CORAL : The Inside Story

Shocking Hacks Revealed

Praveen Seshadri   Shaun Flisakowski   Seymour Hersh

June 17, 1996

This is an attempt at documenting the code for CORAL. In part, it is intended to give a broad idea of the code layout to unsuspecting ingenues who may wander blithely into these dangerous waters. In part, it should be a map of the system, marking the dangerous reefs for the use of experienced CORAL hackers too (though this is a dying breed)[1].

[1]Since much of this document has been written out of memory without direct access to the actual system, there could be inaccuracies, which hopefully are minor. Please feel free to modify, augment, trim or otherwise assault this document. The authors would like to declare that all files, classes, function names and code fragments described herein are purely fictional and any similarity with any real program is unintentional.

Contents

1 Broad Overview: What does CORAL do?

CORAL is a deductive database system. Relations of different kinds are supported:

1. First, there are stored (or *base*) relations.
2. Then there are view (or *derived*) relations. These relations are expressed as rules that specify how they are to be computed. When a query is asked on such a relation, the actual relation is computed based on the rules.
3. Certain commands like print, display_timer, etc. are builtin relations.

The interface presented to the user is a Prolog-style interpreter interface. Commands can be typed in at the prompt, or can be read from a file using the 'consult(*file*).' command. If you want more detailed information about the look and feel of the system, the CORAL User's Manual makes great bedtime reading. This document is more of an implementation primer.

Every stored *relation* is composed of *tuples*. Users are allowed to define any number of derived relations in *modules*. Any number of base relations may also exist. The modules serve as a means of encapsulating the definitions of derived relations. All kinds of relations are accessed via a uniform interface: *get_next_tuple()*. Without going into the details of the exact structure of each tuple, lets look at query evaluation. Assume that the user types the query ?p(X). In other words, she wants to get all values of X, such that a tuple p(X) exists. First assume that p is a base relation: so structures corresponding to each of the tuples are stored in a relation structure. The arity of the relation 'p' is assumed to be one. If the arity were two, the query would have been ?p(X,Y). A tuple itself is composed of *arguments*, which could be of several types.

To answer the query, the system retrieves all tuples from the relation 'p' that match the pattern specified in the query. Formally, a unification is performed between the query and the tuples in the relation, and when there is a successful unification, the bindings corresponding to the query variables (in this case, X), are returned as the answer.

There is the notion of an *iterator* over a relation. When all the matching tuples need to be returned, an iterator is set up over the relation. The interface to a relation is "get_next_tuple". It takes the query pattern (i.e. the argument list), and returns the next tuple from the relation that matches the query pattern. Exactly how this happens depends upon the nature of the relation, and how it is implemented.

So far, what has been presented informally is:

1. CORAL manages a bunch of workspaces.
2. Each workspace is a collection of relations.
3. Each relation is (conceptually) a collection of tuples.
4. Each tuple is a list of arguments
5. Arguments can be of various types. Basic types like integers and strings, as well as complex types like sets, functors and relations are supported.

Some notation:

1. A relation is always specified using lower case letters.
2. A variable always begins with an UPPER CASE letter.
3. All CORAL commands/rules end with a period.

2 System Structure

Here's how the code is laid out:

```
coral           :  the root of the tree.
coral/bin       :  binaries (or links to them).
coral/doc       :  some documentation.
coral/EXAMPLES  :  example programs, test\_suite
coral/lib       :  coral library (used by magic, etc.)
     /help      :  help files for on-line help
coral/include   :  all system-wide header files
coral/src       :  All the source code is under this subtree.
  clients/tcl   :  Tk/Tcl code for Explain and the Coral Client.
  tools/translator:  code for CORAL to C++ translator.
  tools/magic   :  code for the magic rewriting program.
  coral/*       :  code for the main CORAL system.
  class/rdb     :  code for RDB.
```

All C++ files should be compilable using ATT cfront or g++. However, the primary support will be for g++. Machine compatibility should be checked with g++ on Sun Sparc, HP700, Solaris for both Suns and PCs, and Linux.

The main CORAL program calls *init_coral()* to initialize the database system, and then calls the parser.

Files to look at:

- *main.C : top level file with main() program. Invokes yyparse.*
- *CoralInit.C : file that contains the init_coral() function.*
- *Identify.h : Trys to identify the platform type via compiler directives.*

2.1 Read At Your Own Risk

Any code that is specific to any particular machine needs to be included within #ifdef statements. The standard macro definitions are HPUX, SUN, SOLAR, MIPS, LINUX, or OTHER. More are added as necessary. These will be defined in Identify.h, based on machine dependent flags provided by the compiler. They can also be provided by changing the compile time flags in $CORALROOT/CONFIG.

3 User Interface

Currently, CORAL supports a *command-line interface* similar to Prolog. Users can enter commands at the CORAL prompt, or consult a file containing commands. Input can be of two kinds: imperative commands, and declarative view definitions.

View definitions are made within a *module*. The keywords 'module' and 'end_module' delineate the declarative view definitions from the rest of the CORAL commands. In addition to rules, a module may possess a number of *annotations*, which are user specified hints to the system. These hints can be used to specify rewriting techniques, evaluation strategies, and much more.

Imperative statements can take the form of imperative rules or commands. See the appropriate sections for details.

There are a number of system defaults that constitute the execution environment. These are stored in the global variable **exEnv**, defined in *ExecutionEnv.h*. Each module inherits these defaults, and they are used to process the module, unless other defaults are specified explicitly by the use of annotations.

Annotations apply locally to a single module. They do not change the system defaults in the exEnv. System defaults are compiled into the system, they can be changed from the prompt though. Also, there is a CORAL startup file called *.coralrc* which can be placed in the current directory, or in the home directory, with any CORAL commands that should be executed at startup.

Any bunch of commands, module definitions, etc. can be typed into a file, and read into the system by typing 'consult(*filename*).'. This command merely redirects input to come from the file specified, so that it is as though the contents of the file are typed in at the prompt.

Files to look at:

- *ExecutionEnv.C : defaults for execution environment.*
- *.coralrc : CORAL startup file with aliases and user defaults.*
- *main.C : top level file with main() program. Invokes yyparse().*

4 Code Etiquette

There are certain conventions that have evolved as CORAL was being developed. I'll attempt to list some of them here :

- All externally visible functions, classes, and variables should be prefixed with **Cor_**, to avoid naming problems when Coral is used with other systems. This is a temporary measure until C++ *namespaces* are in the standard and implemented reliably.

- We currently use the CVS system for version control. This avoids any problems with breaking locks and contention, and assists programmers in merging their changes.

- Make sure that you run the regression test suites before checking in your modified files. At all times, the checked in version should be consistent.

- When you are checking in a large bunch of files together, put in a detailed description of all the changes you are making. If you are making conceptual changes to how Coral works, add some external documentation in /p/coral/doc as well.

- All debugging-type code should be enclosed within an #ifdef DEBUG. Similarly, there are flags for TRACE, EXPLAIN and PROFILE, and other things, check in CONFIG for the full list. In any external release, the DEBUG flag is assumed to be off. Whenever adding new code, it should be kept in mind that the current CVS version should be ready for release at a day's notice.

- Right now, the system compiles with g++ as well as CC on DECstations, SPARCs and HP 700s. We want it to stay that way. Try not to use features that are specific to a machine, or a compiler. For example, if you want to use C++ templates, I'd rather you worked around it, until template code is stable for CC and g++. Backward compatibility is also important; I get messages from people using AT&T C-front version 1.*.

- It is important that algorithms, etc. be provided as comments. Right now, much of the code is poorly commented, but there's no point adding to the problem. If necessary, append your initials or your name to the comments.

- Different conventions have been used in different parts of the code for function names, variable names, etc. It is probable that you will be extending existing parts of the system. In that case, please try to be environmentally friendly, and stick to the naming conventions used there already. If you are adding a brand new component to the system, try to use some standard approach all the same. Same holds for indenting: much of the code has several levels of nesting: therefore, keep indenting small. Need I say this: keep variable names meaningful, minimize globals and gotos.

- We have learned the hard way that it makes sense to write efficient code the first time through. Too often, an inefficient hack has been thrown in, and has then been forgotten. If you want to do this, put in a comment with a label on it, and before you check in your code, grep for the label, and make sure you fix all inefficiencies. It is a lot more painful to go back later and try to improve stuff (especially if you're not the one that wrote it in the first place).

- Remember that we are still primarily a main-memory system. We **have** to keep our code efficient, especially the run-time code. Gross hacks to this end are undesirable, but the definition of gross changes depending on how much efficiency it will buy.

5 The Parser and Scanner

Users' input is processed by a scanner and parser. The scanner gets its input from the top of a stack of input streams. The stack is initially created from the files specified as command line arguments when coral was invoked. These are placed on the stack, and at the bottom is stdin. At any time, if a file is consulted, all that happens is that the file is opened, the stream and filename are placed on the stack, and the input of the scanner (yyin) is redirected to come from the new input stream.

The parser uses a structure called the parserStruct to store whatever it is reading in. All CORAL constructs(rules, queries, facts, etc.) end with a period. Each parserStruct corresponds to one CORAL construct. However, an entire module is treated as one parserStruct. The parser operates as follows : (a) read in an entire parserStruct (b) process the parserStruct. *Thus there is an imperative semantics in that one parserStruct is processed before the next is parsed.* This requires that an entire module be treated as a single parserStruct. Thus the parse operates in two modes : if it has seen a "module blah." statement, it enters in-module mode. In this mode, fresh statements just keep getting added to the parserStruct. When an "end_module." is seen, it leaves the in-module mode, and the parserStruct is ready to be processed.

The parserStruct has a field that holds a function pointer. This function will be called to process the parserStruct. It is the responsibility of the parser to assign the correct function. This is done using parserStruct.assign_func(), and then it is processed by saying parserStruct.execute() which invokes the assigned function. The functions are defined in *ParseActions.C*.

The parser and scanner also maintain a history mechanism via a history buffer.

Files to look at:

- *ParserStruct.h : structures for passing info from scanner to parser.*
- *Gram.y : parser. Reads in each period-terminated line into a ParserStruct. The parserStruct could represent a query, a command, an entire module, or an annotation.*
- *CArg.y : parser. Reads in data files only.*
- *ParseUtils.C : Contains parser utility functions.*
- *Scanner.l : tokenizing scanner.*
- *ParseActions.C : interface of the parser to the rest of the system. The contents of the* **parserStruct** *global variable are treated as (a) a CORAL tuple to be inserted, or (b) a module to be compiled, or (c) an imperative rule to be fired, or (d) a query, to be executed.*
- *History.C : Manages the history mechanism.*

5.1 General Info

The parser for coral is generated from the file *Gram.y* using **bison**. yacc cannot be used, as the parser has been made re-entrant, which requires the use of the *pure_parser* bison-only directive. The parser produced by bison is placed into the files *Gram.tab.c* and *Gram.tab.h*.

The scanners for coral (also known as lexers) are generated from the file *Scanner.l* using **flex**. Two scanners are produced from this file, one for streams and one for strings (char *). The scanner for streams, called yylex(), is placed in *lex.yy.c*. The scanner for strings, called sslex(), is placed in *lex.ss.c*. The same file can generate two lexers by the the use of the flex modifier "-Pss" which indicates to flex that the lexer generated should use the prefix "ss" on all functions, as oposed to the default of "yy".

By using the definition STRING_PARSE, different macros are generated for the input routines, in this case, getting the input from a character buffer. Two seperate lexers have been created since lexers are small, but the calls to the input routines can be very frequent, an extra test for the type of item being scanned at the input routine would be excessively costly at run-time.

5.2 Scanner

The scanner recognizes tokens in the input stream and returns a symbolic constant indicating which token is found. The coral scanner also places the text of the command in parseEnv->commandbuf and creates and CArg for it if the token has a value (like a number). The scanner basically consists of a number of statements, consisting of an optional state, a pattern, and C code indicating what should be done when that pattern is found in the input stream.

States are changed via the C command: "BEGIN(*state_name*);" (i.e. BEGIN(C_COMMENT);). The scanner always starts in state INITIAL.

The function yywrap() (or sswrap() for the string lexer), is a special function called when the lexer reaches the end of the current input file. This function would normally return 0 to indicate that there is more to parse (and a new file has been provided), and 1 to indicate that parsing is done. Instead, this function currently prepares the next file to be parsed (if any), but always returns 1 so that yyparse() will return after each item on the parse stack is parsed.

For more info about flex and lexing, see the man pages for *flex* and *flexdoc*.

5.3 Parser

The parsing routine for coral, yyparse(), has been modified recently to be re-entrant. A call to yyparse() will process the top item on the stack, taking whatever actions is called for during that parse. When the file or buffer is consumed, it is popped off the stack and the next item, if any, is prepared for parsing. At this point, the parser always returns. This is different from the way it was previously done, in which the entire stack would be consumed by a single call to yyparse(). This change has been made to give access to the full advantage of re-entrant parsing, an item can be put on the stack and parsed without worrying about control not returning at the end of that parse.

The parser for coral can parse either from a stream or a character buffer (char *). These items are placed onto the ParseStack, and then yyparse() is called. To place a file onto the stack, use handle_new_file(), for a character buffer, use handle_new_buff(). Files and buffers can be arbitraily intermixed on the stack, all calls provided handle either case. This duplicity is provided by using a function pointer to choose which lexer should be used for the item currently being parsed, there is one lexer for files and one for strings.

Re-entrant parsing is achieved in coral through the use of a ParseStack, which is a stack of ParserEnv's (parsing envirornments). All information needed for the current parse is stored in the ParserEnv. There is a global variable parseEnv, that always points at the top-most ParserEnv on the stack, allowing faster reference to that item. Among other things, the ParserEnv keeps track of the lexer that should be used for this parse (a function pointer to it), and lexing buffers, to allow the lexers to easily switch between scanning from different sources.

The parse stack can be accessed directly by using the push(), pop(), and put_on_bottom() functions. After using these functions, the reset_position() method should be called to set parseEnv to the new top of the stack, and prepare for parsing. The put_on_bottom() function has been added to make it more convienent to parse a list of files in the order they are listed. These functions are slightly more efficient if several files will be parsed, but in most cases handle_next_file() or handle_next_buff() should be used.

A method parse(), has been added to the ParseStack object to simplify calls to yyparse(). It is basically the same yyparse() loop seen everywhere, but a little more general. A call to it can specify to consume the entire stack, or only parse a certain number of items off the top (default is one).

For more info about bison and parsing, use the *info* command and look under bison. The O'Reily book "Lex and Yacc" is also a good reference.

5.4 Data Parser

There is a second parser that reads only data and table files, (.F or .T file extensions). The bison code for it is in *CArg.y*. This was done for speed reasons, when loading a large data file, there is no reason to create all the parsing structures that are used in the full Coral parser.

The parser understands both the table file and data file formats by ignoring commas between CArgs that are not part of a functor term.

5.5 Addendum

The right way to write parser rules is to pass information between them using the parser stack (yystack) which is accessed by the $ commands. The coral parser has not yet been completely rewritten in this fashion, and many actions associated with the rules directly modify the contents of the global Parser-Struct. I believe that if the code is properly written, the parser actions need never access ParserStruct except at the top level. -Praveen

6 User Commands, BuiltinRelations, and Solvers

6.1 Overview

Many user commands that can be executed from the Coral prompt; Type 'help(commands).' to get on-line help about them. These commands are implemented as *BuiltinRelations*. This is a special subclass of *Relation*, that presents the same get-next-tuple interface. However, get-next-tuple, or insert-tuple into the relation results in a command being executed. For example, at the Coral prompt, if we type 'consult(anc.P).' , this is treated by the parser as the insert of the tuple '(anc.P)' of arity 1 into the relation 'consult'. The insert_tuple method is called on the BuiltinRelation consult, and this results in the file anc.P being consulted by the system. So builtin relations are viewed as commands by the users, but as just another kind of relation by the system.

All arithmetic operations are also builtin relations. Similarly, all set manipulation predicates are builtins, as well as the string manipulation predicates.

Conceptually, these builtin commands are considered by the coral system to be relations, the same as everything else. They are each a type of *BuiltinRelation* and having an associated *Solver* function within Coral that performs the desired operation. By attempting to query or insert facts into these special relations cause the normal Coral mode of operation to invoke these solver functions.

The solvers are all stored in the solvers subdirectory of the the coral source tree, and are split into different files according to their approximate functionality.

An *Operator* is similar, but less general, there are only a few types of operators, each with very specialized forms. Operators are used to provide the functionality of Addition, Multiplication, Bit-Shifting, etc.

Files to look at:

- *Solvers.C*
- *InitBuiltins.C*
- *DBSolvers.C*
- *IO_Solvers.C*
- *Operators.C*
- *SetSolvers.C*
- *BuiltinSymbols.C*
- *All other files that end with 'Solver'.*

7 Rewriting Transformations

Modules with rules defining views may be rewritten for optimization. A number of rewritings are supported. The primary rewriting technique is the *magic sets rewriting*, and its variant, the *supplementary magic sets rewriting*. By convention, view definitions are stored in files with a .P suffix. They are consulted using the *consult* command. If the file *blah.P* is consulted and rewritten, the rewritten output is stored in *blah.P.M* for ease of debugging, and for the user's edification.

The rewriting techniques run in separate processes independent of the main CORAL process. CORAL writes out the module definition to a temporary file, and then invokes the rewriting process on it. The rewriting produces a transformed module in which the rules have been rewritten for optimization. This rewritten module is now consulted.

At the end of a module, the parser calls *parserStruct.execute()*. If the module has the rewriting flag on, then the contents of the parserStruct get written out to a temporary file, and the appropriate rewriting process is called on it. The function that does this is *perform_rewriting* in *Rewriting.C*. The ouput of the rewriting is then fed back into the input of the scanner, and this time the module has the rewriting flag turned off, so that it is not sent for rewriting again.

7.1 Read At Your Own Risk

One thing to note is that when the module is written out to the temporary file for rewriting to occur, all the system default annotations are added to it, though of course, the defaults in the original definition have precedence. The way this is done is : parserStruct has fields for all the possible annotations. These get initialized to the system defaults contained in exEnv (execution environment). Now when the module is parsed, any annotations in the module change the fields in the parserStruct (**not** in exEnv). When the module is written out to the temporary, all the annotations in the parserStruct are written along with it. This is essential, since the rewriting process does not know what the current set of defaults for CORAL are.

Another subtle point is that if there are multiple modules defined in *blah.P*, all their rewritten versions need to be stored in *blah.P.M* (because this is what is meaningful to the user). However, as far as the parser is concerned, it is as though each of these modules was being independently typed in at the prompt. To overcome this, the filename info is stored along with the input stream on the scanner input stack.

Further subtle point: if a module exports a number of query forms, it is treated as though there were a copy of the module consulted for each exported query form.

Yet another subtle point: there's absolutely nothing subtle about any of this. Just checking if you were awake.

Files to look at:

- *Rewriting.C : contains perform_rewriting()*
- *ParseUtils.C : contains EndModuleFunction*
- *Unparse.C : contains routines to print out a module*

8 Imperative Commands

From the CORAL prompt, imperative commands may be specified. They fall into the following categories: a) queries b) assign rules c) append rules d) delete rules.

For the syntax of these, type 'help(rules).' within CORAL. These rules are evaluated using the *same evaluation code* as rules within a module. This is important, because the optimizations performed for rules within a module are applied to imperative rules as well (backtracking analysis is especially important).

A seminaive rule structure is created from the rule typed in at the prompt. This rule is now evaluated using the *SemiNaive::evaluate* method. In order to do this however, some initial setup needs to be performed so that the evaluate code can work for both imperative rules, and rules within modules.

Append rules are treated similar to rules within modules. For assign rules, the contents of the head relation are first deleted, and the rule is then applied. For delete rules, the head relation is first changed to a temporary relation. All the tuples to be deleted are collected in this temporary. Finally, they are deleted from the original head relation.

Queries are converted into rules by adding a special head predicate called *print_<output-type>*, i.e. *print_bindings*. These are builtin relations defined in *IO_Solvers.C*. The print call is added to the query in *ParseActions.C*. Each time a head fact is generated, the code tries to insert it into the builtin relation, and the builtin prints out the tuple instead.

8.1 Read At Your Own Risk

Currently, if an imperative rule is specified that is recursive, it is not handled properly. Coral presently prints an error message indicating the problem. This could instead be handled properly if we were to create a temporary for the head relation.

9 Declarative Modules: Compile Time

This section describes how a module is compiled into intermediate structures. The module under consideration has been passed through the rewriting stages, and has been consulted by the parser. As part of the EndModuleFunction, the function *interpret_module* is called. This resides in *Compile.C*. All compile time routines reside in *Compile.C* or *CompileRule.C*.

Each module is compiled into a QFModuleData structure. There are two subclasses of QFModuleData, one for materialized modules, and one for pipelined modules. Here we consider the materialized modules only. Look for pipelined modules in the section on pipelining. For materialized (i.e. bottom-up) modules, a QFMaterializedModuleData structure is created. Since its name is so long and painful, I just refer to it as a QFM in this section.

Since a QFM corresponds to a module, it is associated with a structure called a Table. This represent a table of the relations used by the module. Whenever the module code needs to access a relation, it is made to access an offset in the table. Thus at run-time, appropriate relation pointers can be plugged into the table, and voila, the relations get accessed. So as a first step, a Table structure is created for the module. Also, the ParserStruct is processed and RuleInfo structures corresponding to each module rule are created. Each RuleInfo struct is supposed to correspond to one syntactic rule defined in the module. This is as opposed to a seminaive rule of which there could be several for each syntactic rule (remember, deltas, non-linear rules, do-re-mi ...). The RuleInfo is merely a data structure representing a syntactic rule entity. All the RuleInfos of a module are grouped into a RuleInfoArray struct.

Just a step back to get the logical view of things first : what is a module compilation supposed to do? (this is the DAILY DOUBLE). The predicates in a module need to broken into strongly connected components or SCCs. Within each SCC, the rules corresponding to the predicates need to be prepared to be evaluated in a seminaive fashion. This is exactly what happens when a module is compiled. Stepping through the code is pretty simple (if you believe that, talk to Scott Sterner). But seriously, compiling a module involves:

- performing an scc analysis on the predicates in the module. The list of predicates is available as a field of the ParserStruct. The scc analysis uses some static structures(SccWork, etc.) declared inside *Compile.C*. The algorithm used is Tarjan's algorithm (at this point, we expect the reader to know that its complexity is O(N), or else call toll free 1-800-COMPLEX). All it does is provide each predicate with an scc number corresponding to the topological numbering of the sccs.

- Now for each scc, an SccInfo structure is created, corresponding to the scc. Each SccInfo struct contains an array of seminaive rules, as well as a number of additional fields and arrays that are used for optimizing execution. While most of these are created at compile time, it makes sense to explain them in conjunction with the explanation of the run-time execution. The function *CreateSNStructs* performs the creation of the seminaive structures. This function looks at each predicate in the scc and for the rules defining that predicate, creates seminaive structures. Finally after the SccInfo struct has been completely created, it is added to a list of SccInfos that the QFM contains. This list is maintained in topological order of sccs (note that during execution, the lowest scc needs to be executed first).

- Every predicate in an scc has a bunch of rules defining it. Some of these rules may be recursive.[2] Every SemiNaive rule has a pointer to a corresponding RuleInfo (or is it an offset into the RuleInfoArray) that specifies the syntactic rule. Also, it keeps information about the join order in a predOrders array. This array contains a permutation of the join order, and all access to the predicates of the seminaive rule should be made through this array. The current join order uses the heuristic that the delta relation will be small, and hence is moved up front. Further (I think I put this in), we try to move literals that have common variables with the delta ahead of those that don't. Take a look at comments in the code that handles this in initPredOrders(). The SemiNaive rule also contains backtracking information to be used at run-time. This information is vital for efficient performance, and more details of this are in the section on backtracking. There is also a function that generates the appropriate index annotations for the efficient execution of the rule. These annotations are appended to the index annotation list of the QFM. The comments in the code explain the logic behind the index creation.

- Wherever possible, we try to perform stuff at compile-time instead of at run-time. This is why you will find a number of arrays called *-offsets. These are pre-computed offsets into lists or arrays of stuff. For instance, there is a pred_offsets array in the SemiNaive structure that contains for every literal, the offset of the corresponding predicate in the Predicates array. If you don't understand the computation of these array right away, don't worry. You will when you have to. However, this stuff is very important. Pipelining was speeded up by almost 25% by moving an offset computation from the run-time code to the compile time code. If you're competing with Quintus, **nothing** is cheap.

- Finally the QFM is complete, it is inserted into the DerivedRelation table as the ModuleData structure corresponding to the exported query form. The solver routine also has to be entered along with it. We have the same solver for all bottom-up modules, and its called (quaintly) "do_query". More about that in the section on run-time stuff.

9.1 Read At Your Own Risk

Compile.C and *CompileRule.C* evolved from what used to be *interp.C*. The third branch is *Execute.C* which contains run-time execution code.

This is only for Coral Gunslingers of the High Speech. The Table was originally called a Frame (the name was subsequently changed as a humanitarian gesture, after Raghu started chairing AI sessions). However, as a result of this, most of the code has declarations of the form 'Table *F'. I'm sorry about that, but rescuing Raghu seemed more important at the time.

[2] Pop Quiz: If there is more than one predicate in an scc, can there be a predicate in the scc without a recursive rule?

You might ask: why is there a separate syntactic structure (the Rule class) used for rules in the ParserStruct and rewriting, and a separate structure (the RuleInfo class) used at compile-time. Good question: integrating the two is your next job.

Remember that the same code used to compile and execute SemiNaive rules is also used for imperative rules. The imperative rules are just massaged to create SemiNaive rules, and then the same compilation occurs. It is your responsibility if you change this code to ensure that it works on both kinds of rules. In order to be able to figure out where the code has been invoked from, I have used the field of the SemiNaive structure that points back to the enclosing SccInfo. If the code is being called during a module compilation, then this pointer will be non-NULL and point to the SccInfo. On the other hand, for an imperative rule, there is no SccInfo, and no QFM, and so the pointer is NULL. This is checked for example to determine whether to generate index annotations (meaningless for an imperative rule).

10 Pipelined Modules: Compile Time

A pipelined module is one that has the pipelining annotation turned on. It should be evaluated in a top-down, left-to-right fashion as Prolog does. Thus when interpret_module is called, it is able to recognize that this flag has been set in the ParserStruct, and invokes *interpret_pipelined().* This function is defined in *Pipelined.C* along with all the other code for compiling and executing pipelined modules. The compilation of a pipelined module results in the creation of a QFPipelinedModuleData structure that represents the module. In some respects, this is similar to the compilation of a bottom-up module. However, there are no SCCs. Instead of SemiNaive rules, structures called PipelinedExecInfos (or PEIs) are created. These PEIs contain backtracking info, pred_offset arrays, etc. However, since the mode of execution is fundamentally different, many of the structures are different. The compilation of pipelined modules is a lot simpler and cleaner than bottom-up modules, and is relatively easy to understand from the code.

Files to look at:

- *Pipelined.C*
- *Pipelined.h*

10.1 Read At Your Own Risk

At this stage, we have reasonable confidence in the pipelining code, and it is probably as lean and mean as it is going to get. So I'm leaving this section without much detail, since there should not be much reason to modify this code (also I haven't looked at it in a while, and don't remember it too well).

One problem that was encountered was that pipelining has to dynamically allocate memory since there may be recursive invocations of rules within a module. Actually, a new PEI is created for each invocation. At some point, Vish tried to see if he could preallocate at least one copy of each PEI. However, this was dicey, and though most of the code is already there, it has been commented out, because there was a bug in it. This might be something worth considering.

As a rule, pipelined modules should not have memory leaks. When a module computation is complete, the allocated memory should be totally reclaimed.

If you need more information about this, you might want to check in the Bratko Prolog book, as it discusses Prolog evaluation.

11 Relation Class Hierarchy and Implementation

There are currently three major types of Relations, all derived from the base class *Relation*:

- Storage Relations that *store* data, usually explictly.
- Derived Relations that store rules, from which the data can be *derived*.
- Builtin Relations that implement Coral's *builtin* commands.

Storage Relations are represented by the StorageRelation base class, there are quite a few relations of this type, some examples being: HashSimpleRelation, SortedRelation, and SockRelation. If you're considering making a new relation type, StorageRelation is almost certainly the base class you want to derive from.

There are currently only two types of Derived Relations: DerivedRelation, the base class, and PipelinedDerivedRelation. It is unlikely that any new Derived Relation types will need to be added in the future.

If you're considering adding a new Builtin Relation type, what you really need is probably a *Solver*; read the section on Solvers right away.

Files to look at:

- *Relation.h : The base class.*
- *StorageRelation.h : For data.*
- *DerivedRelation.h : For rules.*
- *BuiltinRelation.h : For specific commands.*

12 Backtracking Analysis

Backtracking is a fundamental concept to the way that CORAL solves logic programs. Rules are evaluated from left-to-right, when a literal fails, this particular instatiation of this rule will fail, and so CORAL moves backward in the rule (to the left), to attempt to find new bindings for variables that may still succeed. A naive strategy for moving backward is to always move back to the previous literal. Using this strategy, all combinations are attempted.

12.1 Overview of backtracking

To avoid further evaluation where it cannot possibly be fruitful, CORAL performs *backtracking analysis*. The main concept here is that the bindings for the literal that failed must change before it can succeed, therefore in many cases, you could skip back multiple literals at once. How far to skip backward for a literal that fails can be in fact be computed at compile-time.

A similar concept can be applied when a rule succeeds (when duplicates are being eliminated). Add in an understanding of *cuts*, which disallow backtracking that crosses them, and the majority of the current backtracking implementation should be clear.

A more recent addition to CORAL's backtrack analysis, *Binding-Time Analysis*, is due to the fact that non-ground facts (facts containing unbound variables) are no longer allowed. With this restriction, CORAL can determine at compile-time *how* each literal will end up being evaluated, and can call a specialized function to perform this directly (as opposed to figuring out what to do with a literal at run-time).

Remember that backtracking over more than a single literal is basically only an optimization, the answers obtained should always be the same as if obtained by the naive backtracking strategy mentioned above.

12.2 Specifics

- First-Failure Backtrack Point:
  Where to backtrack to in the rule when a right-hand-side literal fails the first time it is reached. Execution should backtrack to the rightmost point that generates fresh bindings for it.

  ```
  p(X,Y,C) :- q(X,Z), r(Z,A,B), s(Z,C), t(A,B,C).
  ```

  If `s(Z,C)` fails the first time it is reached, execution should backtrack to `q(X,Z)`, to generate a new binding for `Z`. (If `s(Z,C)` has already returned successfully, you would use the next-failure backtrack point.)

- Next-Failure Backtrack Point:
  Where to backtrack to in the rule when a right-hand-side literal fails after having been reached after it has suceeded at least once. In this case, execution should backtrack to the rightmost point that generates fresh bindings for the literal or any literal after it in the rule execution order.

  ```
  p(X,C) :- q(X,Z), r(A,B), s(Z,C), t(A,B,C).
  ```

  If `s(Z,C)` fails on a get-next, execution should backtrack to `r(A,B)`, not to `q(X,Z)`. Note that if the head literal is allowing duplicates (i.e. a multiset relation), then we would always backtrack to only the previous literal as we need to allow the entire rule to succeed as many times as possible.

- Success Backtrack Point: Where to backtrack to when the entire rule has succeeded. We want to backtrack to the right-most literal that generates new bindings for the head literal (assuming duplicate elimination)

  ```
  p(X,Y) :- q(X,Z), r(A,B), s(Z,C), t(A,B).
  ```

  After the rule has suceeded, we would backtrack to `q(X,Z)`, which can result in a new binding for `X`. If the head is allowing duplicates, we would only backtrack to the previous literal, `t(A,B)`.

The backtracking analysis has been extended to also perform *Binding Analysis*. A variable is bound at the first literal it is seen in. This information is used to perform *matching*, *assignment*, or an *equality check* in order to avoid performing a full *unification* against facts in the database (which are guarenteed to be ground). To this end, the *SemiNaive* structure stores an array to identify what type of unification should be used when unifying against a term known to be ground; for other cases, the hint in the array is simply ignored, and full unification is used.

Files to look at:

- *BindTime.[Ch] : Backtracking and BindTime Analysis.*
- *BacktrackInfoArray.h*
- *CompileRule.C*
- *SemiNaive.h : Stores the backtrack points, and unification functions.*
- *HashIndex.C : Unification against ground facts.*

13 SCC - Strongly Connected Components

Considering the rules within a module, notice that interdependancies sometimes exist between some of the relations. A *dependancy graph* is a directed graph showing the dependancies between relations. An directed arc exists in the graph pointing from p to q iff relation p directly depends on q, i.e. q is mentioned in the body of a rule defining p.

A *strongly connected component*, or *SCC*, is a region of this graph that is as small as possible such that there are no dependancy cycles between SCC's. In other words, an SCC encapsulates all cycles between rules within itself. Within an SCC, all the relations, because of their interdependance on one another, must be solved simultaneously.

The SCC's form a *directed acyclic graph*, or dag, and the SCC's can be evaluated independantly of one another, in a bottom-up fashion. Each SCC is evaluated individually using *SemiNaive Evaluation*, which is discussed in the next section.

Files to look at:

- *Scc.h*
- *SccInfo.h*

14 SemiNaive Evaluation

SemiNaive Evaluation is the method used to evaluate an individual SCC. Because of the ordering of the evaluation of SCC's, when this SCC is being evaluated, all relations outside this SCC that this SCC depends on have already been computed. SemiNaive evaluation is a type of fixpoint evaluation that can solve a set of rules that may contain inter-dependancies, or recursion. Coral has marked each rule that is involved in recursion in the SCC-analysis phase.

The evaluation starts by firing each rule in the SCC that is not marked as recursive. These rules will already have their dependancies satisfied due to the ordering of SCC evaluation, so there is no benefit to evaluating these rules more than once.

Now the recursive rules are evaluted, we iterate through the recursive rules, with each one firing as many times as it can with the current information available. We continue to iterate over these rules as long as new facts are being created.

Each recursive relation being evaluated is conceptually split into two parts (via an *RMark*), there is the *old* part, which are the tuples computed in previous iterations, and the *delta*, which are the tuples that we have computed in this iteration. At the end of each pass, the tuples in the delta are added to the old part of the relation, and the deltas are emptied. If the deltas for all relations within this SCC are empty at the end of an iteration, we have reached a *fixpoint*, and evaluation can stop, as further evaluation will not discover any new facts.

The deltas are implemented through two *RMarks*, (Relation Marks), for each relation. An RMark can be considered to be a conceptual mark seperating the two portions of the relation (when visualized as a table of facts). One RMark seperates the old part from the delta, with the other marking the end of the delta. If the user has specified (via an annotation in this module) that deltas should be indexed, the delta is actually implemented as a seperate relation which is then indexed. Otherwise, the delta is implemented as a simple list of tuples that are not yet added to the main relation, though all the low-level primitives (such as a scan over the relation) consider them to be part of it. At the end of each iteration, the *move_all_entries()* function is called, and these tuples are moved from the delta into the main relation.

Files to look at:

- *SemiNaive.h*
- *SemiNaiveEval.C*
- *SccInfo.C*

15 Lazy Evaluation

When the user asks a query on a relation, and CORAL's interactive mode is turned on, a single tuple is shown to the user, and CORAL requests if they want to see more. When *Lazy Evaluation* is turned on, CORAL suspends the top level query as soon as an iteration has produced some facts. The state of the not-yet-completed computation is stored in a structure, and a pointer is placed in the iterator. If the user requests more facts, the get_next call for the iterator will continue the computation when it runs out of tuples.

Currently, this is an annotation, though it should really be a run-time setting, as it has no effect on compilation, or on the evaluation of any except the top-level SCC.

15.1 Read At Your Own Risk

Previously, the lazy evaluation code used the name *Eager* in many places, apparently as a synonym for *Lazy*. I suspect this was a left-over from a change to the interface that never fully propagated through the code. I believe I have eliminated all references to this in both code and comments, but be on your guard, as the Red Menace doesn't sleep.

16 Unification and Binding Environments

Given two structures that may contain variables, *unification* determines if the two structures can be *unified* (made equal), by some instantiation of variables.

A related concept is *matching*, that is unification for which one of the structures is know to be ground (contains no variables).

A *BindEnv* is a binding environment, for a set of variables it indicates what they are each bound to. A *Term* is a argument (CArg), along with a BindEnv that may bind some of the variables within the argument. A BindEnv maps each variable to a Term, so these structures are mutually recursive, allowing for multiple levels of binding.

A *Trail* is a stack of BindEnv's, which can be used during evaluation, and then popped back to a fixed *StackMark* to return the environment to its previous state.

When we are interested in unifying against a fact from the database, we can assume the database fact is fully ground, and so depending on the form of the first argument, we can decide at compile time what action should be taken when we attempt to unify at run-time:

- If the argument is a variable, and it has not yet been bound in this rule, then we can *assign* (bind) this value to the variable.
- If the argument is a variable, and it is already bound, we want to check for equality at this point.
- If the argument is not a variable, but is not constant (it contains variables), then we must use *matching*.
- Otherwise, the argument must be constant, and we can simply check for equality.

Files to look at:

- *BindEnv.C*
- *StackMark.h*
- *Term.C*
- *Trail.C*
- *Unify.C*
- *HashIndex.C*
- *BindTime.C*

17 Handling Non-Ground Terms

Non-ground terms have been eliminated from the current version of Coral. Some remanents of them still exist in various parts of the code, and you will see an occasional comment mentioning them.

The remanents should be found and removed, if we add non-ground terms back in at some point, it should be done in such a way that it can be completely ifdef'd out with no effect on the speed of the ground-only version.

18 Indexing Mechanisms

Indexing is done via hashing by default for relations in main memory, different index forms are created as needed before evaluation. The current forms of indexes that currently exist for a specific base relations are displayed in the output from the 'list_rels.' command. The form indicates the arguments that are considered bound for an index, and thus contribute to the hash value for that tuple in that index. Chaining is used for collisions. The system watches the total number of tuples within a relation, when the average number of entries (statistically) in a bucket gets too high, the index doubles in size, and all entries are immediately rehashed.

Every relation gets at least one "index", as it is the only place to store tuples. This default index is completely bound (i.e. every argument contributes to the hash value used).

If external Exodus relations are used, Exodus handles the index creation and management as well.

19 CArg - The Coral Argument

A Coral Argument, known as a *CArg*, is Coral's basic type; it can hold a number, string, list, variable, set, or other implemented type. The CArg class is implemented as a C-style union, rather than a C++ class hierarchy, for efficiency reasons. This may seem like a minor savings, but this class is used extreemely heavily, as at the lowest level it is the basic container for all data that is stored in the system.

The CArg union has been organized to minimize *padding*, and to consume at most 16 bytes on all systems Coral is available on. *Padding* is empty space between the fields of an allocated structure, that is added by the compiler to preserve alignment.

Be careful when modifying this structure, and check with *sizeof()* to ensure minimal padding is present.

A quick review of C's alignment / padding rules, as they relate to this class, may be helpful:

- On many systems, simple types (such as int, double, etc.) are required to be aligned on a boundry dependant on their size. For instance, on a Sun, an 8-byte double must be aligned to an 8-byte boundry.
- The alignment restriction mentioned above remains the case when the simple type is stored within a struct or union. The compiler adds padding to the struct as necessary to ensure this is the case.
- The rule for structure padding is that the structure will be padded out to the size the type would occupy as an element of an array of such types. The alignment requirement for a structure type

will be at least as stringent as for the component having the most stringent requirements. In other words, a structure that contains an object requiring alignment, will always have a size that is a multiple of that alignment. So for instance, if a structure (like CArg) contains a *double* which requires 8-byte alignment, its size after padding will be a multiple of 8. Currently, the size of CArg is 16 bytes, adding a single char variable to the end of the structure would change its size to 24, not 17.

- Fields are placed in memory in the same order that they are listed in the structure declaration. Considering this, and the above rules, it becomes obvious that not all field orderings are equally efficient in their use of memory. For example, assume an architecture that requires 8-byte alignment of doubles, and 4-byte alignment of ints, and the structures:

```
struct lean {                 struct fat {
  int    i1;                    int    i1;
  int    i2;                    double  d;
  double  d;                    int    i2;
};                            };
```

  The lean structure requires 16 bytes, and contains no padding, while the fat structure consumes 24 bytes, and contains 4 bytes of padding between *i1* and *d*, and another 4 bytes padding at the end of the structure (after *i2*).

Most of the types that can be held in a CArg are fairly self-explanatory, I would like to comment on a few things, just to help fill this section in a bit:

- **Symbols** - There are two symbol types for storing strings, SYMBOL and TINY_SYMBOL. The SYMBOL type stores the actual string in a *StrRefTable*, and keeps a pointer to this reference-counted string. The TINY_SYMBOL version is only for short strings, which can fit within the CArg itself, and no reference counting is used for these.
- **Functors** are implemented as a name and an arglist. Move along, there's nothing to see here.
- **Lists** are implemented via a *CArgStack*, which holds the items of the list in reverse-order. This is done so that the stack's push/pop operators correspond to cons/car for the list.
- **Expressions** use a CArg for the operator node, with pointers to the CArg operands, the operands can be expressions as well, allowing expression trees to be built.
- **Sets** are implemented as a CArg holding a pointer to a unnamed Relation. Duplicate elimination is turned on for sets, off for multisets.
- **Aggregates** are represented as Grouping Args early on, and as an Aggregate Result during evaluation. Look at the section on Aggregates for more information.

Files to look at:

- *CArg.C*
- *CArg.h*
- *CArg.i Inlined methods of CArg class.*

- *CArgUtil.C*
- *Stack.h*
- *ArgList.h*
- *StrRefTable.h*

20 Handling Aggregates and Negation via Ordered Search

Coral supports aggregation in the head of rules, the functions currently implemented are, **min**, **max**, **any**, **sum**, **prod**, **count**, **avg**, and simple grouping. There are also versions of each (that make sense) with the *distinct* suffix, indicating that duplicates should be eliminated before the aggregation is performed. There are also operators, so that each of these functions can also be performed on (multi)sets in the body of rules; note that the implementation of the operators is (currently) completely distinct from the aggregation part of the code.

Each form of aggregation is seen as a special type of argument. It is parsed as a normal functor, then converted to the special form in *create_functor()*.

When an insert is attempted on a relation that is performing some type of grouping, *insert_grouping()* is called to perform the insert. If this is the first tuple inserted with unique values in the non-grouping columns, it is inserted as normal; otherwise the *update_grouping()* function is called to update the **stored** tuple **in-place**. The reason this works (it was quite a shock), is because this tuple is guarenteed to not be accessed until this aggregation is complete as long as the *monotonic* annotation is off. The exact reason for this reason is not entirely clear to me, even after hearing Raghu explain it a few times.

If either aggregation or negation is used in a module, *Ordered Search* evaluation is used. There is a paper on the Coral web site detailing the theory and implementation of ordered search, but the code within Coral is basically write-only. The similarity between aggregation and negation is that all tuples in a grouping, or of the form being negated, must be known to have been computed, before the negation/aggregation can be evaluated. i.e. If the literal 'not p(2,Y)' appears in a rule, we must have already computed enough of p to determine that no tuple of p with a first argument of 2 exists, or will be still be computed.

Files to look at:

- *CArg.[Chi] : Definition of Aggregation CArg types.*
- *CArgUtil.C : update_grouping function.*
- *ParseUtils.C : create_functor function.*
- *StorageRelation.C : insert_grouping function.*

21 Client/Server

21.1 Overview

A Coral client connects to the Coral server. At this point, the client is the user *nobody*, the default user. The client may now use the *login* command to make new users, obtaining a user_id (good for this session only) for each sucessful login. A limited number of users may log in, after which logins fail until a user is removed by logging out. The client can switch between its user's using the *user* command. Each user has an independent envirornment, and a user may not resume another user's suspended queries. A client remains the same user until they explictly switch with the *user* command.

Presently, the Coral server switches between connections non-preemptively. After a command (or a few) a client should execute the command *yield*, allowing other clients to connect and process commands. This is only a temporary fix, eventually (I hope) the server will switch between the clients itself.

When Coral is not in server mode, it sets itself up with a single "connection" to stdin/stdout, it has the same interface as Coral running in the server mode.

Only full commands should be sent to the server (ending with '.'), partial commands will cause the server to block inside its parsing routines, waiting for the rest of the command.

who is the only command that allows a user on one connection to see the existence of users on other connections. All other commands assume the user is referring to their own connection, which provides security by offering no way in which to name users on other connections. The *list_sus(all)* command, which lists all suspended queries, has also been modified to list only the queries of the current connection.

21.2 Specifics

To provide a seperate envirornment for each user on each connection, new envirornment data structures are created for each user or connection as needed. To simplify their use, and to remain consistent with their previous use, global pointers are provided to the data structures of the current user, these are automatically updated during the context switch between different users and connections:

- *exEnv : The execution envirornment of the current user.
- *parseStack : The ParseStack of the current user.
- *parseEnv : The ParserEnv of the current user (top of parseStack)

Some new pointers have also been added:

- *curUser : The current User.
- *curConn : The current Connection.
- *clients : An array of all connections (ConnectArray).

NOTE: The variables *curUser*, *curConn*, and *clients* are valid even if Coral is not running in server mode. This eliminates the need to check what mode Coral is in. In most cases, you can simply do things the "server way" and the stand-alone coral will probably work as well.

stdin, *stdout*, and *stderr* should not be used for anything except debugging information, stdout and stderr will (at best) be going into a log file for the server. The variables *exEnv->in_strm*, *exEnv->out_strm*, and *exEnv->err_strm* should be used instead, these connect back to the current client. In truth, *exEnv->in_strm* should never be used without checking that coral is not in server mode, otherwise the server will block until the question is answered.

21.3 Command Reference

Note: All commands which take a user_id also take the login name of a user. While the login name is not necessarily unique, it probably will often be in practice. In the case it is not unique coral will use the first user that has a matching name. Clients should use the numerical user_id if possible to avoid forcing coral to search through a list of user names.

- **login**(*user name*, *password*)
  Echos back a user id and the password, which the client should save. The user id is used in many of the commands as an identifier. The client will need the password again if the user attempts to run explain. If the current user is *nobody*, login also switches to the created. Note: currently coral does not check the password.
- **logout**() or **logout**(*user_id*)
  If a user_id argument is specified this command logs that user out. With no argument it logs out the current user (and does not disconnect), allowing the space for this user to be reclaimed for a new user on this connection. If the current user of a connection logs themself out, the connection switches to user *nobody* as if the command *user(0)* had been used. When a client disconnects, any users logged in on that connection are automatically logged out.
- **quit**()
  If this is the top level, logs out all users on this connection and closes the connection. If not at the top level, it has the normal coral behavior (aborting the current item at the top of the parse stack).
- **user**(*user_id*)
  Switch to the user specified by user_id. If the user_id is not valid, gives a warning message, and remains as the current user. The user_id of 0 is the default user for a connection, known as *nobody* and can always be safely switched to.
- **who**() or **who**(*conn_id*)
  List all users logged in on all connections. If a connection id (*conn_id*) is specified, lists only the users on that connection.
- **sync**() or **sync**(*number*)
  This is a request from the client for synchronization at this point, a number or string may optionally be specified for validation of the sync response.
- **yield**()
  Yield the coral server. This command releases the coral server to check for input on other connections.

- **whoami**()
  Print the current user's login name.
- **quit**()
  Logs out all users on this connection and disconnects the client.

21.3.1 Suspension Commands

- **answer**() or **answer**(*number*)
  Print out the current user's number of answers per query, this is the number of answers that a query will give before it suspends. If a number is given, sets the number of answers per query to that number. answer(0) chooses the system default number of answers per query.
- **kill**(*resume_id*)
  Kills a suspended query by id. A user can not kill another user's suspended queries.
- **list_sus**() or **list_sus**(*user_id*) or **list_sus**(all)
  List the suspended queries for the current user. With a user_id as an argument, lists the suspended queries for that user. With an argument of "all", lists all suspended queries for this connection. Listing the suspended queries for other connections is not allowed.
- **resume**(*resume_id*)
  Resume a suspended query. A user can only resume their own query.

21.4 Client Protocol

The requests presently being sent to the server by the coral client are of this form:

```
<any coral commands>
...
sync(n).
yield.
```

The sync command passes back a number or string to allow the detection of the end of this answer. Something like this is needed when the user is allowed to give the coral commands directly. If the user would give multiple commands in one line it is difficult for the client to figure out when the commands are "done" without this command.

Additional commands are available to the client for dealing with users and suspended querey. These are: login,logout, user, who, answer, resume, list_sus, kill, sync. These are discussed in detail in the section above.

The server sends back the same info that it would print to the screen when running interactively. Special messages for the client are preceeded by a 5 character identifier (ending with a colon). Any message without a prefix is considered to fall into the "MESG:" catagory.

These fixed length identifiers are intended to be easy to parse, keeping the clients simple.

- OKAY:
  A command finished, the client can prompt the user for more input now.

- ACK_:
  Acknowledge your presence, by sending the server a (possibly empty) command.
- CERR:*string*
  Usually an error message, indicating a command failed.
- MESG:*string*
  An answer to a query, or anything else that should be shown to the user.
- USER:*number*
  The numerical user id assigned to this client's user.
- LOGN:*string*
  The login of the user.
- PSWD:*string*
  The password of the user.
- R_ID:*number*
  A resume id.
- PRMT:*string*
  Indicates the prompt may be changed to the string given. This is only a suggestion, the client may ignore it if it wants.
- NEWS:*string*
  The name of the current workspace (it changed).
- SYNC:*number*
  The number returned is the same one used in the *sync* command for verification.
- EXPL:*conn_id user_id*
  This info is needed by the *explain* program, which the server wants you to start now.
- EXIT:
  The server is requesting the client disconnect now.

There are constants for the above defined in *ClientProtocol.h*.

There is also a short paper about creating clients (written by Wei)
in `/p/coral/doc/client_programmer_guide`.

Files to look at:

- *ClientProtocol.h : Header defining the client/server protocol strings.*
- **ConnectArray.h : The array of connections.*
- *Connection.[Ch] : A single connection.*
- *User.[Ch] : A single user.*
- *CoralStream.h : A stream to talk over (in general).*
- *SockStream.h : A special stream for use with sockets.*

- *WodiStream.h : A special stream for use with Wodi.*
- *PvmStream.h : A special stream for use with PVM.*
- *CoralClient.C : A simple Coral client.*
- *clients/tcl/Coral : The* **real** *Coral client.*
- *doc/client_programmer_guide : A guide for creating clients.*
- *doc/client.README : A user's guide to using Coral in client/server mode.*

22 C++ and Tk/Tcl Interfaces

There is a document explaining what is considered to be part of the "C++ Interface" in $CORAL-ROOT/doc/Interface. It is mostly the same classes CORAL uses internally, but simplified, being an interface that we should avoid changing in the future. I am loathe to repeat that information here, and would introduce the possibility of having the two versions out of sync. I strongly recommend you look it over if you are getting started with Coral though, as it gives a through overview of many of the low-level classes.

The C++ Interface has been changed from being a seperate set of functions to being a watered-down version of the classes Coral uses internally, many of the previous (depreciated) functions are still present in *Interface.C*.

The Interface document explains the usage of the Coral classes, **CArg**, **ArgList**, **CArgStack**, **Tuple**, and **Relation**. Some of the more complex methods of these classes have been purposely omitted from the document as a normal user of Coral would probably not find them very useful. If a user determines the time has come for them graduate to a Coral hacker, they can read the source like the rest of us.

The class **ScanDesc** is added to the C++ Interface to provide a simple interface for scans over relations, as opposed to the more complex **TupleIterator** interface. In particular, the recently added open_scan() function makes opening a complex scan fairly easy for a beginner.

The Tk/Tcl Interface is even simpler. It is defined in `clients/coralsh/coral/Coral.S`.

Both the C++ and Tk/Tcl programming interfaces allow normal Coral code fragments to be embedded in the source code, executing them when they are reached. For the C++ Interface, the users source code (in a file prefixed with '.S') is first passed through the *translator* program, which produces C++ source with the embedded Coral removed. The translator uses the **ExecutionList** class to score each embedded section; the place in the source that was occupied by embedded Coral is replaced with code to compile (if it hasn't been already) and execute this ExecutionList. Since the translator predates Coral's ability to parse from strings, the Coral source is being written to a file, and parsed from there.

Files to look at:

- *doc/Interface : A User's Guide to Coral's C++ and Tcl Interfaces.*
- *tools/translator/* : The source code for the translator.*
- *Interface.[Ch] : ExecutionList and ScanDesc class definitions.*

- *CArg.[Chi] : User access to this class is allowed.*
- *ArgList.[Ch] : User access to this class is allowed.*
- *Tuple.[Ch] : User access to this class is allowed.*
- *Stack.[Ch] : User access to this class is allowed.*
- *Relation.[Ch] : User access to this class is allowed.*
- *Coral.S : Tk/Tcl Coral Interface.*
- *Coral_Utils.S : Some utilities for use by Coral.S.*

23 Memory Reuse

During the evaluation of a query, the current implementation of Coral allocates and frees many small chunks of memory. The standard C memory allocation functions, malloc/free, are prone to fragmentation and (because of their generalality) are rather inefficient for small chunks.

To help minimize these problems, Coral reuses small chunks of memory rather than immediately freeing them to the system. This reuse is only performed if Coral is compiled with the REUSE_MEMORY compile-time flag. When this definition is set, the most heavily-used classes have their new/delete operators overloaded to use our own memory allocator, the *Entity_Array* class.

The *Entity_Array* only operates on chunks that are multiples of 8, all requests for memory are rounded up to the next multiple of 8 for allocation purposes (ceil(size/8)). When memory is freed, a pointer to the the chunk is stored in a global array, with the index being floor(size/8)-1. Within each array entry, the memory chunks are chained togather using the actual chunk to store the pointer to the next one in the chain. When a chunk of memory is requested, the array if first checked to see if we have one that size already, if not, the standard allocator is used to obtain one.

23.1 Read At Your Own Risk

The allocator operates faster than standard malloc/free in most cases, but there are some drawbacks to its use. If the free lists are swapped out to disk, we will cause cache misses by allocating it, we use LIFO ordering of the chunks to attempt to minimize misses.

Files to look at:

- *Entity.[Ch] : Coral's memory allocator and storage.*