The default is an iterative bottom-up fixpoint evaluation following some source-to-source program transformations. The program transformations essentially propagate bindings during the evaluation of a rule; once some of the body literals are "solved" (i.e., the variables in these literals are bound to some terms), the resulting bindings are used to restrict the computation for subsequent literals. The order in which literals are solved is called the *sideways-information-passing* or *sip* order. The default sip-order is left-to-right in the body of a rule. [2] We note that sip-ordering affects only efficiency, and not the semantics of a program. However, some restrictions on programs containing negation and grouping (Sections 5 and 6) are presented in terms of the sip-order.

[2]Indeed, this is the only sip-order that can be specified in the current release of CORAL. More flexible specification of sip-orders will be supported in the next release.

5 Declarative Language Features: Negation

We now present the use of negation in CORAL through illustrative examples.

The keyword `not` is used as a prefix to indicate a negated body literal. For instance, given a predicate *parent*, we can check that a is not a parent of b by using `not` $parent(a, b)$. Such a literal can be used in a query, or in the body of a rule. The following program, in file declne1.P, defines p tuples to be $q1$ tuples that are not in $q2$.

```
module declne_eg1.
export p(f).

q1(1).
q1(2).
q2(1).
p(X) :- q1(X), not q2(X).

end_module.
```

5.1 Non-Floundering Programs

We require that programs satisfy the following *non-floundering* restriction:

If a variable appears in an argument of a negated literal in a rule body, it must be bound to a ground (i.e. variable-free) term `before` *the literal is evaluated. (Remember that the default sip-order of body literals is left-to-right.)*

If the non-floundering restriction is not satisfied, there is the possibility that a variable in the negated literal is bound to a non-ground term when the negated literal is evaluated, and is subsequently instantiated further. Consider the program in file declne2.P:

```
module declne_eg2a.
export notintoys(f).

% Illustrates dangers of floundering programs.

person(john).
person(susan).
employed(susan,marketing).
```

```
notintoys(X) :- person(X), not employed(X,Y), Y=toys.

end_module.
```

The logical reading of the program suggests that *notintoys*(*susan*) is true, since *susan* is not employed in the *toys* department. Nonetheless, the answer set is empty.

Many programs that do not meet the non-floundering restriction are nonetheless meaningful, as the following program, in file declne2.P, suggests. Unemployed people are defined as people who are not employed.

```
module declne_eg2b.
export unemployed(f).

% This program, while it does not meet the non-floundering restriction,
% nonetheless behaves reasonably.  The key point is that variables
% in the negated literal are not further instantiated later in the rule.

person(john).
person(susan).
employed(susan,marketing).

unemployed(X) :- person(X), not employed(X,Y).

end_module.
```

Note that the variable Y in the negated literal does not appear elsewhere in the rule; the rule is read as " x is unemployed if *person*(x) is true and there is no Y value y such that *employed*(x, y) is true".

In general, programs that do not meet the non-floundering restriction must be written with care; variables in a non-ground literal must not be further instantiated later in the rule execution.

5.2 Negation and Recursion

Of course, negation can be used in conjunction with recursive definitions. The following program is in file declne3.P:

```
module declne_eg3.
```

```
export t(ff).

e(1,6).
e(6,7).
e(7,6).
e(2,3).
e(2,4).
e(3,5).
e(4,6).

congested(4).

t(X,Y) :- t(X,Z), e(Z,Y), not congested(Z).
t(X,Y) :- e(X,Y).

end_module.
```

The relation t is defined as the transitive closure of the edges in relation e, with the restriction that paths through congested nodes cannot be extended. Notice that $t(2,6)$ and $t(2,7)$ are not computed; the only path from 2 to 6 or 7 is via 4, which is a congested node.

The following program, in file declne4.P, is an extension of the previous program:

```
module declne_eg4.
export t(ff).

e(1,6).
e(6,7).
e(7,6).
e(2,3).
e(2,4).
e(3,5).
e(4,6).

congested(4).
congested(Y) :- congested(X), e(X,Y).

t(X,Y) :- t(X,Z), e(Z,Y), not congested(Z).
t(X,Y) :- e(X,Y).
end_module.
```

The notion of congestion has been extended to mean that any node reachable from a congested node is also congested. Notice that $t(1,7)$ is no longer computed; the only path is via 6, which, now, is also classified as being congested. (A similar remark holds for $t(4,7)$.)

All of the examples that we have discussed so far are *stratified*, that is, if a predicate p is defined in terms of *not* q, then q is not defined in terms of p.

The next example illustrates non-stratified negation. Let us suppose that we have a complex mechanism constructed out of a number of components that may themselves be constructed from smaller components. Let the component-of relationship be expressed in the relation *part*. A component is known to be working either if it has been (successfully) tested or if it is constructed from smaller components, and all the smaller components are known to be working. This is expressed by the following program, in file declne5.P:

```
module declne_eg5.
export working(f).
%@pipelining. %  works with or without this annotation;
%  choice influences speed, and is dependent upon data

%  We can infer that a part is working if it is tested, or if
%  it contains no suspect subpart.  Recursively, a part is
%  defined to be suspect if it contains a subpart that is
%  not known to be working.

working(X) :- tested(X).
working(X) :- part(X,_), not has_suspect_part(X).
has_suspect_part(X) :- part(X,Y), not working(Y).

end_module.
```

part(a,b). part(a,c). part(b,d). part(b,e). part(c,f). part(c,g).

tested(d). tested(e). tested(g).

Note that the predicate *working* is defined negatively in terms of itself. However, the *part* relation is acyclic (i.e., it is a hierarchy), and hence the *working* status of a component is defined negatively in terms of subcomponents, but not negatively in terms of itself.

The above program illustrates the class of *left-to-right modularly stratified programs* ([Ros90, Bry89]). Intuitively, a modularly stratified program is such that in the answers and subgoals generated for the program, there are no cycles through negation.

In summary, CORAL supports the well-founded model semantics [VRS91] for programs with negation that are non-floundering and modularly stratified. For a more detailed discussion of modularly stratified programs, we refer the reader to the CORAL overview.

5.3 Negation in Pipelined Modules

CORAL provides a mode of evaluation called "pipelining" that closely mimics Prolog-style evaluation. This is discussed in Section 10. Negation in such modules also follows the Prolog negation-as-failure treatment. The reader is cautioned that this semantics does not always coincide with the modularly-stratified semantics. For instance, the program in file declne4.P is modularly stratified, indeed even stratified. However, pipelined evaluation of this program will not terminate.

However, it does coincide for programs in which there is no cycle, positive or negative, of goals. This is indeed the case for the program in declne5.P. In such cases, the user has a choice of two evaluation strategies. In the "working" program, the nature of the data determines which method is faster — if the hierarchy is "almost" a tree, pipelining is faster; if it is a dag with many paths between nodes on average, ordered search is faster. (The program in file decse7.P, discussed in Section 6, illustrates a similar trade-off.)

5.4 A Note on Efficiency

When a predicate p is defined in terms of *not* q and both p and q are defined in the same module, by default the evaluation is carried out using the Ordered Search algorithm. The overheads associated with this algorithm increase with the number of rules in the module. Thus, whenever q does not depend upon p, it is a good idea to define q in a separate module; more generally, it is a good idea to minimize the number of rules in a module that contains negation. (We have ignored this guideline in writing the small example programs for this section.) We will examine this issue more closely in Section 8 after introducing the module mechanism.

6 Declarative Language Features: Sets and Multisets

Sets and multisets are encountered in CORAL in two distinct forms. First, sets and multisets appear in CORAL is as special kinds of terms. Second, the collection of tuples with a given predicate name can be viewed as a set, and is commonly called a *relation* (or sometimes relation instance). If several copies of a given tuple are retained, such a collection is a multiset.

The first aspect of sets and multisets in CORAL is discussed below. The second aspect is discussed in Section 6.5.

6.1 An Important Restriction

The following important restriction in CORAL allows for efficient implementation: *A multiset term is restricted to be ground and to match only another (identical) ground multiset term or a variable. General matching or unification of sets (where one or both of the sets can have variables, respectively) is not supported.*

This forces the user to take a constructive approach to multiset manipulation, and CORAL provides a number of built-in operations to facilitate this. We will see the effects of this restriction often in the rest of this section.

6.2 Creating Multisets

The first point to note is that CORAL directly supports only multisets; set semantics is supported via a *set* operator. The program in file declse1.P illustrates two different ways of creating a multiset value:

```
module declse_eg1.
export phones(ff), employees2(ff),
       works(fff), employees(ff), budget(ff), sals(ff), numsals(ff).

works(joe,toy,20).
works(jim,toy,20).
works(jane,sales,30).
```

```
% the following rule illustrates enumeration

phones(Dept,{"2-5171","2-6172"}) :- employees1(Dept,{jim,joe}).

% The following (commented-out) rule is illegal! A variable cannot
% appear between curly braces.

% employees(Dept,{E}) :- works(E,Dept,Sal).

% To achieve the same effect, the following rule can be used.

employees(Dept,S) :- works(E,Dept,Sal), create_set(S), add_elem(S,E).


% employees2 illustrates grouping

employees2(Dept,<E>) :- works(E,Dept,Sal).

% budget illustrates multiset operator in head

budget(Dept,sum(<Sal>)) :- works(E,Dept,Sal).

% In budget, the salary 20 is counted twice since it is earned
% by two people.  The following rule computes the set of distinct
% salaries in each department.

sals(Dept,makeset(<Sal>)) :- works(E,Dept,Sal).

% illustrates use of multiset operators in body (count);
% a large suite of such operators is available in CORAL

numsals(Dept,N) :- sals(Dept,S), count(S,N).


end_module.
```

The simplest way to create a multiset is by *enumeration*. This is accomplished by listing the elements between { and }. The multiset {jim,joe} in the example illustrates this. Such a multiset-term can appear in the body or head of a rule, as the rule defining *phones* illustrates. It is important to note that the arguments in between { and } must be constants; even variables that will be bound to constants when execution reaches this literal are not allowed. The predicate *employees* illustrates this point. The commented-out rule is illegal because it violates the above rule. A natural reading of the rule would suggest that for each employee, an *employees* tuple is created with a singleton set in the second argument. This can be achieved in CORAL using the *create_set* and *add_elem* built-in predicates.

The second way to create a multiset is through the *grouping* operator $< ... >$. This operator can only appear in the head of a rule. There are some restrictions in the current version of CORAL: only one occurrence of grouping per rule; and if grouping is used to define a predicate, there must be no other rule defining the predicate. (The last restriction is rather conservative; only some programs with multiple rules defining a predicate using grouping cause problems.)

The predicate *employees*2 illustrates grouping. One tuple is created per department, and the second field contains a multiset of the people who work in that department. In contrast, the predicate *employees* contains one tuple per employee, with the first argument being a department name and the second argument being a singleton multiset with the employee's name.

The rule defining *budget* illustrates the utility of having multisets as opposed to sets. Since grouping generates a multiset, the salary 20 is included twice, and the budget of the toy department is computed to be 40. If grouping had been defined to compute a set, the second copy of 20 would have been discarded, and the budget would have been computed to be 20. (This is the semantics adopted in LDL.) In contrast, consider the rule defining *sals*. Here, we want to compute the set of distinct salaries; this is achieved by forcing grouping to create a set rather than a multiset. (CORAL does this on the fly without first creating a multiset.)

We will examine grouping in more detail in Section 6.6.

6.3 Multiset Operators

The rule defining *sals* in the previous example illustrated the use of the *makeset* operator in the head of a rule to modify the result of the grouping operator. Other operations that can be used similarly in the head include: *min*, *max*, *count*, *sum*, *product* and *avg*. For instance, if *makeset* were replaced by *sum*, we would obtain the sum of all salaries, as illustrated by the rule for *budget*. If it were replaced by *product*, we would obtain the product 20*20*30. If we wished to compute the number of distinct salaries, it seems natural to say $count(makeset(< Sal >))$ in the head; such nesting of aggregates is currently not supported in CORAL.

The rule defining *numsals* illustrates the use of a multiset operator (*count*, in this example) in the body of a rule. Several built-ins are provided for manipulating multiset values. These include multiset versions of union, intersection, difference, testing for subset, sum, min, max, product, average, etc. We refer the reader to the CORAL overview [RSSS93] for a full listing.

The user should keep in mind the following important point:
All the set/multiset operators in CORAL are non-destructive.

For example, $difference(S1, S2, S)$ binds S to a multiset in which the cardinality of an element is its cardinality in S1 minus its cardinality in S2, if it appears more often in S1, and zero otherwise. The multisets S1 and S2 are not changed, and continue to exist. This allows us to view all the multiset operations declaratively, and has some implementation advantages as well. On the other hand, this has a significant effect on the use of memory: once a set or multiset value is created, it is not destroyed until the end of the module execution.

6.3.1 The member Multiset Operator

We now examine an important multiset operator in some detail. Consider the program in declse2.P:

```
module declse_eg2.

export okteam(f), team(f), engineer(f), pilot(f), doctor(f).

engineer(joe).
engineer(jack).
pilot(amy).
pilot(jack).
doctor(jack).
doctor(paula).

team({jack}).
```

```
team({amy,joe}).
team({amy,joe,paula}).
team({amy,joe,paula,jack}).

% An okteam must contain a doctor, a pilot and an engineer.
% They need not be different people; however, the size of the
% team must be at most 3.

okteam(S) :- team(S), count(S,C), C <= 3, member(S,X), member(S,Y),
member(S,Z), engineer(X), pilot(Y), doctor(Z).

end_module.
```

This program illustrates the use of an important multiset operator, the *member* predicate. When called with the first argument bound to a multiset and the second argument a variable, as in this example, it succeeds repeatedly with the second argument bound in turn to the elements of the multiset. Thus, this program illustrates how we can take a multiset and express conditions involving its elements.

In LDL, *okteam* can be defined by the following rule:

okteam(S) :- team(X,Y,Z), engineer(X), pilot(Y), doctor(Z).

The absence of the *count* literal is a technical detail; the important difference is that the argument of *team* in the body is a set that is specified using a template containing variables. This is possible since LDL supports set-matching. In CORAL, as we noted earlier, this is not supported.

The *member* predicate is very powerful. It can also be used to step through the tuples in a relation. This is a useful capability when the relation to be scanned is only specified at run-time. Consider the program in file declse3.P:

```
module declse_eg3a.

export professional(f), scan(bf), scan2(bff), engineer(f), couple(ff).

engineer(joe).
engineer(jack).

couple(joe,jill).
```

```
professional(X) :- engineer(X).
professional(X) :- pilot(X).

temp(X) :- engineer(X).

% X can be bound to any unary exported predicate name: professional,
% engineer or pilot.  However, binding it to temp will not work.

scan(X,U) :- member(X,U).

% X can be bound to any binary exported predicate (e.g. couple)

scan2(X,U,V) :- member(X,U,V).

end_module.


module declse_eg3b.

export pilot(f).

pilot(amy).
pilot(jack).

end_module.
```

The predicate *member* is used to define both scan predicates. In scan, a binary version of *member* is used; in scan2, a ternary version is used. For reasons that have to do with the module structure, the first argument of *member* cannot be bound to a local predicate name; it *must* be bound to an exported or base predicate name (or to a multiset).

Finally, we note that there is a predicate called *call* supported in CORAL. It is similar to *member* but with some important differences. When a literal $call(T)$ is evaluated, T must be bound to a term of the form $p(A1, A2, ..., An)$, where p is the name of an n-ary predicate. It is important that p be a predicate name. A literal of the form $call(X(A1, A2, ..., An))$ will not be parsed by CORAL (and thus, it is irrelevant whether X will be bound to a predicate name at run-time). This restriction is due to the way CORAL deals with functor terms; it can be worked around using *univ*, as illustrated in the program in file pipe12.P (which is discussed later since it involves some features that have not yet been introduced). In general, *member* is more versatile since the first argument can be predicate name, a multiset, a relation, or a variable that is bound at run-time to one of these three kinds of arguments. However, the number of arguments of the predicate to be queried must be known in advance, unlike for *call*.

6.3.2 Converting Lists to Multisets

A list value can easily be converted into a multiset value, as the following program, in file declse32.P, illustrates:

```
module declse_eg33.

export list_to_set(bf).

% The first argument must be bound to a list; returns a multiset
% with the same elements.

list_to_set(L, <X>) :- element_of(L, X).

end_module.

module declse_eg33.
@pipelining.

export element_of(bf).

element_of([H|T], H).
element_of([H|T], X) :- element_of(T, X).

end_module.
```

Converting set values into lists is less straightforward; we discuss this in Section 10.

6.4 Multisets in Pipelined Modules

CORAL provides a mode of evaluation called "pipelining" that closely mimics Prolog-style evaluation. This is discussed in Section 9. While enumeration can be used to create multisets in a pipelined module, grouping is not currently supported. The built-in multiset operators can still be used in rule bodies.

The following program in declse4.P can be used to see what goes wrong when grouping is used with pipelining:

```
module declse_eg4a.
export sals1(ff), sals2(ff).
@pipelining.

works(joe,toy,20).
works(jim,toy,20).
works(jane,sales,30).

% grouping (< ... >) is not supported in pipelined modules.
% try these predicates to see what happens ...

sals1(Dept,<Sal>) :- works(E,Dept,Sal).
sals2(Dept,makeset(<Sal>)) :- works(E,Dept,Sal).


end_module.
```

Both *sals*1 and *sals*2 lead to errors when queried.

On the other hand, consider this program, also in declse4.P:

```
module declse_eg4b.
export friends(ff), known(ff), connected(ff).
@pipelining.

% these rules illustrate that sets can be manipulated effectively
% in conjunction with pipelining although grouping is not allowed.
```

```
friends(john, {joe,sue}).
friends(joe, {jill,jack}).
friends(jack, {carl}).

known(X,Z) :- friends(X,S), member(S,Z).
known(X,Z) :- friends(X,S1), member(S1,Y), friends(Y,S), member(S,Z).

% however, the following definition causes an infinite loop
% with pipelining!  try it without pipelining; it works fine.

connected(X,Z) :- friends(X,S), member(S,Z).
connected(X,Z) :- connected(X,Y), friends(Y,S), member(S,Z).

end_module.
```

The predicate *known* illustrates the use of built-in multiset operators in a pipelined module. A recursive variant of *known*, called *connected*, gets into an infinite loop since it is left-recursive; this is an inherent problem with pipelining, and is not related to the use of *member*. (The program runs fine if evaluated without pipelining; and Prolog, which uses an evaluation method similar to pipelining, will also go into an infinite loop.)

6.5 Relations as Multisets

The collection of tuples with a given predicate name is commonly called a *relation* (or sometimes *relation instance*). If several copies of a given tuple are retained, such a collection is a multiset. Although such duplicate values are not permitted in the traditional database definition of a relation, in most commercial systems, a relation-as-multiset notion is also supported. This is achieved by not checking for duplicate tuples and retaining one copy of each tuple per derivation. A similar notion of relation-as-multiset is supported in CORAL, and we now discuss this feature.

Consider the program in declse5.P:

```
module declse_eg5.
export pathcnt(bff,fff).

/* The multiset annotation is critical.  Turn it off and
```

```
see what happens.  The data is in declse5.F  */

@multiset.

/* This program computes the number of paths between two nodes.
It will work correctly as long as the edge relation is acyclic.  */

pathcnt(Source,Dest,count(<Dest>)) :- path(Source,Dest).

path(Source,Dest) :- edge(Source,Dest).
path(Source,Dest) :- edge(Source,Dest1), path(Dest1, Dest).

end_module.
```

For a restricted class of programs, [3] the *multiset* annotation ensures that the number of times a fact is generated is equal to the number of *derivation trees* for it. In this example, the number of derivation trees for a fact $path(a, b)$ is equal to the number of paths from a to b. Notice that two facets of multisets in CORAL are central to this program: (1) the relation *path* is a multiset (due to the multiset annotation), and (2) the grouping in the definition of pathcnt is a multiset operation.

A more sophisticated example is given in declse6.P:

```
module declse_eg6.
export partcnt(bff), partcnt(fff).

/*  Computes total number of occurrences of Subpart in Part.
The assembly relation must not contain cycles!
Input data in declse6.F  */

/*  The multiset annotation below is critical.
Turn it off and see what happens.  */

@multiset.

partcnt(Part,Subpart,sum(<Q>)) :- subparts(Part,Subpart,Q).
```

[3] Every head variable must appear in the body, and if a variable appears twice in a body literal, or inside a structured argument, it must also appear before that literal in the sip order.

```
subparts(Part,Subpart,Quantity) :- assembly(Part,Subpart,Quantity).
subparts(Part,Subpart,Quantity) :- assembly(Part,Subpart1,Quantity1),
subparts(Subpart1,Subpart,Quantity2),
Quantity = Quantity1 * Quantity2.

end_module.
```

The number of copies of a Subpart in a Part depends upon the number of paths from Part to Subpart (since each part indicates a different use of the Subpart) as well as the "quantity" labels of each edge (which indicate the number of copies of a Subpart in a given use).

Finally, we note that the multiset annotation cannot be used in conjunction with pipelining. Since pipelining does not materialize the generated tuples, the issue of duplicate checking does not arise.

6.6 Multiset Grouping and Stratification

The use of the multiset grouping operation in the head of the rule intuitively suggests that all relevant facts for the body literals must be computed before any head fact can be derived. For instance, consider the program in declne1.P. The predicate *budget* computes, for each department, the sum of all salaries paid to workers in the department. Clearly, all the tuples in *works* with a given department name must be known before the corresponding *budget* tuple is generated. We say that the *budget* fact "depends" upon each of these *works* facts. In this simple program, the relation *works* is independent of *budget*, and therefore there is a *stratification* with respect to the *depends* relationship.

This is very similar to the concept of stratification discussed with respect to negation in Section 5. Indeed, this similarity is not accidental; in both cases, the generation of a tuple requires us to be certain that we have already generated all tuples that could affect the contents of the tuple being generated. Thus, it is natural to expect that the Ordered Search technique used to evaluate programs with negation can also be used to evaluate programs with grouping. This is indeed how grouping is implemented in CORAL; consequently, the class of programs supported includes those that are modularly stratified with respect to a *depends* relationship induced by uses of negation or grouping. (It is possible that negation and grouping are both used in a single module.)

The following program illustrates a program that is modularly stratified with respect to grouping. It is in file declse7.P:

```
module declse_eg7a.
export bom(bf,ff).

/* This program computes the total cost of a composite part by summing
the total costs of its subparts.  This is often called the
bill-of-materials problem. Note that the program is not stratified.
It will work correctly as long as the assembly relation is acyclic
(i.e. the part-subpart relationship, closed transitively, is acyclic).
There are input data sets in declse7.F and declse72.F */

bom(Part,sum(<C>)) :- subpart_cost(Part,SubPart,C).

subpart_cost(Part,Part,Cost) :- basic_part(Part,Cost).
subpart_cost(Part,Subpart,Cost) :- assembly(Part,Subpart,Quantity),
bom(Subpart, TotalSubcost),
Cost = Quantity * TotalSubcost.

end_module.
```

This is a version of the well-known bill-of-materials problem. The cost of a part is defined as the sum of the costs of its components. If the part hierarchy is acyclic, the *depends* relationship is clearly acylic, and this is a modularly stratified program. The same file also contains a more efficient program that exploits the module mechanism in CORAL. We discuss this program in Section 8.

CORAL also allows the user to write several programs that are not even modularly stratified, but that have an intuitive semantics due to some monotonicity with respect to the use of aggregation. These are discussed in Section 7.

6.7 A Note on Efficiency

Grouping is implemented using Ordered Search. Thus, as for negation, if a module contains grouping it is a good idea to minimize the number of rules in the module. We will examine this issue more closely in Section 8 after introducing the module mechanism.

7 Declarative Language Features: Advanced

In this section, we discuss several advanced features supported in CORAL declarative modules.

These include head-updates, aggregate selections, prioritization and monotonicity. While non-ground terms are supported by default, and head-updates can be viewed as a different kind of rule, multiset relations, aggregate selections, [4] prioritization and monotonicity are examples of *annotations* in CORAL. We discuss these annotations here since they have a great impact on the semantics of a program. We will discuss annotations that primarily affect efficiency in Section 9.

7.1 Non-ground Facts

CORAL supports non-ground facts:

n >`equal(X,X).`

The above fact can be used to define equality, although this is unnecessary since "=" is available as a built-in predicate. The query `?equal(5, 6)` will fail, whereas each of the following queries will succeed: `?equal(5, 5)`, `?equal(30.2, 30.2)`, `?equal("Madison", "Madison")`.

The *@return_unify* annotation will improve performance on programs that generate non-ground facts. (It is not the default, and must be explicitly requested.)

The meaning of a non-ground fact is that it denotes an infinite collection of facts obtained by replacing each variable by a ground term. That is, the variables are universally quantified. For example, equal(X,X) should be read as follows: for all values x of X, equal(x,x) is true.

Another well-known example of a program that uses non-ground data structures is a program that appends two lists in constant time. In file declad1.P, we have:

```
module declad_eg1.
export dappend(bbf).

% appends the two given lists; lists must be in difference-list form.
```

[4] Aggregate selections in CORAL are general enough to subsume the *choice* operator described in [RSS92b].

```
% sample query:  ?dappend(dlist([1|[2|[3|X]]],X),dlist([4|[5|Y]],Y),Z).

dappend(dlist(X,Y), dlist(Y,V), dlist(X,V)).

end_module.
```

As the sample input illustrates, the lists are in *difference-list* form. A difference list is essentially a list represented as the difference of two lists, where the second list is just a variable. For example, the list [1, 2, 3] would be represented as dlist([1 —[2 —[3 —X]]],X). Thus, the representation consists of a non-ground term. The append program just unifies the second list with the variable in the first list; this is similar to switching a tail pointer in Lisp, but has a logical semantics.

For a more detailed example using non-ground terms, we have the following parsing program in declad2.P:

```
module declad_eg2.
export sentence(fbb).

/* This program can be made more efficient by using either
@pipelining.
or
@return_unify.
*/

/*  This is the translation of a DCG that handles the
syntax and meaning of a small subset of natural language,
taken from Bratko's Prolog book, pg 455.
The last two arguments of ``sentence'' represent a sentence
in difference list form, and the first argument is instantiated
to a parse tree for this sentence.

Sample queries:  ?sentence(S,[a,man,paints],[]).
 ?sentence(S,[every,man,that,paints,admires,monet],[]).
*/

sentence(S,List1,Rest) :-
noun_phrase(X,Assn,S,List1,List2),
verb_phrase(X,Assn,List2,Rest).
```

```
noun_phrase(X,Assn,S,List1,Rest) :-
determiner(X,Prop12,Assn,S,List1,List2),
noun(X,Prop1,List2,List3),
rel_clause(X,Prop1,Prop12,List3,Rest).
noun_phrase(X,Assn,Assn,List1,Rest) :-
proper_noun(X,List1,Rest).

verb_phrase(X,Assn,List1,Rest) :-
trans_verb(X,Y,Assn1,List1,List2),
noun_phrase(Y,Assn1,Assn,List2,Rest).
verb_phrase(X,Assn,List1,Rest) :-
intrans_verb(X,Assn,List1,Rest).

rel_clause(X,Prop1,and(Prop1,Prop2),[that|List1],Rest) :-
verb_phrase(X,Prop2,List1,Rest).
rel_clause(X,Prop1,Prop1,List1,List1).

determiner(X,Prop,Assn,all(X,implies(Prop,Assn)),[every|List1],List1).
determiner(X,Prop,Assn,exists(X,and(Prop,Assn)),[a|List1],List1).

noun(X,man(X),[man|List1],List1).
noun(X,woman(X),[woman|List1],List1).

proper_noun(john,[john|List1],List1).
proper_noun(annie,[annie|List1],List1).
proper_noun(monet,[monet|List1],List1).

trans_verb(X,Y,likes(X,Y),[likes|List1],List1).
trans_verb(X,Y,admires(X,Y),[admires|List1],List1).

intrans_verb(X,paints(X),[paints|List1],List1).

end_module.
```

We refer the reader to Bratko's book [Bra90], from where this example was taken, for a detailed discussion. When parsing, it is common to identify a structure (e.g. noun-phrase) before all its "slots" (e.g. the noun-phrase matches the identifier john in some input sentence) are filled in. In such situations, it is useful to generate non-ground facts that describe the deduced structure with the unknown slots containing variables.

7.2 Head Deletes

CORAL provides insert, delete and update operations, but in general their use is restricted to the command line or extended C++ code. The reason is that we view declarative modules as the equivalent of a query language, analogous to the SELECT statements of SQL, and the intermixing of inserts/updates/deletes with query language constructs tends to destroy the non-operational flavor of the query language. We note that any CORAL command, including insert/delete/update, can be invoked from a rule in a declarative module; however, no guarantee is offered as to the semantics, which depends upon when and how often these commands get executed. The exception to this rule occurs with pipelined execution, where the order of evaluation is fixed.

However, we allow a limited use of updates in materialized declarative modules, primarily as a means to increase efficiency. When a rule is used to generate a fact, other facts (of base predicates or predicates defined in the current module) can be deleted as a side-effect.

As a simple example, consider the program in declad3.P:

```
module declad_eg3.
export twopower(ff).

@no_rewriting.

twopower(X+1,C*2), del twopower(X,C) :- twopower(X,C), X < 20 .
twopower(0,1).

end_module.
```

The program computes powers of 2. Once we compute 2**5, for example, the fact corresponding to 2**4 is no longer needed; the delete command in the rule head discards it. The semantics of head delete is operational: when a fact is inferred, the fact specified for deletion is discarded as a side-effect. (More precisely, the deletes are done at the end of the iteration, when the "Seminaive" updates are carried out.)

Another example is the following program to sum the elements of a list, also in file declad3.P:

```
module declad_eg4.
export sum(bf).

% This program sums the elements in a list.

list_sum(L,L,0).
list_sum(L,L1,N1+X), del list_sum(L,[X|L1],N1)
            :- list_sum(L, [X|L1], N1).

sum(L,N) :- list_sum(L,_,N).

end_module.
```

In this program, a program transformation is used to propagate the query binding (i.e. the list to be summed). Such program transformations generate "adorned" versions of predicates, and the question of how to adorn the delete literal arises. If the delete literal also appears in the body, as in this example, the delete literal is given the same adornment as the body literal. If this default is not desired, or if the delete is specified on a literal that does not also appear in the body, the adorned form must be specified explicitly. (As we saw in the previous example, the issue of adornments does not arise if no rewriting is done; since the default is to rewrite the program, the @no_rewriting annotation must be specified.)

The following program, in file declad4.P, illustrates that the use of head deletes can lead to very operational programs:

```
module declad_eg4a.
export sort1(f).
@no_rewriting.

% In each iteration, the old sort1 fact is deleted and a new one,
% containing exactly one new X-value from p (the least cost tuple),
% is generated.  Clearly, no additional rewriting is to be performed.
% Further, if either of these head deletes is omitted, the set of
% answers changes.  This is a very operational program, and is
```

```
% perhaps best viewed as a production-rule program.

sort1([]).
sort1([X|[C|L]]), del sort1(L), del choose(C), del p(X,C) :-
sort1(L), choose(C), p(X,C).

% The fixpoint evaluation uses the scc-by-scc optimization:
% predicates are divided into sccs, and are evaluated in
% scc-by-scc order.
% This has the advantage that at each stage, only a small number of
% rules is examined, and further, a predicate from a lower scc can be
% treated as a base predicate, i.e. assumed to be fully evaluated,
% in higher sccs --- an optimization that significantly reduces
% the number of ``semi-naive'' rules.  In this program, the following
% (commented out) rule is evaluated just once, prior to application
% of the sort1 rules.  However, for the program to work as intended,
% this rule must be re-evaluated after each change to p.  This effect
% is achieved by placing the rule in a separate module.  Again, the
% operational nature of this program is underscored.

% choose(min(<C>)) :- p(X,C).

end_module.

module declad_eg4a2.
export choose(f).

choose(min(<C>)) :- p(X,C).

end_module.
```

One could argue that the program is nonetheless intuitive and compact. However, the program is probably best understood in operational terms, rather than in terms of (operational!) modifications to the least model semantics.

We note that head deletions are not supported in pipelined modules since it is not clear how head deletions should affect the order of backtracking in pipelined evaluation. (Program transformations and pipelined evaluation are discussed in Section 9, in the context of annotations that control evaluation.)

The use of head deletes is sound, in the following sense, for programs that do not use negation or set-grouping: Any fact computed by a program using the update operation will also be computed by the version of the program with the update removed. Although the semantics of using this operation is in general non-deterministic, it is deterministic in many cases. A more detailed discussion of head updates is presented in the overview paper [RSSS93].

7.3 Prioritization

Sometimes, it is useful to be able to control the order in which the tuples in a relation get used in making derivations. The *prioritize* annotation gives a user such control. The following sorting program, in declad4.P, illustrates this:

```
module declad_eg4b.
export sort2(f).
@no_rewriting.

% In each iteration, the old sort2 fact is deleted and a new one,
% containing exactly one new X-value from p (the least cost tuple),
% is generated.  Clearly, no additional rewriting is to be performed.
% Further, if either head delete is omitted, the set of answers changes.
% This is a very % operational program, and is perhaps best viewed
% as a production-rule program.  The prioritize annotation ensures
% that scc-by-scc evaluation is overridden (i.e. the rules are
% evaluated together), and that in each iteration,
% the q relation contains a single tuple, with the order in which the
% tuples are seen governed by the min(C) condition.

sort2([]).
sort2([X|[C|L]]), del sort2(L), del q(X,C) :- sort2(L), q(X,C).

% The effect of prioritize is to add exactly one least cost q tuple
%to the delta in each iteration. (Additional q facts are made
% available only when no facts can be derived without such an addition.)

@prioritize q(X,C) min(C).

% The following rule is needed since only derived predicates
```

```
% can be prioritized.

q(X,C) :- p(X,C).

end_module.
```

As another example, consider the following program, also in declad4.P:

```
module declad_eg4c.
export sort3(b).

%  This program takes a binary predicate name as input, and prints
% the tuples in it, sorted by the second column. Again, there
% is an operational flavor.

sort3(P) :- q(P,X,C), printf(X,C).

@prioritize q(P,X,C) min(C).

q(P,X,C) :- member(P,X,C).

end_module.
```

As these programs suggest, the prioritize annotation often leads to operational reasoning and must be used sparingly and with caution. In the next section, on aggregate selections, we will see another program that illustrates the use of prioritization, in a way that affects only efficiency, not semantics.

7.4 Aggregate Selections

In many situations, it suffices to retain just one of several computed facts. The *@aggregate_selection* annotation is designed to facilitate this.

The following program in declad5.P illustrates aggregate selections:

```
module declad_eg5a.
export spanning_tree(bfff,ffff).
```

```
/*  Selects (in a non-deterministic manner) spanning trees.
    spanning_tree(root, start, end, cost)
    is true if (start, end, cost) is an edge in a selected spanning
    tree rooted at root.   Can also be queried with root free.
    A variant of this program is discussed by Ganguly et al. in
    a PODS92 paper.

    Assumes a base relation edge(start, end, cost);
    (sample set in declad5.F)
 */

% The program below essentially does transitive closure to "reach" all
% nodes. For each node reached via more than one incoming arc, only
% one of these arcs is retained, thanks to the aggregate selection.

spanning_tree(R,nil,R,0) :- is_source(R).
spanning_tree(R,X,Y,C) :- spanning_tree(R,Z,X,C1), edge(X,Y,C).

% The following definition of is_source ensures that the "tree" does not
% contain cycles involving the source node.  It requires special entries
% in the edge relation for potential source nodes.

is_source(R) :- edge(nil, R, _).

@aggregate_selection spanning_tree (R,X,Y,C) (R,Y) any(X,C).

% the above aggregate selection is the same as:
% @choice spanning_tree (R,X,Y,C) (R,Y).

end_module.
```

The aggregate selection states that in the relation for spanning_tree, if two tuples have the same R,Y values, we can retain either and discard the other. This implies that, for a given root R, a spanning tree is constructed by choosing an X,C pair for each Y; that is, by choosing exactly one incoming edge for each node. (We note that the *choice* annotation described in [RSSS93, RSS92b] is just a special case of aggregate selections using *any*, and it has been eliminated in favor of *any*.) A minimum cost spanning tree can be constructed by ensuring that least-cost edges are chosen to extend the spanning tree, subject to the requirement that the "tree" property be preserved. This program, also in declad5.P, is essentially Prim's algorithm:

```
module declad_eg5b.
export min_spanning_tree2(bfff,ffff).

/*  Selects (in a non-deterministic manner) minimum cost spanning trees.
    min_spanning_tree2(root, start, end, cost)
    is true if (start, end, cost) is an edge in a selected spanning
    tree rooted at root.   Can also be queried with root free.

    At each step, the set of edges of the form (from,to) with 'from' in
    the current spanning tree is examined.  Of these, the edges with the
    least cost are chosen and added to the tree, subject to an
    aggregate selection on min_spanning_tree that ensures that each
    node has only one immediate predecessor in a tree.
    In considering candidate edges for addition to the tree,
    previously chosen edges are ignored in order to allow
    more expensive edges to be considered; the @monotonic
    annotation ensures that the current set of 'chosen'
    tuples is used in the negated literal.  (The default is
    to completely evaluate 'chosen'; this would be
    inappropriate.)  This definition of 'chosen' ensures
    that the current least cost edges are added to the tree at
    each step.  Contrast this with the min_spanning_tree2 program,
    which is WRONG.

    Assumes a base relation edge(start, end, cost)

*/

min_spanning_tree(R,nil,R,0) :- is_source(R).
min_spanning_tree(R,X,Y,C):-
min_spanning_tree(R,Z,X,C1), chosen(R,X,Y,C).

@aggregate_selection min_spanning_tree (R,X,Y,C) (R,Y) any(X,C).

is_source(R) :- edge(nil, R, _).
```

```
chosen(R,X,Y,C) :- candidate(R,X,Y,C), not chosen(R,X,Y,_).
candidate(R,X,Y,C) :- min_spanning_tree(R,_,X,_), edge(X,Y,C).

@monotonic.
@aggregate_selection chosen (R,X,Y,C) (R,X,Y) min(C).

end_module.
```

At each stage, tuples that are "connected" to the current spanning tree are considered as candidates for extending the tree. The least-cost candidates are chosen for possible addition to the tree; if their addition preserves the tree property (i.e. that each node except the root contain exactly one immediate predecessor), they are actually added. Contrast this with the (incorrect) *min_spanning_tree2* program, which is seemingly a simpler but equivalent version.

Another program that illustrates the use of aggregate selections is in file declad6.P:

```
module declad_eg6a.
export mincost(bbf), shortest_path(bbff).

/*  This program illustrates the use of aggregation.
The predicate mincost computes the cost of the shortest path between
two points; shortest_path computes shortest paths between two points.
Note that this program works even on cyclic graphs as long as there
are no negative cycles.  The reason is that the min operation in the
aggregate selection retains only the current shortest paths.

Incidentally, note that making the definition of cost left-linear
would speed the program up.

The input for this program is an edge relation; sample inputs are
provided in files declad6.cyc.F and declad6.acyc.F.  */

cost(X,Y,C) :- edge (X,Y,C).
cost(X,Y,C) :- edge(X,Z,C1), cost(Z,Y,C2), C=C1+C2.

@aggregate_selection cost[bbf] (X,Y,C) (X,Y) min(C).
```

```
% The grouping in the following rule ensures that mincost contains,
% for each X,Y pair, the actual least cost of a connecting path (and
% not just the cost of a currently known least cost path).

mincost(X,Y,min(<C>)) :- cost(X,Y,C).

shortest_path(X,Y,C,[X,Y]) :- mincost(X,Y,C), edge(X,Y,C).
shortest_path(X,Y,C,[X|P]) :- mincost(X,Y,C), edge(X,Z,C1),
shortest_path(Z,Y,C2,P), C=C1+C2.

% The following annotation indicates that for a given cost, we only
% want one path between a given pair of points.

@aggregate_selection shortest_path[bbff] (X,Y,C,P) (X,Y,C) any(P).

end_module.
```

This program computes shortest paths in a graph by first computing the cost of the shortest path and then selecting a path with this least cost, for each pair of nodes. The point to note in this program is the use of grouping in the definition of mincost; this ensures that in each mincost fact, the value in the third argument is indeed the cost of the shortest path between the nodes that correspond to the first two arguments. (In contrast, the aggregate selection on cost merely influences duplicate elimination. It is possible that intermediate cost facts — which are used in deriving other cost facts — have values in the third argument that don't correspond to the shortest path between the nodes in the first two arguments.)

The following program, also in declad6.P, is more elegant, and also illustrates the use of prioritization:

```
module declad_eg6b.
export spath(bbff).

/*  Illustrates monotonic programs.  Although the 3rd argument value
may not be the least cost between the nodes in the first two arguments
for intermediate spath facts, this is indeed true of all spath facts
when the computation terminates.  (Contrast this with the previous
program.)
```

```
Note that this program works even on cyclic graphs as long as there
are no negative cycles.  The reason is that the min operation  in the
aggregate selection retains only the current shortest paths.

The input for this program is an edge relation; sample inputs are
provided in files declad6.cyc.F and declad6.acyc.F.  */

spath(X,Y,C,[X,Y]) :-  edge(X,Y,C).
spath(X,Y,C,[X|P]) :-  edge(X,Z,C1), spath(Z,Y,C2,P), C=C1+C2.

@aggregate_selection spath(X,Y,C,P) (X,Y) min(C).
@aggregate_selection spath(X,Y,C,P) (X,Y,C) any(P).

% The following annotation ensures that the (current) shortest path
% is expanded at each step. (This may actually slow down the
% computation if the overhead involved is not offset by the
% gains for a given data set.)

@prioritize spath(X,Y,C,P) min(C).

end_module.
```

Paths are generated while retaining only the shortest known path between a pair of nodes. Again, intermediate spath facts may not have the shortest path cost in the third argument, but when the computation terminates, the third argument of each spath fact does contain the shortest path between the nodes that correspond to the first two arguments. In addition, for any pair of nodes and cost, a single path is retained, and spath facts are prioritized by cost. This results in O(ElogV) evaluation, closely resembling Dijkstra's algorithm. In practice, the prioritization has a significant overhead, and on a given data set, the prioritization annotation may actually slow the program down (sometimes considerably!).

7.4.1 Aggregate Selections and Single Answer Queries

Sometimes, it is sufficient to compute one answer to a query. Aggregate selections are very useful in dealing with such queries. CORAL implements *intelligent backtracking*, and the use of aggregate selections is taken advantage of to avoid further computation after an answer has been found. The following program, in file declad7.P, illustrates the use of aggregate selections for single-answer queries. To understand what happens, execute the query with trace on (type *trace_on*.).

```
module declad_eg7.

export p(bf).

% Try ?p(1,X); the computation stops after the first
% iteration over the recursive rule.  Execute with trace on to
% see what happens; try it with and without the annotation.

@aggregate_selection p(X,Y) (X) any(Y).

p(X,Y) :- b(X,Y).
p(X,Z) :- b(X,Y), p(Y,Z).

end_module.

b(1,2).
b(1,3).
b(1,4).
b(2,5).
b(5,6).
b(6,7).
```

While this is a very useful technique, it has limitations. In essence, the aggregate selection is a simple modification to the duplicate checking. It is not sophisticated enough to ensure that an answer is definitely generated if it exists, as the following example, in file declad72.P, illustrates:

```
module declad_eg72.

export p(bf).

% Even if only one answer is desired, the following aggregate selection
% could lead to some answers not being computed.  For example, with
% ?p(1,X), the facts p(2,4) and p(4,5) are needed to compute p(1,5),
% which is the only answer.  However, CORAL will (non-deterministically)
% retain only one of the facts p(2,3) and p(2,4); if it retains $p(2,3)$,
```

```
% as it does in the current implementation, the answer is not generated.

%@aggregate_selection p(X,Y) (X) any(Y).

p(X,Y) :- b(X,Y).
p(X,W) :- c(X,U), p(U,V), p(V,W).

end_module.

c(1,2).
b(2,3).
b(2,4).
b(4,5).
```

A technique for getting around this problem is presented in Section 10.

7.5 Monotonic Programs

Monotonic programs are a class of programs with aggregation and grouping that have an intuitive semantics, even though they are not even modularly stratified. They have been examined in [RS92, Van92].

We have already seen a monotonic program in the second version of the program for shortest path (in declad6.P). Since the aggregation operator was *min*, we were able to write the program using an aggregate selection. Our next example, in declad8.P, illustrates a monotonic program in which the aggregate operator is *count*; it cannot be expressed using aggregate selections. Instead, we must use the annotation @*monotonic*.

```
module declad_eg8.
export coming(f).

/*  This program, due to Ross and Sagiv, illustrates monotonic programs.
Some rather rude invitees inform the host that they will come only
if they are guaranteed that at least k other guests that they know
are going to come.  The irritated host can use the following program
to determine those guests to whom such a guarantee can be extended.

The ``knows'' relation need not be acyclic; try this program with
```

```
different instances for this relation.  A sample input is in declad8.F */

@monotonic.

coming(X) :-  requires(X,0).
coming(X) :- requires(X,K), kc(X,N), N >= K.

% The monotonic annotation influences the following rule: intermediate
% kc facts could contain underestimates in the second column.  (An
% attempt to obtain an exact count --- which is done by default, in the
% absence of the monotonic annotation --- would lead to an infinite
% loop if knows contains cycles.)

kc(X, count(<Y>)) :- knows(X,Y), coming(Y).


end_module.
```

Although the default semantics for a rule with grouping would force complete evaluation of *kc* for each X value, the monotonic annotation overrides this. Intermediate *kc* facts with underestimates in the second argument — based upon the set of currently known coming facts — can be used to generate new coming facts.

Another well-known monotonic program is the company control program, in declad9.P:

```
module declad_eg9.
export controls(ff).

/* This is the well-known company control program.  owns(X,Y,N)
means that company X owns a fraction N of the shares of Y.
A company X controls company Y if the sum of the shares in Y owned
by X and companies controlled by X is greater than half the shares of Y.

This is a monotonic program.  A sample data file is declad9.F  */

@monotonic.

% controlsvia(X,Y,Z,N) means that X controls fraction N of the shares
```

```
% of Z via intermediary Y.

controlsvia(X,X,Y,N) :- owns(X,Y,N).
controlsvia(X,Y,Z,N) :- controls(X,Y), owns(Y,Z,N).

% controlmin(X,Z,S) means that X controls (at least) fraction S
% of the shares of Z

controlmin(X,Z, sum(<M>)) :- controlsvia(X,Y,Z,M).

controls(X,Y) :- controlmin(X,Y,N), N > 0.5.

end_module.
```

Here, the aggregate operator is sum, and again, it cannot be written using aggregate selections. In running the program on the sample input file, note that since our semantics is two-valued, $controls(ge, hp)$ — for example — is not derived and hence is false.

8 Modules in CORAL

In this section, we discuss the module facility in CORAL. Modules provide a way, as the name suggests, to modularize code. In addition, modules serve as the unit of compilation in CORAL, and provide the basis for incremental program development and testing. Modules also provide a clean way to mix and match different execution alternatives.

8.1 A Note on Module Names

The choice of module names is not significant in the current implementation. The file mod1.P contains

```
module start_eg1.
export grandchild(bf).

grandchild(Z,X) :- parent(X,Y), parent(Y,Z).

end_module.
```

The name of this module is start_eg1, and so is the name of the module in file start1.P. However, if you type

n >`consult(mod1.P).`

the module is correctly compiled and results in the addition of a definition for the predicate grandchild, even if you have already consulted start1.P. To verify this, you can use the `list_rels` command. (This command lists the names of all exported predicates and base predicates.) Note, however, that the *explanation* facility uses module names in creating names for dump files. (Type $help(explain)$. for details about the explanation facility.)

However, note that two modules cannot export the same query form (although they can export different query forms for the same predicate). If you consult two modules that export the same query form, only the second definition is retained; a warning message to this effect is printed.

8.2 Inter-Module Calls

If p is exported by Module M1 and used in Module M2, p is treated essentially as a base (EDB) predicate in M1. That is, p is simply treated as an explicitly enumerated collection of facts. When M2 is executed and needs p facts, the definition of p exported by M1 is used to create (the required subset of) this collection of facts.

To understand what happens, suppose that p is indeed a base predicate, and appears in a rule of M2. Evaluation within a rule proceeds left-to-right [5] and can be thought of as a nested-loops join. (While this is not entirely accurate with respect to pipelined evaluation, for example, it is an accurate enough description for our purposes.) When evaluation reaches the p literal, a scan is opened on p. A p tuple retrieved by the scan is used to instantiate the rule. When evaluation returns to the p literal on backtracking, [6] the scan on p is advanced to *get the next* p tuple.

This "get-next" interface to a base relation p via a scan is essentially the interface presented to M2 by M1 — M1 is the module in which p is defined, to return to our example — regardless of the nature of M1. We emphasize that the user need not be concerned about the details of how "get-next" requests are generated etc. This is just an abstraction of how the evaluation proceeds, and is presented here for clarity of exposition.

An important consequence of this interface is that p is fully evaluated as evaluation repeatedly reaches the p literal upon backtracking (in some rule of M2). (More precisely, the part of p that is relevant to this p literal is fully evaluated.) Thus, the following rule governs inter-module calls:

Inter-Module Calls: The calling module will wait until the called module returns answers to the subquery. The called module presents a scan-like interface, and returns all answers to the subquery upon repeated "get-next" requests.

This is independent of the evaluation modes of the two modules involved. The point at which the called module returns answers, however, depends on its evaluation mode.

If the called module is pipelined, an answer is returned as soon as it is found, and the computation of the called module is suspended until another answer is requested by the caller. The use of certain features, "save modules" (see below), "head updates" and "aggregate selections" (see Section 7), can result in all answers being computed before any answers are returned by the called module. Otherwise, answers are returned at the end of each fixpoint iteration in the called module; further iterations are carried out if more answers are requested by the calling module. At the level of the top-most query,

[5] More generally, in sip-order.

[6] Backtracking is only intra-rule unless evaluation in M2 is pipelined.

this results in answers being available at the end of each iteration.

8.3 Negation, Grouping and Module Structure

CORAL provides several alternative ways to deal with programs containing negation. (Everything that we say here holds for multiset-grouping as well.)

First, CORAL provides an evaluation mechanism called Ordered Search [RSS92a] that evaluates programs with left-to-right modularly stratified negation. By default, a program with negation or grouping is evaluated using this method. (See Sections 5 and 6.)

Second, a module can be evaluated using pipelining. This can be done by adding the annotation `@pipelining` to the module. In this case, evaluation proceeds very much like Prolog, and the semantics for negation is negation-as-failure!

Third, the inter-module call mechanism can be used to write programs with negation or grouping. Recall that the called module presents a scan-like interface and computes all answers to the subquery as needed. This can be used to achieve the effect of *stratified* program evaluation. (Indeed, it can be used to correctly evaluate some modularly stratified programs.)

The file mod2.P in directory *doc/examples* illustrates these options.

```
/*  We provide a number of definitions of the even function.  */

module even0.
export even0(b).

/* A definition with non-stratified negation.  It is implemented using
Ordered Search.  */

even0(0).
even0(X) :- X > 1, Y = X-1, not even0(Y).

end_module.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
module even1.
export even1(b).
@pipelining.

/*  The evaluation method is pipelining, and the semantics for negation
is negation as failure.  In this example, the modular stratified
semantics (which holds in non-pipelined modules) and negation as
failure happen to coincide.  */

even1(0).
even1(X) :- X > 1, not even1(X-1).

end_module.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


module even2.
export even2(b).

/*  The same function defined without negation.  Of course,
not all definitions with negation can be replaced by a
negation-free definition.  */

even2(0).
even2(Y) :- Y>0, X = Y-2, even2(X).

end_module.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


module even22.
export even22(b).
@ordered_search.

/*  Identical to even2, except for the use of Ordered Search.
The slow-down relative to even2 indicates the overhead of
doing Ordered Search.  */
```

```
even22(0).
even22(Y) :- Y>0, X = Y-2, even22(X).

end_module.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

module even3.
export even3(b).

/*  This module, in conjunction with Module even33, indicates how the
modularly stratified semantics can be realized for
some programs without using Ordered Search.  The idea is to set up
a chain of inter-module calls, each of which is fully evaluated.
Of course, a cycle of calls would result in infinite looping.
In contrast, the Ordered Search mechanism deals with cycles, except
cycles involving a negated call, in which case the modularly stratified
semantics is itself undefined.  (In this program, it so happens
that every chain of calls includes negated calls.  If it is possible that
there is some cycle of positive calls, but no cycle of negative calls,
this ``call based'' approach to negation would result in an infinite
loop; Ordered Search will still compute the well-founded model, since
the program is  modularly stratified.)

This program is also less efficient than even0 (which uses Ordered
Search) due to the overhead of inter-module calls.  */

even3(0).
even3(X) :- X>1, Y = X-1, not even33(Y).

end_module.

module even33.
export even33(b).

even33(X) :- even3(X).

end_module.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

An important point to note is that Ordered Search is less efficient than ordinary fixpoint evaluation, as seen in the difference between $even2$ and $even22$; it is also slower than pipelined evaluation ($even1$). However, using inter-module calls extensively is even more expensive, as illustrated in the difference between $even0$ and $even3$!

These observations suggest two guidelines:

- If a program is non-stratified — i.e. a predicate is defined negatively in terms of itself — pipelined evaluation is very efficient, but provides negation-as-failure semantics. If the well-founded model semantics is desired, Ordered Search is the appropriate evaluation method, and inter-module calls should not be used.
- Ordered Search is relatively expensive, and modules that use it should be kept as small as possible by moving unrelated predicates into other modules.

The program in mod3.P illustrates the high cost of Ordered Search when the number of rules is large:

```
module mod_eg3.
export t(ff).

/*  This program is stratified, but the negated literal ``congested'' is
also defined in this module, and so CORAL uses Ordered Search.
The number of rules in this modules is unnecessarily large, and
this slows down execution a lot; compare it to the program in mod4.P.
(Note that each e fact and each congested fact is effectively a rule,
since these facts are listed inside the module!)
The answers for this program should be identical to the answers for the
program in declne4.P; the extra facts in congested are irrelevant.  */



e(1,6).
e(6,7).
e(2,3).
e(2,4).
e(3,5).
e(4,6).
```

```
congested(10).
congested(11).
congested(12).
congested(13).
congested(15).
congested(16).
congested(17).
congested(18).
congested(19).
congested(20).
congested(21).
congested(22).
congested(23).
congested(25).
congested(26).
congested(27).
congested(28).
congested(29).
congested(30).

congested(4).
congested(Y) :- congested(X), e(X,Y).

t(X,Y) :- t(X,Z), e(Z,Y), not congested(Z).
t(X,Y) :- e(X,Y).

end_module.
```

The program in mod4.P is identical to the one in mod3.P — Ordered Search is still used — except that the size of the module with negation is greatly reduced:

```
module mod_eg4.
export t(ff).

/*  This program is stratified, but the negated literal congested1 is
defined in this module.  CORAL uses Ordered Search in module mod_eg4.
This program improves upon the one in mod3.P by reducing the
size of the module in which Ordered Search is used.
```

```
The answers for this program should be identical to the answers for the
program in declne4.P; the extra facts in congested are irrelevant.  */


congested1(X) :- congested(X).

t(X,Y) :- t(X,Z), e(Z,Y), not congested1(Z).
t(X,Y) :- e(X,Y).

end_module.

module mod_eg42.
export congested(f), e(ff).


e(1,6).
e(6,7).
e(2,3).
e(2,4).
e(3,5).
e(4,6).

congested(10).
congested(11).
congested(12).
congested(13).
congested(15).
congested(16).
congested(17).
congested(18).
congested(19).
congested(20).
congested(21).
congested(22).
congested(23).
congested(25).
congested(26).
congested(27).
congested(28).
congested(29).
```

```
congested(30).

congested(4).
congested(Y) :- congested(X), e(X,Y).

end_module.
```

There is a special case to be considered, namely the class of *stratified programs*. If the program is such that q does not depend upon p whenever p depends. negatively upon q, the program is said to be stratified. In this case, the definition of q should be placed in a module that is different from the module in which p is defined. (If q in turn depends negatively upon r, the definition of r should be placed in a third module, and so on.) If this is done, none of the modules needs to use Ordered Search — and in fact CORAL detects this situation and avoids using OS — and there is a significant increase in efficiency. (The effect of inter-module calls is minimal in this special case.)

The program in mod3.P is stratified; the program in mod5.P exploits this.

```
module mod_eg5.
export t(ff).

/*  This program is stratified.  Since the only negated literal is
``congested'', and it is defined in a separate module, Ordered Search
is not used.  The answers for this program should be identical to
the answers for the program in declne4.P; the extra facts in
congested are irrelevant.  */

e(1,6).
e(6,7).
e(2,3).
e(2,4).
e(3,5).
e(4,6).

t(X,Y) :- t(X,Z), e(Z,Y), not congested(Z).
t(X,Y) :- e(X,Y).

end_module.
```

```
module mod_eg52.
export congested(f).


congested(10).
congested(11).
congested(12).
congested(13).
congested(15).
congested(16).
congested(17).
congested(18).
congested(19).
congested(20).
congested(21).
congested(22).
congested(23).
congested(25).
congested(26).
congested(27).
congested(28).
congested(29).
congested(30).

congested(4).
congested(Y) :- congested(X), e(X,Y).

end_module.
```

Ordered Search is no longer used, and this program is even faster than mod4.P. Clearly, stratified programs should be organized into modules so as to avoid the need for Ordered Search.

The guidelines offered here for programs with negation hold equally for programs with grouping, since Ordered Search is again the evaluation method used by default. Since grouping is not supported in pipelined modules, Ordered Search is the only applicable evaluation method if grouping is used in conjunction with recursion. In such programs, it is important to keep the number of rules in a module with Ordered Search small. On the other hand, for programs in which the use of grouping is stratified, the use of Ordered Search can be avoided by using the module structure judiciously. This point is illustrated by the programs in files mod6.P and mod7.P. The program in mod6.P requires the use of Ordered Search:

```
module mod_eg6.
export t(f).

/*  This program illustrates that the guidelines for programs
with negation also hold for programs with grouping, since both
are implemented using Ordered Search.

Compare this program with the one in mod7.P  */


congested(10).
congested(11).
congested(12).
congested(13).
congested(15).
congested(16).
congested(17).
congested(18).
congested(19).
congested(20).
congested(21).
congested(22).
congested(23).
congested(25).
congested(26).
congested(27).
congested(28).
congested(29).
congested(30).


e(1,6).
e(6,7).
e(2,3).
e(2,4).
e(3,5).
e(4,6).
```

```
congested(Y) :- congested(X), e(X,Y).

t(<X>) :- congested(X).

end_module.
```

The program in mod7.P does not require the use of Ordered Search, and executes much faster:

```
module mod_eg7.
export t2(f).

/*  This program is a variant of the one in mod6.P; the
only difference is that the definition of ``congested''
has been moved to a separate module. Note the increase in speed. */


t2(<X>) :- congested(X).

end_module.



module mod_eg72.
export congested(f).

congested(10).
congested(11).
congested(12).
congested(13).
congested(15).
congested(16).
congested(17).
congested(18).
congested(19).
congested(20).
congested(21).
congested(22).
```

```
congested(23).
congested(25).
congested(26).
congested(27).
congested(28).
congested(29).
congested(30).

e(1,6).
e(6,7).
e(2,3).
e(2,4).
e(3,5).
e(4,6).

congested(Y) :- congested(X), e(X,Y).


end_module.
```

The program in mod8.P is similar to the one in mod7.P; the only difference is that *congested* is now a base predicate. From the perspective of the calling module (in which *t*2 is defined), this distinction is irrelevant.

Finally, we note that in some modularly stratified programs, the use of Ordered Search can be avoided by using inter-module calls. At the cost of some (often subtle) operational reasoning, this gives us a gain in efficiency. In Section 6, we considered the bill-of-materials problem. In addition to the program discussed in that section, file declse7.P contains the following program:

```
module declse_eg7b.
export bomb(bf,ff).

/*  The following program is equivalent to the previous one.
However, it relies upon the inter-module calling mechanism rather
than Ordered Search to ensure that all subpart costs are determined
before the bom rule is applied.  Thus, some operational reasoning
```

```
is required to understand why this works as expected.

It is faster than the previous program on some data sets, for
example the one in declse7.F, and slower on others, for
example the one in declse72.F.  The use of pipelining
in the module defining subpart cost is an optimization.  While
it speeds this program up, it does not affect the basic nature
of the trade-off illustrated by these two data sets.  */

bomb(Part,sum(<C>)) :- subpart_cost(Part,SubPart,C).

end_module.

module subpart.
export subpart_cost(bff,fff).
@pipelining. % This is an optimization.

% The following annotation (commented out) should NOT be used!
% This module is involved in a cycle of inter-module calls.
% @save_module.

subpart_cost(Part,Part,Cost) :- basic_part(Part,Cost).
subpart_cost(Part,Subpart,Cost) :- assembly(Part,Subpart,Quantity),
bomb(Subpart, TotalSubcost),
Cost = Quantity * TotalSubcost.

end_module.
```

The cost of a part is defined as the sum of the costs of its components. If the part hierarchy is acyclic, the *depends* relationship is clearly acylic, and this is a modularly stratified program. In the above program however, the cost of a part is computed by first generating calls that compute the cost of each of its subparts and then adding these costs. The acyclic part-subpart relationship ensures that there is no cycle of inter-module calls.

It is faster than the previous program on some data sets, for example the one in declse7.F. In general, it is faster if the data set is "almost a tree". However, it is slower if the input is a dag with many paths into each node, on average. For example, it is slower than the previous program on the data set in declse72.F. The reason is that each call of the form $bomb(bike, X)$ is re-computed; recall that all facts computed in a call are discarded at the end of the call. (See Section 8.5 for more discussion of this example.)

8.4 Head Deletes, Prioritization and Module Structure

Head deletes and prioritization are discussed in Section 7. These features are not supported in pipelined modules. When these features are used, an important optimization of fixpoint evaluation called *SCC-by-SCC* evaluation is not applicable. (This is also true when Ordered Search is used, and is one of the reasons for the relative inefficiency of OS.)

Again, the rule of thumb is to reduce the size of a module that contains these features.

8.5 The Save Module Annotation

As discussed above, when a module is called, a scan-like interface is presented to the caller. Answers are returned in response to "get-next" requests until there are no more answers. At this point, by default, all facts generated in the computation of the called module are discarded, and the call is complete.

This default strategy of discarding facts is not always appropriate. For example, one consequence of moving the definition of the predicate *congested* to a separate module is that queries on this predicate generated in the evaluation of the rule for t are re-computed from scratch. Thus, our guidelines for reducing the size of a module that is evaluated with Ordered Search (or indeed the desire to keep modules small to increase clarity and locality in the program) could lead to much repeated computation due to the default strategy of discarding facts at the end of module evaluation.

The save module feature addresses this problem. By specifying @*save_module* in a module, the default is overridden, and all computed facts are saved when a call on this module terminates. Subsequent calls re-use saved facts, and avoid repeating previously made inferences.

Consider the program in mod9.P:

```
module mod_eg9.
export t(ff).

/* The only difference between this program and the one in mod4.P is
in the use of the save_module annotation, in the next module.  */
```

```
congested1(X) :- congested(X).

t(X,Y) :- t(X,Z), e(Z,Y), not congested1(Z).
t(X,Y) :- e(X,Y).

end_module.

module mod_eg92.
export congested(f), e(ff).

/*  The save module annotation below is the only difference between
this program and the one in mod4.P.  This ensures that congested is
not evaluated repeatedly each time that it is called.  */

@save_module.

e(1,6).
e(6,7).
e(2,3).
e(2,4).
e(3,5).
e(4,6).

congested(10).
congested(11).
congested(12).
congested(13).
congested(15).
congested(16).
congested(17).
congested(18).
congested(19).
congested(20).
congested(21).
congested(22).
congested(23).
congested(25).
congested(26).
```

```
congested(27).
congested(28).
congested(29).
congested(30).

congested(4).
congested(Y) :- congested(X), e(X,Y).

end_module.
```

This program is identical to the one in mod4.P except for the use of the save module feature. Note that it is a little faster. (It would be much faster if the definition of *congested* were more complicated and took a significant amount of time to re-compute.)

This annotation, however, cannot be used if the module is evaluated using pipelining.

Further, there is the following restriction on the use of the save_module feature: *If a module uses the save_module feature, it should not be involved in a cycle of inter-module calls.*

Consider the *bomb* program in declse7.P again. It does a lot of repeated computation over different calls to a module when the data set is a dag. It seems natural to try and fix this problem by adding a @save_module annotation. This will not work due to the restriction that none of the modules in a cycle of inter-module calls can have the @save_module annotation.

9 Declarative Language Features: Annotations and Control

CORAL provides several execution alternatives, and default choices are made by the system so that users are not forced to make explicit choices. The default settings for the CORAL execution environment can be viewed using $display\_defaults()$, and can be modified by users from the command interface using $set()$, $clear()$ and $assign()$.

Annotations provide the same flexibility on a per-module basis. A user can specify annotations in a CORAL module in order to provide hints, or directives, for efficient execution. An annotation specifies a default environment setting for the module in which it occurs.

We have discussed several annotations that can be used to alter the *semantics* of a program in other sections. Here we present annotations whose primary use is to increase efficiency. (This distinction is often blurred; some annotations intended to alter the semantics also have an impact on efficiency, and some annotations intended to improve efficiency could alter the semantics of a program, unless they are used with care. In particular, it can be argued that *prioritize* and *head deletes* lead to operational programs and should be viewed as control annotations. Nonetheless, we have chosen to classify them as annotations intended to achieve a different semantics, and present them in Section 7.)

To measure the effect of annotations on the execution, it is necessary to time programs. The built-in predicate *cputime(X)* returns the current time. (Timing commands are also discussed in Section 12.)

9.1 Program Transformation

Several program transformations are supported in CORAL. These include Magic Templates [Ram88], Supplementary Magic Templates [BR87], Supplementary Magic with Indexing [RS91] and Factoring [NRSU89, KRS90].

9.1.1 Basic Rewriting Techniques

Supplementary Magic Templates is chosen as the default rewriting technique. However, the choice of program transformations can be controlled completely by the user via annotations.

Consider the well-known "same generation" program in file declac1.P:

```
module declac_eg1.
export sg(bf,ff).

/*  This is the well known same-generation program.
It is a good program on which to see the effect of various
program transformations.  The default is supplementary magic.
You can see the rewritten program by consulting this file (declac1.P)
and then examining the file declac1.P.M.

Type help(annotations). to find out how to choose other program
transformations through the use of annotations.  For example,
adding the annotation "@ magic." would result in the use of magic
rather than supplementary magic.
There are two data sets in declac1a.F and declac1b.F */

sg(X,Y) :- parent(X,XParent), sg(XParent,YParent), parent(Y,YParent).
sg(X,X) :- parent(X,Y).
sg(X,X) :- parent(Y,X).

end_module.
```

Consult this file. CORAL optimizes this program by doing source-to-source transformations and choosing execution defaults. Since the program does not specify any annotations, the defaults are chosen. See the file declac1.P.M (which is created when file declac1.P is consulted) to see the result of this optimization. In addition, the consult stores an in-memory description of this optimized program. This internal description is used to drive the CORAL interpreter when a query is executed.

Notice that there are *two* modules in declac1.P.M, even though declac1.P contains only one module. If a module in the user's program exports several predicates or several adorned forms for a predicate, CORAL creates one module per exported query form internally. In case you wish to skip over the rewriting phase, you can directly consult the .P.M file.

The basic program transformation is one of the following: Magic, Supplementary Magic, Supplementary Magic with Indexing [7]

[7]Incidentally, the "Indexing" in Supplementary Magic with Indexing does not refer to the creation

8

Factoring is a special transformation that is not always applicable; for instance, it is not applicable on this program. If @factoring is specified when it is not applicable, CORAL reverts to the default (which is supplementary magic unless the user overrides this from the command line by using the *set* command).

The user is invited to run this program with @magic, @no_rewriting, @sup_magic_indexing, and @factoring to see what happens. Time queries using the two datasets (in declac1a.F and declac1b.F), and examine the declac1.P.M file.

(If you don't want the answers printed out while timing, use a query like this: $?sg(X, Y), fail.$)

9.1.2 Passing Bindings in Conjunction with No Rewriting

The annotation @no_rewriting ensures that none of the program transformations are applied, and that the fixpoint of the original program is computed directly. This is clearly the appropriate strategy when the entire exported relation is to be computed, and it is sometimes the best strategy even when the query specifies a selection. For example, if the query involves all subparts of a given part, and the given part is near the root of the parts hierarchy, the effort of "marking" the "relevant" subparts may be greater than the reduction in computation made possible by ignoring subparts that are not relevant.

Sometimes, however, it might be desirable to use bindings available in the query, although not in the way that any of CORAL's rewriting algorithms would do so. It is possible to achieve this through a useful trick, which we illustrate in the program in file declac12.P:

```
module declac_eg1a.

%  Illustrates use of magic fact in conjunction with no_rewriting

export anc(bf).
```

of indexes in the usual sense. This transformation is a variation of Supplementary Magic in which each goal ("magic" fact) is given a distinct integer "id", and this id, also added to "supplementary" facts, is used as a special index. It is useful when programs involve complex data structures or Factoring.

[8]The version of Factoring used in CORAL is the one described in [KRS90].

```
@no_rewriting.   % No program transformation is done...

                % ...so this rule must be added to define anc_bf
anc_bf(X,Y) :- anc(X,Y).

anc(X,Y) :- m_anc_bf(X),  % The fact m_anc_bf(5) is automatically
    parent(X,Y).   % added when computing answers to
% ?anc(5,X); this rule takes advantage
% of this fact.

anc(X,Y) :- anc(X,Z), parent(Z,Y).

end_module.
```

There are three points to note here. First, although no rewriting is done, a "magic" fact corresponding to the query is always added at the beginning of materialized evaluation of a module. Thus, a user who understands this and wishes to access the bindings in the query can place a "magic" literal accordingly. Second, without rewriting, the only defined query form is $anc[ff]$, and there will be an error if $anc[bf]$ is exported. To circumvent this, the user must explicitly add a rule defining $anc_b f$; the set of tuples computed for this predicate is returned by CORAL as the answer set. (Note that $anc[bf]$ is not acceptable notation here; CORAL expects to see the notation $anc_b f$.) Third, each (external) call on *anc* sets up an execution of this module; the module is thus solved one (external) goal at a time.

9.1.3 Factoring

The program in file declac2.P contains a transitive closure program that can be used to see the effects of factoring:

```
module declac_eg2.

/*  This program computes transitive closure.  Although existential query
optimization is not applicable for the query form anc(bf), it is
applicable for the query form anc1(f).  Look at the module defining
anc1(f) in file declac2.P.M after consulting this file (declac2.P)
to see the result of existential query optimization.
```

```
Factoring is applicable for both query forms.  Again, you can see what
effect this has by examining the .P.M file.  (The @factoring annotation
is commented out below; you must include it for factoring to be
effected.) Using the data sets in declac1a.F and declac1b.F, time the
query ?anc(1,X) and ?anc1(X) and see the effect of the optimizations.  */

export anc(bf), anc1(f).

% @factoring+.

anc1(X) :- anc(X,Y). %  Y can be replaced by _ ("don't care" variable)

anc(X,Y) :- parent(X,Y).
anc(X,Z) :- parent(X,Y), anc(Y,Z).

end_module.
```

This program can also be used to make one other point. In CORAL, rules are evaluated from left-to-right (the default "sips"), and an argument in a literal is considered "bound" if it contains a constant or a variable that appears to the left of this literal in the rule. The rules defining a predicate are specialized for each such "adorned form" of the predicate.

Consider the program in declac2.P, with the export statement modified to just export *anc[ff]*. Since we want the entire anc relation, the best strategy is to simply evaluate the original rules. However, the adorned form *anc[bf]* is generated from the recursive rule, and additional rules are generated. While the heuristic of aggressively propagating bindings is often justified, there is a cost associated with propagating bindings — additional "magic" predicates are introduced, and added to rules — and this cost is sometimes not justified.

CORAL therefore provides the user with a way to control the set of adorned forms for which specialized rules are generated. The user can add an annotation of the form *@allowed_adornments predicate_name[adornment].*

(If more than one adornment is to be allowed for a predicate, several such annotations can be listed in a single annotation statement.) This refines the rewriting phase as follows. When a new adorned form is generated it is first checked against the list of allowed adornments (which, by default, includes all adornments). If it is not in the list, an adornment with fewer bound arguments is chosen from the list instead; a warning is issued if the list does not contain an adornment with fewer bound arguments. In our example program, the user could add, for instance, @*allowed_adornments* *anc*[*ff*]. When *anc*[*bf*] is generated, it is replaced by *anc*[*ff*]. Thus, no rules defining *anc*[*bf*] are added, as is clear from the .P.M file. If *anc*[*fb*] were the only allowed adornment, it would lead to an error. (It is worth running the *anc*[*ff*] query with and without the @*allowed_adornments* *anc*[*ff*] annotation to see the difference in speed.)

9.1.4 Negation and Grouping

Finally, we note that the rewriting algorithm treates programs with negation and grouping specially. The following simple program in declac3.P illustrates this:

```
module declac_eg3.
export p(f), q(f).

% This program is intended to illustrate the rewriting algorithms
% in the presence of negation and grouping.  Consult this file
% and look at the .P.M file.

b1(5).
b1(6).
b2(6).

p(X) :- b1(X), not b2(X).

q(<X>) :- b1(X), b2(X).

end_module.
```

The changes in the rewriting step provide guidance for the run-time system.

9.2 Controlling the Mode of Execution

CORAL provides a number of annotations that affect the run-time execution strategy. These are often, but not always, orthogonal to the rewriting phase.

CORAL's suite of run-time strategies can be broadly classified into bottom-up fixpoint evaluation strategies or (top-down) pipelining; we discuss pipelining in Section 10. There are three different fixpoint strategies: Basic seminaive evaluation (the default), Predicate-wise Seminaive (a refinement of Basic Seminaive; specified using @*psn*), and Ordered Search, which is used for programs with negation or multiset grouping (in conjunction with a modified rewriting phase). Ordered Search is discussed in Sections 5 and 6.

Basic Seminaive evaluation is essentially a repeated application of rules until no new facts are generated. After each iteration (in which all rules are tried), the generated facts are checked against previously generated facts to see if any new facts have been generated; if not, execution terminates. There are two refinements:

- If the original rules were iterated upon, derivations would be repeated in subsequent iterations. To avoid this, a "seminaive rewriting" is applied (to the result of the source-to-source rewritings discussed in the previous sections). The set of seminaive-rewritten rules is then iterated upon. Essentially, new predicates are introduced to classify tuples as "generated in previous iterations" and "generated in current iteration", and these are used to avoid repeated inferences. The result of this rewriting step is not visible in .P.M files; it is only reflected in the internal CORAL data structures that represent the program.
- Rather than include all rules in a single iteration, the program obtained after the source-to-source rewriting (not semi-naive rewriting) is analysed, and predicates are grouped into maximal mutually recursive sets, or strongly-connected-components (scc's). The sccs are evaluated one at a time, beginning with those sccs at the "leaves" of the scc structure. Within each SCC, fixpoint evaluation is used.

Predicate-wise Seminaive is a variant in which tuples generated within an iteration are made available for use before the end of the iteration [RSS90]. This results in a smaller number of iterations, each of which produces more facts (relative to corresponding iterations in BSN), in the presence of sccs with multiple recursive predicates.

9.2.1 Intelligent Backtracking

CORAL also implements intra-rule *intelligent backtracking* (IB) in conjunction with both pipelined [9] and materialized execution. To explain IB, we note that at run-time evaluation in CORAL can be thought of as a *nested-loops index join*. While rewriting may transform the original rules, this characterization is true of fixpoint evaluation (on the transformed rules) as well as pipelining (always on the original rules, since this annotation turns off

[9] Pipelined execution is discussed in Section 10.

all rewriting). At each point in the evaluation of a rule, we either try to get the *first* matching tuple for the next literal (i.e., a tuple that matches the current bindings for rule variables), or to get the *next* matching tuple. If either of these operations fails, we must *backtrack* in the rule since the current rule bindings cannot be extended further. Usually, backtracking just gets another tuple for the current literal, or if there are no more tuples for the current literal, backs up by one literal. Intelligent backtracking uses some simple analysis to determine that backtracking can "back up" further, thus avoiding some repeated (and demonstrably fruitless) computation.

Try out the program in declac4.P to get a feel for pipelining and intelligent backtracking, as well as the rewriting transformations:

```
module decalc_eg4.
export joinans1(), joinans2(), joinans3(), joinans4().

/* This is a collection of simple joins.  It can be used to get
some idea of CORAL performance relative to systems like LDL or
Quintus Prolog on non-recursive Datalog programs.  While comparing
with Prolog, use @check_subsumption-, since Prolog does not do
any subsumption checks.

Try each of the following annotations to see how they perform.
@no_rewriting and @pipelining should behave similarly on this program.
@magic will be a little worse since it creates filters (magic relations)
that provide no restriction.  @sup_magic will be similar to magic
on joinans1 and joinans2, but will be faster on joinans3.
If you examine the file declac4.P.M after compiling this program, you can
understand why.  @sup_magic essentially creates intermediate
relations corresponding to execution points in the body of a rule;
in joinans3, for example, this means that there is a unary relation with
argument X after the first two joins.  Thus, the rest of the rule is
evaluated once per distinct X-value, unlike the other approaches, in
which the rest of the rule is evaluated once per (X,Y,Z)-value.  On
joinans1, however, supplementary magic behaves very like magic due
to an implementation heuristic --- the creation of intermediate
relations stops when there are no more derived predicates in the
rule body.  Essentially, the intuition is that it is likely to be
easier to recompute a join of base predicates than to cache it.
On this program, with large parent relations, this is a poor
heuristic.  The program joinans3 indicates how to get around this
```

```
heuristic:  If you want the supplementary optimization, judiciously
introduce derived predicates that are copies of base relations.  */

%@sup_magic.   %  The default.
%@sup_magic_indexing.
%@magic.
%@pipelining.
%@no_rewriting.

%@non_ground_facts -. % This enables the use of intelligent backtracking.
% In CORAL, IB is implemented in conjunction with
% rewriting/fixpoint evaluation as well as pipelining.

% joinans1 involves joins on the first column; Quintus Prolog (and LDL)
% automatically index on the first argument.

joinans1 :- parent(X,Y), parent(X,Z), parent(X,W), fail.

% joinans2 involves a join on the first column of parent, but
% introduces an intermediate relation.

joinans2 :- parent1(X,Y), parent1(X,Z), parent1(X,W), fail.

parent1(X,Y) :- parent(X,Y).

% If intelligent backtracking is used, on joinans1 and joinans2 CORAL
% would detect that the rule could never succeed when it first
% got to the fail literal, and would not do any further computation;
% thus it is very fast.  Int. backtracking does not help in joinans3.
% This program also illustrates the fact that sup_magic can be
% worse than the other methods.  Since the intermediate relation
% after the second body literal is now a relation of (X,Y,Z)-values,
% the only effect of the supplementary rewriting is to cache a
% lot of intermediate relations.

joinans3 :- parent1(X,Y), parent1(X,Z), parent1(X,W), fail4(W,X,Y,Z).

% joinans4 joins on the second column, just for some variety.
% Note that the necessary indices are automatically created. Use
% the list_relations command before and after executing the query
```

```
% ?joinans4 to see what indices are added while evaluating this query.

joinans4 :- parent(Y,X), parent(Z,X), parent(W,X), fail.

end_module.
```

9.2.2 Indexing in CORAL

CORAL automatically generates several indexes on base predicates as well as predicates defined via rules. The user can specify additional indexes, although CORAL usually does a good enough job that the user does not have to do so. (Of course, CORAL ensures that the same index is not created twice.)

Indexes can be specified on a single field or on a combination of fields. Several indices can be specified on a single relation. All indices are automatically maintained under inserts and deletes (for base predicates) and as facts are computed (for predicates defined by rules).

Indexes can be specified on both in-memory and persistent predicates; index specification has the same syntax. Persistent indexes are B+ tree indexes, using the Exodus implementation. In-memory indexes are primarily hash-based. In addition, *pattern-form* indexes can be created for in-memory relations.

The following program in declac5.P illustrates some of these indexing features:

```
module declac_eg5a.

export anc(ff).

@allowed_adornments anc[ff].  % keeps the rewritten program simple; the
      % adorned form anc_bf is not optimized.

/*  This program computes transitive closure.  We use it here to
illustrate the indexing facilities in CORAL.  Actually, the index
declarations in this program are not needed, since CORAL automatically
generates them!  The goal here is to illustrate the syntax.  Inputs
in declac1a.F and declac1b.F */

@make_index anc[ff] (bf).  % create an index on the first argument
```

```
   % of the derived predicate anc_ff
   % (automatically maintained as tuples are added.)

@make_index anc[ff] (X,Y) (X). % means the same thing.  even if both
% these annotations are specified,
% only one copy of the index is created.

@index_deltas +.   % states that the set of new anc tuples (delta) is to
% be indexed.  (this is the default.)
% run ?anc(X,Y) on the data in declac1a.F with
% @index_deltas+ and @index_deltas- and see the difference!

anc(X,Y) :- parent(X,Y).
anc(X,Z) :- anc(X,Y), anc(Y,Z).

end_module.
```

As we noted earlier, the set of tuples in a predicate is actually classified into two sets: those generated in the current iteration and those generated earlier. The set generated in the current iteration is called the *delta* subset. A natural question is whether the delta relations should be indexed or not. By default, CORAL indexes the delta relations. However, if the number of tuples generated in any one iteration is small, it is better not to index the delta relations. The following program, also in declac5.P, provides such an example, and also illustrates pattern-form indices:

```
module declac_eg5b.
export append(bbf).

/*  This is the familiar program for appending two lists.  By moving
the structure in the bound argument into the body, we avoid creating
some structures at run-time.  However, append is really a program
best suited for Prolog-style evaluation.  In CORAL, this is
approximated by using pipelining.

We use this program to illustrate some aspects of indexing in CORAL.
A sample input set is in declac5.F  */

@index_deltas -. % in each iteration, just one tuple is generated.
% clearly, indexing deltas is not a good idea.
```

```
@check_subsumption -.   % each tuple (including the magic tuples) is
% generated precisely once; the overhead of
% subsumption checks can be avoided.

% The following annotation is not needed; this index is automatically
% created by CORAL.  we have included it for two reasons.  (1) It shows
% the syntax for a PATTERN-FORM index.  (2) It underscores the point
% that CORAL generates indexes on the rewritten program.  (In fact,
% index creation is done by analysing the program after seminaive
% rewriting!)

% The annotation specifies an index in which each tuple is required to
% meet the pattern ([X|Y],Z) in the bound arguments; an index on (Y,Z)
% pairs is created.

@make_index m_append [bbf] ([X|Y],Z) (Y,Z).

append([], X, X).
append(U, Z, [X|W]) :- U=[X|Y], append(Y, Z, W).

end_module.
```

Type *list_rels.* after consulting declac5.P to see the indexes created on anc. Consult declac1a.F, which contains a sample parent relation, you execute the query ?anc(X,Y) (preferably, ?anc(X,Y),fail. unless you want to see all answers!) and then type list_rels. You will see that indexes have been created on parent.

9.2.3 Duplicate Checks

In Section 6, we discussed the @multiset annotation when we considered relations that are multisets of tuples. The multiset semantics is ensured by performing duplicate checks on the "magic" predicates but not the original program predicates.

CORAL also allows duplicate checks (more generally, subsumption checks) to be turned off in the entire module, or on arbitrary predicates, using the *@check_subsumption* annotation. This allows a user to selectively avoid the cost of subsumption checks, but there could be a negative effect if many facts are derived repeatedly. As an extreme case, not checking magic predicates for duplicate derivations could lead to non-termination even if the set of distinct facts is finite (because of cyclic derivations).

The @check_subsumption annotation is illustrated in the second program in declac5.P. For more on the syntax of annotations for duplicate checks, we refer the reader to the overview document [RSSS93].

9.2.4 Lazy Evaluation

By default, any answers generated at the end of an iteration are returned, and CORAL waits to be prompted for more answers before continuing with further iterations. A similar interface exists for inter-module calls. This can be overridden, forcing CORAL to evaluate all answers before returning any, by specifying "@lazy_evaluation -."

CORAL automatically turns off lazy evaluation in some situations, e.g., modules that contain prioritization, head deletes, and the @*monotonic* annotation.

10 Declarative Language Features: Pipelined Evaluation

The program in file declac4.P, in addition to further illustrating Magic, Supplementary Magic and Supplementary Magic with Indexing, introduces a new execution annotation — *@pipelining.* CORAL's default strategy of rewriting with some variant of Magic and then evaluating the fixpoint can be be characterized in database terminology as *materialization*, wherein all intermediate tuples are saved, or materialized. The overhead of materialization is balanced against the potential for avoiding recomputation, and, perhaps more importantly, completeness of the evaluation strategy. Prolog's evaluation strategy, in contrast, does not materialize intermediate tuples. Rather, bindings generated by solving a goal are passed along in a "pipelined" fashion to restrict subsequent computation. The @pipelining annotation instructs CORAL to use a pipelined, non-materialized evaluation technique. (No rewriting is used in conjunction with pipelining.) There are many differences with respect to a WAM-based Prolog engine, since our goal was to implement pipelining in the most convenient possible way within the CORAL implementation framework, rather than to implement pipelining as efficiently as possible. However, the resulting execution bears a close resemblance to Prolog execution.

CORAL supports the infamous, but very useful, *cut* (!) operator in pipelined modules. CORAL also implements intra-rule *intelligent backtracking* (IB) in conjunction with both pipelined and materialized execution (see Section 9). In this, it differs from most Prolog systems.

10.1 Negation and Grouping in Pipelined Modules

Multiset grouping is not supported in pipelined modules.

Negation is supported with negation-as-failure semantics as in Prolog. For a discussion, see Sections 5 and 6.

10.2 Pipelined Evaluation and Embedded Commands

Pipelined evaluation gives the user a guarantee about the order in which rules will be executed, unlike materialized evaluation, in which no operational guarantees are available. This is very useful when CORAL *commands*, for example inserts and deletes of tuples, are embedded in rules. CORAL does not support Prolog features such as *assert* and *retract*, but the use of commands such as *insert* and *delete* is quite similar (but with fewer guarantees as to ordering of facts).

```
module pipe1.
@pipelining.

export raise1(b), raise2(b).

% Give everyone in the named department a 10 percent raise.
% To understand this program, it is important to note that builtins
% (such as insert/delete here) do not succeed repeatedly on backtracking.

% dept(Dname, Ename, Sal) is a base relation to be updated.
% Note that there is no guarantee as to the order of tuples in dept,
% or the location of inserted tuples.  Thus, raise1 may give some
% employee two raises!

raise1(Dept) :-  dept(Dept, Emp, Sal), Newsal = 1.1*Sal,
delete(dept(Dept, Emp, Sal)),
insert(dept(Dept, Emp, Newsal)).

% This ensures that each employee receives exactly one raise.

raise2(Dept) :-  find_new_sals(Dept), add(dept, dept2).

find_new_sals(Dept) :-  dept(Dept, Emp, Sal), Newsal = 1.1*Sal,
delete(dept(Dept, Emp, Sal)),
insert(dept2(Dept, Emp, Newsal)).

add(dept, dept2) :- dept2(Dept, Emp, Sal),
delete(dept2(Dept, Emp, Sal)),
insert(dept(Dept, Emp, Sal)).

end_module.


% sample data

dept(toys, sue, 30).
```

```
dept(toys, john, 20).
dept(bolts, henry, 30).
dept(nuts, sarah, 40).
```

There are two main points to note in this program. First, the fact that pipelined evaluation proceeds left-to-right within each rule, and considers rules in the order listed, allows the user to place insert/delete commands in appropriate locations. This is important since these operations have side-effects.

Second, unlike, say, Prolog, no guarantee is made as to the ordering of facts in a base relation, or the location at which a tuple is inserted. Thus, in $raise1$, it is possible that a *dept* tuple representing a raise is inserted at a point ahead of the open scan on *dept* (due to the first body literal). This raises the possibility that an employee receives more than one raise! To circumvent this, in $raise2$ the tuples representing raises are inserted into a temporary relation and later added to the main relation.

10.2.1 Indexing and the Order of Insertions

In the above example, it can be argued that $raise1$ is simpler, and will work correctly if the inserted tuples are added "behind" all currently open scans (so that these scans do not see the newly inserted tuples). (Such behaviour would be similar to Prolog's $asserta$.) *This is guaranteed in CORAL only if all open scans on the relation use an "all-free" (all arguments free) index.* This is one situation in which CORAL's defaults can be intrusive — CORAL tries to identify useful indexes and automatically creates them. (And of course, the "best" available index is always used.) In particular, a relation with subsumption checking on always has an "all-bound" index. Further, if a literal is such that some variable in it appears earlier in the rule, an index will be generated. For example, if X appears in a rule ahead of literal $emp(X, Y)$, then an index will be created on the first argument of *emp*.

10.2.2 Converting Multisets to Lists

As mentioned in Section 6, converting a multiset value into a list value is not as straightforward as converting a list into a multiset. We illustrate this in the following program, in file pipe12.P:

```
module pipe_eg12.
```

```
@pipelining.

export multiset_to_list(bf).

% The first argument must be bound to a multiset; returns a list
% with the same elements.

multiset_to_list(S, L) :-
        insert_mset(S),
convert(temp, [], L), !.

convert(R, Lin, Lout) :-
                % The CORAL parser doesnt accept variables in
% places where it expects a constant;
% hence this kludge
univ(T, [R,X]),
call(T), % T= R(X); X is bound by this call
delete(T),
convert(R, [X|Lin], Lout).

convert(_, L, L).

end_module.

module insert_mset.

export insert_mset(b).

insert_mset(S) :-
clear(check_subsumption),
member(S, X), insert(temp(X)).

end_module.
```

This program also illustrates another point. The CORAL parser will currently not accept a term with a variable in place of a functor. So, in situations where the functor will be known at run time but not at compile time, the user must program around this restriction using *univ*, as illustrated here.

10.3 The Cut Operator

The *cut* operator (!) prunes backtracking during pipelined evaluation. We refer the reader to any Prolog textbook for a detailed discussion of cut, and simply present the basic idea through the following example, in file pipe2.P:

```
module pipe_eg2.
@pipelining.

export tested1(f), tested2(f), tested3(f).

%  This is a simple example of a generate-and-test program that
%  uses cut (!) to stop after producing one solution.

tested1(X) :- generate(X,Y), test(Y), !.

%  Compare tested1 with tested2, which does not use the cut.

tested2(X) :- generate(X,Y), test(Y).

generate(X,Y) :- candidates(X,Y).
test(Y) :-  candidates(X,Z), candidates(U,V), Z>V, Y>Z, Y<6.

%  tested3 shows the operational reasoning involved in using
%  cuts: although it is logically equivalent to the previous
%  definitions, it produces no answer.

tested3(X) :- generate(X,Y), test3(Y), !, Y<6.

test3(Y) :-  candidates(X,Z), candidates(U,V), Z>V, Y>Z.

end_module.

% sample data

candidates(a,2).
candidates(a,1).
candidates(b,3).
candidates(b,4).
```

```
candidates(c,5).
candidates(d,6).
candidates(d,7).
```

The logical reading of the rules for *tested*1 and *tested*2 implies that b, c and d are solutions. However, if only one solution is desired, it is desirable to terminate the computation once it is produced. We know that a solution has been obtained after successfully instantiating the *test* literal; the cut following it indicates that if control goes past this point and subsequently backtracks back to this point, the rule should "fail". Thus, after a solution is generated, and control returns to the point of the cut via (trivial) backtracking, the rule is deemed to "fail", and no further solutions are generated.

Such reasoning is quite operational. The rules for *tested*3 are logically equivalent to those for *tested*1 and *tested*; however, it produces no solutions. Tracing the execution reveals that $generate(X, Y)$, $test(Y)$ succeeds with X=d, Y=6 first; however, $Y < 6$ fails, and control returns to the point of the cut. By definition of the cut, the rule "fails", and no solutions are generated. (If the first success had been with, say, X=c, Y=5, this program would have generated a solution. However, CORAL offers no guarantees on the order in which base facts — in this program, the *candidate* facts — are considered.)

10.3.1 Cuts and Single Answers for Materialized Execution

As noted in Section 7, while the @aggregate_selection annotation is a useful tool for dealing with single-answer queries, it must be used with care to ensure that an answer is found if one exists. Further, on some programs, to ensure that an answer is found, we have to avoid the use of aggregate selections, as the program in declad7a.P illustrates.

The cut operator is a well-known tool for terminating a pipelined evaluation after generating an answer (as illutrated by the program in pipe2.P). In CORAL, the cut can also be used to achieve similar results for a query that is evaluated using materialization. CORAL supports a mode of evaluation called *lazy evaluation* in which answers are returned at the end of each iteration, and further computation proceeds by request. This is very appropriate when the user is only interested in one answer to the top-level query. If it suffices to have one answer for an intermediate query, the cut operator can be used to achieve the same effect, as the following program, in pipe3.P, demonstrates:

```
module pipe_eg3.
@pipelining.
```

```
export psingle(bf).

psingle(X,Y) :- p(X,Y).

end_module.

module pipe_eg3a.

export p(bf).

p(X,Y) :- b(X,Y).
p(X,Z) :- b(X,Y), p(Y,Z).

end_module.

b(1,2).
b(1,3).
b(1,4).
b(2,5).
b(5,6).
```

In particular, note how this program improves upon just using the aggregate selection; contrast the execution of $?p(1,X)$ versus $?psingle(1,X)$. Of course, by modifying the data, the difference can be made arbitrarily large.

10.4 Ordering of Base Facts

As we have noted several times, CORAL offers no guarantees on the relative order of facts in base predicates. Among other things, this allows CORAL to use flexible indexing strategies. However, it is sometimes desirable to have base facts examined in a specific order. This can be achieved by placing the collection of facts in a pipelined module. If this is done, each fact is considered a rule, and rules in pipelined module are evaluated in the order listed. We emphasize that this is less efficient than treating the set of facts as tuples in a base relation, and should be done only when the order of facts is essential to the program. Further, to the extent that the program depends upon such ordering, the logical reading of the program is compromised. With these caveats, we illustrate how an ordering of facts can be achieved in the following program, in file pipe4.P:

```
module pipe_eg4.
@pipelining.

export p_ordered(f), p_random(f).

%  This program illustrates how the relative order
%  of a collection of facts can be enforced by
%  making them rules in a pipelined module.

p_ordered(X) :-  b_ordered(X).
p_random(X) :-  b_random(X).

end_module.

module pipe_eg4a.
@pipelining.

export b_ordered(f).

b_ordered(1).
b_ordered(2).
b_ordered(3).
b_ordered(4).

end_module.

b_random(1).
b_random(2).
b_random(3).
b_random(4).
```

10.5 Pipelined Evaluation and Exported Query Forms

One other point about pipelined evaluation is worth noting. If a module is evaluated using pipelining, all query forms on the exported predicates are solved using the same evaluation technique. In effect, this pipelined module exports all adorned forms for the exported predicates, regardless of what forms are actually listed in the export statement. (This is an artefact of the CORAL implementation. If the user wants query form $p[bf]$

evaluated using pipelining, and $p[ff]$ evaluated using some other strategy, say, this can only be achieved by renaming one of these uses of p.)

11 Some Useful Built-in Predicates

CORAL provides a collection of "built-in" predicates that can be used in programs just like ordinary (user-defined) predicates. Many of these have been discussed in other sections already.

11.1 Arithmetic Built-Ins

Arithmetic expressions can be constructed using +, -, *, /, `mod` (for modulus) using the usual infix syntax. In addition, `abs` is a binary predicate that binds its second argument to the absolute value of its first argument (which must always be bound when the literal is evaluated). Another useful predicate is the ternary `pow`. The expression pow(A,B,C) succeeds with $A^B = C$. It can be evaluated with any pair of the arguments bound; thus, in addition to computing A^B, it can be used to compute the Bth root of C and log C to the base A. Some important guidelines for using arithmetic predicates are discussed in Section 3.

11.2 Multiset Operators

A number of operators are provided for manipulating multiset values. These are discussed in Section 6 and in the overview [RSSS93]. We list these operators here for convenience: `member`, `unionsum`, `unionmax`, `inter`, `difference`, `subset`, `make_set`, `create_set`, `add_elem`, `count`, `sum`, `avg`, `min`, `max`, `prod`.

We note that an arithmetic expression cannot be listed as an element of a set.

For example, the following, typed in at the command line, would cause an error:

$n >$ `?U = 4, 7*9.`

However,

$n >$ `?U = 4,7,9, prod(U,X).`

results in X being bound to 252. (By the way, such command line interaction is a good way to get familiar with the suite of built-ins.)

11.3 Metaprogramming in CORAL

Some Prolog-style builtins for metaprogramming are available in CORAL. These include: *member, call, univ, functor, is_string, is_num, is_const, is_var, is_functor, and is_list.* Type *help*(*builtins*) to get more information on these builtins.

The predicate *member* can be used to solve goals whose predicate name is only known at run-time. When a literal $member(X, A1, A2, ..., An)$ is evaluated, X must be bound to a relation (or relation name) of arity n, [10] and the remaining arguments can be any terms. This generates a goal $?X(A1, A2, ..., An)$. The relation X can be base or derived. In the latter case, any binding of the A's is used in evaluating the goal; thus only the relevant portion of X is computed.

The predicate *call* is similar to *member*, with slightly different syntax. The literal $call(X(A1, A2, ..., An))$ generates the goal $?X(A1, A2, ..., An)$; X must be bound to a predicate name (not a set). While *member* is more versatile in some ways, *call* can be used in conjunction with *univ* to solve goals whose arglist (and even arity) is only known at run-time.

The suite of routines of the form "is_ ..." test whether an argument is of a particular kind; for example, $?is\_var(A)$ succeeds if A is a variable. These should be used in materialized modules with some caution; re-ordering of literals is possible during fixpoint evaluation, and, for example, a variable that is expected to be unbound gets bound, or vice-versa.

The builtin *univ* is useful for two distinct tasks: (1) to extract the arguments of a functor term, and (2) to create a functor term whose functor name is only available at run-time. The builtin *functor*, which is closely related, extracts the functor name and arity from a functor term.

$n >$ `?X=p, member(X,U).`

11.4 Miscellaneous Operators

Bitwise or, xor, and, rshift and lshift operators are provided, with C syntax.

There is also a special built-in called *fail* that has arity 0. This can be used to force a rule to fail. Logically, it is identical to a relation with no tuples, but is a little more efficient in practice. It is very useful, for example, if we want to time a query and suppress

[10] X can also be bound to a multiset; see 6 for a discussion of *member* used this way.

printing of answers:

$n >$ `?anc(X,Y), fail.`

This would result in the entire anc relation being computed (assuming that a module defining this predicate has already been consulted), but no X,Y bindings being printed.

12 CORAL Commands

The *help* command provides a menu-driven help facility. When *help.* is typed at the CORAL command interface, several topics are listed. Specific help on one of these topics can be requested by typing *help(topic)*. In this section, we provide an overview of the available commands, and we recommend that the reader supplement this material by using the *help* command.

12.1 Executing Commands

Commands can be executed from the CORAL prompt, of course. In addition, commands can be embedded in C++ or Tcl code (see Section 13) or used in declarative rules in modules. When commands are used in declarative modules that are evaluated using materialization, no guarantees are offered as to the order of execution; this makes it very difficult to write meaningful programs. However, using commands in pipelined modules can be quite useful. See Section 10 for a detailed discussion of this point.

12.2 Commands Discussed Elsewhere

Several commands are discussed in other sections of this tutorial (for example, *shell, list_rels*). Some commands are straightforward, and adequately documented in the help files. For example, *help(io)* will give you information on *print* and other commands for input and output. (We note that several nice I/O features are supported, e.g., answers can be output in sorted order, relations can be input/output in fact form or table form, etc.) We will not discuss them further in this section.

12.3 Consult

The *consult* command has already been discussed in earlier chapters, and extensive information is available via on-line help. However, it is an important command, and a couple of points are worth emphasizing. First, a consulted file can contain a series of CORAL commands. These are simply redirected to the CORAL input, and the result is identical to typing them in at the prompt. In particular, nested *consult*s are handled correctly. Second, When persistent relations are being manipulated, transaction semantics are guaranteed for the persistent relations at the granularity of a single user command. In other words, a single user-level consult command is treated as a transaction, independent of whether there are any nested consults.

12.4 Commands That Modify the Execution Defaults

A number of execution parameters have their values set through defaults, and can be modified by the user.

display_defaults: This command lists the current values of all parameters that control execution, and indicates which of them can be reset.

assign: This command can be used to reset the value of an execution parameter.

set and *clear*: These commands are similar to *assign*; they just provide a convenient way to change the values of parameters that are ON/OFF flags.

With respect to the above commands, it is important to keep in mind that *consult* effectively "compiles" queries. If some execution parameters are changed (for example, we could do `set(psn)` to require that a variant of semi-naive evaluation be used as the default), these only affect subsequent consults, and the behaviour of previously consulted queries is not changed.

Another useful command is *alias_rel*. Many CORAL command names are verbose, and once a user gets familiar with them, typing these long names can get to be irksome. The *alias_rel* command allows users to rename commands as they please. Indeed, by placing a series of *alias_rel* commands in a file called .coralrc in their home directory, the renaming can be made to persist across sessions. All commands in the *.coralrc* file are executed each time CORAL is invoked; any command whatsoever can be placed in this file. CORAL looks for the *.coralrc* file in the current directory first, and then in the home directory. This can be taken advantage of to customize the execution of CORAL in different directories, for example, run in verbose mode in a "debugging" directory, load in some data files automatically in a "test" directory, etc.

The *help* command can be used to get more details on any of the above commands.

12.5 Commands for Debugging

CORAL provides some basic tools for debugging of programs. These include an *explanation* package as well as *tracing* and *profiling* capabilities.

12.5.1 Explanations

CORAL provides a powerful proof-tree based explanation mechanism[?] for declarative modules evaluated using materialization. (This is orthogonal to the choice of rewriting techniques and fixpoint evaluation method, the use of save module, etc.; however, it does not work with pipelining.)

In materialized evaluation, the user's program is (possibly) rewritten, and the rewritten program is evaluated by iteratively instantiating rules to generate new facts. If the command *explain_on().* is typed in at the CORAL prompt, each rule instantiation that generates a fact is forwarded to the *Explain* program for display. The forwarding can be turned off with *explain_off().*

These commands can be called with an exported predicate name as argument if only instantiantions relevant to queries on that predicate (i.e. instantiantions carried out in the module that exports the predicate) are to be forwarded.

The *Explain* program collects and displays the information forwarded to it, allowing the user to analyze the set of derivations graphically. "Derivation trees" can be "grown" and "pruned" dynamically on the screen, thereby providing a visual *explanation* of just how facts were generated.

We note that the derivations are recorded in the exact form that they are carried out. Thus, if the user's program was rewritten by the system, the recorded derivations reflect the rewriting, and it can sometimes be hard to see the mapping between the original and rewritten programs. We suggest that while using the explanation facility, programs be run with only the @*magic*. rewriting annotation. The mapping between the original program and the program rewritten using this algorithm is simple, and the user should be able to reason essentially in terms of the original program when presented with derivations of the rewritten program.

You will notice that some of the predicate names in the derivations appear with an $m\_$ as a prefix, [11] and most predicate names have suffixes like bbf etc. A fact of the form $m\_p\_bf(a)$ indicates that there was a subgoal $?p(a, X)$ during the course of the program evaluation. A fact $p\_bf(a, 5)$ indicates that $p(a, 5)$ was computed in response to the query fact $m\_p\_bf(a)$. The suffixes indicate which arguments were bound and which were free in the subgoal. For instance, we had m_p_bf since the first argument was bound to a and the second argument was a free variable. The same suffix $\_bf$ is also present in answers to the subgoal m_p_bf.

[11] "m_" stands for "magic".

The easiest way to learn about the explanation facility is to use it, and we encourage the reader to do so. For more information, type *help(explain).* at the CORAL prompt.

12.5.2 Tracing

A `trace` facility is provided that does the following:

1. It lets the user know what rules are being evaluated, and
2. It prints out answers and subgoals as they are generated, to let the user know how the computation is proceeding.

You can type:

`trace_on().`

This prints out every fact when it is derived. *In addition, if CORAL is not running in* `quiet` *mode, each rule is printed out when it is applied.*

`trace_off().`

turns off this feature.

It is possible to trace individual exported predicates rather than tracing all predicates, using:

`trace_on(exported_predicate_name).`

All predicates in the module where the predicate is defined are automatically traced.

Trace output is printed by default on stderr. The assign command can be used to direct trace output to any file by changing the value of the *trace_file* parameter.

Further details about this facility can be obtained using `help(trace).` from the CORAL prompt.

WARNING: The current implementation of trace sometimes prints out variable names incorrectly. There can be two variables of the same name in different contexts (bindenvs). When printing out When such variables are printed, they must be renamed, and the output routines do this. But the debugging trace routines do not do this currently (for reasons of efficiency), and you may find that two distinct variables are printed with the same name.

12.5.3 Profiling

CORAL also provides some high-level profiling facilities. The unit of profiling is the unification operation. Unification of two atomic terms counts as one unification, while, for example, unification of $f(X, Y)$ and $f(a, b)$ counts as three unifications, one at the outer level and two at the inner level. Profiling also lets the user know how efficient the indexing is, by keeping counts of the number of tuples that the indexing operation tried to unify, and the number that actually unified and were retrieved. In addition, other counts such as number of successful applications of each rule, and the number of unsuccessful attempts at using a rule are also maintained. All this information put together gives users a fair idea of where their programs are spending the most time, and helps them optimize programs accordingly.

Type `help(profile).` to find out more about profiling.

12.5.4 Timing Information

The following commands are useful in timing an execution[?].

reset_timer This command sets the timer to 0.

display_timer This command prints out the time elapsed since the last reset_timer command (or start of the CORAL session).

In addition, there is a unary built-in predicate called *cputime* that returns the time elapsed since the previous reset_timer command. This can be used to obtain timing information within a rule.

12.6 Commands that Manipulate Relations and Workspaces

Relations can be created, inserted into, deleted from, and indexed from the command line[?]. Insertion and deletion can be done on a per-tuple basis or, using rule-like syntax, in a set-oriented fashion. Type `help(relations).` to get more information on these commands.

CORAL supports a *workspace* concept[?]. A workspace is a collection of relations; either exported from a module or base. At all times, there is a *current workspace*. All queries are evaluated against the current database, and when a file is consulted, the relations defined in it are added to the current workspace. New workspaces can be created, and the current workspace can be changed. Relations defined in a workspace can be emptied of all tuples, closed (i.e. tuples are not visible), opened (tuples visible again), opened in another workspace (tuples visible, but not copied), or copied to another workspace (physically copied). The current state of a workspace can also be *saved* in a file and restored later.

The workspace facility is especially useful during long sessions and for hypothetical reasoning. For example, a new workspace can be created and some of the relations defined in other relations can be opened in this workspace. Additional relations can be defined, and possibly, several queries executed in this new workspace. So long as only new relations are modified, the changes in this workspace are not visible in the other workspaces. Thus, hypothetical changes can safely be made in the new workspace. Hypothetical insertions are easily accomplished by defining a new relation to contain all tuples in an old relation plus the inserted tuples. Hypothetical deletes, however, require the old relation to be copied (not just opened) in the new workspace; otherwise, a new relation containing the deleted tuples must be defined, and a second relation (which corresponds to the original relation after the deletions) must be defined in terms of the original relation and the relation contining the deleted tuples using negation.

We note that all *persistent relations*[?] are stored in a special workspace called `exodus_ws`[?]. Of course, they can be opened in other workspaces as well. (If such a workspace is saved, only the names of persistent relations are saved. Thus, if the persistent relations are subsequently modified, these changes will be reflected upon restoring the saved workspace.)

13 CORAL and C++

The CORAL system has been integrated with C++ in order to support a combination of imperative and declarative programming styles. We have extended C++ by providing a collection of new classes (relations, tuples, args and scan descriptors) and a suite of associated methods. In addition, there is a way to embed CORAL commands in C++ code. This extended C++ can be used in conjunction with the declarative language features of CORAL in two distinct ways:

- Relations can be computed in a declarative style using declarative modules, and then manipulated in imperative fashion in extended C++ without breaking the relation abstraction. In this mode of usage, typically there is a main program (and possibly some built-in definitions) written in C++ that call upon CORAL for the evaluation of some relations defined in CORAL modules. The main program is compiled (along with associated built-ins, and after some preprocessing) and executed from the Unix prompt, and the CORAL command interface is not used.

- New built-in predicates can be defined using extended C++. These built-ins can be used in declarative CORAL code and incrementally loaded from the CORAL command interface.

Thus, declarative code can call extended C++ code and vice-versa. We discuss the above two modes further in the following sections.

13.1 Adding Relations to C++

We have extended C++ by adding a collection of classes and associated methods. The new classes are:

- **Relation** This allows access to relations from C++. Relation values can be constructed through a series of explicit inserts and deletes, or through a call to a declarative CORAL module. The associated methods allow manipulation of relation values from C++ without breaking the relation abstraction.

- **Tuple** A relation is a collection — set or multiset — of tuples.

- **CArg** A tuple, in turn, is a list of args. A number of methods are provided to construct and take apart arguments and argument lists.

- **C_ScanDesc** This abstraction allows us to support relational scans in C++ code. A C_ScanDesc object is essentially a cursor.

In addition to the new classes, any sequence of commands that can be typed in at the CORAL command interface can be embedded in C++ code. The code must be bracketed by `\[` and `\]`.

The collection of new classes and the associated methods is documented in the interface specification. The following simple program is in impmod1.S, and is a good example of the use of declarative CORAL from imperative C++ :

```
/*
 * Example of a C++ program that uses declarative CORAL.
 */

#include <stdio.h>

main(int argc, char**argv)
{
  int i = 2; double j = 4.23;
  printf("hello there\n");
  init_coral(argv[0]);

  for (i = 0; i < 3; i++) {
    fprintf(stderr, "entering iteration %d\n", i);
    /*
     * here is the embedded CORAL code !
     *
     * note that the start and end markers must each be
     * on a separate line that has no other non-whitespace
     * characters.
     *
     * Each time through the loop, the parameter i
     * that is passed to the declarative CORAL code will
     * vary, while j remains the same. Hence the query
     * ?grows(X,Y) will give successively increasing answers
     * as more facts are added to the grows() relation.
     * The query ?static(X,Y) returns the same answer each
     * time through the loop.
     */
```

```
    \[
      grows(($int)$i, 1).
      static(2, ($double)$j).
      ?grows(X,Y).
      ?static(X,Y).
        \]
  }
  printf("bye there\n");
  exit_coral();
}
```

There are a quite a few important things to note in this example. The first is that a file containing C++ code with embedded CORAL code must first be passed through the CORAL preprocessor and then compiled. The file *make.sample* in the *interface* directory provides a template. (It is also included in Appendix **??**.) Before CORAL can be called from C++, it has to be initialized by calling init_coral(), with the name of the calling program as the argument. Before the program terminates, exit_coral() should be called.

The values of C++ variables can be passed to CORAL by using the following syntax :

```
 \[
parent(($int)$i, ($int)$j).
 \]
```

The types that can be passed in this fashion are: int, double, string (char*), CArg*, and chunk (void*).

It should be noted that in the process of translating the C++ program with embedded CORAL code, some auxiliary files may be created that will be used at run-time.

Another example, which is in impmod2.S, illustrates the new classes that have been added to C++ as part of the CORAL interface:

```
/*
 * Example of the use of the Relation, C_ScanDesc, Tuple and Arg classes
 * and some functions provided in the interface.
 */

#include <stdio.h>
```

```
main(int argc, char**argv)
{
  char *rel_name = "data";
  int rel_arity = 2;

  init_coral(argv[0]);

  /* First consult the data file which contains facts of the
   * form data(1,2), data(2,3), etc.
   *
   * The aim of the progam is to add the values of the first
   * argument of each fact, and print the sum.
   */
  \[
consult(data.F).
  \]

  Relation *rel = find_relation(rel_name, rel_arity);

  C_ScanDesc *scan = new C_ScanDesc(rel);

  Tuple *tuple;
  int sum = 0;

  /*
   * Iterate over the tuples in the relation
   */
  for (tuple = scan->next_tuple(); !(scan->no_match());
 tuple = scan->next_tuple()) {
    if (!is_int((*tuple)[0])) {
      fprintf(stderr, "non-integer first field !\n");
      exit 1;
    }
    sum += make_int((*tuple)[0]);
  }

  printf("Sum is %d\n", sum);
  exit_coral();
}
```

This example uses a few functions like `find_relation` and `is_int` that are part of the interface specification. The complete interface specification is provided in the appendix. However, this simple program demonstrates the fact that the C_ScanDesc abstraction, along with the Relation, Tuple and CArg abstractions, gives the C++ programmer a convenient way of accessing data stored in CORAL relations. Scans can be set up in a totally identical fashion on both base and derived relations. (Note that it is easy to materialize a derived relation, if desired, by using an imperative rule with ":=")

A suite of routines is provided for converting CORAL terms into C++ values and vice-versa. (A full listing is given in the interface specification.) One restriction in the current interface is that a very limited abstraction of variables is presented to the user. Variables can be used as selections for a query (say, via repeated variables) or in a scan, but variables cannot be returned as answers (i.e., the presence of non-ground terms is hidden at the interface). Presenting the abstraction of non-ground terms would require that binding environments be provided as a basic abstraction, and this would make the interface rather complex.

The interface is dealt with in detail in an appendix. There are also some sample programs in the `interface` directory that might be useful. This interface has already been used to develop an *explanation* facility for CORAL, and we are in the process of developing other applications using it.

13.2 Defining Built-Ins

As we have already seen, predicates exported from one CORAL module can be used freely in other modules. Sometimes, it may be desirable to define a predicate using extended C++, rather than the declarative language supported within CORAL modules. The facility for defining *built-in predicates* is intended to fill this need. [12]

Defining a new built-in predicate is a straightforward process. A *_coral_export* statement is used to declare the arguments of the predicate being defined. The definition can use full extended C++. We require that built-in definitions be in files whose names have a ".S" suffix. Such definitions can be consulted from the CORAL prompt, just like .P files (which contain module definitions) or files containing facts or tables. The .S file is pre-processed into a C++ file, compiled to produce a .o file, and read into a newly

[12]The name "built-in" is used because this is indeed the way that many of CORAL's system-defined built-in predicates (for arithmetic, set manipulation, etc.) are implemented. We note that any such built-in can be accessed from C++ code as well.

allocated region in the data area of the executing CORAL system. It is also possible to directly consult a pre-processed .C file or .o file, and avoid repeating the pre-processing and compilation steps.

The following program is in impmod3.S:

```
/* Example of C++ code that is to be incrementally loaded into CORAL */

#include <stdio.h>

/* The export statement below says that this function defines a built-in
predicate whose first argument is a double and whose second argument is
also a double.  Note that the output value of this function is
automatically mapped to the second argument of the built-in, which
is therefore to be called with a bf binding pattern.  For example,
we could have myfunc(X,Y) appear in rule in a declarative module.
X must be bound to a double.  */

_coral_export double myfunc(double);

double myfunc(double x)
{
  return x*2 ;
}
```

It is a simple built-in definition, and should require little explanation. We note that the return value of the C++ function myfunc is automatically mapped into the second argument of the built-in predicate myfunc that results when this definition is consulted. This built-in can only be called with the first argument bound to a double; the second argument can be free or bound. If the second argument is bound, the computed value is compared with the binding.

The following are the only types that can be used in a _coral_export declaration: int, short, long, float, double, char* and CArg*. User-defined types are not allowed. The export mechanism makes it easy to pass values of these limited types between CORAL and C++ code. It is important to note that the translator currently does *no* type checking, or even attempt to check if the exported function is defined in the file; it is a purely syntactic filter.

CArg* is a catch-all type; it can be used to pass bitmaps, relations, C++ structs, or just about anything. It is especially convenient for passing structured CORAL terms (e.g. lists) to a builtin defined using extended C++, as the following example, in impmod4.S, illustrates:

```
/*
 * Example of a builtin relation definition that is to be
 * incrementally loaded. This builtin demonstrates the use of
 * Arg* to allow arbitrary CORAL structured arguments
 * to be manipulated by user-defined code.
 */

#include <stdio.h>

/*
 * The builtin sum_list(X, Y) takes a list X as its first argument
 * and returns the summation of the list in the second argument.
 */

_coral_export int sum_list(Arg *);

int sum_list(Arg *input_list)
{
  int sum = 0;
  Arg *temp;

  /* first check that the input argument is indeed a list */
  if (!input_list || !(is_list(input_list))) {
    fprintf(stderr, "WARNING ! : non-list argument \n");
    return -1;
  }

  /* iterate through the list, summing the elements */
  while (input_list) {

    temp = make_car(input_list);

    /* check that each list element is an integer */
    if (!temp || !is_int(temp)) {
      fprintf(stderr, "WARNING ! : non-integer list member \n");
```

```
      return -1;
    }

    sum += make_int(temp);
    input_list = make_cdr(input_list);
  }

  return sum;
}
```

Built-in definitions can be incrementally loaded from the CORAL command interface, as we mentioned earlier. They can also be compiled with the main C++ program, as discussed in the previous section.

14 Extensibility in CORAL

The CORAL architecture is designed to be extensible. New relation and index implementations can be added; the support for persistent relations is based upon this aspect of CORAL. New data types can be added; for example, a bitmap type with special equality and display operations can be defined. In addition to the architecture of the system, the integration with C++ allows for the definition of sophisticated methods associated with the new types.

We anticipate that the most common use of CORAL's extensibility will be the addition of new types tailored to a particular application domain. To create a new type[13], the user must carry out the following three steps:

- Declare the new type as a subclass of the system class ConstArg.
- Define several virtual functions (methods) for the new type. These include: an operator '==' (which takes an object of type Arg as parameter), printon (which takes a file as a parameter), hash which returns a hash value, copy which creates a copy of the object, and delete which is called when the system no longer needs the object.
- Define built-ins (see Section 13) to create, destroy, and manipulate values of the new type.

We refer the reader to the overview document [RSSS93] for more details on extensibility in CORAL.

14.1 Arrays

As a case study, we consider the addition of an *array* data type to CORAL. The first two steps in defining the array data type are best understood by carefully examining the file *array.C*, which is included in Appendix **??**. This file is compiled and linked with the CORAL system to add support for arrays to the system.

The third step is the most important in terms of understanding what additional capabilities have been provided to the user. The built-ins to manipulate arrays are as follows. We use the convention that input values start with lower case and output values with upper case.

[13]Objects of the user-defined type must be "constants", i.e., they cannot contain variables within them.

```
array(Array, size)
// Binds 'Array' to an array of size 'size'.
// Array offsets start from 0.
// BEWARE: only ground values may be stored in arrays.

array(array, Size)
// Unifies Size with the size of array (which may be a
// logical array (see below)).

logical_array(Array, size)
// Binds 'Array' to a logical array of size 'size'.
// A logical array permits efficient logical_bind (see below).
// Array offsets start from 0.
// BEWARE: only ground values may be stored in arrays.

logical_array(array, Size)
// Unifies Size with the size of array.  array can be of
// any array type, not necessarily logical_array.

bind(array, index, value)
// Binds the index'th element of array to value
// BEWARE:  the operation is destructive, even on logical
// arrays.  The operation can alter stored facts.

logical_bind(old_array, index, value, New_array)
// Creates new_array as version of old_array and
// Binds the index'th element of new_array to value.
// The versioning operation is efficient in the case of
// logical arrays, but is inefficient for regular arrays
// since it involves copying out the entire array.

lookup(array, index, Value)
// Unifies Value to the index'th element of array.
// Fails if index is out of range, without a warning msg.
```

The above suite of built-ins provides the interface specification for two distinct types of array data structures, both of which require all elements to be ground data structures (of any kind). First, a notion of *logical arrays* is supported.

It is easy to understand the concept of a logical array by analogy with lists and multisets. If L is a list and E is an element, we can define a predicate *append_element* as follows:

```
append_element([], E, [E]).
append_element([X|Y], E, [X|W]) :- append(Y, E, W).
```

Consider a goal ?*append_element*$(L1, 5, L2)$, where $L1$ is bound to a (say ground) list value. The element 5 is appended to the list, and the resulting list is $L2$. The important point to note is that list $L1$ is not changed — at least, with respect to a logical reading of the program. Thus, if $L1$ is used later in the same rule that contains the above goal, it denotes the same list as before. The implementation can be carried out in several ways, some involving a change to $L1$'s representation, as long as the logical reading is not affected.

Similarly, the multiset built-ins do not modify the arguments. For example, $inter(S1, S2, S)$ makes S be the intersection of $S1$ and $S2$, but multisets $S1$ and $S2$ are not changed.

In summary, the list and multiset data structures in CORAL are "logical" or "non-destructive" data-structures, in the sense that operations on them are non-destructive. Logical arrays are similar; the operations on them (*logical_array*, *lookup* and *logical_bind*) are non-destructive. The *logical_array* built-in can be used to create a logical array of a given size or to check the size of a given logical array. The *logical_bind* operator, in a way that is similar to the multiset operator *inter*, for example, creates a new array that is identical to the old array except that the value of the ith element is changed; the old array is not affected. Logical arrays are implemented as balanced tree-like structures, and *lookup* and *logical_bind* both take time that is logarithmic in the size of the array.

The following program, in extens1.P, illustrates the use of logical arrays:

```
module extens_eg1.
export lcumsum(bf).

% Adds up the elements of input array, and stores the cumulative sum
% of elements 0 through I in the I+1 st element of the result array.
% The logical array data structure is used.
% A sample session is as follows:
%
%--------------------------------
% lstore(X) := logical_array(X,5), bind(X,0,0), bind(X,1,1),
%        bind(X,2,2), bind(X,3,3), bind(X,4,4).
```

```
% consult(extens1.P).
% ?lstore(X), lcumsum(X,Y).
%---------------------------------

lcumsum(OldArray,NewArray) :- array(OldArray,Size),
tempcumsum(OldArray,Size, NewArray,Size-1).

                       /* 0-th element of result is itself */
tempcumsum(OldArray,Size, OldArray, 0).
tempcumsum(OldArray,Size, NewArray,I+1) :-
tempcumsum(OldArray,Size, TempArray,I),
I < Size,
                lookup(OldArray,I+1,Val),
                lookup(TempArray,I,Sum),
                logical_bind(TempArray,I+1,Sum+Val,NewArray).

end_module.
```

The second kind of array that is supported in CORAL is the familar array data structure found in imperative languages. The *array* built-in is used to create such an array. The *bind* built-in destructively updates an array by changing the value of an element, in constant time. (The *lookup* operation is also constant time for destructive-assignment arrays.) This operation should be contrasted with the *logical_bind* operation. (We note that the *logical_bind* operation can also be used on destructive-assingment arrays — which are created with *array* — but it is implemented via copying and is thus not as efficient as on logical arrays. Also, the *bind* operation can be used to make a destructive update to a logical array.) Destructive arrays are quite useful in conjunction with pipelined execution, where the order of execution is predictable.

The following program, in extens2.P, illustrates the use of destructive-assignment arrays:

```
module extens_eg3.
export foreach(fbb).

% The for-each definition below is a useful template for iterating
% over arrays (both logical and destructive).
```

```
% A sample session is included below:
%
%--------------------------------
% store(X) := array(X,5), bind(X,0,0), bind(X,1,1), bind(X,2,2),
%        bind(X,3,3), bind(X,4,4).
% consult(extensj.P).
% ?store(X), foreach(I,0,4), lookup(X,I,Val), print(Val).
%
%---------------------------------

% Generates all integer values in the range Low to High, both inclusive.

foreach(Low,Low,High) :- Low <= High.
foreach(I+1,Low,High) :- foreach(I,Low,High), I < High.

end_module.
```

In extens3.P, we present a predicate that is quite useful in dealing with arrays:

```
module extens_eg3.
export foreach(fbb).

% The for-each definition below is a useful template for iterating
% over arrays (both logical and destructive).
% A sample session is included below:
%
%--------------------------------
% store(X) := array(X,5), bind(X,0,0), bind(X,1,1), bind(X,2,2),
%        bind(X,3,3), bind(X,4,4).
% consult(extensj.P).
% ?store(X), foreach(I,0,4), lookup(X,I,Val), print(Val).
%
%---------------------------------

% Generates all integer values in the range Low to High, both inclusive.

foreach(Low,Low,High) :- Low <= High.
foreach(I+1,Low,High) :- foreach(I,Low,High), I < High.
```

```
end_module.
```

Finally, we note that there is no special syntax for entering array values directly. (In contrast, the [...] notation is available for lists and the { ... } notation is available for sets.) However, it is quite easy to create an array value from a relation, as the following program, in extens4.P, illustrates.

```
module extens_eg4.
export input_array(bbf).

% This predicate allows for convenient creation of a
% logical array value from data presented as a relation.
% By changing the logical_array literal to an array
% literal, we can create destructive arrays.

input_array(In_rel, Size, NewArray) :-
logical_array(NewArray,Size), member(In_rel,I,V), bind(NewArray,I,V).


end_module.

% Given the following facts, the goal
% ``?input_array(input,5,A)'' creates a new
% array A that contains the elements a--e.

input(0,a).
input(1,b).
input(2,c).
input(3,d).
input(4,e).
```

15 Programming in CORAL: Some Guidelines

CORAL is a high-level language, and the compiler attempts to optimize programs to ensure efficient evaluation. However, completely automatic optimization in a language this powerful can only be an ideal, and the compiler often selects a less than optimal evaluation strategy. As with any programming language, writing efficient programs is an art that must be learned. With some broad understanding of the evaluation strategies used in CORAL, however, you can usually make a program more efficient (without changing the underlying logic extensively, if at all) by using the following guidelines:

1. Remember that CORAL evaluates rules in left-to-right order; avoid cross-products.
2. The default evaluation is bottom-up fixpoint evaluation, after (Supplementary) Magic rewriting. Consider whether top-down backtracking ("pipelining") might be better, or whether you can provide hints to make the fixpoint evaluation run faster.
3. Make use of the module facility to ensure that each portion of a program is evaluated using the best possible strategy.
4. Use C++ to define builtins or to utilize special data structures when appropriate.

Typically, a good organization of a program into a set of modules, together with a few hints, will ensure that the program runs fast. We discuss the above points in more detail in the rest of this section.

15.1 Order of Literals in a Rule

Remember that execution proceeds left-to-right within each rule. Thus, more selective predicates (i.e. those with fewer tuples) should be placed to the left. In particular, try and avoid rules like this:

- p(X) :- q(X), r(Y), X=2*Y.

If both q and r contain 1000 tuples, the left-to-right evaluation of this rule would create a million intermediate X-Y pairs. On the other hand, the following version of the rule eliminates the problem:

- p(X) :- q(X), X=2*Y, r(Y).

15.2 Modules

The use of modules is one of the most powerful techniques available in CORAL for improving program structure and efficiency. There are two aspects of module evaluation to consider:

1. The choice of evaluation method in a module is independent of how the module is called. The default is Seminaive fixpoint evaluation following Supplementary Magic rewriting, but Ordered Search is used if the module contains a negated literal (if the negated predicate is defined in the same module) or grouping (if any predicate in the body of the rule with grouping is defined in the same module).

2. If one module calls another, the caller waits until the called module returns an answer; on backtracking, the called module attempts to generate all answers. (See Section 8 for more details on inter-module calls.)

A simple rule of thumb: *The use of Ordered Search should be minimized, and pipelining should be considered when possible.* Sections **??** address this point in more detail. Organize your program into modules in such a way that the most appropriate evaluation strategy is used; for example, make sure that rules to be evaluated using pipelining are not mixed with rules to be evaluated using Ordered Search in the same module.

15.3 Pipelining vs. Materialization

Pipelined evaluation is very similar to Prolog-style evaluation, and is essentially top-down, depth-first backtracking. It has the risk of infinite loops on some programs (e.g. left-linear recursion), and being inefficient due to repeated subgoals (e.g. fibonacci), but it is often much more efficient than materialization (e.g. append). It is a good idea to run a program with pipelining (@pipelining+), perhaps on a smaller representative dataset, to see if this offers a significant improvement over the default materialized evaluation. This is especially the case for programs where infinite loops are not a concern (e.g. non-recursive programs).

One restriction to note is that for Materialized modules, any existing bindings for the arguments to the exported function must be fully ground at the time of the call. They may of course, be free as well. This restriction does not exist for pipelined modules.

15.3.1 Materialization

Materialized evaluation consists of rewriting the original program (to propagate bindings in the query) followed by evaluating the fixpoint of the rewritten program.

Rewriting Algorithms

The default rewriting strategy is Supplementary Magic (@supmagic+), and it works well for queries with bound arguments in which subgoals are generated multiple times. The Factoring rewriting can be much faster for some programs, and is worth trying (@factoring+). If there are no bound arguments, no rewriting may be the best approach (@no_rewriting+).

Fixpoint Evaluation

Semi-naive evaluation is the basic fixpoint evaluation algorithm. The following issues are worth considering. If facts are not likely to be generated in multiple iterations, testing for duplicates may not be worthwhile, and can be turned off (@check_subsumption-). A good compromise is to check for duplicate goals but not to check duplicates for other facts (@multiset+). If several facts are generated in each iteration, it is worth indexing the set of newly generated facts, and this is the default. However, if only a few facts are generated in each iteration, this can be turned off (@index_deltas-).

15.4 Using C++ and CORAL Effectively

While the declarative subset of CORAL offers the advantages of a high-level rule-based language, to get the most out of the CORAL system, you must use the C++ interface effectively. Especially for large applications, it may be necessary to write some code in C++. This may be to define some core algorithms, or to define specialized data structures. Within CORAL rules, the only data structures are constants, logical variables and terms; you may sometimes want to define new data types (e.g. arrays). CORAL is an extensible system that allows you to do this. Relations in CORAL are implemented as structures with hash-based indices. Sometimes, a different kind of relation implementation might be useful (e.g., relations sorted on a specified field). Again, this can be done in CORAL. Adding new data types and relation implementations to CORAL is discussed in Section 14.

By identifying critical data structures and code that can be handled efficiently (and sometimes, more simply) using C++, the utility of the CORAL system can be greatly increased for large, data intensive applications. The use of C++ is discussed in Section 13.

16 Current Status

The following issues are not handled satisfactorily in the current version of CORAL. We hope to address them soon:

Arithmetic Expressions : Functions such as *plus* are now unidirectional, i.e., X+Y = Z causes an error unless X and Y are bound. We plan to make such functions behave more uniformly to the extent possible without constraint solving, e.g. X+Y = Z should work correctly as long as some pair of variables is bound. We also intend to re-order arithmetic literals as early as possible during the evaluation of a rule.

Persistent Relations : Currently, derived relations have to be in-memory, i.e., a relation that is defined in a module via rules and is evaluated using materialization cannot be stored on disk using Exodus. We plan to remedy this shortly. Memory management is flaky; as tuples are read in from disk, the values in these tuples are copied into main memory, which can therefore fill up quickly. While we have tried to minimize the amount of copying, in the long term, we will eliminate such copying.

Specification of sips : This is currently flaky.

Unification : Occur checks are not implemented currently.

Some longer term directions are listed below:

Support for Some Specialized Relations : In particular, we plan to support ordered relations and array relations. An ordered relation permits efficient ordered scans, and an array relation allows indexed access to tuples.

Object-Orientation : We have a design in place for CORAL++, which extends CORAL with support for named fields, objects, classes, methods, inheritance and encapsulation.

Language Issues : We would like to add some features to the declarative language such as disjunction (Prolog's ";") in rule bodies, rules with multiple heads, and an if-then-else construct. We also want to eliminate some of the syntactic restrictions currently placed upon rules that use grouping (< ... >).

References

[BR87] Catriel Beeri and Raghu Ramakrishnan. On the power of Magic. In *Proceedings of the ACM Symposium on Principles of Database Systems*, pages 269–283, San Diego, California, March 1987.

[Bra90] I. Bratko. *Prolog Programming for Artificial Intelligence.* Addison-Wesley, 1990.

[Bry89] Francois Bry. Logic programming as constructivism: A formalization and its application to databases. In *Proceedings of the ACM SIGACT-SIGART-SIGMOD Symposium on Principles of Database Systems*, pages 34–50, Philadelphia, Pennsylvania, March 1989.

[CDRS86] Michael Carey, David DeWitt, Joel Richardson, and Eugene Shekita. Object and file management in the EXODUS extensible database system. In *Proceedings of the International Conference on Very Large Databases*, August 1986.

[KRS90] D. Kemp, K. Ramamohanarao, and Z. Somogyi. Right-, left-, and multi-linear rule transformations that maintain context information. In *Proceedings of the International Conference on Very Large Databases*, pages 380–391, Brisbane, Australia, 1990.

[Llo87] J. W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, second edition, 1987.

[NRSU89] Jeffrey F. Naughton, Raghu Ramakrishnan, Yehoshua Sagiv, and Jeffrey D. Ullman. Argument reduction through factoring. In *Proceedings of the Fifteenth International Conference on Very Large Databases*, pages 173–182, Amsterdam, The Netherlands, August 1989.

[Ram88] Raghu Ramakrishnan. Magic Templates: A spellbinding approach to logic programs. In *Proceedings of the International Conference on Logic Programming*, pages 140–159, Seattle, Washington, August 1988.

[RBK88] Raghu Ramakrishnan, Catriel Beeri, and Ravi Krishnamurthy. Optimizing existential Datalog queries. In *Proceedings of the ACM Symposium on Principles of Database Systems*, pages 89–102, Austin, Texas, March 1988.

[Ros90] Kenneth Ross. Modular Stratification and Magic Sets for DATALOG programs with negation. In *Proceedings of the ACM Symposium on Principles of Database Systems*, pages 161–171, 1990.

[RS91] Raghu Ramakrishnan and S. Sudarshan. Top-Down vs. Bottom-Up Revisited. In *Proceedings of the International Logic Programming Symposium*, 1991.

[RS92] Kenneth Ross and Yehoshua Sagiv. Monotonic aggregation in deductive databases. In *Proceedings of the ACM Symposium on Principles of Database Systems*, pages 114–126, 1992.

[RSS90] Raghu Ramakrishnan, Divesh Srivastava, and S. Sudarshan. Rule ordering in bottom-up fixpoint evaluation of logic programs. In *Proceedings of the Sixteenth International Conference on Very Large Databases*, August 1990.

[RSS92a] Raghu Ramakrishnan, Divesh Srivastava, and S. Sudarshan. Controlling the search in bottom-up evaluation. In *Proceedings of the Joint International Conference and Symposium on Logic Programming*, 1992.

[RSS92b] Raghu Ramakrishnan, Divesh Srivastava, and S. Sudarshan. CORAL: Control, Relations and Logic. In *Proceedings of the International Conference on Very Large Databases*, 1992.

[RSSS92] Raghu Ramakrishnan, Divesh Srivastava, S. Sudarshan, and Praveen Seshadri. Implementation of the CORAL deductive database system. Submitted, 1992.

[RSSS93] Raghu Ramakrishnan, Praveen Seshadri, Divesh Srivastava, and S. Sudarshan. An overview of coral. Manuscript (full version of [RSS92b], which appeared in VLDB92)., 1993.

[Van92] A. Van Gelder. The well-founded semantics of aggregation. In *Proceedings of the ACM Symposium on Principles of Database Systems*, pages 127–138, 1992.

[VRS91] A. Van Gelder, K. Ross, and J. S. Schlipf. Unfounded sets and well-founded semantics for general logic programs. *Journal of the ACM*, 38(3):620–650, 1991.

A The CORAL Installation Guide

This chapter of the manual is not valid for Coral Release 1.2, please use the file install.chap included at the top level of the coral distribution instead.

CORAL is an experimental logic-programming language implemented using bottom-up techniques.

The document "INSTALL" at the top level of the CORAL distribution describes how to install CORAL. The release contains premade executables for many different platforms.

You can obtain a copy of the latest release from: http://www.cs.wisc.edu/coral/

or via ftp at: ftp.cs.wisc.edu:/coral/

If you run into any problems installing CORAL, you can contact the CORAL maintainer at:

coral.cs.wisc.edu

We hope you enjoy using CORAL; your feedback would be appreciated. If you extend CORAL, and want to make your changes available to the world, you can get in touch with Raghu Ramakrishnan at:

raghu@cs.wisc.edu

or the CORAL maintainer at:

coral@cs.wisc.edu.

B The CORAL–C++ Interface Specification

The document "doc/Interface" included with the CORAL distribution describes using CORAL with C++ and/or Tk/Tcl.

Index

.coralrc, 101

allowed adornments, 80
annotations, 76
arithmetic expressions, 97

basic seminaive evaluation, 82
BSN, 82
builtin predicates, 97

command aliases, 101
commands, 100
consult, 100
current status, 120

debugging, 101
duplicate checking, 87

execution defaults, 76, 101

factoring, 79
future extensions, 120

grouping, 81

help, 100

index generation, 85
indexing delta relations, 86
input/output, 100
intelligent backtracking, 82

lazy evaluation, 88

metaprogramming, 98
multiset semantics, 87
multisets, 97

negation, 81

predicate seminaive, 82
profile favility, 104
program transformation, 76
PSN, 82

rewriting strategies, 76
rewritingno, 78

scc analysis, 82
seminaive evaluation, 82
subsumption checking, 87

trace facility, 103
transaction semantics, 100