Query Processing

For better understanding of the query process, we will use the following query as example:

assign a; call c; while w; variable v;

Select <c.procName, a> such that Parent\*(w, a) and Uses(a, v) pattern a(\_, \_”2\*y + 3”\_) with c.procName = v.varName (1)

## Query Validation

Query validation process is handled by the QueryPreprocessor (QPP). QPP receives a query in string form, and tries to read from left to right until the end of the string. During reading, if it meets a block of symbols which fits to a PQL grammar rule, the QPP breaks the block apart from the string and validate that block using the corresponding grammar rule. The reading-breaking-validating process continues until no more new block of symbols can be found; thus, this is a top-down approach.

For each undivided block of symbols, QPP will validate it using the corresponding grammar rule. If any rule is violated, the QPP will sent back information to QueryRepresentation (QR). Otherwise, the information of the block will be saved into QR, either in SymbolTable or QueryTree.

For the example query at (1), the validation process is shown as below:

(each time a block of symbol is found, it is enclosed by a square bracket)

* 1st iteration:

assign a; call c; while w; variable v; Select <c.procName, a> such that Parent\*(w, a) and Uses(a, v) pattern a(\_, \_”2\*y + 3”\_) with c.procName = v.varName

* 2nd iteration:

[assign a;] [call c;] [while w;] [variable v;] [Select <c.procName, a> such that Parent\*(w, a) and Uses(a, v) pattern a(\_, \_”2\*y + 3”\_) with c.procName = v.varName]

* 3rd iteration:

[assign a;] [call c;] [while w;] [variable v;] [Select [<c.procName, a>] [such that Parent\*(w, a)] [and Uses(a, v)] [pattern a(\_, \_”2\*y + 3”\_)] [with c.procName = v.varName]]

* 4th iteration:

[assign a;] [call c;] [while w;] [variable v;] [Select [<[c.procName], [ a ]>] [such that [Parent\*(w, a)]] [and [Uses(a, v)]] [pattern [a(\_, \_”2\*y + 3”\_)]] [with [c.procName = v.varName]]]

And so on. For each block of symbols, QPP will call the function corresponding to the grammar rule for this block, such as preprocessDeclaration() for [assign a;] or preprocessAttrRef() for [c.procName]

## Query Evaluation

### Data Representation (QR)

The QueryRepresentation (QR) is a class to store the necessary information of the query for evaluation process. Since a query contains two parts: a list of symbol declarations and a query part, the QR save those information into 2 data structures:

1. SymbolTable: storing all declaration in the query. A declaration is separated and saved into 2 parts: the declared entity and symbol name.
2. QueryTree: storing the query part, starting from the keyword: “Select” until the end of the query. Each symbols of the query part is stored into a node, and linked with others to form a query tree.
3. In addition to those 2 data structure, QR also saves a BOOLEAN value to indicate whether the query is free of grammar errors or not. Later the QueryEvaluator will check this value first to decide to ignore the query evaluation in case the query has grammar problem.

For example, using the query at (1), we have the following data:

SymbolTable:

|  |  |  |
| --- | --- | --- |
| Id | Type | Name |
| 0 | assign | a |
| 1 | call | c |
| 2 | while | w |
| 3 | variable | v |

QueryTree:

:Select

Tuple:Result

C:QuerySymbol

procName:Attribute

a:QuerySymbol

:SuchThat

:ParentS

W: QuerySymbol

A: QuerySymbol

:SuchThat

:Uses

A:QuerySymbol

V:QuerySymbol

:Pattern

A:QuerySymbol

:Underline

:Underline

:Plus

:Times

2:Const

Y:Variable

3:Const

:With

C:QuerySymbol

procName:Attribute

V:QuerySymbol

varName:Attribute

### Basic Query Evaluation

#### Manage the temporary results

In the old version (CS3201), we don’t have a data structure to manage the temporary result. The temporary result will be create as a vector inside the QueryEvaluator. During the evaluation process, the vector will keep being updated until there is no more unsolved clause or an unsatisfiable clause is met. The QueryEvaluator will try to use the vector to update the final result, and destroy it after that.

However, this method has limitations since it does not allow to review the throw-away temporary results, which is difficult for programmers during debugging and updating the program. Moreover, as QueryEvaluator only uses one vector to store temporary data and use it to solve all clause, it raises the complexity of the evaluation process because not all of those data are needed in solving a certain clause. For example,

Hence, we have added two new classes: ResultTable and ResultManager to handle those problems. While ResultTable keeps values of a list of symbols used in the query, the ResultManager is created to maintain a list of ResultTables. It interacts with the QueryEvaluator during evaluation, records and returns data when QE demands. The main features of ResultManager is:

1. Return concise and non-duplicated data to the QE: before evaluating of a clause, the QE will ask RM for data relating to that clause. The RM, after receiving a list of symbols used in the clause, will extract the data from all ResultTables in list. The extracted data is of the given symbols only, and no data is duplicated.
2. Minimize the space complexity as much as possible: After evaluation, a new ResultTable will be sent from QE to RM. Before saving it to the ResultTables list, RM will try to merge the existed tables with the new one by checking for shared symbols. If there are symbols shared between an existed table and the new table, RM will merge those two table together. Otherwise, RM will do nothing. In this way, each symbol will be saved in only 1 table, and non-related symbols (symbols which are not used in same clause) will be kept in different tables. This approach will cut down time and space to maintain and extract data from RM.

For instance, when evaluating the query at (1), if we use the old approach, the time complexity will be O(N4 P) where N is the number of values for a query symbol and P is the time for a PKB’s operation. It is very costly since we try all combinations of 4 given query symbols.

On the other hand, with the new approach, the symbols’ data are extracted from the RM before the evalutation, hence cost of solving each clause is O(N2P) at most. After that, when we insert the new RT back to RM, time complexity is O(N4M) for merging, where M is merging time of 2 rows. However, since the number of values in a row is usually small, we can safely assume that M<<P. Thus, the new approach will run faster than the old one.

For example, supposed the evaluation of query (1) is now at pattern a(\_, \_”2\*y + 3”\_). At that time, RM contains 1 RT

|  |  |  |  |
| --- | --- | --- | --- |
| Id | w | a | v |
| 0 | 1 | 2 | x |
| 1 | 1 | 4 | t |
| 2 | 1 | 8 | y |
| 3 | 3 | 4 | t |
| 4 | 3 | 8 | y |
| 5 | 5 | 8 | y |

To solve the clause, QE asks RM to extract value of symbol “a”. RM will return a table:

|  |  |
| --- | --- |
| Id | a |
| 0 | 2 |
| 1 | 4 |
| 2 | 8 |

After evaluation, the table now contains only data satisfying the clause

|  |  |
| --- | --- |
| Id | a |
| 0 | 8 |

The QE then asks the RM to insert this new table into its table list. RM must merge the existed table with this new one. The table below is the merged table:

|  |  |  |  |
| --- | --- | --- | --- |
| Id | w | a | v |
| 0 | 1 | 8 | y |
| 1 | 3 | 8 | y |
| 2 | 5 | 8 | y |

### Optimization

The optimization process takes place between the preprocessing-validating and evaluating. After a SymbolTable and a QueryTree of the given query is created, a class named QueryOptimizer will be called to optimize the query. The optimization process includes two actions:

1. Rank all clauses of the given query based on 2 features: their clause’s type and the number of query symbols used in each clause. In our program, we give higher rank to “with” clause, then “such that” and “pattern” clauses. And for number of query symbols, the higher it is, the lower rank the clause will have. For instance, a clause using 2 query symbols will have lower ranking than another with 2 underlines and 1 query symbol.
2. Sort the query tree again using the ranking of its clauses. We assume that the clauses with higher ranking will have better time cost during evaluation, and try to solve those clauses first. Thus, after the sorting, the query tree will have its clause nodes re-arranged, with higher-ranked clause node in the front, following by the lower ones.

For the example query, without optimization, the order of the QueryTree’s clause nodes is shown in part 6.1.1 (such-that -> such-that -> pattern -> with)

After optimization, the order will look like this:

Pattern -> with -> such-that -> such-that

The pattern clause, while having low ranking for its type, is placed at first since it only contains 1 query symbol. For the rest ones, each of them uses 2 query symbols, thus the order is mainly based on their types.

### Design Decisions

1. QueryRepresentator:
   1. SymbolTable:
   2. QueryTree

Inherits from the parent class Tree, which is also used for building AST. By using the same data structure, we can compare 2 trees easier, which is used in solving pattern clause

1. QueryOptimizer:

No data structure. The data is taken from QueryPreprocessor after validation. There are 2 main functions in QO:

* RankTree(): give a ranking score to any clause node met during travelling the tree. The method applies depth first search.
* SortTree(): based on the rankings, QO will sort the clauses node from high ranking to lower ones. The method uses merge sort for sorting.

1. QueryEvaluator:

Not much change from the old version.

1. ResultTable and ResultManager:
   1. ResultTable:

Save a list of temporary results in a table form. It provides basic methods to insert/ get/ delete data of the table, and one more method call extractTable(vector<string> symbols) to extract data of symbols in the input.

* 1. ResultManager:

Save a list of pointers to different ResultTable. It provides 2 methods to insert and extract data from its tables.