Visual COKO: A Debugger for Query Optimizer Development

Daniel Abadi
Brandeis University
dna@cs.brandeis.edu

Mitch Cherniack
Brandeis University
mfc@cs.brandeis.edu

I. Introduction

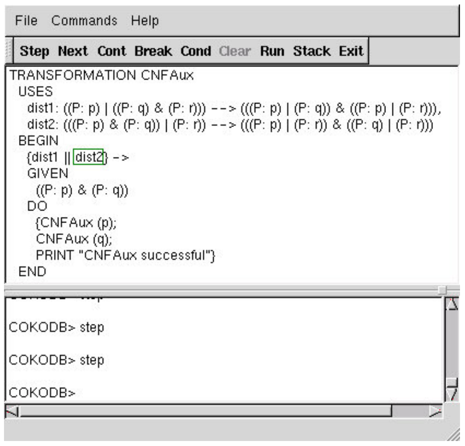
Query optimization generates plans to retrieve data requested by queries, and query rewriting (rewriting a query expression into an equivalent form to prepare it for plan generation) is typically the first step. COKO-KOLA introduced a new approach to query rewriting that enables query rewriters to be formally verified using an automated theorem prover [CZ96]. KOLA is a language for expressing term 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 [CZ98].

COKO is a programming language designed for query optimizer development. Programming languages require debuggers, and in this proposal, we illustrate our 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.

II. An Example

Figure ** (show code for CNFAux in Figure) shows a COKO transformation that converts a KOLA predicate into CNF. The transformation lists two KOLA rules in its "USES" section (dist1 and dist2) and an algorithm between keywords BEGIN and END. When called on a parse tree for a KOLA predicate, CNFAux first fires dist1 and dist2. If either succeeds, the query is transformed into the form "P & Q" and CNFAux is recursively called on P and Q.

Below we demonstrate how Visual COKO can be used to trace the execution of CNFAux on the KOLA predicate, "(pk & qk & rk) | sk". ('&' and '|' are the KOLA equivalents of $\wedge$ and $\vee$ respectively.) In the screenshot below, the left window displays the CNFAux transformation (with the green rectangle identifying the COKO statement currently being executed) and the right window displays the query tree for the KOLA predicate above.

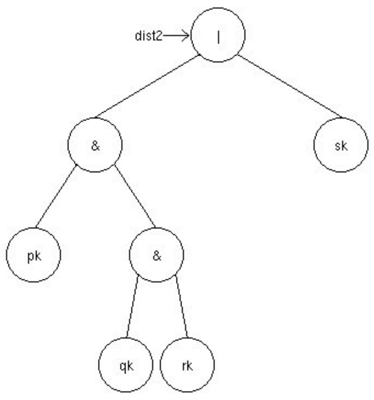


```
TRANSFORMATION CNFAux
  USES
    dist1: ((P: p) | ((P: q) & (P: r))) --> (((P: p) | (P: q)) & ((P: p) | (P: r))),
    dist2: (((P: p) & (P: q)) | (P: r)) --> (((P: p) | (P: r)) & ((P: q) | (P: r)))
  BEGIN
    {dist1 || dist2} ->
    GIVEN
      ((P: p) & (P: q))
    DO
      {CNFAux (p);
      CNFAux (q);
      PRINT "CNFAux successful"}
  END
```

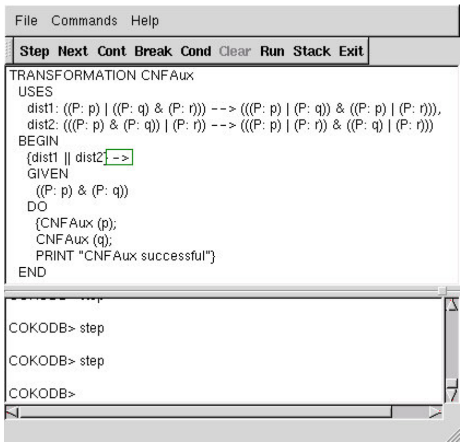
```
COKODB> step

COKODB> step

COKODB>
```



Control of transformation execution is by way of 'next' and 'step' commands, as with traditional debuggers such as gdb. When the 'next' button is pressed, rule dist2 is fired, and the screenshot below is displayed. The query tree is transformed as a result of the successful rule firing; it is now of the form P & Q.

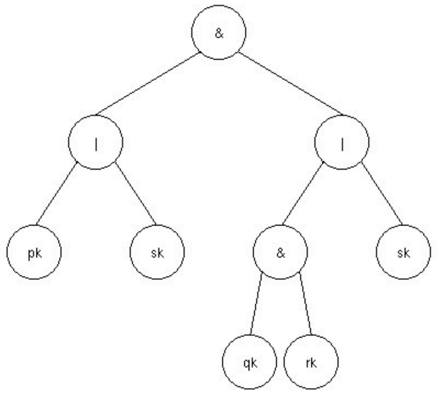


```
TRANSFORMATION CNFAux
  USES
    dist1: ((P: p) | ((P: q) & (P: r))) --> (((P: p) | (P: q)) & ((P: p) | (P: r))),
    dist2: (((P: p) & (P: q)) | (P: r)) --> (((P: p) | (P: r)) & ((P: q) | (P: r)))
  BEGIN
    {dist1 || dist2} ->
    GIVEN
      ((P: p) & (P: q))
    DO
      {CNFAux (p);
      CNFAux (q);
      PRINT "CNFAux successful"}
  END
```

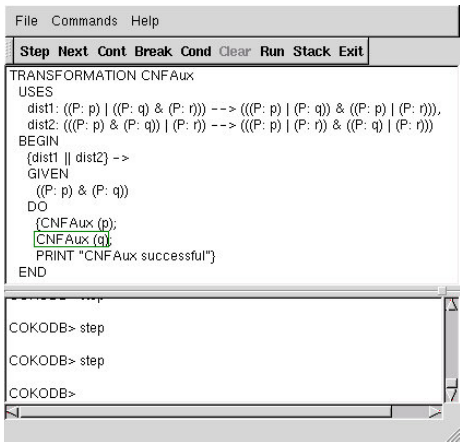
```
COKODB> step

COKODB> step

COKODB>
```



A few steps later, CNFAux is about to be called recursively on the right branch, as shown below:

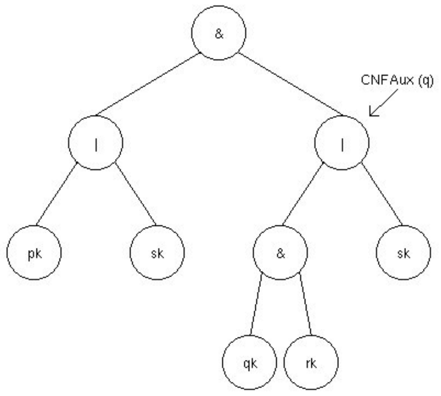


```
TRANSFORMATION CNFAux
  USES
    dist1: ((P: p) | ((P: q) & (P: r))) --> (((P: p) | (P: q)) & ((P: p) | (P: r))),
    dist2: (((P: p) & (P: q)) | (P: r)) --> (((P: p) | (P: r)) & ((P: q) | (P: r)))
  BEGIN
    {dist1 || dist2} ->
    GIVEN
      ((P: p) & (P: q))
    DO
      {CNFAux (p);
      CNFAux (q);
      PRINT "CNFAux successful"}
  END
```

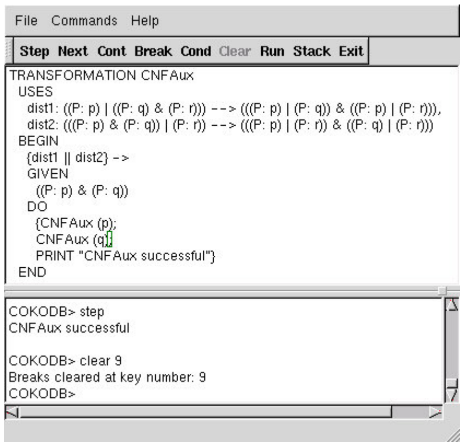
```
COKODB> step

COKODB> step

COKODB>
```



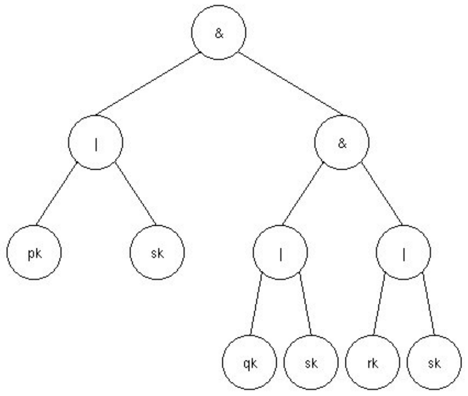
The recursive call to CNFAux shows the original predicate has been converted to CNF.



```
TRANSFORMATION CNFAux
  USES
    dist1: ((P: p) | ((P: q) & (P: r))) --> (((P: p) | (P: q)) & ((P: p) | (P: r))),
    dist2: (((P: p) & (P: q)) | (P: r)) --> (((P: p) | (P: r)) & ((P: q) | (P: r)))
  BEGIN
    {dist1 || dist2} ->
    GIVEN
      ((P: p) & (P: q))
    DO
      {CNFAux (p);
      CNFAux (q);
      PRINT "CNFAux successful"}
  END
```

```
COKODB> step
CNFAux successful

COKODB> clear 9
Breaks cleared at key number: 9
COKODB>
```



III. The Tool

Visual COKO Commands	
Command	Function
Step	Initiates execution of the next COKO statement
Next	Initiates execution of the next COKO statement in the current transformation
Break	Causes debugger to pause execution once the indicated statement is reached
Continue	Continue execution of transformation until the next break statement
Clear	Removes break from indicated statement
Condition	Causes debugger to break after indicated statement if that statement returned a success value of true
Set success on/off	Causes debugger to display success value after each executed statement
Set rule on/off	Causes debugger to display rule information after each attempt to fire a rule

Visual COKO provides a similar interface to the standard debugging tool, gdb. Like gdb, Visual COKO includes the standard program execution control commands: *step*, *next*, *continue*, *break*, and *clear*. These commands parallel the C debugging commands in that, for the *step* and *next* commands, a call to another COKO transformation is treated like a function call.

Aside from providing control of transformation execution, Visual COKO also displays the current state of the rewriting computation after each step. This is accomplished by displaying the parse tree for the query expression being rewritten (expressed in KOLA). Visual COKO has three display options. The first option is to display the entire query tree. The second option is to display only the current branch that the transformation is working on. The final option is to display a branch of the tree that is bound to a temporary variable that will be later used in pattern-matching.

Like the term rewriting rules that they generalize, COKO transformations are either successful or unsuccessful upon firing. Similarly, every statement within a COKO transformation also succeeds or fails. These success values determine the flow of control in a COKO program. For example, entire sections of COKO code might be contingent on a successful pattern-match statement. The *set success on* command will display the success value of every completed statement as they occur. By reporting the success values as they are generated, Visual COKO assists the user in locating logical errors in control flow.

The debugger also provides a conditional break command. Unlike gdb where conditions are based on data values, Visual COKO bases conditions on the success or failure statement execution. That is, the debugger will stop on a statement with a conditional break only if the execution of that statement succeeded. This is useful for monitoring rule firings, because most attempts to fire rules fail, and therefore execution would stop infrequently. Conditional breaks are inserted by way of the *condition* command.

We propose to demonstrate Visual COKO. Visual COKO enables a query optimization developer to visually trace the execution of a COKO transformation, one step at a time. In functioning both as a debugger and as a visual aid, Visual COKO facilitates the query optimizer development process.