Visual COKO: A Debugger for Query Optimizer Development

Daniel Abadi

May 2002

Abstract

Query optimization generates plans to retrieve data requested by queries, and query rewriting (rewriting a query expression into an equivalent form to prepare it for plan generation) is typically the first step. COKO-KOLA introduced a new approach to query rewriting that enables query rewrites to be formally verified using an automated theorem prover. KOLA [1] is a language for expressing rewriting rules that can be "fired" on query expressions. COKO is a language for expressing query rewriting transformations that are too complex to express with simple KOLA rules [2].

COKO is a programming language designed for query optimizer development. Programming languages require debuggers, and this paper describes a COKO debugger: Visual COKO. Visual COKO enables a query optimization developer to visually trace the execution of a COKO transformation. At every step of the transformation, the developer can view a tree-display that illustrates how the original query expression has evolved.

Rule-based query rewriting and the COKO-KOLA project are described for background. Then the COKO syntax is summarized from the point of view of the COKO programmer using an example query transformation that converts query predicates to conjunctive normal form. Visual COKO is described and instructions for its use are presented. Finally, a description of its implementation is given.

Background

Rule-Based Optimization

Optimization is the middle step of query processing – the procedure of extracting data from a database. After the user has submitted a query in a usable language such as SQL,

the query processor first parses and translates the query into an internal representation language (often a relational algebra), It then sends this representation to the query optimizer which searches through possible algorithms for executing the query, and generates a plan that is as efficient as possible. Finally, it executes the query according to the prescribed plan. Not all databases perform query processing in this way. [6] and [7] show examples of query processing algorithms that merge the query optimization and query execution steps so that a query is reoptimized during execution the environment evolves.

A complex query that accesses many tables in a large database can take a long time to execute, so creating an efficient plan can sometimes increase execution time by orders of magnitude. Thus, a sophisticated query optimizer is an essential part of database systems. There are two aspects of query optimization. The first aspect is to find the best ordering of operators (selections, projections, joins, etc.) in the internal representation of the query that will lead to efficient execution. For example, selection operators might be moved so that they will be performed before join operators (so that the join operators will have a smaller input). The second aspect is to annotate each with the best algorithm for its execution. For example, it might be decided to use a sort-merge join or a nested-loop join for a particular join operator. These two parts of query optimization are often interleaved – different operator algorithms might be best for different inputs to the operator (depending on where the operator is located in the query expression tree). For example, a sort-merge join might only be used if the inputs to the join operator are already sorted on the join attribute. The work described in this paper focuses on the first aspect of query optimization – the generation of the layout and order of operators in the query expression tree.

Although user languages such as SQL are typically declarative (meaning that the user does not indicate how the data should be extracted, just what data should be extracted from the database), the structure of an SQL query will lead to a particular translation into an internal representation that contains an initial ordering of query operators. However, one cannot expect the user to write queries that suggest efficient processing plans. Query

rewriting is the process of searching for expressions that are logically equivalent (meaning that it will always return the same result) to the initial expression, but lead to more efficient execution plans.

One method for performing query writing is to use equivalence rules to generate expressions that are equivalent to the initial expression. If the initial expression matches the left-hand side of the rule, the expression can be transformed according to the right-hand side of the rule. One example of such a rule is the commutative property of selection operators: $\sigma_{\theta 1}(\sigma_{\theta 2}(E)) \Rightarrow \sigma_{\theta 2}(\sigma_{\theta 1}(E))$ where σ is the selection operator and θ_1 and θ_2 are the predicates upon which to select. For example, this rule could be used to transform $\sigma_{salary > 40}(\sigma_{salary < 80}(employees))$ to $\sigma_{salary < 80}(\sigma_{salary > 40}(employees))$. The primary advantage of expressing query rewrites using rules is that it makes the optimizer easily extensible: in order to change the functioning of the optimizer, one only has to add and subtract the rules that it uses to generate equivalent expressions. [4] and [5] show examples of extensible optimizers that use rules to rewrite queries.

The COKO-KOLA project

Although the above discussion asserted that rule-based optimizers are extensible, there can be drawbacks to using rules to rewrite queries. Sometimes, the rewritten query does not return the same result as the original query. [3] described the infamous “count bug”, an example of a rewrite rule that unnests nested queries (rewrites SQL queries with a nested SQL query in the WHERE clause as a join) that returns incorrect results when the aggregation operation COUNT returns a value of 0. Clearly, it is imperative for rewrite rules to return the expected results– rules should be formulated so that they be proved correct.

The problem is that most internal query representation languages make the process of reasoning and proving the correctness of rules very difficult. Cherniack and Zdonik showed in [1] that the choice of the representation determines the effectiveness with which rules can express rewrites. In particular, variable-based languages make the

process of identifying matches of patterns in the rule and the query and the rearrangement of these patterns more complicated because of the presence of free variables in the matched patterns that make its meaning context dependent. For example, consider the following rewrite rule expressing in SQL:

SELECT distinct X.name → SELECT distinct X.name
FROM X, Y FROM X

The above rule says that we don't need to join X with Y if we only need an attribute from X. The problem with this rule is the presence of variables X and Y, making the rule context dependent. A replacement of X with Y in the SELECT clause would give the rule an entirely different meaning. One would need code to supplement this rule to make it more general (i.e. the code would express that X and Y are two tables, and the X in the SELECT clause is referring to the first of the two tables in the FROM clause). [1] also introduced a new language for query representation, KOLA, that removed variables from the query by using combinators. KOLA's functions and predicates are either predefined primitives, or are built using formers from other functions and predicates. The above query, expressed in KOLA, would be:

join (K(true), name o π1) ! [X, Y] → iterate (K(true), name) ! X

In the above rule, ! implies a function call, the function is to its left, and the parameters are to its right. Intuitively, the left-hand side of the above rule says that there is a join of X and Y and is taking the "name" attribute for each value of X (for specific details on the syntax of KOLA see [1]) . The right hand side just takes the "name" attribute for each element in X (it avoids the join like in the SQL rule). In this KOLA rule, the only place where the variables appear are in the function parameters. The function itself (to the left of the !) has no variables. This facilitates the proof of this rewrite rule, and indeed [1] showed that this can be done using an automatic theorem prover.

Rules alone are not always sufficient to express complicated rewrites. Take the example of normalization. In many cases, in order to fire a rule on a query, the query must be normalized so that query can follow a predictable pattern to be matched with the left hand side of a rule. One simple rule is not sufficient to normalize a query, because typically an algorithm of adjustments must be made to complete the normalization. Normalizations are thus too general to be expressed with just one rule, or even a set of rules. [2] introduced the concept of a COKO transformation, which can perform these more complex rewrites by integrating a set of KOLA rules with a language that can express an algorithm for their firing. COKO transformations retain the property of ease to prove correct because the only way the transformation can affect a query is through the firing of KOLA rules – the COKO language simply states an algorithm for their firing.

Even though a COKO transformation can be correct from the point of view that the results of the transformed query will be the same as the results of the initial query, a faulty COKO code will lead to unexpected resulting queries. It will return the "correct" answer, but it might not be in the normalized form that the programmer intended. Visual COKO facilitates the COKO programming process by providing the ability to visually trace the execution of a COKO transformation, one step at a time. Before Visual COKO is presented, the language that it debugs: COKO is described in more detail.

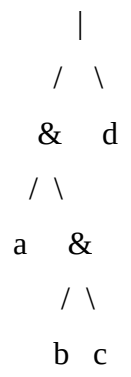
The COKO language

In order to understand the use of Visual COKO, the language that it debugs must be explained in more detail. In this section, the COKO syntax is presented, in the context of a COKO transformation that converts a KOLA predicate to CNF (conjunction normal form). The COKO code that performs this normalization is given.

In order to convert a predicate to CNF (assuming the query does not contain negation) all one needs are two base KOLA rules:
rule1: (p & q) | r –> (p | r) & (q | r)
rule2: (p & q) | r –>(p | r) & (q | r)

and the following algorithm:

Assume the input predicate is in the form of a tree, for example:



traverse the tree from the bottom up in a post-order pass. For each visited node, attempt to fire rule1. If that fails, then attempt to fire rule2. If either rule fired, then attempt to fire both rules on both children of this subtree and continue down on each child until both rules fail. The COKO code for this algorithm is the following:

```
TRANSFORMATION CNF
 USES
  CNFAux
 BEGIN
  BU {CNFAux}
 END

TRANSFORMATION CNFAux
 USES
  rule1: p | (q & r) –> (p | q) & (p | r),
  rule2: (p & q) | r –> (p | r) & (q | r)
BEGIN
  {rule1 || rule2} –>
```

```
   GIVEN p & q DO {CNFAux (p); CNFAux (q)}
END
```

The basic structure of a COKO transformation, is the keyword TRANSFORMATION followed by the name of the transformation, the keyword USES followed by a list (separated by commas) of the rules or other COKO transformations that this transformation will be using, and the keywords BEGIN and END surrounding the body of the transformation algorithm.

CNFAux uses the two rules discussed earlier. It declares them in the USES section using the format <rule-name>: <rule>. The rules are named so that they can be used later (in the line {rule1 || rule2}) without having to rewrite the entire rule. COKO does not require that these rules be declared in advance as the following example shows. Another way that CNFAux could have been programmed is as follows:

```
TRANSFORMATION CNFAux
 USES

BEGIN
  { [p | (q & r) –> (p | q) & (p | r)] || [(p & q) | r –> (p | r) & (q | r)]} –>
    GIVEN p & q DO {CNFAux (p); CNFAux (q)}
END
```

However, this is messy and difficult to read, so in general one declares rules ahead of time in the USES section.

The body of CNFAux is best explained piece by piece. Once again the body is:

```
{rule1 || rule2} –>
   GIVEN p & q DO {CNFAux (p); CNFAux (q)}
```

A parser would break this line up into statements. In the above code, rule1 is a statement, rule2 is a statement, rule1 || rule2 is a statement, CNFAux (p) is a statement, CNFAux (q) is a statement, CNFAux (p); CNFAux (q) is a statement, GIVEN p & q DO {CNFAux (p); CNFAux (q)} is a statement, and finally, {rule1 || rule2} –> GIVEN p & q DO {CNFAux (p); CNFAux (q)} is a statement. Each of these statements will be looked at in turn. First, however, it is important to note that in addition to altering the input KOLA tree, execution of each statement returns a boolean success value indicating whether or not that statement was successful in its execution.

The first statement that the parser sees is rule1. An identifier such as this means that rule1 must be naming either a rule or another COKO transformation. From looking at the USES section, it is deduced that rule1 is a rule. This is code for an attempt to fire rule1 on the current KOLA tree. If the pattern for the rule does not match the pattern of the tree, then this attempt will fail (this statement will return a success value of false). If the patterns match, the rule will fire (the KOLA tree will transform) and a success value of true would be returned.

{rule1 || rule2} is a special type of statement, named a disjunctive multistatement. A disjunctive multistatement takes two statements (in this case rule1 and rule2) and will only execute the second statement (in this case rule2) if the first statement returned a success value of false. The success value of a disjunctive multistatement is true if either one of the two input statements returned a success value of true. In this example, rule1 is fired on the tree. If it failed, then rule2 is fired.

As stated above, CNFAux (p) is a statement. This type of statement (an identifier followed by a variable name in parenthesis) is similar to the simple type of identifier statement (e.g. rule1) discussed above. CNFAux must either be a rule or a transformation (in this case, it is a recursive call to the current transformation). The variable in parenthesis is a pointer to the KOLA tree upon with the transformation call should be executed. In this case, CNFAux will be called on the KOLA tree rooted at the node

pointed by p. If the code had been just CNFAux (without the '(p)') then a pointer to the current KOLA tree would have been sent to CNFAux, instead of a tree rooted at p.

{CNFAux (p); CNFAux (q)} is a special type of statement, named a sequential multistatement. A sequential multistatement takes two statements (in this case CNFAux (p) and CNFAux (q)) and will only execute both statements in order. Like a disjunctive multi-statement, the success value of a sequential multistatement is true if either one of the two input statements returned a success value of true.

GIVEN p & q DO {CNFAux (p); CNFAux (q)} is also a special type of statement. The format of a GIVEN statement is GIVEN <equations> DO <statement>. The <equations> tag is a set of bindings of variables to pointers to various nodes of the KOLA tree. An equation is of the form <COKO expression with pattern variables> = <KOLA expression>. An equation can also take the form of just < COKO expression with pattern variables> where the right side is assumed to be a pointer to the current KOLA tree. In this example, there is one equation (p & q) with no right hand side, so an attempt to match p and q to the current KOLA tree leads to p pointing to the left child of the current tree, and q pointing to the right child of the current tree. With these bindings in place, the DO statement ('{CNFAux (p); CNFAux (q)}') can now be executed with p and q pointing to the correct trees. A GIVEN statement returns a success value of true if all of the equations succeed in pattern matching and the DO statement returns a success value of true.

The two statements discussed thus far can be put together using the symbol –> .

```
{rule1 || rule2} –>
   GIVEN p & q DO {CNFAux (p); CNFAux (q)}
```

Once again, this is a special type of statement, named a conjunctive multistatement. A conjunctive multistatement takes two statements (in this case '{rule1 || rule2}' and 'GIVEN p & q DO {CNFAux (p); CNFAux (q)}') and will only execute the second statement if and only if the first statement returned a success value of true. The success

value of a conjunctive multi-statement is true if the first of the two input statements returned a success value of true.

So now the body of CNFAux has been completely explained. Rule 1 fired on the current KOLA tree. If it fails, then rule2 is fired. If one of them succeeded, then p and q are matched to be the left and right children of the current tree, and CNFAux is called recursively on each subtree.

Now, the body of transformation CNF will be examined. Once again, the body was simply:
BU {CNFAux}
CNFAux is an identifier that, as stated above, must either be a rule or a transformation. In this case it is a transformation declared in the USES section. BU {<statement>} is a special kind of statement that executes <statement> on every node of the parse tree of the KOLA expression in a bottom up (post-order) traversal. In this example, CNFAux is called on every node of the input KOLA parse tree in a bottom up fashion. A BU statement returns a success value of true if <statement> succeeded on any of the subtrees that it was called upon.

A call to CNF will transform the input KOLA predicate tree into a KOLA tree in CNF (assuming there were no negations). The above discussion included most of the COKO syntactical constructs. The remaining COKO statements are included below.

TD {<statement>} will execute <statement> on every node of the current KOLA tree in a top-down (pre-order) traversal. A TD statement returns a success value of true if <statement> succeeded on any of the subtrees.

TRUEv <statement> will execute <statement> but will always return a success value of true. Likewise, FALSEv <statement> will execute <statement> but will always return a success value of false.

REPEAT <statement> will continually execute <statement> until <statement> returns a success value of true.

Visual COKO

Visual COKO enables a query optimization developer to visually trace the execution of a COKO transformation, one step at a time. It functions as a debugger by providing commands that control the execution of the COKO code, allowing the programmer to step through the code and to find the source that is causing the transformation to behave unexpectedly. It functions as a visual aid by displaying the original KOLA query parse tree (and various branches of this tree) throughout execution so that the user can visually observe and understand how the COKO code and KOLA rules transform the original query. By functioning both as a debugger and a visual aid, Visual COKO greatly facilitates the COKO programming process.

First, Visual COKO's functionality as a debugger will be described, and its commands for use presented. Then, Visual COKO's functionality as a visual tool will be explained.

Debugger Commands

Introduction

The Visual COKO debugger commands are based on, and work similarly to the gdb commands. There are two key differences between common programming languages (for example: C or JAVA) and COKO which make the commands work slightly differently. First is the use of the semicolon. In C or Java, semicolons can be ignored (at least in the sense of someone reading and understanding code – of course semicolons are a syntactical necessity) – the language parser simply uses them in order to differentiate between programming statements. But in COKO, semicolons are inherently much more important to the language – they are used as connections between two statements, and are the mechanism for constructing COKO transformations from smaller sequential COKO statements. As described earlier, a semicolon is a COKO statement in its own right; and

like any other COKO statement has a success value (defined by the following: the semicolon is successful if one of the two COKO statements it connects is successful). Given the importance of the semicolon (termed a sequential multistatement), the debugger can not skip over them (as in C or JAVA), nor treat them as part of one of the two statements it connects, but must provide the capabilities to stop on a semicolon like any other COKO statement.

A second difference between COKO and other common programming languages is the concept of the subroutine (sometimes referred to as functions, methods, or procedures in various languages). A COKO transformation works similarly to standard subroutines in that it contains a subset of COKO commands and can be called from a different COKO transformation in a similar fashion as a procedure call. What makes it slightly different from C methods or Pascal functions is that transformations are restricted in that they can only be called with one parameter (the KOLA tree upon which it will work) and can only return one value (that very same KOLA tree with its appropriate alterations). Each transformation is designed to be able to work alone, as if it was the only transformation working on the query. Nevertheless, from a debugging point of view, COKO transformations will be treated like standard subroutines with respect to commands such as step and next (that either enter or skip over subroutine calls respectively).

Breakable Statements

The debugger only provides the capabilities to stop on certain COKO statements, termed breakable statements. The debugger can stop on a rule or transformation firing statement, a multistatement, a GIVEN statement and its respective matching equations, or a print statement. The debugger will not stop on any other COKO statement. Some examples of statements that the debugger will not stop on are REPEAT statements, bottom-up or top-down statements, and TRUEv/FALSEv statements. Note that this is different from standard debuggers such as gdb. Debuggers tend to stop on *while* or *repeat* statements once for each time through the loop. The decision as to what statements to stop, or not to stop on were made from using the command line version of the debugger. In stepping through a transformation, it seemed wasteful to stop every time on a REPEAT or BU

statement since the statement itself wasn't going to accomplish anything. Clearly it is the statements inside these loops that perform the work. Thus, the user is not given any more power over the execution of the transformation if given the ability to stop on loop commands – it just slows down the stepping process. So the general heuristic used was that the debugger only stops on statements that actually perform some task (such as firing a rule, matching an equation, or printing to the screen). The debugger does not stop on statements that simply describe future execution.

The exceptions to this heuristic are the multistatements on which the debugger provides capabilities to stop, even though they don't actually perform a task. The reasoning for this exception is that it was found useful to stop after the first of the two statements that the multistatements connect in order to check its success value. Since the success value of this statement often determines whether or not the second statement will be executed, pausing in between these two statements will allow the programmer to check the success value of the first statement and prepare for the execution of the second statement (for instance by placing a break in the appropriate location). Given that the decisions as to which statements the debugger should stop on were made using sample runs of the debugger, it is entirely possible that it will be found on future runs of the debugger that it should stop on REPEAT, BU, or TD statements. If this is found, it will be relatively straightforward to make it possible for the debugger to stop on these statements. Details will be given in the implementation section.

Execution Control Commands

break: The user can indicate a COKO statement that the debugger should stop upon reaching. Placing a break on a statement in the current transformation will cause the debugger to pause transformation execution before executing the indicated statement. In addition, it will stop on that statement any time that transformation is called – not just in the current version. So the break is placed on the transformation, not an instance of the transformation. A break is indicated in Visual COKO with a red rectangle around the statement. To choose the statement upon which to break, the user right clicks on the statement and selects the command *break*.

condition: This command works similarly to *break*. However, it will only break on the indicated statement if it generated a positive success value (remember that every COKO statement evaluates to either true or false). Since it must evaluate the statement to test its success value, this command differs from *break*, which stops before the execution of the statement. If the conditioned statement evaluates to true, the debugger will stop on the next breakable statement after the indicated one (otherwise it will not stop). The conditional statement is particularly useful for COKO rule firing statements. A conditional break is indicated in Visual COKO with a blue rectangle around the statement. To choose the statement upon which to place a conditional break, the user right clicks on the statement and selects the command *condition*.

clear: This command removes both regular and conditional breaks on the indicated statement. To choose the statement upon which to clear a break, the user right clicks on the statement and selects the command *clear*.

continue: When the debugger is in a paused state, the user can use the continue command to cause the debugger to continue execution of the transformation until the next statement upon which a break is indicated (either regular or conditional).

step: This works similarly to the standard debugger command. The debugger will execute the current statement that it was previously stopped on and will go on to the next COKO statement. If the statement is breakable (see the above discussion), the debugger will stop. Otherwise, the debugger will continue until it reaches a breakable statement. If the executed COKO statement was a transformation call, the debugger will stop on the first breakable statement inside the new transformation. Visual COKO will also display the contents of this new transformation in the current transformation window.

next: Again, this works similarly to the standard debugger command. The debugger will execute the current statement that it was previously stopped on and will continue to the next breakable statement in the current transformation. The only difference between *next*

and *step* is that if the current statement was a transformation execution, the debugger will not enter that transformation, but instead perform that transformation and return to next statement of the current transformation. If the *next* command is performed on the last statement of a transformation, it acts identically to the *continue* command (see above).

run: This command restarts execution of the COKO transformation.

Textual Output Commands

The following commands are initiated by typing the command into the console below the current transformation in Visual COKO, and all textual output from the command is written into the same console window. These commands are not commonly used and are left over from a command line version of Visual COKO that relied solely on textual output. The information derived from issuing these commands can usually be found visually, without having to wade through the text in the console window.

set success on: This command causes the success value of each COKO breakable statement that is executed to be displayed to the console. The format of the output is: Statement: <transformation name>::<statement number>|<statement> evaluated to <boolean value>. The success value of each statement is printed to the screen directly after it has finished executing. It should be noted that the order of statements with their success values is not necessarily (and indeed not usually) the same as the order of statements in the transformation. The reason is that the success value of a multistatement can not usually be determined until both of its statements have been evaluated (the exception is the conditional multistatement whose success value could theoretically be determined after evaluating just the first statement since the success value of the conditional multistatement is the same as the success value of its first statement). Thus, the success value of the second statement of a multistatement will be outputted before the success value of the multistatement itself. A possible future extension of Visual COKO would be to have the success value of a statement be indicated directly in the current

transformation window, perhaps by changing the color of the COKO statement text. Under this implementation, the statement text will not be colored in order, nor will each step immediately color the statement that it executed with a true or false value (for example if the statement executed was a multistatement), but it would be easier to match success values with specific COKO statements in the transformation. *set success off* turns off this textual output feature.

set rule on: This command causes state and success information to be displayed to the console every time a rule is fired. The format of the output is:
ATTEMPT TO EXECUTE RULE: <KOLA rule> on tree <KOLA tree> ATTEMPT <succeeded or failed> Result State is: <resulting KOLA tree>. Like the success feature described above, the rule output can be difficult to read at times. This is principally because the outputted text is a KOLA expression. Textual KOLA expressions tend to be much more challenging to read than the same expressions in graphical tree format, especially as the expressions become more complex. An additional limitation of this command is that matching information is not displayed – just the original KOLA rule and the KOLA tree. There is no attempt to display attempts of matching the variables in the rules with the branches of the KOLA parse tree of the query. Future versions of Visual COKO could change this feature to be more of a visual tool. The process of matching variables can be displayed directly on the graphical display of the KOLA tree with variables pointing to their corresponding branches.

output <stmt>: This command will output to the console one of three KOLA expressions. If <stmt> is *root*, it will print the textual representation of the entire KOLA query tree. If <stmt> is a variable name (usually used inside a GIVEN statement), the KOLA expression bound to that variable will be outputted. If <stmt> is *current* (or empty) the part of the KOLA query tree that the transformation is currently working on is outputted. The *output* command is used rarely because the entire query tree (root) and the current query tree are always visually displayed as separate windows. Furthermore, whenever a variable is bound to a branch of the tree, it will be displayed directly on the tree with an arrow the branch to which it is bound.

Additional Visual Features

stack: This command brings up a new window listing the current transformation, and each of its calling transformations above it. This allows the user to observe the path of transformation calls that lead to the execution of the current execution. This stack window is especially useful for recursive transformation calls – it is possible to detect the current level of recursion.

In addition to simply being able to view the names of the transformations that called the current transformation, one can also view the transformations themselves. By right clicking on the name of the transformation and clicking *expand*, the transformation will be opened in a new window. Breaks can be added or cleared from this opened transformation. This is especially useful in the situation where the current transformation is long and boring – for example it might be a cleaning transformation that performs rule firings in a bottom up or top down fashion on a large query tree. Rather than stepping through this boring transformation, one can right-click on the transformation that called it and put a break on the statement following the statement that called the current transformation. By then clicking on the *continue* command, the rest of the current transformation can be skipped, the debugger will stop at the next statement of the calling transformation. A common scenario where this happens is if the user accidentally pushed the *step* button when she meant to push the *next* button.

Graphical KOLA Tree Display

As described above, KOLA is a difficult language to understand, but is particularly difficult to read in its textual form. The task of comprehending the KOLA is greatly facilitated if, instead of wading through commas, parenthesis, and long lines of text, a picture of the KOLA parse tree is displayed. While this graphical representation does not illuminate the semantics of KOLA subexpressions (if the user does not understand how iterate works, he will not be helped by seeing iterate in the KOLA parse tree

representation), it will make clear the parameters of each KOLA expression and eliminates the commas and parenthesis that makes KOLA so difficult to read.

Throughout the execution of a transformation, Visual COKO keeps track of the KOLA query that is being transformed, and the current branch of the KOLA query upon which the transformation is working on. Both parse trees of these KOLA expressions are displayed in separate windows. The root KOLA parse tree is displayed as a default, but the current KOLA tree is kept hidden by default (to display this tree, simply click on the appropriate option in the view menu). Often the KOLA queries can be quite large, and might have trouble fitting into one of these display windows. In such a scenario, each node of the tree (ands its corresponding internal text) might be so small that it is difficult to read. There are two options that the user can pursue. First is to expand the display window. The tree will automatically expand with the window, and each node will get bigger with each additional increment of window size. The second option is to use the zoom feature. The user can right click on any area of the tree, and select *ZOOM*. Scroll bars will appear in the window, and the tree will expand by a factor of approximately 125% (so it will not all fit in the window).

In addition to the current and root KOLA trees there are two other types of display windows that show branches of the KOLA query tree. The first is a branch isolator feature. The user can right click on a node of any display of a KOLA tree and isolate that branch (with the selected node as the root) in a new window. Subsequent changes that occur on that branch on the original query tree will also be seen in this isolation. This isolation feature is similar to the zoom feature, in that it functions to make a portion of the tree look larger. However, it differs in that it opens up a new window with the enlarged tree rather than enlarging it in the current window and adding scroll bars. An isolation window is kept open until the user closes the window – at which point the isolation is deleted. Note that if the branch upon which the isolation was based is eliminated, the isolation window simply displays the text “nothing to display”.

The second type of display window is a variable display window. If there are variables currently bound to portions of the tree, they can be isolated by typing the command: *display* <variable-name> into the console. A new window will open containing the KOLA tree that the variable is bound to. This variable display window is automatically closed as soon as the variable becomes out of scope.

Implementation

While the commands and visual interface of command line COKODB and Visual COKO are based on gdb and ddd, the implementation is not. The COKO debugger is written directly into the COKO language so that the language itself can be run in regular or debugging mode. This implementation of COKO and its debugger is explained in the following section.

The COKO language is written in a very object oriented style. There is a class: CStmt, from which every COKO statement inherits. These COKO statements can have other COKO statements as member variables upon which that COKO statement works. A COKO statement object tree can be constructed by representing each of these COKO statement variables as children of that statement. A COKO transformation is a pointer to the root COKO statement of such an object tree.

Each COKO statement contains an Exec method that is called to perform the activity of that statement upon the current environment (an instance of a CState class). To execute a COKO transformation, the Exec method of the root statement is called, which will call the Exec method of each of its children in an order that depends on what type of statement this root is. Thus, the COKO statement tree is traversed and the Exec method is called on every node of this tree.

Debugging methods are programmed directly into these COKO statements. An ExecDebug method was added to every COKO statement that mirrors the actions of the corresponding Exec method except that it calls ExecDebug on its children instead of

Exec. In addition, it checks with the debug controller (an instance of the DDebugController class) to see if it should execution of the current statement should pause (break) program execution and control given to the debug controller which can prompt the user for debugging commands.

Various other debugging methods were added to these COKO statements. The rep_string method is used to display the textual representation of that statement only (in contrast to the representation method which is used to display the textual representation of the entire COKO transformation by returning not only the representation of the current statement, but also calling the representation method of each of its children). The number_self method is used to number the breakable statements in a transformation by assigning the current statement a number (its key) and then numbering its children. The output_self method will display to the debugging window the key number and its representation. All of these methods are used by the debug controller to display information about the current statement to the user.

It can be seen from the above description that the COKO debugger is integrated closely with the COKO language itself. The above described COKO statements and their debugging methods can be found in the /src/coko/PC folder. However, there are many classes (including the debug controller alluded to above) that were written for the debugger only, and are separate from the COKO language. These classes can be found in the /src/coko/DE folder.

The debug controller (DEDebugController.C) is the central class to the COKO debugger. When execution of the COKO transformation is paused, the debug controller gains control and prompts the user for commands. It then analyzes these commands and performs the necessary actions. For instance, if the user typed “break 8” into the command line debugger (or the console of the visual debugger) then the debugger will find the current transformation in the COKO transformation stack, and will place the bool: true in the 8^{th} index of the break array for this transformation. The debugger then retains control and asks the user for the next command. If the command is step, next, or

continue, then appropriate variables are set, and control is returned to the COKO program execution (the ExecDebug method that gave control to the debug controller in the first place).

If the visual option is turned on (Visual COKO), then the debug controller interfaces with a visual window controller: VC.C. This visual window class uses QT to provide a visual interface to the user so that the user can use the debugger with more ease. When a program is paused, control is still passed to the debug controller, but the debug controller then passes this control to the visual window controller which waits for user input. Once the user inputs a command (by pressing a button, right clicking the mouse, selecting a menu option, etc) the visual window controller sends control back to the debug controller which analyzes the command and decides whether to give control back to the visual window controller, or to send control back to program execution. Thus, the debug controller and the visual window controller are the two central components of the COKO debugger. For this reason, a description of each of the methods contained in these two classes are given below. A quick glance at the code below will show that these two classes perform many functions and have many methods. The best way to familiarize oneself with the debugging code is to start in the central method of the debug controller: breakLoop and look one-by-one at the methods it calls. This method also interfaces with the visual window controller if the visual flag is turned on, and so walking through this breakLoop method will result in looking at most of the classes in both the debug and visual window controllers.

DEDebugController.C methods:

```
public:

  DDebugController(int, int, char**);
  // Constructor

  ~DDebugController();
  // Destructor

  void doBreak(CStmt*, CState*);
  // This method is called from any COKO statement class that inherits from
  // CStmt. It is called if its previous call to check_for_break returned true.
  // If so, then it passes along a pointer to itself (stmt) and a pointer to
  // the current state (s). The key number of the COKO statment inside the
  // current transformation and the rep_string of the statement are found from
```

```
// stmt and this information, along with s, is passed along to the breakLoop
// method which takes control and allows the user to indicate his/her wishes
// while the program execution has paused.

void doBreak(CEqn*, CState*);
// This method is called from any COKO statement class that inherits from
// CEqn. It is called if its previous call to check_for_break returned true.
// If so, then it passes along a pointer to itself (eqn) and a pointer to
// the current state (s). The key number of the equation inside the current
// transformation and the rep_string of the equation are found from eqn and
// this information, along with s, is passed along to the breakLoop method
// which takes control and allows the user to indicate his/her wishes while
// the program execution has paused.

void start(CRuleBlock*);
// This method is called to start the debugging process on the root
// transformation. Two options are possible: this is the first time through
// this transformation (the user just started up Visual COKO or Command Line
// COKO) or this is not the first time through this transfomration (the user
// restarted the debugger). In the former case, the current transformation is
// added to the _curr and _break_rbList and a welcome line is printed to the
// screen. In the latter case, the same thing occurs but the _break_rbList
// is not reinitialized (so that the breaks that the user placed on various
// statements will be remembered even though execution got restarted).

void stop();
// This method is called when the debugger is finished executing the root
// transformation. This root transfomration (rule block) is simply popped off
// the _curr_rbList stack.

void format_output(char*, char*);
// This method outputs to the textual display window the two parameters
// separated by a '|' character.

void format_output(int, char*);
// This method outputs to the textual display window the two parameters
// separated by a '|' character.

int check_for_break(int);
// This method checks to see if the debugger should break and take control
// before executing the next COKO statement. It is called at the beginning of
// every ExecDebug method of each COKO statement class. This method takes an
// int (key_number) and finds if there is a break on that key (statement of
// COKO code) in the current transformation (the transformation that is
// currently executing). Alternatively, it checks if the debugger should
// break for other reasons (such as if the user typed "step" or next". If any
// of these reasons are valid, it returns true indicating that the debugger
// should break. Otherwise it returns false.

NonType* setToExec(NonType*, char*, char*);
// This method sets the global variable toExec when a COKO transformation
// is about to call a sub-transformation on a branch of the tree (toExec
// stands for to execute). toExec points to this branch of the tree. In
// addition, the variable num of the root of this branch is set to be
// negative so that an arrow can be drawn in the visual window to indicate
// that a subroutine is about to be called. This branch with the new variable
// num is returned. In addition, the global variable "execName" is set so
// that the visual window can use this as the label of the arrow pointing to
// the tree branch.

NonType* resetOldVariableNum(NonType*, char*, CState*);
// This method takes a NonType (pointer to a KOLA tree) and resets all of the
// variable-nums of each node in the tree back to being a positive number (ie
```

```
// if that node was marked (the mark is done by giving the varibale-num of
// that node a negative number) as the node upon which a sub-transformation
// was about to be called, then we now remove this "mark"). This method is
// currently called from the PCRuleInvokeStmt class after the debugger has
// returned control to this class and just before the transformation call
// actually occurs. The char* key paramater is the name of the variable that
// is bound to the branch of the tree that the sub-transformation is being
// called upon. For this method, I only care whether or not key is NULL. If
// it is not NULL, then s->Store has to be rebuilt to reflect these changes
// in variableNum - otherwise the change was only to the root and the tree
// does not have to be rebuilt. The function returns back the toExec tree
// with the updated variableNums.

void check_for_condition(CState*, char*, int);
// This method is called from the COKO statement classes after they are
// finished to check if the success flag is set. If it is, then the success
// variable in the CState parameter s checked and printed to the screen. In
// addition, if the success value was true, this method checks to see if
// there was a conditional break set on that statement (in which case a
// global variable is set saying that the debugger should stop as soon as it
// can).

int ruleOn();
// This 1-line method simply returns the value of the rule_on flag.

//void checkAndDisplaySuccess();

int really_execute();
// This method is called from every ExecDebug method in every COKO statement
// to make sure that it should really go ahead with the execution. The reason
// why this check has to be made is that if the user wants to restart the
// program, no more execution should occur until the the current execution
// unwinds from all its recursive calls and the while loop and begin again.

int do_restart();
// Once the outer while loop is ready to restart the COKO program (ready to
// begin execution again, do_restart will be called to reinitialze the
// restart variable to 0). But, before restarting, it checks to see if it
// really should restart by checking the value of the restart boolean.

void setReadyToGo();
// This method sets the readyToGo member variable to equal 1.

void repInitialize(int, int, int);
// This method takes textual display information about the current
// transformation including the number of lines (line) columns (col) and
// statements (num_stmts) and sets corresponding member variables to equal
// these parameter values. In addition, this method initializes an array
// (currLCArray) of size num_stmts that will eventually contain the line and
// column number of the beginning and end of each statement in the textual
// represenation of the current transformation.

void updateArray(int, int, int, int, int);
// This method is called from the representationWDI mehtod of various COKO
// breakable statement classes. This method will update the currLCArray with
// input statment number (kn) the exact location (in terms of characters)
// inside the QT textarea window of this statement (using the start line
// (sL), end line (eL), start column (sC) and end column (eC)). This method
// also updates where we are in the text area using the end line and end
// column.

void updateLine(int, int);
// This method is called if CMulStmtCon or CMulStmtDis added a '{' to the
```

```
// textual representation. If that is the case then every statement on that
// line must be moved over by one character to account for this '{'.

int getLine();
// Returns the current line number in the text area that is being used to
// display the contents of the current transformation.

void setLine(int);
// Sets the current line number in the text area that is being used to
// display the contents of the current transformation.

int getCol();
// Returns the current column number in the text area that is being used to
// display the contents of the current transformation.

void setCol(int);
// Sets the current column number in the text area that is being used to
// display the contents of the current transformation.

void outputDBArray();
// This method is for debugging purposed only. It displays to cout the
// current contents of currLCArray and currLine and currCol.

void parseInput(const char*, char**, int);
// This method is called by the command line version of the debugger to pasre
// the char* charArrayToParse input string into an array of char* with each
// element of the array pointing to 1 word (space delimited)

void printInRightPlace(char*);
// This method prints whatever is stored in the Char* parameter (toPrint) in
// the correct location. If visual mode is turned on, then this text is
// printed to the console window. Otherwise, it is printed directly to the
// screen (cout).

int findCurrent(CRuleBlock* actuallyFindThisOne=NULL, char* transName=NULL);
// This is an important method that goes through the break rule block list
// (this list maintains all of the transformations that have been called so
// far and any break information that the debugger might need to know about
// each transformation) and by default finds the one that is being currently
// run (the top element in the current rule clock stack). Alternatively, if
// a char* transformation name is given (transName) it will find that
// transformation (and not the current one) or if a pointer to a RuleBlock
// is given it will get the name of the RuleBlock by following the pointer
// and calling the get_name() method and will find that transformation.
// The key assumption here is that transformation name is a key. This method
// returns a boolean value indicating whether it found that transformation in
// the break list and additionally will result in the iterator of the global
// variable _break_rbList pointing to the approporate element.

void testRunAgain();
// Once the COKO program has finished executing, this method will be called
// to check if the user wants to run the program again, or just finish.

void histPush(NonType*, int, int, char*, CIEnvStack*, NonType*);
// This method simply passes along all of its parameters to the push method
// of member variable _histList (which is an instance of HistoryList and is
// used to reconstruct the global KOLA tree from the current KOLA tree and
// the environment stack stored in this _histList variable).

void histPop();
// This method simply calls the pop() method of the _histList member variable

NonType* histUnravel(CState*);
```

```
  // This method takes a pointer to the current state (s) and unravels the
  // histList stack from the current state to the root of the stack. Each
  // element in this histList stack is the current environment when a
  // transformation was called on a branch of the tree resulting in a new
  // environment to be initialized that contains only information pertaining
  // to this new "zoomed in" portion of the tree. Thus this method can be seen
  // as unraveling all of these environments to "zoom out" to get the original
  // KOLA tree as it now stands. Since changes may have been made in each
  // "zoomed in" frame, care must be take to make sure that this new version of
  // this branch of the KOLA tree replaces the old branch in the older
  // envirment in the stack frame above.
  //
  // This method is called when the user types: "display root" in the command
  // line debugger, and in the root tree display in the visual debugger

  void setTempStmt(CStmt*);
  // This method sets member variable tempStmt to CStmt* parameter stmt. This
  // is done because unfortunately, updateDEP was written so that it is a
  // member method of the CStmt class, and sometimes the debugger needs to call
  // this method and doesn't have an instance of a CStmt around to call it on.
  // Since updateDEP doesn't actually affect the stmt that it was called on, it
  // doesn't matter what statement is sent to this method - as long as it will
  // always be non-NUll while the debugger is running. Really, updateDEP should
  // have been declared static when it was originally written.

  LCSubstring* getCurrLCArray();
  // This method returns currLCArray (the variable that contains the location
  // of each statement in the text area that is displaying the current
  // transformation).

  int getNumStmts();
  // This method returns the number of breakable statements in the current
  // transformation.

  void updatePaintWindow(CState*);
  // This method tells the visual display window (vis) to update itself,
  // sending the variable symbol list (_symList), and the current tree
  // (s->Store()) which is found using the State parameter to this method (s).

private:

  void doPrintOST();
  // This method displays to the screen whatever is stored so far in the member
  // variable ost. If visual mode is turned on, then this text is printed to
  // the console window. Otherwise, it is printed directly to the screen
  // (cout).

  void breakLoop(int, char*, CState*);
  // This is the central and most important method in this class! If the visual
  // option is turned on, then this method gathers all of the current
  // information: the transformation code, the global tree, the current tree,
  // and all trees bound by variables, and sends this information to vis (the
  // visual coko display controller) which displays this information to the
  // user. In addition, this method sets the variableNums of all of the nodes
  // in these trees so that the tree drawer can display arrows pointing to all
  // bound variables and so that the current node can be painted green. If the
  // visual option is not turned on, then the only information displayed is the
  // current statement upon which the COKO transformation has stopped. Further
  // information must be explicitly requested by the user.
  //
  // Once the information has been display, the method loops while the user
  // types commands that do not return control back to program execution (such
  // as break x, display y, print z, etc) - performing those commands and
```

```
  // waiting for the next command. Otherwise (if the user typed step, next, or
  // continue) global variables are set and control is returned back to the
  // COKO program execution.

  void getInput(string*);
  // This method is called by the command line version of the debugger to get
  // a line that the user inputted.

  void parseInput(string*, char**, int);
  // This method is called by the command line version of the debugger to pasre
  // the char* charArrayToParse input string into an array of char* with each
  // element of the array pointing to 1 word (space delimited)

  void outputInput(string*, int);
  // This method is simply a debugging method used to make sure that I am
  // parsing the input correctly.

  void doUndefinedCommand(char*);
  // If the user typed a command that is not recognized by the debugger, this
  // method will be called to inform the user.

  void doBreakOptions();
  // While the command was renamed to "list", the method name wasn't. This
  // method is only called in the command line version of the debugger. It
  // displays the current transformation that is being debugged to the user and
  // beneath it displays the same transformation with numbers before all the
  // breakable statements. That way, it is easy for the user to select which
  // statement upon which to break.

  void doAddBreak(char* whereToBreak, int key_number, CRuleBlock* toAddBreak =
NULL);
  // This method will add a break to the indicated key number. If the parameter
  // whereToBreak is NULL then the parameter key_number is the indication of
  // which statement to add the break. Otherwise the integer conversion from
  // the char* whereToBreak is used. The default transformation to add the
  // break is the current transformation. Otherwise, a RuleBlock* must be
  // passed to indicate which transformation to add the break.

  void doAddBreakSpecific(char*, char*, int);
  // This method will add a break to the indicated key number to the
  // transformation transName. If the parameter whereToBreak is NULL then the
  // parameter key_number is the indication of which statement to add the
  // break. Otherwise the integer conversion from the char* whereToBreak is
  // used. The parameter transName is expected to come in as "<transname>" so
  // the < and > have to be eliminated before searching for the transformation
  // in the transformation list _transList.

  void doAddConditionBreak(char*, int);
  // This method will add a conditional break to the indicated key number. If
  // the parameter whereToBreak is NULL then the parameter key_number is the
  // indication of which statement to add the break. Otherwise the integer
  // conversion from the char* whereToBreak is used.

  void doAddConditionBreakSpecific(char*, char*, int);
  // This method will add a condition break to the indicated key number to the
  // transformation transName. If the parameter whereToBreak is NULL then the
  // parameter key_number is the indication of which statement to add the
  // break. Otherwise the integer conversion from the char* whereToBreak is
  // used. The parameter transName is expected to come in as "<transname>" so
  // the < and > have to be eliminated before searching for the transformation
  // in the transformation list _transList.

  void doPrint(char*, CState *);
```

```
// This method will print in textual form the KOLA tree depending on toPrint.
// If toPrint is null or is current, this method will print to the screen the
// textual version of the current KOLA tree. If toPrint is "root", then this
// method will print the KOLA tree starting from the query root. Otherwise,
// it will print the KOLA tree bound to the variable inidicated in doPrint.

void doDisplay(char*, CState *);
// This method will display in textual form (or if Visual COKO is running
// then in visual form) a KOLA tree which is indicated by the toDisplay
// parameter. If toDisplay is null or is current, this method will display
// the textual (or visual) version of the current KOLA tree. If toPrint is
// "root", then this method will display the KOLA tree starting from the
// query root. Otherwise, it will print the KOLA tree bound to the variable
// inidicated in doDisplay. The biggest difference between the doDisplay and
// doPrint methods is the display means continually display it (don't just
// print it once). For instance, if a variable is to be displayed, it should
// be displayed every time the debugger stops until the variable is no longer
// in context. This method accomplishes this by adding the variable name to a
// display list which is checked every time the debgger stops. There is one
// variable list in every rule block in _curr_rbList and accessed using the
// topSym() method.

void doUndisplay(char*, CState *);
// This method turns of the KOLA tree indicated by the toDisplay paramater
// that the doDisplay method turned on. If it was a variable that was being
// displayed, it is removed from the current rule block list. Otherwise,
// the approporiate flag is turned off.

void doClearBreak(char*, int);
// This method will clear a break at the indicated key number. If the
// parameter whereToClear is NULL then the parameter key_number is the
// indication of which statement to add the break. Otherwise the integer
// conversion from the char* whereToClear is used.

void doClearBreakSpecific(char*, char*, int);
// This method will clear a break at the indicated key number in the
// transformation transName. If the parameter whereToClear is NULL then the
// parameter key_number is the indication of which statement to clear the
// break. Otherwise the integer conversion from the char* whereToClear is
// used. The parameter transName is expected to come in as "<transname>" so
// the < and > have to be eliminated before searching for the transformation
// in the transformation list _transList.

void doHelp(char*);
// If the user typed the command: "help" this method will be called. It
// simply displays a list of all the command the user can type. At some
// point, this should be updated so that the user can type "help clear" or
// "help print" and get more detailed instructions on those specific
// commands.

int analyzeInput(char**, int, CState*);
// This method is called once the input is parsed to perform the appropriate
// command. It checkes the first parameter (which in every case is the
// command) and calls the appropriate method depending on the command. Some
// commands don't need separate methods, so they are dealt with directly in
// this method. An integer is returned that specifies whether the command
// should return control back to the COKO execution (such as continue or
// next) or if the user should retain control (as in the case of most other
// commands). 0 means return control to the execution, and 1 means retain
// control in the debugger.

void outputNode(int, char*, CIdent*);
// This method is used only in the command line version of the debugger. It
```

```
// takes a CIdent (that is holding the value of a variable) and outputs this
// variable (val) and its textual value along with its display number (num).

void displayAndDeleteList(SymbolList*, CState*);
// This method takes a symbol list (sym) and checks to see if each variable
// in this list is currently bound in the current state (s). If it is, then
// it is displayed. Otherwise, it is removed from the list. It also displays
// the current and root trees of the appropriate flags are set to true

void reaffirmAndDeleteList(CState* s);
// This method is used only in Visual COKO. The _svl is a list of all windows
// currently being displayed by the visual debugger besides the root,
// current, and isolate tree display windows (leaving just windows that
// display branches of the tree that are bound to variables). For each window
// specified, this method looks up the value of the variable and sets the
// approprate variable in the window so that the correct tree will be
// displayed. In other words, this method "refreshes" all of the variable
// display windows.

void doSet(char*, char*);
// If the user typed a command that started with the word "set", this method
// will be called. There are three possible flags that the user can set:
// set rule on/off, set pattern on/off, and set success on/off. Set rule on
// will display to the screen information about rule firing every time COKO
// attempts to fire a rule. Set pattern on right now does nothing. Set
// success on will print to the screen the success value of every COKO
// statement after that statement is finished executing. This method just
// checks the value of the second parameter (par1) for "rule". "pattern", and
// "success" (and prints an error if it is not one of those three) and sets
// the appropriate flag to on or off depending on the thrid parameter (par2).

void doStack();
// If the user typed the command: "stack" this method is callled and will
// call the displayStack in vis (if Visual COKO) or on the _curr_rbList
// directly if the command line version is being run. These methods will
// simply print to the screen in some way the stack of all the COKO
// transformation calls up to the current COKO transformation.

void doExit();
// If the user typed the command: "exit" or "quit" this method is callled and
// will ask the user if they really want to exit. If they do, a system exit
// is called.

int doRun();
// If the user typed the command: "run" this method is called and will ask
// the user if they really want to restart the program. If they do, the
// necessary variables are reset, the restart flag is set to true, and the
// program continues. Later on, the program will check the value of the
// restart flag (using the really_execute method) to stop excecuting the
// current run.

NonType* findNodeFromNumHelper(NonType*, int);
// This method initializes the member variable histCounter to 0 and then
// calls the findNodeFromNum method which finds the "num"th element in a
// preorder traversal of the parameter: tree.

NonType* findNodeFromNum(NonType*, int);
// This method recurses through the KOLA tree parameter (tree) counting each
// node that it visits. Once it has visited num nodes, it returns the tree
// that is rooted at the current node.

void visMarkVariables(CState*);
// This method takes a State (s) and goes through the environment of that
```

```
// state. For each variable that is found in the environment, it assigns a
// variableNum to that node starting from 4, and then counting by twos
// upwards (4, 6, 8, 10, etc). Before setting the variableNum of the NonType
// (KOLA tree) stored in that variable, it checks to see what it used to be.
// If it was negative (meaning that a COKO transformation is about to be
// called on this node) this information is preserved by assigning it a new
// variableNum that is -1 times what it would have otherwise given it (ie
// instead of giving it 4, it gives it -4). Once the variableNum has been
// changed for each KOLA tree pointed to by each variable, s->Store is
// rebuilt to reflect these changes in variableNums (this is because
// unfortunately, the COKO system was built without the debugger in mind, and
// for this reason the KOLA trees that are pointed to by each variable in the
// environment are only copies of the branches of the corresponding current
// tree, so any changes made to these copies must also be made to the current
// KOLA tree in s->Store).

void scaleVariableNums(NonType*);
// This method takes a KOLA tree input (currNT) and scales the variableNum
// of each node in the tree to a number between [-2, 1]. In so doing, it
// retains the sign of the variableNum (if it was negative before it will
// remain negative and if it was positive before it will remain positive or
// 0) and whether or not it was divisible by 2 (odd numbers remain odd, even
// numbers remain even). Thus the mapping is:
// negative and odd  --> -1
// negative and even --> -2
// positive and odd  --> 1
// positive and even --> 0
//
// By scaling variableNums to be within [-2, 1] this method takes away
// magnitude information from the variableNums where each variable number
// corresponds to a variable in the environment. This magnitude information
// is set in the method visMarkVariables, which is usually called after the
// the variableNums have been scaled in this method.

NonType* findCurr(NonType*);
// This method looks through its KOLA tree parameter (currNT) and finds the
// parent of the node whose variableNum is not even (which signifies that
// that is the "current" node of the tree and should be drawn with the color
// green on the visual display of that tree). It returns a pointer to this
// parent.

int findCurrWhich(NonType*);
// This method looks through its KOLA tree parameter (currNT) and finds the
// parent of the node whose variableNum is not even (which signifies that
// that is the "current" node of the tree and should be drawn with the color
// green on the visual display of that tree). After finding the parent of
// this node, it returns which child of the parent this node is (ie it
// returns 1 if it is the first child of the parent, 2 if it is the second
// child, etc). This information is used in the breakLoop method to replace
// this branch of the tree (from this special node downwards) with s->Store.
// But in order to call the replaceChild method on the parent, the second
// parameter needs to know which child - hence this method is called.

void resetVariables();
// This method is called whenever the debugger is restarted. Right now, all
// it does is set tempStmt back to NULL, but my guess is that certain other
// variables must be reset as well, and that is why the program crashes
// sometimes when the user restarts debugger execution (by pressing the run
// button). This method should be revisited.

void doTrack(char*);
// If the user typed the command: "track <transname>" this method is called
// and will place the specified transformation on the tracking list. Once on
```

```
  // the tracking list, the user can place breaks on that transformation even
  // if it is not the current tranformation that is being executed.
```

VC.C methods:

```
public:

  MyMainWindow(/*NonType*, CRuleBlock*, */DDebugController*, RuleBlockList*,
RuleBlockList*, DTransformationList*);
  // constructor

  void displayRBInfo(char*, int, LCSubstring *, int, NonType*, NonType*,
SymbolList*, char*, CState*);
  // This is the central method of this class. It is called from the breakLoop
  // method of the debug controller after it has figured out the values of all
  // of the above parameters. The first 3 parameters are used to display the
  // text of the current transformation in main window, the next 4 parameters
  // are used to display the root and current KOLA trees in their respective
  // windows. The final parameter is the current state of the system and is
  // used by the setCurrInformation method to find the values of variables.
  // After passing all of this information to their respective windows, these
  // windows are refreshed.

  char** getParsedArray();
  // This method returns the parsedArray variable. It is called from the debug
  // controller when it is ready to analyze the input.

  bool close(bool);
  // Calls the quit() slot.

  void insertConsoleText(const char*);
  // This method takes some text (toInsert) and inserts it at the end of the
  // console window.

  void dealWithSelection(QString*);
  // This method is called when the user typed in a command. This command is
  // passed through the QString* parameter str. This string is then parsed by
  // the debugger and control is returned to the Debug Controller.

  void clearParsedArray();
  // This method deletes every element in the member variable parsedArray and
  // frees its corresponding memory.

  void setCurrInformation(SymbolList*, char*, NonType*, CState* s);
  // This method updates the important variables in every paint window that
  // displays the KOLA trees and the central widget (textual transformation
  // display) window. It deletes windows that are no longer useful and
  // refreshes the ones that are.

  void enableDisable(int);
  // This method makes sure that both the break/condition or clear options
  // aren't available at the same time. If the current key number has any kind
  // of break on it, then the break and condition buttons and commands (in the
  // command menu) are disabled while the clear button and command is enabled.
  // If the key number has not break, then the reverse is done.

  int checkDoAgain(int);
  // This method takes an integer parameter (which) that specifies which of the
  // three dialog boxes to display. The corrosponding dialog box is then
  // the displyed, and the user response is returned.

  int getDisplayRT();
```

```
// This method checks to see if the root KOLA tree display window is
// currently being displayed. If so, then it returns true, otherwise false.

int getDisplayCT();
// This method checks to see if the current KOLA tree display window is
// currently being displayed. If so, then it returns true, otherwise false.

void setDisplayRT(int);
// This method takes a variable int (x) and will display the root KOLA tree
// display window to true if x == 0. Otherwise, it will hide this window. It
// updates the view menu accordingly.

void setDisplayCT(int);
// This method takes a variable int (x) and will display the current KOLA
// tree display window to true if x == 0. Otherwise, it will hide this
// window. It updates the view menu accordingly.

void setSVL(DSymbolVisList* svl);
// This method sets the member variable _svl to be equal to the symbol visual
// display list parameter (svl)

QPopupMenu* getView();
// This method returns the view popupmenu stored in member variable: view

int insertIntoView(char*);
// This method inserts into the view menu an new item labeled by the
// parameter: toInsert. It then connects this item SIGNAL to the SLOT:
// showItem. Finally, it returns the index of this new inserted item in the
// view menu.

void removeFromView(int);
// This method removes from the view menu the item with the index
// corresponding to the int parameter: index.

NonType* doIsolate(NonType*,DWhichList*);
// This method refreshes the KOLA tree isolation starting from a particular
// root tree (rootTree) using a particular Which List (wl). Each element
// in wl is an integer indicating how to travese the root tree from the
// given point by telling it which child to follow. For instance, if the
// which list contains: 2 1 1, then this means that the isolation is the
// second child of the first child of the first child of the root tree. This
// method returns the tree derived from this isolation algorithm.

void insertIntoSVL(char*,DWhichList*,int,DScrollWidget*,DPaintWindow*,int);
// This method simply passes along these parameters to the insert method of
// the symbol visual display list. This list contains the windows of all of
// the additional windows besides the root KOLA tree and current KOLA tree
// display windows (hence isolations and variable display windows).

void displayStack();
// This method recalculates what should be in the stack window by calling the
// displayStack method on the _curr_rbList and then sets the text of the
// stack display window to be this returned text. The stack window is then
// display and the view menu is updated accordingly.

void dbGo();
// This method simply tells the debug controller that it is ok to regain
// control by calling its setReadyToGo method.

void removeFromTransList(char*);
// If the user closed a transformation window, we assume that the user no
// longer is interested in it, so it gets removed from the list of viewable
// transformations.
```

```
  void expandTrans(char* transName = NULL);
  // This method will expand a transformation (meaning it will open a new
  // window with the transformation code). Which transformation is expanded can
  // be indicated in two ways. The first way is by simply passing the name of
  // the transformation to the method (parameter transName). The other way is
  // by double clicking the appropriate line in the stack window which this
  // method can figure out by seeing which line is active in the stack display
  // window.

  void updatePaintWindows(NonType*, NonType*, SymbolList*, char*);
  // This takes a pointer to the root KOLA tree (rtTree) and current KOLA tree
  // (ctTree) and sets these to be the trees of the root display and current
  // display windows. In addition, it passes the SymbolList (sym) and execName
  // string to these windows. Finally it repaints them.

  void closeAllSVLWindows();
  // This method goes though each element in the symbol visual display list
  // (_svl) and closes and deletes each window.

public slots:

  void quit();
  // This slot gets called if the user tried to exit the program (by pressing
  // the exit button, by closing the window, etc). This method places the word
  // quit on the console and returns control to the debug controller which
  // will make sure that the user really wants to quit.

  void step();
  // This slot gets called if the step button is pressed. The word "step" is
  // written to the console and control is given back to the debug controller.

  void next();
  // This slot gets called if the next button is pressed. The word "next" is
  // written to the console and control is given back to the debug controller.

  void cont();
  // This slot gets called if the continue button is pressed. The word
  // "continue" is written to the console and control is given back to the
  // debug controller.

  void doBreak(); //breaking naming convention because break name not allowed
  // This slot gets called if the break button was pressed. It figures out
  // which key number the user wanted the place the break upon, and enters
  // break <kn> into the console, updates the parsedArray, and returns control
  // to the Debug Controller.

  void doCondition();
  // This slot gets called if the condition button was pressed. It figures out
  // which key number the user wanted the place the conditional clear break
  // upon, and enters condition <kn> into the console, updates the parsedArray,
  // and returns control to the Debug Controller.

  void doClear();
  // This slot gets called if the clear button was pressed. It figures out
  // which key number the user wanted the clear the break upon, and enters
  // clear <kn> into the console, updates the parsedArray, and returns control
  // to the Debug Controller.

  void doRun();
  // This slot gets called if the run button was pressed. It places the word
  // "run" on the console and returns control to the debug controller which
  // which make sure that the user really wanted to restart execution of the
```

```
// root transformation.

void toggleStack();
// This slot gets called if the stack option in the view menu is selected.
// If the stack window was currently being displayed, it gets hidden.
// Otherwise it gets displayed.

void doStack();
// This slot gets called if the stack window is to be refreshed or displayed.
// It places the word "stack" on the console and returns control to the debug
// controller.

void showRT();
// This slot will display the root KOLA tree window if it is currently
// hidden, and will hide it if it is currently being displayed. The view
// menu is updated accordingly.

void showCT();
// This slot will display the current KOLA tree window if it is currently
// hidden, and will hide it if it is currently being displayed. The view
// menu is updated accordingly.

void showItem(int);
// This slot takes an int (index) which is the index of an item in the view
// menu and will display that item if it is currently hidden, and will hide
// it if it is currently being displayed. The view menu is updated
// accordingly.

void showTransformation(int);
// This slot takes an int (index) which is the index of an item (in
// particular a transformation) in the view menu and will display that
// transformation if it is currently hidden, and will hide it if it is
// currently being displayed. The view menu is updated accordingly.

void doExpandTrans();
// This slot is called from the popup menu derived from right-clicking on
// the stack window. It will put the words "track <name-of-trans> to the
// console and allow the debugger to execute the command.
```

Conclusion

Visual COKO enables a query optimization developer to visually trace the execution of a COKO transformation, one step at a time. It functions as a debugger by providing commands that control the execution of the COKO code, allowing the programmer to step through the code and to find the source that is causing the transformation to behave unexpectedly. The debugger commands described (step, break, continue, etc.) are based on the commands of common debuggers such as *gdb*. Visual COKO functions as a visual aid by displaying the original KOLA query parse tree, along with branches of this tree corresponding to variable bindings, zoomed views, and the current branch that the transfomration is working on, throughout execution. These visual tools allow the user to

visually observe and understand how the COKO code and KOLA rules transform the original query. By functioning both as a debugger and a visual aid, Visual COKO greatly facilitates the COKO programming process and hence the development of a query rewriting engine.

References

[1] M. Cherniack and S.B. Zdonik. Rule languages and internal algebras for rule-based optimizers. In Proceedings of the ACM SIGMOD *International Conference on Management of Data*, Montreal, Quebec, June, 1996.

[2] M. Cherniack and S.B. Zdonik. Changing the Rules: Transformations for rule-based optimizers. In Proceedings of the ACM *SIGMOD International Conference on Management of Data*, Seattle, WA, June, 1998.

[3] R. Ganski and H. Wong. Optimization of Nested SQL Queries Revisited. In Proceedings of the ACM *SIGMOD International Conference on Management of Data*, pages 23-33, San Francisco, 1998.

[4] G. Graefe and D.J. DeWitt. The EXODUS Optimizer Generator. In Proceedings ACM-SIGMOD *International Conference on Management of Data*, pages 160-172, San Francisco,1987.

[5] L.M. Haas, J.C. Freytag, G.M. Lohman, and H. Pirahesh. Extensible Query Processing in Starburst. In Proceedings ACM-SIGMOD *International Conference on Management of Data*, pages 377-388, San Francisco,1987.

[6] J. Hellerstein and R. Avnur. Eddies: Continuously Adaptive Query Processing. In Proceedings of the ACM *SIGMOD International Conference on Management of Data*, pages 261-272, Dallas, 2000.

[7] N. Kabra and D.J. DeWitt. Efficient Mid-Query Re-Optimization of Sub-Optimal Query Execution Plans. In Proceedings ACM-SIGMOD *International Conference on Management of Data*, pages 106-117, Seattle, 1998.