Design Documentation – COKO Compiler

Joon Suk Lee

May, 1998

1. Introduction

Since the processing times between two equivalent queries can vary and users of the database system usually input a query that is not in the most efficient form, it is the responsibility of the system to transform the input query into an equivalent query that can be computed more efficiently. Query-to-query transformation is a query optimization technique that is widely used in object databases as well as relational databases. However, not all transformations preserve the equivalency between two queries (input query and output query) and produce more efficient output queries. Correctness and efficiency are the metric, which determine the usefulness of transformations.

Rule-based optimizers and optimizer generators use rules to specify transformations of queries. Rules act directly on query representations, which typically are based on query algebras. KOLA is a combinator-based algebra rather than a variable-based algebra. While variable-based algebras use variables to name manipulated data, combinator-based does not use variables. Combinator-based algebras have several advantages over variable-based algebras. Optimizers that use variable-based algebras require supplementary codes to express rules and these code fragments make rules difficult to understand and prove correct (correctness of the rules depend on the correctness of the code fragments.) Details of the KOLA algebra and its advantages over variable-based algebras are described in [CZ96].

Rewrite rules are declarative expressions of transformations. Rewrite rules consist of matching part where variable unification occurs and building part where transformed queries are constructed. COKO is a language with which to express transformations that transform KOLA queries. A COKO transformation is a set of KOLA rewrite rules as well as a firing algorithm controlling their firing. The COKO compiler described here can compile COKO transformations and generate C/C++ code, which then can be compiled and executed to carry out actual transformations of KOLA queries. This documentation describes the details of COKO and COKO compiler.

2. Background

2.1 KOLA

KOLA is a variable-free query representation for rule optimizers. By removing variables from query representations, KOLA avoids the problems that variable-based algebras introduced. KOLA query representations do not require supplementary code fragments for rewrite rules and therefore do not impede the formation of declarative rules. The KOLA operators are listed in the following table. Most parts of the table are taken straightly from the paper [Che97]. (A few more KOLA operators are added.) A more detailed description of KOLA and the KOLA data model can be found on section 3 of the paper [Che97].

Operator	Description	Semantics
Basic Function Primitives		
ID	identity	**id** ! $x = x$
#1	projection (1)	**#1** ! $[x, y] = x$
#2	projection (2)	**#2** ! $[x, y] = y$
SHIFTL	Shift Left	**shiftl** ! $[x,[y, z]] = [[x, y], z]$
SHIFTR	Shift Right	**shiftr** ! $[[x, y], z] = [x,[y, z]]$
TWIST	twist operator	**twist** ! $[x, y] = [y, x]$
Int and Float Function Primitives (i and j denote integers or floats)		
ABS	absolute value	**abs** ! $i = \lvert i \rvert$
ADD	addition	**add** ! $[i, j] = i + j$
MINUS	subtraction	**minus** ! $[i, j] = i - j$
MUL	multiplication	**mul** ! $[i, j] = i * j$
DIV	division	**div** ! $[i, j] = i / j$
MOD	modulus	**mod** ! $[i, j] = i \bmod j$
SQUARE	squaring	**square** ! $i = i * i$
INVERSE	reciprocal	**inverse** ! $i = \frac{1}{i}$
String Function Primitives (s and t denote strings (array of chars))		
AT	string indexing	**at** ! $[s, i] = s[i]$
CONCAT	string concatenation	**concat** ! $[s, t] = s \,\Vert\, t$
PREFIX	string prefixing	**prefix** ! $[s, i] = s[1...i]$
SUBSTR	substring	**substr** ! $[s,[i, j]] = s[i...j]$
Bag Function Primitives (A and B denote bags. X denotes a bag of bags. For any type t, $\lVert t \rVert$ denotes the type of bags whose elements are all of type t)		
ELT	element extraction	**elt** ! $\lVert x \rVert = x$
SINGLETON	singleton	**singleton** ! $x = \lVert x \rVert$
SET	duplication removal	**set** ! $A = \{x \mid x \in A\}$
FLATTEN	bag flattening	**flatten** ! X = $\lVert (x)^{A(x) \bullet X(A)} \mid x \in A, A \in X \rVert$
PW	pair with	**pw** ! $[x, B] = \lVert ([x, y])^{B(y)} \mid y \in B \rVert$
CARTPROD	cartesian product	**cartprod !** $[A, B] = \lVert ([x, y])^{A(x).B(y)} \mid x \in A, y \in B \rVert$
UNI	bag union	**uni** ! $[A, B] = \lVert (x)^{A(x)+B(y)} \mid x \in A, x \in B \rVert$
INT	bag intersection	**int** ! $[A, B] = \lVert (x)^{\min(A(x), B(y))} \mid x \in A, x \in B \rVert$
DIF	bag difference	**dif** ! $[A, B] = \lVert (x)^{\max(A(x) - B(y), 0)} \mid x \in A, x \in B \rVert$
INS	insertion	**ins** ! $[x, A] =$ **uni** ! $[\lVert x \rVert, A]$

Operator	Description	Semantics
Basic Function Formers		
o	composition	$(f \circ g) \ ! \ x = f \ ! \ (g \ ! \ x)$
$\langle \ \rangle$	pairing	$\langle f, g \rangle ! x = [f \ ! \ x, g \ ! \ x]$
$\times$	products	$(f \times g) ! [x,y] = [f ! x, g ! y]$
K_f	constant function	$K_f(x) ! y = x$
C_f	curried function	$C_f(f,x) ! y = f \ ! \ [x,y]$
CON	conditional function	$\mathbf{con}(p,f,g) ! x = \begin{cases} f!x, \ if \ p?x \\ g!x, \ else \end{cases}$
PCON	partial conditional function	$\Vert \mathbf{pcon}(p,f) ! x \Vert = \begin{cases} \Vert f!x \Vert, \ if \ p?x \\ f, \ else \end{cases}$
Query Function Formers (* $\otimes$ is assumed to be commutative and associative and the expression, $x \otimes y$ is equivalent to $\otimes ! \ [x,y]$.)		
ITERATE	iteration	$\mathbf{iterate}(p,f) ! A = \Vert (f!x)^{A(x)} \mid x \in A, p?x \Vert$
ITER	iteration2	$\mathbf{iter}(p,f) ! [x,B] =$ $\Vert (f![x,y])^{B(y)} \mid y \in B, p?[x,y] \Vert$
JOIN	join	$\mathbf{join}(p,f) ! [A,B] =$ $\Vert (f![x,y])^{A(x) \bullet B(y)} \mid x \in A, y \in B, p?[x,y] \Vert$
LSJOIN	l. semijoin	$\mathbf{lsjoin}(p,f) ! [A,B] =$ $\Vert (f!x)^{A(x)} \mid x \in A, p?[x,B] \Vert$
RSJOIN	r. semijoin	$\mathbf{rsjoin}(p,f) ! [A,B] =$ $\Vert (f!y)^{B(y)} \mid y \in B, p?[y,A] \Vert$
NJOIN	nest join	$\mathbf{njoin}(p,g,h) ! [A,B] =$ $\{[x, h \ ! \ \Vert (g!y)^{B(y)} \mid y \in B, p?[x,y] \Vert] \mid x \in A\}$
UNNEST	unnest	$\mathbf{unnest}(f,g) ! A =$ $\Vert ([f!x,y])^{A(x) \bullet B(y)} \mid x \in A, B = (g!x), y \in B \Vert$
PFOLD	partial aggregate	$\mathbf{pfold}(f,\otimes) ! A =$ $\begin{cases} (f!x_1) \otimes \ldots \otimes (f!x_n) \\ such \ that \ A = \Vert x_1,\ldots,x_n \Vert, n \geq 1 \ * \end{cases}$
FOLD	total aggregate	$\mathbf{fold}(x, \quad) ! A =$ $\begin{cases} , \qquad f \\ \otimes(\ ! \ _1) \otimes \ldots \ (\ ! \ _n) \\ = \Vert \ _1, \ldots, \ \Vert, \ \geq 1 \ * \end{cases}$

Operator	Description	Semantics
Basic Predicate Primitive (x and y are the same type)		
EQ	equality	**eq** ? $=$ $=$
		$=$ $=$ $=$ $=$
		$(p \oplus f)?x = p?(f \; ! \; x)$
&	conjunction	$(p \,\&\, q)?x = (p?x) \wedge (q?x)$
|	disjunction	$(p \mid q)?x = (p?x) \vee (q?x)$
~	negation	$\sim(p)?x = \neg(p?x)$
INV	inverse	**inv**$(p)?[x,y] = p?[y,x]$
$\times$	products	$(p \times q)?[x,y] = (p?x) \wedge (q?y)$
K_p	constant predicate	$K_p(b)?x = b$
C_p	curried predicate	$C_p(p,x)?y = p?[x,y]$
Query Predicate Formers		
exists	$\exists$	**exists**$(p)?A = \exists x(x \in A \wedge p?x)$
forall	$\forall$	**forall**$(p)?A = \forall x(x \in A \Rightarrow p?x)$
ex	$\exists_2$	**ex**$(p)?[x,B] = \exists y(y \in B \wedge p?[x,y])$
fa	$\forall_2$	**fa**$(p)?[x,B] = \forall y(y \in B \Rightarrow p?[x,y])$

Table1: KOLA Operators

2.2 COKO

A rewrite rule is of the form K1 ⇒ K2 such that both K1 and K2 are KOLA expressions supplemented with unification variables. Note that these variables are not query variables. They stand for arbitrary KOLA subexpressions and are part of the rule language rather than the query language. The term COKO expression will be used to denote rule language variables together with KOLA expressions.

While a rewrite rule applies to a particular subtree of a query expression, it is often necessary to express a more global technique for transforming queries. For example, we might want to apply a given rewrite to all possible subtrees in a given expression, or we might want to express a technique for unnnesting correlated subqueries. In each of these cases, something more than rewrite rules is required. In order to explicitly program, algorithms for expressing complex transformations, COKO was developed. COKO uses rules as primitives and surrounds their firing with built-in control structures, making rule sequencing very directed and efficient.

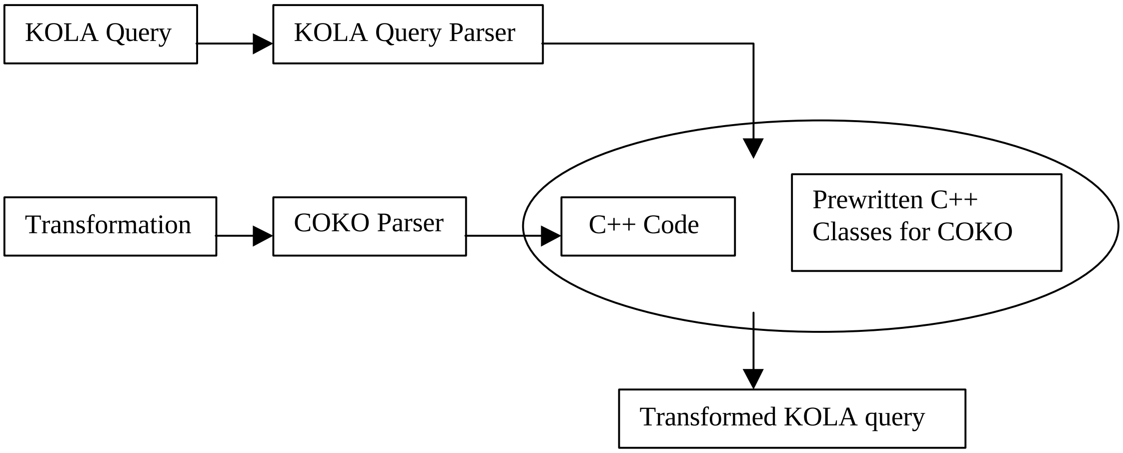
2.3 COKO Compiler

We have implemented a compiler for COKO that generates C++ classes from COKO transformations. Objects of these generated classes manipulate KOLA trees according to the firing algorithm of the compiled COKO transformation. The compiler's design is purely object-oriented. The current version of the compiler used several UNIX tools including Lex/Yacc (Bison/Flex), STL and Sicstus Prolog in its development.

3. COKO without Semantics

3.1 Design Overview

The following diagram shows the context within which COKO transformations are used. A KOLA parser parses a KOLA textual query into a KOLA parse tree. A notation "`[ ID o ID ]`" will be used to denote a parse tree equivalent of "`ID o ID`" through out this documentation. For example, a KOLA textual query, "`ID o ID ! 3`" is parsed into "`[ID o ID ! 3]`" by the KOLA parser. The COKO compiler parses COKO transformations into C++ classes. Computation of these C++ classes then act on the parse trees to manipulate them into derived forms.



3.2 Grammar

3.2.1 Success Value and Current Tree

As stated before, rewrite rules are declarative expressions of transformations. By applying these rewrite rules successively to a query, one can achieve a desired query-to-query transformation. A single action of applying a rewrite rule to a query is called rule firing. While each rewrite rule is treated as a lemma that is assumed to be always true, each rule firing has a success value. The success value tells whether a rule firing is succeeded or not. For example, firing a rule "`f o g → g`" on a KOLA tree, "`[ID o ID]`" will result in a successful transformation of the input tree to the output tree, "`[ID]`". In this case, the success value of the rule firing is set to be true. Note that even in a case, which the rule itself is algebraically absurd, the correctness of the rule has no effect on determining the success value of the rule firing. The correctness of rules can be verified by the theorem prover. (A more detailed explanation of the rule firing will be stated in later section.)

KOLA transformations are achieved by utilizing and manipulating the current tree (**curr**). The current tree is entire local KOLA tree. More accurately, **curr** is a default unification variable that always gets matched with a local root of KOLA tree. For example, suppose that we have an input KOLA tree, "`[ID o ID ! 3]`" and a COKO transformation which includes a rule called *fuse*. If *fuse* is successfully fired on the subtree, "`[ID o ID]`" then the rule firing of *fuse* will results in the transformed tree,"`[ID]`". When we first enter the transformation, current tree is “`[ID o ID ! 3]`”. However, as we enter the matching part of the rule *fuse*, the current tree becomes “`[ID o ID]`”. Similarly, by the time rule firing is completed, the current tree is “`[ID]`”. By modifying and passing the current tree between transformations and between rewrite rules, the result of rule firing and transformation is preserved.

3.2.2 Transformation

Transformations are the basic building blocks of COKO. A transformation consists of a set of KOLA rewrite rules accompanied by a firing algorithm that specifies their firing. A COKO transformation is specified by the keyword, “**transformation**” followed by the name of the transformation, optional declarations and a main body. The declaration section consists of two parts: a rule declaration part (“**USES**” part) and a property inference part (“**OptInfer**” part). “**Uses**” section defines KOLA rules that can be fired in the body of the transformation. COKO properties are reviewed in chapter 4. The main body includes a firing algorithm for the use of KOLA firing rules. The body of transformation is delineated by the keywords, **BEGIN** and **END.**

In general, a COKO transformation has the following form:

```
TRANSFORMATION transformation-name

       Uses declaration

       OptInfer declaration

BEGIN

       Stmts

End
```

Figure 3.2.2.1: General Transformation Structure
(Italics indicate optional parts.)

An example COKO transformation is shown in Figure 3.2.1.2. The “**Uses**” declaration section includes lines 2 and 3 of Figure 3.2.1.2. Lines 4 to 6 of Figure 3.2.1.2 form the main body of the transformation. Notice that line 3 is a KOLA rule, which gets fired in the main body (line 5). The “**OptInfer**” declaration not used in this example will be reviewed in next chapter.

The transformation shown in Figure 3.2.1.2 applies the rule, *fuse* to an input query. For example, successful rule firing of *fuse* on the tree, “`[ID o ID]`” will result in transformed query tree, “`[ID]`”.

```
1   Transformation Simple1
2   Uses
3     fuse: g o ID → g
4  Begin
5     fuse
6  End
```

Figure 3.2.2.2: Transformation Simple1

3.2.3 Uses

In this part of a transformation, one lists KOLA rewrite rules and/or other transformations used in the transformation firing algorithm. This section is introduced by the keyword, “**Uses**” followed by one or more KOLA rewrite rules and/or auxiliary COKO transformation names. Every rule must be proceeded by a name and ‘**:**’. For example, “`fuse`” in line 3 of Figure 3.2.3.2 is a rule name, and “`g o ID → g`” is a rule. When auxiliary transformation are declared in the section, only the name of the transformation is needed. For example, “`Simple1`” on line 5 of Figure 3.2.3.2 is an auxiliary COKO transformation name. All the identifiers used in a Uses section must be distinct. Also, notice that a comma separates each rule.

In general, the syntax of a “**Uses**” declaration section is as follows:

```
<USES Section> → USES <UseList>
<UsesList> → <UseList> , <Use>
<Use> → <Rule-Name> : <Rule>
<Use> → <Transformation-Name>
```

Figure 3.2.3.1: Uses section Syntax

such that <Rule-Name> and <Transformation-Name> are identifiers which indicates rule names or auxiliary COKO transformation names.

```
1. Transformation Simple2
2. Uses
3.   fuse: g o ID → g,
4.   fuse2: ID o g → g,
5.   Simple1
6. Begin
7.   Simple1
8. End
```

Figure 3.2.3.2: Transformation Simple2

3.2.4 Main Body

A main body of a transformation includes a firing algorithm for controlling the firing of KOLA rules. For example, line 5 of Figure 3.2.2.2 indicates that rule “`fuse`” gets fired on current KOLA tree. A main body of a transformation consists of the keyword “**Begin**” followed by optional Stmts (statements) and the keyword “**End**.” The success value of the body determines the success value of the transformation.

3.2.5 Rule Firings

A rule is of the form “E1 → E2,” where E1 and E2 are rule expressions denoting arbitrary KOLA predicates, functions, Booleans, or objects. A rule expression is any KOLA expression (function expression, predicate expression, Boolean expression, or object expression), potentially with variables and anonymous variables (DON’T CAREs.) Notice that E1 and E2 are not KOLA expressions (KOLA is variable-free algebra and KOLA expressions do not include any variables) but COKO expressions, which include unification variables. The terms, pattern, COKO expression and rule expression can be used interchangeably to denote same thing.

The left-hand side (head) of a rule (E1) is a matching part where variable unification between KOLA trees and rule expressions occurs. The variables in rule expressions get matched with some subtree of the current KOLA tree. As a result of unification, variable-to-KOLA tree bindings are built and stored in COKO environment. The right-hand side (tail) of a rule (E2) is a building part where a transformed KOLA query is built using the bindings stored in COKO environment. By replacing variables in right-hand side of the rule to a matching subtrees, the transformed KOLA tree is built. COKO environment and variable-to-KOLA tree bindings are explained fully in later sections.

3.2.6 Single Statements (Stmt)

A statement is a single action that affects a KOLA tree in place. Also, a statement returns a Boolean value that indicates whether or not the statement was successful. A statement can be one of the following forms.

i. IDENT

IDENT names either a rule or a transformation. A named rule must be declared in the **USES** section of the current transformation. A named transformation must be either the name of the current transformation or a name of another transformation declared in the **USES** section. Only rules named in the current transformation can be referenced in the body of the transformation. For example, *Transformation Bad1* of Figure 3.2.6.1 is not a legal transformation since *fuse* is not defined in *Transformation Bad1*. *Transformation Bad2* is not legal because *fuse2* is not defined in the **USES** section of the transformation.

```
Transformation Bad1
Uses
  Simple1
Begin
  fuse
End
```

Figure 3.2.6.1: Transformation Bad1

```
Transformation Bad2
Uses
  fuse: g o ID → g
Begin
  fuse2
End
```

Figure 3.2.6.2: Transformation Bad2

The semantics of a named statement is to transform the current KOLA tree according to the rule or transformation named IDENT. If IDENT names a rule, a successful firing of the rule sets success value of the statement to TRUE and has a side effect of changing the form of current KOLA tree. An unsuccessful firing of the rule sets success value of the statement to FALSE and has no effect on the current KOLA tree. If IDENT names an externally defined transformation, the statement's success value is equal to the success value of the external transformation named IDENT.

ii. IDENT (variable name)

This is almost same as statement type i., except that a rule or transformation named IDENT is not applied to the current KOLA tree but a subtree of the current tree that is pointed by a variable name. For example, the following transformation, *Apply-fuse* applies a rule *fuse* to the tree that is matched with f. Suppose that the current tree is "`[ID o ID ! 3]`". As a result of executing the satement, "**`GIVEN`**` f ! _O `**`DO`**` fuse(f)`", f is matched with the sub tree, "`[ID o ID]`". (The **GIVEN** statement will be fully explained later in this chapter.) By applying the rule *fuse* to f, f will be replaced by "`[ID]`" and the current tree will be replaced by "`[ID ! 3]`".

```
Transformation Apply-fuse
Uses
  fuse: g o ID → g
Begin
  GIVEN f ! _O DO fuse(f)
End
```

Figure 3.2.6.3: Transformation Apply-fuse

iii. [KOLA rewrite rule]

This is same as statement type i. where IDENT named a rule, except that the rule is expressed directly within […]. For example, the following transformation is equivalent *to transformation simple1* in Figure 3.2.2.2.

```
Transformation Same-as-Simple1
Begin
  [g o ID → g]
End
```

Figure 3.2.6.4: Transformation Same-as-Simple1

A defined rule in a **USES** section works as a macro. By naming and defining a rule in the **USES** section of a COKO transformation, one can use the same rule more that once without the need for writing out the same rule every time one wants to use the rule. Even in a case where a defined rule is used once in a body, it is preferred to use a named rule for readability. For example, *transformation CNF* (Figure 3.2.6.5) is equivalent to *transformation CNF-equivalent* (Figure 3.2.6.6). It is not only much easier for COKO programmers to write *transformation CNF* than *transformation CNF-equivalent*, but also easier for readers to understand *transformation CNF* than *transformation CNF-equivalent*.

```
Transformation CNF
Uses
  involution:  ~ (~ (p))  --> p,
  deMorgan1:   ~ (p & q)  --> ~ (p) | ~ (q),
  deMorgan2:   ~ (p | q)  --> ~ (p) & ~ (q),
  CNFSel
Begin
  TD {involution || deMorgan1 || deMorgan2};
  BU {involution};
  CNFSel
End
```

Figure 3.2.6.5: Transformation CNF

```
Transformation CNF-equivalent
Uses
  CNFSel
Begin
  TD {[~(~(p)) --> p]
      || [~(p & q) --> ~(p) | ~(q)]
      || [~(p | q) --> ~(p) & ~(q)]};
  BU {[~(~(p)) --> p]};
  CNFSel
End
```

Figure 3.2.6.6: Transformation CNF-equivalent

iv. **[KOLA rewrite rule] (variable name)**

This is same as statements of type ii., where IDENT named a rule, except that the rule is defined directly within […].

v. **IDENT INV**

IDENT INV describes the “inverse” (i.e., right-to-left) firing of a rule. This statement is only legal if IDENT names a rule. (i.e., Inverting a transformation is not allowed.) Also, rules can only be fired inversely if variables used in the head (left-hand side) of a rule are also used in the tail (right-hand side) of the rule. For any invertible rule r: Q **-->** Q', r INV is equivalent to the rule r': Q' **-->** Q. For example, it is possible to invert rule1 in *transformation Inverse* of Figure 3.2.6.7. The inverted form of rule1 is equivalent to rule3. However, it is not possible to invert rule2 because the right-hand-side of the rule (tail of the rule) has only one variable (g) while left-hand-side of the rule (head of the rule) has two (g and f).

```
Transformation Inverse
Uses
  rule1: g o ID → g,
  rule2: #1 ! [f, g] → f ,
  rule3: g → g o ID
Begin
  rule1 INV
End
```

Figure 3.2.6.7: Transformation Inverse

vi. **IDENT INV (variable)**

This is almost same as statements of type v.. The only difference is that a rule named IDENT is not applied to the current KOLA tree but applied to a subtree of the tree that is pointed by a variable name.

vii. **Stmt: PRINT (variable / string / CURRTREE)**

This statement is useful for debugging purposes only. The success value of this statement is always true and execution of this statement has no effect on the current KOLA tree. **PRINT V** (**V** is a variable) will print out a text representation of the KOLA tree that is bound to **V**. **PRINT CURRTREE** will print out a text representation of the current KOLA tree. **PRINT *string*** will print out explicitly named strings. To illustrate, if the current tree is "`[ID o ID ! 3]`", the output of the transformation,

```
Transformation Print-Statements
Begin
  PRINT "hello world\n";
  GIVEN f ! _O DO PRINT f;
  PRINT "\n"
  PRINT CURRTREE
End
```

Figure 3.2.6.8: Transformation Print-Statements

will be

```
Hello world
ID o ID
ID o ID ! 3
```

viii. **GIVEN Eqns DO Stmt**

Rules and transformations need not to be fired on entire KOLA trees and can instead be fired on isolated subtrees of KOLA trees. These subtrees are identified by matching patterns (COKO expressions) to the current KOLA tree using the **GIVEN** statement. A pattern resembles a KOLA expression, but can include pattern variables and anonymous variables that get bound by matching. By naming variables as arguments in subsequent rule or transformation firings, the subtrees bound to these variables can be selectively transformed. Variable matching occurs in ***Eqns*** clause of the **GIVEN** statement. The semantics of the **GIVEN** statement is like this:

```
Process Eqns
If matching process of Eqns succeeded then execute Stmt.
```

The success value for the **GIVEN** statement is true if all equations in ***Eqns*** clause successfully match and a subsequent statement (Stmt) succeed.

ix. **TD Stmt / BU Stmt**

Query trees can be traversed in bottom-up (postorder) or top-down (preorder) fashion. For any statement *S*, "**BU** *S*" performs a bottom-up pass of the KOLA query tree executing *S* on every subtree. (Analogously, "**TD** *S*" executes *S* on every subtree during a top-down pass of the KOLA query tree.) Both traversal statements return a success value of true if *S* succeeds when fired on some subtree visited during the traversal.

Unlike **GIVEN** statement, **TD** and **BU** can not be fired on isolated subtrees of the current KOLA tree instead they are always fired on entire KOLA tree. For example, the following statement is not a legal COKO statement. While first foo(f) is a legal statement, the second one is an error.

`GIVEN f ! _O, f = g o h DO {foo(f); TD foo(f) }`

x. Stmt: TRUEv Stmt / FALSEv Stmt

This statement does nothing to the current tree but it sets return value of the statement. For any statement *S*, "**TRUEv** *S*" always returns true regardless success or failure of *S*. Likewise, "**FALSEv** *S*" always returns false. For example, statement "**TRUEv TD GIVEN** f ! _O, f = g o h **DO** foo (f)" will always succeed regardless of input query. Similarly, "**FALSEv TD GIVEN** f ! _O, f = g o h **DO** foo (f)" will always fail regardless of the input query.

xi. Stmt: REPEAT Stmt

For any statement *S*, "**REPEAT** *S*" or "***** *S*" will fire *S* repeatedly until *S* no longer succeeds. The success value of **REPEAT** statement depends on the success value of the first firing of *S*. For example, firing the following transformation on a KOLA tree, "`[((((ID o ID) o ID) o ID) o ID) ! 3]`" will result in transformed KOLA tree, "`[ID ! 3]`".

```
Transformation Repeat-Test
Uses
  fuse: g o ID → g
Begin
  GIVEN f ! _O DO * fuse(f)
End
```

Figure 3.2.6.9: Transformation Repeat-Test

3.2.7 Compound Statements (Stmts)

A compound statement (Stmts) consists of two statements (Stmt) combined in one of three ways.

- Two statements can be connected sequentially by semicolons. (Sequential Multi Statements)
- Two statements can be connected by conjunction. (Conjunctive Multi Statements)
- Two statements can be connected by disjunction. (Disjunctive Multi Statements)

i. Sequential Multi Statements (S_1; S_2)

Semicolons separate two statements that are to be executed in sequence. That is, the semantics of the sequential compound statement, S_1; S_2 is:

```
Execute S1.
Execute S2.
```

The above compound statement has a success value semantics of true if any one of two statements has a success value semantics of true.

ii. Conjunctive Multi Statements (S_1 -> S_2)

"->" (THEN) separates two statements that are to be executed conditionally on success of preceding statements. That is, the semantics of the conjunctive compound statement, $S_1 \rightarrow S_2$ is:

```
Execute S1.
If S1 succeeds then
Execute S2.
```

A conjunctive compound statement, $S_1 \rightarrow S_2$ is true if S_1 is true.

iii. Disjunctive Multi Statements ($S_1 || S_2$)

"||" separates statements that are to be executed conditionally on the failure of preceding statements. That is, the semantics of the disjunctive compound statemen, $S_1 || S_2$ is:

```
Execute S1.
If S1 fails then
Execute S2.
```

The above statement has a success value of true if either S_1 or S_2 has success value of true.

3.2.8 Eqns

Eqns is a series of equations separated by comma. Eqns is true only when all the equations listed in Eqns are true.

3.2.9 Equations

Equations are used for building environment for COKO. All equations are of the form, "variable = E," where E is a COKO expressions. Equations of the form, "E" should be treated as a special case and should be interpreted as "(**curr** =) E" where **curr** is a default variable denoting current KOLA tree. The processing of an equation results in an attempted match of E with the tree previously bound to variable. Successful matching of an equation adds the variables appearing in E (and the subtrees that they match with) to the environment and sets the equation's success value to true. For example, the successful processing of ***Eqns*** clause of the **GIVEN** statement of the following transformation will adds binding f, binding f1, binding f2 and binding g to the environment.

```
Transformation Equation-Example
Uses
  fuse: g o ID → g
Begin
  GIVEN f ! _O,
        f = f1 o f2,
        f2 = g o ID   DO  fuse (g)
End
```

Figure 3.2.9.1: Transformation Equation-Example

3.3 Variable Scoping for COKO

This section describes variable scoping rule for COKO language. This section includes two main subsections. In the first half of the section, general scoping rule of COKO is explained while the second half is dedicated for explaining one special case of the scoping rule.

3.3.1 General Case.

3.3.1.1 Transformation

- All the variables appear in a transformation are local to the transformation.
- All the variable appears first time in a transformation should be a fresh one.

For example, variable f in *transformation foo* of Figure 3.3.1.1 should not be visible in transformation boo of Figure 3.3.1.2. In other word, f in transformation foo is not the same as f in transformation boo.

```
Transformation foo
Uses
  boo
Begin
  GIVEN f ! _O DO boo(f)
End
```

Figure 3.3.1.1: Transformation foo

```
Transformation boo
Uses
  fuse: g o ID → g
Begin
  GIVEN f o f1 DO fuse(f)
End
```

Figure 3.3.1.2: Transformation boo

3.3.1.2 Rule

- All the variables appear in a rule (in a **Uses** section) are local to the rule.
- All the variable appears first time in a rule should be a fresh one.

For example, f in the rule fuse (in the **Uses** section) of *transformation Variable-Scope1* of Figure 3.3.1.3 should not be visible outside of the rule. In other word, f on line 3 is not the same as f on line 6. Moreover, f in line 3 is not the same one as f in line 4.

```
1. Transformation Variable-Scope1
2. Uses
3.   fuse: f o ID → f,
4.   fuse2: ID o f → f
5. Begin
6.   GIVEN f ! _O DO fuse(f)
7. End
```

Figure 3.3.1.3: Transformation Variable-Scope1

3.3.1.3 Statement & Equation

- Every variable newly appears in a body of the transformation is local to the body.
- A variable name appears in right-hand side of an equation has different meaning as one appears in left-hand side. (A equation is either a form of A = B, or (**curr** =) B.)
- A variable appears in left-hand side of equation has to be always bounded to some part of KOLA tree while a variable name appears in right-hand side is always treated as fresh one.

For example, in *transformation Variable-Scope2* of Figure 3.3.1.4, f in line 5 is a fresh variable, which get matched with a subtree of current KOLA tree. Since f is now bound with some subtree, it is permissible to use f in left-hand side of the equation in line 6. If we change f in line 6 to f1, it will be flagged as an error since f1 is not bound to anything and it is used in left-hand side of the equation.

```
1. Transformation Variable-Scope2
2. Uses
3.  fuse: f o ID → f,
4. Begin
5.  GIVEN f ! _O,
6.        f = g1 o g2,
7.        g1 = g3 o f2  DO  fuse(f)
8. End
```

Figure 3.3.1.4: Transformation Variable-Scope2

A variable appeared in a body of transformation could be referred or used at later time in the transformation until it gets invisible. A variable becomes invisible in two cases. When a rule or transformation is invoked on a variable's ancestors, the variable becomes permanently invisible. When an already bounded variable name is used as a fresh variable, old one becomes temporarily invisible until newer variable name becomes permanently invisible. Notice that no rule or transformation can be invoked on the variable while it is invisible. For example, variable f in line 6 of *transformation Variable-Scope3* of Figure 3.3.1.5 is visible until line 8.

```
1.  Transformation Variable-Scope3
2.  Uses
3.    foo
4.    boo
5.  Begin
6.    GIVEN f ! _O,
7.          f = g1 o g2 DO
8.          { foo;
9.            GIVEN f ! _O,
10.                 f = <g1, g2> DO
11.                 boo }
12. End
```

Figure 3.3.1.5: Transformation Variable-Scope3

However, since foo is invoked on current tree, which is a parent of variable f, in line 8, f gets permanently invisible. f in line 9 is a different one than one in line 6. For example, following transformation is illegal, since f in line 9 is not bounded to anything. f gets invisible after line 8.

```
1.  Transformation Variable-Scope4
2.  Uses
3.    fuse: f o ID --> f,
4.    boo
5.  Begin
6.    GIVEN f ! _O,
7.          f = g1 o g2 DO
8.          { boo ;
9.            fuse(f) }
10. End
```

Figure 3.3.1.6: Transformation Variable-Scope4

In *transformation Variable-Scope5* of Figure 3.3.1.7, f in line 5 becomes temporarily invisible in line 7. A fresh new f masks old f until new f becomes permanently invisible. In line 9, newer f becomes permanently invisible and older f becomes visible again since f1, parent of new f is invoked on a rule fuse.

```
1.  Transformation Variable-Scope5
2.  Uses
3.    fuse: f o ID --> f
4.  Begin
5.    GIVEN f ! _O,
6.          f =  f1 o f2,
7.          f1 = f o g DO
8.          { fuse(f);
9.            fuse(f1);
10.           fuse(f) }
11. End
```

Figure 3.3.1.7: Transformation Variable-Scope5

An already bound variable cannot be used twice in left-hand side of the equation while it's visible. This was done to prevent any possible confusion resulting from variable updating and aliasing. For example, line 7 and 8 of the following transformation will cause a variable aliasing error.

```
Transformation Variable-Scope6
Uses
 foo,
 boo
Begin
 GIVEN f ! _O,
       f = g1 o g2,
       f = g1 o g3 DO foo(f)
End
```

Figure 3.3.1.8: Transformation Variable-Scope6

3.3.1.4 TopDown and BottomUp statements

TD statement and **BU** statement have a unique variable scoping rule unlike other statements. **TD** and **BU** has separated self-contained variable scope. No variable defined outside of **TD** or **BU** can be used in subsequent statement of **TD** and **BU**. As it was stated in previous section, execution of "`GIVEN f ! _O, f = g o h DO {foo(f); TD foo(f) }`" will cause "variable f is not defined" error. This error is caused due to **TD** and **BU**'s unique variable scope.

However, foo(f) in the following example is legal. Since f was defined inside of **TD** statement (in subsequent statement of **TD**), f is referable in **TD**. Only variables that can be used in subsequent statement of **TD** and **BU** are the variables defined in the subsequent statement of **TD** and **BU**.

```
TD GIVEN f ! _O, f = g o h DO foo(f)
```

3.3.2 Special Case

Having multiple occurrence of same variable name in COKO expression is not always practical and even introduces much confusion. However, COKO allows users to use same variable name more than once in some cases of COKO expressions. This section describes when the multiple occurrence of same variable name is allowed and when it is not.

3.3.2.1 Rule

It is possible to use variable name more than once in either sides of a rule, even though this feature of COKO is rarely used. Note that the rules in following transformations are not algebraically correct. However, they are used to demonstrate the usage of multiple variable occurrences.

Invoking *transformation multiple-variable-occurrence1* of Figure 3.3.2.1 to "`[ID o ID ! 3]`" will transform the query tree to "`[(ID o ID) o (ID o ID) ! 3]`". Similarly, invoking *transformation multiple-variable-occurrence2* of Figure 3.3.2.2 to "`[(ID o ID) o (ID o ID) ! 3]`" will transform the query tree to "`[ID o ID ! 3]`".

```
Transformation multiple-variable-occurrence1
Uses
  fuse: g → g o g
Begin
  GIVEN f ! _O DO fuse(f)
End
```

Figure 3.3.2.1: Transformation multiple-variable-occurrence1

```
Transformation multiple-variable-occurrence2
Uses
  fuse: g o g → g
Begin
  GIVEN f ! _O DO fuse(f)
End
```

Figure 3.3.2.2: Transformation multiple-variable-occurrence2

3.3.2.2 Statement & Equation

It is not allowed to have more than one occurrence of the same variable name in statements or equations. By disallowing multiple variable name occurrences, a possible confusion that introduced when a rule or transformation is invoked on a variable with multiple occurrences is avoided. For example, in an equation, "`f = g o g`", two g's are not the same object but they refer to the same COKO tree structure. Problem occurs when a rule gets called on g in this case. It is not clear whether a rule should be invoked on first g, second g or both. Current implementation of COKO prevents this confusion by simply disallowing multiple occurrence of a variable in equations or statements. For example, the following transformation will result in error.

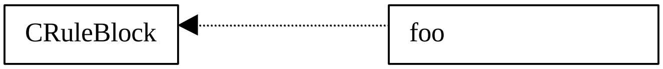
```
Transformation multiple-variable-occurrence-error
Uses
  fuse: g o ID → g
Begin
  GIVEN f ! _O, f = g o g DO fuse(g)
End
```

Figure 3.3.2.3: Transformation multiple-variable-occurrence-error

3.4 Architecture

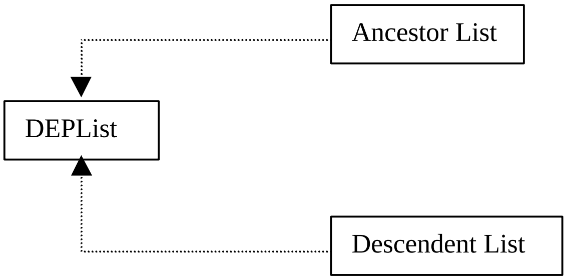
This section describes COKO language and the COKO compiler from a programmer's point of view. All the prewritten C++ classes as well as parser generated C++ classes are covered in this section. Each subsection will include brief description of C++ classes as well as OMT like diagrams, which visually demonstrates class hierarchies.

3.4.1 Transformation



Every transformation is compiled into a C++ class, which is a derived class of **CRuleBlock** class. For example, when we compile transformation foo, COKO compiler will generate a C++ class foo, which is derived from **CRuleBlock**. **CRuleBlock** class is a base class for all the transformations and has one virtually defined method, **Exec**. **Exec** method is a trigger for KOLA query transformation. Calling **Exec** method of each transformation class will trigger execution of the transformation on KOLA query.

3.4.2 Variable Dependency



Since it is allowed to redefine a variable in a GIVEN statement, it is necessary to keep track of their dependency to insure the correctness of KOLA rewrite rules. As it was stated before, an equation is of the form ***(curr =)** E* or *variable name = E* where E is any COKO expression and **curr** is a pointer to local root of the KOLA tree. Note that **curr** is not the root of KOLA input query, but is a local root of the KOLA tree in current state. Suppose the input KOLA query for the following transformations is “`[(ID o ID) o (ID o ID) ! 3]`”. Invoking the first transformation on the input query will result in changes of the KOLA tree to “`[ID o ID ! 3]`”, while invoking the second transformation on the query will result in “variable q2 is not bound to anything yet” error. This is an expected result as described in previous section.

Variable dependency check is achieved by means of attaching two kinds of dependency lists, ancestor list and descendent list, to every variable as well as **curr**. Since **curr** does not have any ancestor, only descendent list (inverse dependency list) is required, while any COKO variable needs ancestor list (dependency list) as well as descendent list (inverse dependency list). For example, **curr** in the first transformation has f, g1 and g2 in its descendent list, while f has **curr** in its ancestor list and g1 and g2 in its descendent list. Similarly, g1 and g2 has empty descendent list and has curr and f in their ancestor list. C++ object class **DEPList** is used in programming to represents dependency list. Every COKO variable is associated with ancestor and descendent list, which are derived classes from **DEPList**.

```
Transformation variable-dependency1
Uses
  foo: f o ID → f
Begin
  GIVEN f ! _O, f = g1 o g2 DO {foo(g2); foo(f)}
End
```

Figure 3.4.2.1: Transformation variable-dependency1

```
Transformation variable-dependency2
Uses
  foo: f o ID → f
Begin
  GIVEN f ! _O, f = g1 o g2 DO {foo(f); foo(g2)}
End
```

Figure 3.4.2.2: Transformation variable-dependency2

3.4.3 Environment and Identifier Search Stack

KOLA tree transformation is achieved by utilizing an environment class which holds information about the current state of the COKO program and a environment stack (identifier search stack) which stores variable-to-KOLA tree bindings.

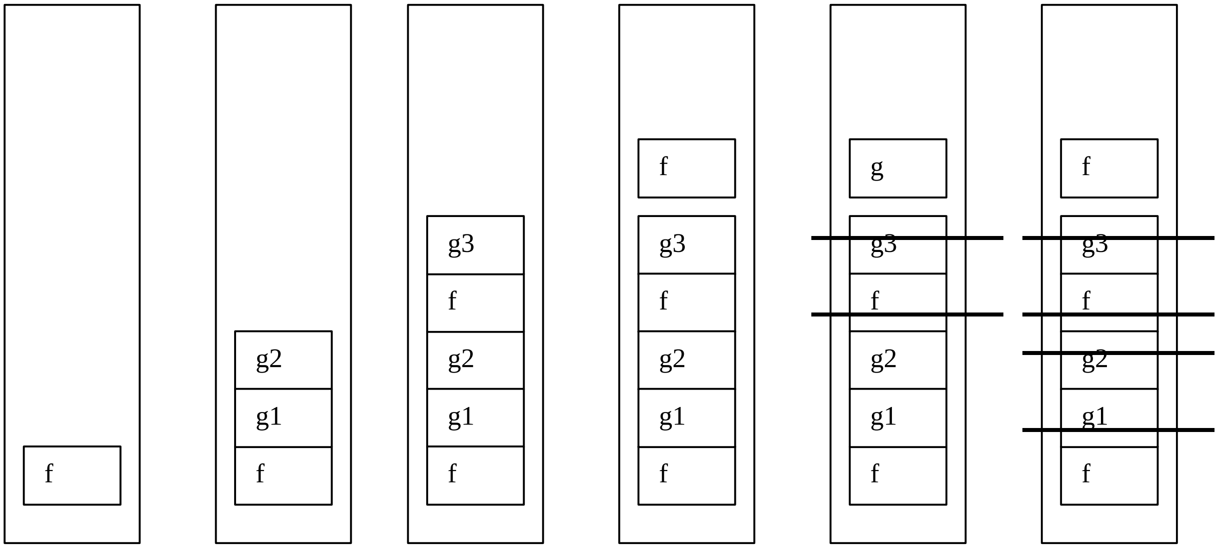
CState is a class for abstraction of the state of COKO program. Usually execution of COKO statement changes the current environment of the program and the result of all the statement executions is monitored by means of passing and receiving **CState**. **CState** includes a pointer to the local root of KOLA tree (**curr**), inverse dependency list for **curr**, a pointer to the environment stack, and a flag for specifying whether the statement execution succeeded or not as its data member.

An environment stack is implemented as a stack of a stack. Using a doubly nested stack eases keeping track of variable dependency and variable scoping. For example, consider the following transformation:

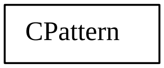
```
Transformation foo
Uses
  fuse: f o ID → f,
  fuse2: ID o g → g
Begin
  GIVEN f ! _O, f = g1 o g2, g2= f o g3 DO {fuse(f); fuse2(g2); fuse(f)}
End
```

Figure 3.4.3.1: Transformation foo

The following diagram will help visualizing changes made to the environment stack according to the execution of the transformation. Each frame of outer stack is used to distinguish different variable scope and each frame of inner stack is used to hold variable-to-KOLA tree bindings. Every time the program enters a new variable scope, an empty frame is inserted into outer stack. By executing the first equation of the GIVEN statement, the variable f to some subtree of current KOLA tree bindings (binding f) will be put into inner stack. Similarly, execution of second equation will put binding g1 and binding g2 into the inner stack. Execution of the third equation will put a new binding f and binding g3 into the inner stack. Note that this new binding f inserted into the inner stack masks old binding f. Since the stack is searched from top to bottom, there is no need to specially mark old bindings as invisible. A new binding f masks old binding f automatically and makes old one invisible until the new binding f gets permanently invisible. Execution of rule fuse requires a new outer stack frame, since the scoping level of body of the transformation and the scoping level of the rule is different. As it is stated earlier, any variable appears in a rule is local to the rule. As a result of processing matching part of the rule fuse, another variable binding f will be added to the new inner stack (a new second frame of the outer stack.) Since the building part of the rule only searches the top frame of the outer stack, there is no difficulty in finding right binding f and the correctness of the rule transformation is preserved. After execution of the rule fuse, all the variable bindings created by the rule will be permanently lost. Since we are using a nested stack structure, there is no need to search for all the variable bindings created by the rule and remove those bindings from the stack one by one. Instead, we can simply pop off the top frame of the outer stack and achieve the same effect. An execution of *fuse2* on a variable g2 will add a new outer stack frame for the use of the rule fuse2 and mark the bindings of the descendant variables of g2 as permanently invisible. As a result of marking a newer binding f invisible, the older binding f will become visible again.



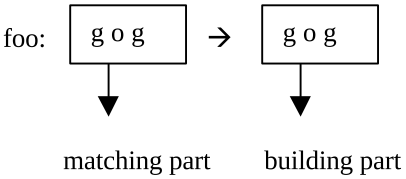
3.4.4 CPattern



CPattern class represents COKO expressions. This class includes a pointer to the actual COKO expression tree as well as several methods for manipulating the tree. Two types of methods are worth noting. One is match and the other is build. There are two different methods for matching. Since multiple occurrence of a same variable name is only permitted for rules and not for the statements and equations, it is necessary to have two different matching mechanism. While **matchSTM** method is used for statement matching, **match** method is used for rule matching.

3.4.5 Rule Matching and Statement Matching

Rule matching method of CPattern checks for match between a KOLA query tree and a pattern tree. Remember that duplicated variable entry is allowed for a rule. Consider the following rule:

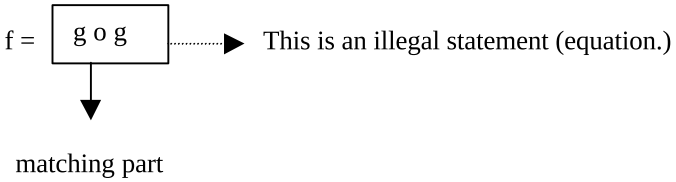


Assume that we apply this rule to a kola tree, "`[ID o ID]`". When unification occurs in a matching part, first we store first g and ID pair into the environment stack. Then the next time we face another g (second one), we search the stack for the matching pair with variable name g and simply check if the matching pair in the stack has same structure as the matching pair we have. In this case, second pair is (g, ID) as well and the rule gets fired successfully.

Now assume that we apply the same rule to a kola tree, "`[ID o (ID o ID)]`". In this case, the rule fails since the second pair is (g, ID o ID), while the first pair is (g, ID). Note that even in a case of failure, the rule itself is legal.

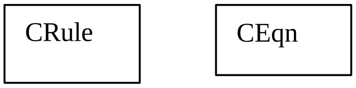
Like rule matching method, statement matching method of CPattern checks for match between a KOLA query tree and a pattern tree. However, no duplicated variable entry is allowed for statement matching. Consider the following statement:

GIVEN f ! _O,



Assume that we have a query tree, "`[ID o ID ! 3]`". Variable f gets bounded to "`[ID o ID]`" and we have to match "`ID o ID`" to "`g o g`" Matching "`ID o ID`" to "`g o g`" is perfectly legal in a rule section. However, it is illegal in statement section of the program. Remember that every variable on the right hand side of an equation must be a fresh one. However, after we match first g to ID, the second g is no longer fresh variable. Therefore, checking a duplicated variable entry is additionally required in statement matching methods.

3.4.6 Rule and Equation

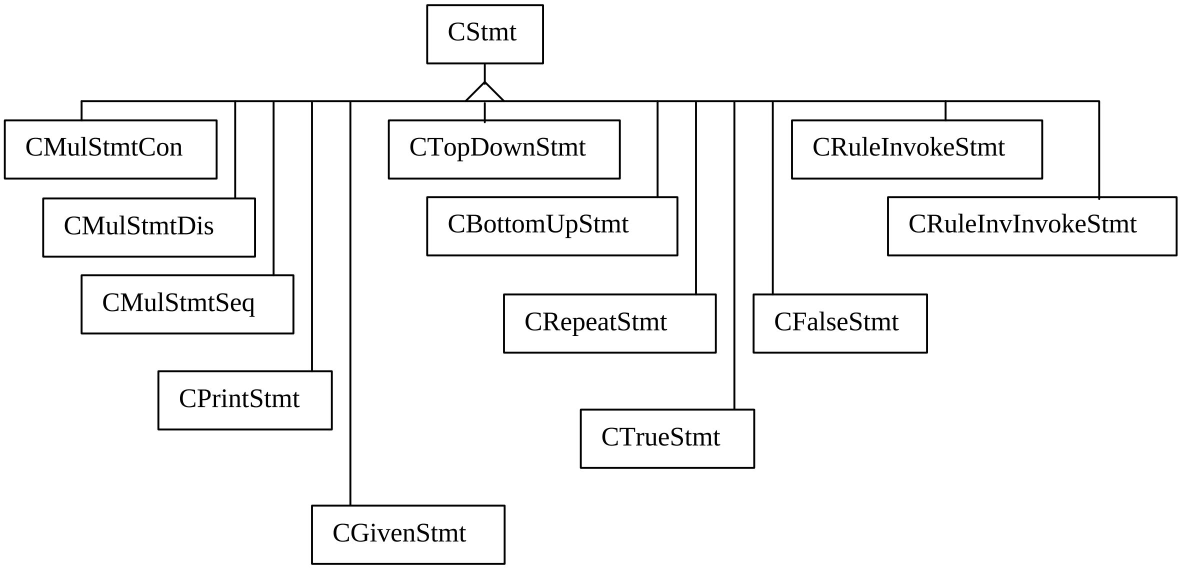


CRule class stores information about KOLA rewrite rules. It includes two **CPattern** as its data member. Each **CPattern** class represents left-hand and right-hand side of the rule respectively. Its **Exec** method is used for rule invocation and **InvExec** method is used for inverse rule invocation. Its **Exec** method calls

match method of the first **CPattern** (left-hand side of the rule) to build variable-to-KOLA tree bindings and then calls **build** method of the second **CPattern** (right-hand side of the rule) to build transformed query. Analogously, its **InvExec** method calls **match** method of the second **CPattern** (right-hand side of the rule) and then calls **build** method of the first **CPattern** (left-hand side of the rule.)

CEqn class stores information about the equation. It includes a character pointer, which represents left-hand side of the equation (variable name) and one **CPattern**, which represents right-hand side of the equation (COKO expression). Its **Exec** method not only calls **matchSTM** method of the **CPattern** to carry out matching process and to build up environment which holds variable-to-KOLA tree bindings but also manipulates dependency list for variables appear in the equation.

3.4.7 Statement



Every COKO statement is implemented with its own C++ class. Each of these classes is a subclass of the abstract class, **CStmt**, and is obligated to define a method **Exec** which takes an environment, **CState**, which includes variable-to-KOLA tree bindings and local root of the KOLA tree, as input and produces a transformed version of this environment as output.

i. CGIVENStmt

The **GIVEN** statement class includes a list of equations and another statement as its data members. The **Exec** method for **GIVEN** adds variable-to-KOLA tree bindings to its input environment by calling **Exec** methods of all equations. In a case of all **Exec** methods of equations succeed, it invokes **Exec** of its statement member sending the newly constructed environment as its argument.

ii. CMulStmtCon

The conjunctive multi statement (**CMulStmtCon**) class includes two statements as its data members. Its **Exec** method invokes **Exec** of its first statement member, then invokes **Exec** of its second statement member if the first call to **Exec** reported success. The success value of the conjunctive multi statement depends on the success value of the first statement member. In other words, if the call to the first statement reports success, the return value of the conjunctive multi statement will be set to true regardless of return value of the second statement.

iii. CMulStmtDis

The disjunctive multi statement (**CMulStmtDis**) class includes two statements as its data members. Its **Exec** method invokes **Exec** of its first statement member, then invokes **Exec** of its second statement member if the first call to **Exec** reported failure. The success value of the disjunctive multi statement is set to true when any one of two statements gets succeeded.

iv. CMulStmtSeq

The sequential multi statement (**CMulStmtSeq**) class includes two statements as its data members. Its **Exec** method invokes **Exec** of its first statement member, then invokes **Exec** of its second statement member regardless success or failure of the first statement. The success value of the disjunctive multi statement is set to true when any one of two statements gets succeeded.

v. CRuleInvokeStmt

Actual rule invocation on a KOLA tree is executed by this statement. This statement includes a pointer to a rule or transformation and a pointer to a variable as its data member. When its variable member is not NULL, a rule or transformation is invoked on the subtree bounded to the variable by calling **Exec** method of the rule or transformation. When the variable pointer is NULL, the rule or transformation is invoked on current KOLA tree.

vi. CRuleInvInvokeStmt

Inverse rule invocation on a KOLA tree is executed by this statement. This statement includes a pointer to a rule and a pointer to a variable as its data member. When its variable member is not NULL, a rule is invoked on the subtree bounded to the variable by calling **InvExec** method of the rule. When the variable pointer is NULL, the rule is invoked on current KOLA tree.

vii. CTopDownStmt

The **TopDown** statement includes a statement as its data member. Its **Exec** method invokes **Exec** method of the statement member on all the subtrees rooted by every node of current KOLA tree. The execution of the statement member on the current tree occurs as the tree is traversed in preorder style. The **TopDown** statement is true when the execution of the statement member succeeds on any of the subtrees.

viii. CBottomUpStmt

The **BottomUp** statement includes a statement as its data member. Its **Exec** method invokes **Exec** method of the statement member on all the subtrees rooted by every node of current KOLA tree. The execution of the statement member on the current tree occurs as the tree is traversed in postorder style. The **BottomUp** statement is true when the execution of the statement member succeeds on any of the subtrees.

ix. CRepeatStmt

The **Repeat** statement includes a statement as its data member. Its **Exec** method invokes **Exec** of the statement member repeatedly until the execution of statement member fails on current environment. Environment is updated each time the statement member is executed. The success value of the **Repeat** statement is same as the success value of the first execution of the statement member.

x. CTrueStmt

The **True** statement has a statement as its data member. Its **Exec** method invokes **Exec** on its statement member. The return value of this statement is always true.

xi. CFalseStmt

The **False** statement has a statement as its data member. Its **Exec** method invokes **Exec** on its statement member. The return value of this statement is always false.

xii. CPrintStmt

The **Print** statement has a string and an integer value as its data members. The integer value indicates the type of print statement. When the integer value is 1, its string member represents a variable name. In this case, its **Exec** method search for a variable-to-KOLA tree binding in the environment stack with a variable name as a key then prints out the text representation of the matching KOLA tree to standard output. When the integer value is 2, its string member represents a text string. In this case, its **Exec** method simply prints out its string member to the standard output. When the integer value is 3, its **Exec** method prints out the text representation of the current KOLA tree to the standard output.

3.5 Interface

CLASS: CRuleBlock
SYNOPSIS: This class is an abstract base class for rule blocks. When the COKO parser generates C++ code for a transformation, it will make the transformation a derived class of CRuleBlock.

Data Member	Description

Methods	Description
virtual CState *Exec(CState *s)	What to do when executed. It does nothing by default.

CLASS: CRule
SYNOPSIS: This class is for KOLA rewrite rules.

Data Member	Description
CPattern* _lhs	Left-hand side of the rule
CPattern* _rhs	Right-hand side of the rule

Methods	Description
CRule (CPattern*, CPattern*)	Constructor for rule representation. Takes two arguments that are patterns of lhs and rhs, respectively.
virtual ~CRule()	Destructor (virtual)
virtual CState *Exec(CState*)	Executes the rule.
CState *InvExec(CState*)	Executes the rule "in reverse".

CLASS: CStmt
SYNOPSIS: This class is an abstract super class for COKO statements

Data Member	Description

Methods	**Description**
virtual ~CStmt(void)	Destructor (virtual)
virtual CState *Exec(CState *s)	What to do when executed
virtual CState* updateDEP (char*, CState*, CIEnvStackDataType*, int)	updates variable dependency list

CLASS: CMulStmtCon
SYNOPSIS: This is a subclass of CStmt. It is used for multiple statements connected by conjunction.

Data Member	**Description**
CStmt *_first	Former statement
CStmt *_second	Later Statement

Methods	**Description**
CMulStmtCon(CStmt *, CStmt *)	Constructor
~CMulStmtCon(void)	Destructor
CState *Exec(CState *)	What to do when executed

CLASS: CMulStmtDis
SYNOPSIS: This is a subclass of CStmt. It is used for multiple statements connected by disjunction.

Data Member	**Description**
CStmt *_first	Former statement
CStmt *_second	Later Statement

Methods	**Description**
CMulStmtDis(CStmt *, CStmt *)	Constructor
~ CMulStmtDis(void)	Destructor
CState *Exec(CState *)	What to do when executed

CLASS: CMulStmtSeq
SYNOPSIS: This is a subclass of CStmt. It is used for multiple statements connected by sequential execution semantics.

Data Member	**Description**
CStmt *_first	Former statement
CStmt *_second	Later Statement

Methods	**Description**
CMulStmtSeq(CStmt *, CStmt *)	Constructor
~ CMulStmtSeq(void)	Destructor
CState *Exec(CState *)	What to do when executed

CLASS: CTopDownStmt
SYNOPSIS: This is a subclass of CStmt.

Data Member	**Description**
CStmt *_stmt	

Methods	**Description**
CTopDownStmt(CStmt *)	Constructor
~CTopDownStmt(void)	Destructor
CState *Exec(CState *)	What to do when executed

CLASS: CBottomUpStmt
SYNOPSIS: This is a subclass of CStmt.

Data Member	Description
CStmt *_stmt	

Methods	Description
CBottomUpStmt(CStmt *)	Constructor
~ CBottomUpStmt(void)	Destructor
CState *Exec(CState *)	What to do when executed

CLASS: CTrueStmt
SYNOPSIS: This is a subclass of CStmt.

Data Member	Description
CStmt *_stmt	

Methods	Description
CTrueStmt(CStmt *)	Constructor
~CTrueStmt(void)	Destructor
CState *Exec(CState *)	What to do when executed

CLASS: C FalseStmt
SYNOPSIS: This is a subclass of CStmt.

Data Member	Description
CStmt *_stmt	

Methods	Description
C FalseStmt(CStmt *)	Constructor
~C FalseStmt(void)	Destructor
CState *Exec(CState *)	What to do when executed

CLASS: CRepeatStmt
SYNOPSIS: This is a subclass of CStmt.

Data Member	Description
CStmt *_stmt	

Methods	Description
CRepeatStmt(CStmt *)	Constructor
~CRepeatStmt(void)	Destructor
CState *Exec(CState *)	What to do when executed

CLASS: CPrintStmt
SYNOPSIS: This is a subclass of CStmt. This class is used for debugging purpose only.

Data Member	Description
int _which_print	What type of print command is it?
char * _iname	Holds variable name or text string.

Methods	Description
CPrintStmt(int, char *)	Constructor
~CPrintStmt (void)	Destructor

CState *Exec(CState *)	What to do when executed

CLASS: CGivenStmt
SYNOPSIS: This is a subclass of CStmt. GivenStmt includes of one or more equations

Data Member	**Description**
CEqn* _eqnList[MAXARG]	equation list
CStmt *_stmt	Given statement body
int _count	Number of equations in the equation list.

Methods	**Description**
CGivenStmt(CStmt* ...)	Constructor
~CGivenStmt(void)	Destructor
CState *Exec(CState *)	What to do when executed

CLASS: CRuleInvokeStmt
SYNOPSIS: This is a subclass of CStmt. It is used for rule invoking statements.

Data Member	**Description**
CRule *_first_r	A pointer to a rule.
CRuleBlock *_first_rb	A pointer to a transfromation. Since we can invoke a rule or transformation, we need two different pointers (one for a rule and another for transformation.) At any given time, just one of those two fields is used.
char * _second	The variable name we are going to invoke a rule or a transformation with.

Methods	**Description**
CRuleInvokeStmt(CRuleBlock*, char *)	Constructor for transformation invoke
CRuleInvokeStmt(CRule*, char *)	Constructor for rule invoke
~CRuleInvokeStmt(void)	Destructor
CState *Exec(CState *)	What to do when executed

CLASS: CRuleInvInvokeStmt
SYNOPSIS: This is a subclass of CStmt. It is used for inverse rule invoking statements.

Data Member	**Description**
CRule *_first	A pointer to a rule.
char * _second	The variable name we are going to invoke a rule with.

Methods	**Description**
CRuleInvInvokeStmt(CRule*, char *)	Constructor
~CRuleInvInvokeStmt(void)	Destructor
CState *Exec(CState *)	What to do when executed.

CLASS: DEPNode
SYNOPSIS: Class for Dependency List.

Data Member	**Description**
char * _value	Holds variable names.
DEPNode * _next	

Methods	**Description**
DEPNode(char * a)	Constructor
~DEPNode(void)	Destructor

CLASS: DEPList
SYNOPSIS: Class for Dependency List.

Data Member	**Description**
DEPNode * _iterator	An iterator used for traversing the list.
DEPNode * _head	A head of the list

Methods	**Description**
DEPList(void)	Constructor
~DEPList(void)	Destructor
DEPNode * find(const char* a)	Find a node with _value = a
void insert(char* a)	Insert a new node with _value = a to the list
char* pop(void)	Pop off a front node from the list. Return a copy of _value.
int empty(void)	Is the list empty?
void initIterator(void)	Initialize _iterator. This method should be used with getValue method
char* getValue(void)	Returns _value of the node that _iterator is pointing to. After a call to this method, _iterator will point to the next node. One can call this method repeatly to get the _value of all the nodes. However, initIterator has to be called before the initial call to this method.

CLASS: EqnNode
SYNOPSIS: Class for Equation List.

Data Member	**Description**
CEqn * _value	Holds an equation.
EqnNode * _next	

Methods	**Description**
EqnNode (CEqn * a)	Constructor
~ EqnNode (void)	Destructor

CLASS: EqnList
SYNOPSIS: Class for Equation List.

Data Member	**Description**
EqnNode * _iterator	An iterator used for traversing the list.
EqnNode * _head	A head of the list
EqnNode * _tail	A tail of the list

Methods	**Description**
EqnList (void)	Constructor
~ EqnList (void)	Destructor
DEPNode * find(const char* a)	Find a node with _value = a
void insert(CEqn* a)	Insert a new node with _value = a to the list
CEqn* top(void)	Returns _value of the top node
int pop(void)	Pop off a front node from the list. Return success value of the pop.

int empty(void)	Is the list empty?
void initIterator(void)	Initialize _iterator. This method should be used with getValue method
CEqn* getValue(void)	Returns _value of the node that _iterator is pointing to. After a call to this method, _iterator will point to the next node. One can call this method repeaty to get the _value of all the nodes. However, initIterator has to be called before the initial call to this method.
CEqn* iteratorValue(void)	Returns _value of the node that _iterator is pointing to. This one is different from getValue in that this one does not advance _iterator after its call.

CLASS: CEqn
SYNOPSIS: Class for equations

Data Member	**Description**
char * _lhs	Stores left hand side of an equation (variable name)
CPattern * _rhs	Stores right hand side of an equation (pattern)
int _evalFlag	Indicates whether an equation has been evaluated or not. If an equation is not evaluated then _evalFlag is 0. Otherwise, _evalFlag is 1. We need _evalFlag to undo any changes that caused by evaluating equations. We need to know which equation is evaluated and which is not.

Methods	**Description**
void SetEF(void)	Set _evalFlag (_evalFlag = 1)
void UnsetEF(void)	Unset _evalFlag (_evalFlag = 0)
CEqn(char*, CPattern*)	Constructor
~CEqn(void)	Destructor
CState* undoDEP(CState *)	Undo any changes on dependency list
void endOfEval(void)	Evaluation of equation ended. Unset _evalFlag as it was.
CState* Exec(CState *)	What to do when executed

CLASS: CPattern
SYNOPSIS: Class for COKO expressions

Data Member	**Description**
NonType* rrclass	A pointer to an actual COKO expression tree.
CIEnvStackDataType* _savedStackTop	Save the position of stack top. This is used for current pattern only.

Methods	**Description**
CPattern (NonType *)	Constructor for pattern representation
virtual ~CPattern(void)	Destructor
virtual CState* match(CState*)	This method is for rule matching only. Duplicated variable entry is allowed here.
virtual CState* matchSTM(CState*)	This match method is used for statement matching. No duplicated variable entry is allowed.
CState* DepListCheck(char*, CState*)	Dependency List Check.
CState* undoDEP(char*, CState*)	Undo any changes made to dependency list.
virtual void SetStackPointer(CIEnvStackDataType* stackPt)	Sets a stack pointer. This one is used to remember current stack top.

virtual CState* build(CState*)	This method is for building a transformed KOLA query.
virtual CState* buildCurrent(CState*)	This calls buildFromHere methods.
virtual CState* buildFromHere(CState*, CIEnvStackDataType*)	Instead of searching a environment stack from the top frame to build transformed query, this method searches the environment stack from the given point.
NonType* getNT()	Returns a pointer to rrclass. This method is only used for debugging purpose.

CLASS: CIdent
SYNOPSIS: This class actually wraps triple, (_Parent, _Child, _Which). _Parent points to the parent of _Child, and _Which specifies which child of the _parent. Actually _Parent field is not used for now. It is always NULL. Only _Child and _Which is used for COKO compiler. However, _Parent field is kept for possible future usage.

Data Member	**Description**
NonType *_Parent	Pointer to the Parent.
NonType *_Child	Pointer to the top node of a KOLA subtree which the variable matches.
int _Which	Which child of the parent?

Methods	**Description**
CIdent(void)	Parameterless Constructor
CIdent(NonType *, int)	Constructor with child and which
CIdent(NonType *, NonType *, int)	Constructor with parent, child and which
~CIdent(void)	Destructor
NonType *Child(void)	Returns a copy of _Child.
NonType *Child_Look_Up(void)	Returns a pointer to _Child.
int Which(void)	Returns _Which

CLASS: CIEnvValueType
SYNOPSIS: _key corresponds to the name of the identifier, and _content stores CIdent type entry matching the _key.

Data Member	**Description**
char *_key	Identifier name string
CIdent * _content	The content
DEPList * _DEPList	Dependency list
DEPList * _InvDEPList	Inverse dependency list
CPattern * _pattern	Pattern associated with the variable. We need this for reconstructing job after invoking any rule to the variable.

Methods	**Description**
int insertDepList(char * a)	Insert an item into the dependency list
int cleanDepList(void)	Delete all item from the dependency list
int emptyDepList(void)	Is dependency list empty?
int findDepList(const char*)	Is item in the dependency list?
char* popInvDepList(void)	Pop an item from inverse dependency list and return it
int insertInvDepList(char * a)	Insert entry into the inverse dependency list
int cleanInvDepList(void)	Delete all item from the inverse dependency list
int emptyInvDepList(void)	Is inverse dependency list empty?

int findInvDepList(const char*)	Is item in the inverse dependency list?
const char * GetKey(void)	Returns _key
CIdent * GetContent(void)	Returns _content
DEPList * GetList(void)	Returns _DEPList
DEPList * GetInvList(void)	Returns _InvDEPList
CPattern * GetPattern(void)	Returns _pattern
void SetKey(char *)	_key modifier
void SetContent(CIdent *)	_content modifier
void SetPattern(CPattern*)	_pattern modifier
CIEnvValueType(const char *, CIdent *)	Constructor with _key and _content
~CIEnvValueType(void)	Destructor

CLASS: CIEnvNodeType
SYNOPSIS: CIEnvValueType's will be stored as linked list. CIEnvNodeType defines structure of each node in the stack.

Data Member	**Description**
CIEnvValueType * _data	The real data item
CIEnvNodeType *_pre	
CIEnvNodeType *_next	

Methods	**Description**
CIEnvNodeType(void)	Parameterless constructor
CIEnvNodeType(CIEnvValueType * data)	Constructor for a node with data
~CIEnvNodeType(void)	Destructor
CIEnvValueType * GetData(void)	Returns _data
void SetData(CIdent*)	_data modifier
int insertDepList(char* a)	Insert an item into the dependency list of _data.
int emptyDepList(void)	Is dependency list of _data empty?
char* popInvDepList(void)	Pop an item from inverse dependency list of _data and return it.
int findDepList(const char*)	Is item in the dependency list of _data?
int insertInvDepList(char* a)	Insert entry into the inverse dependency list of _data.
int emptyInvDepList(void)	Is inverse dependency list of _data empty?
int findInvDepList(const char*)	Is item in the inverse dependency list of _data?
CPattern* GetPattern(void)	Get _pattern of _data.
void SetPattern(CPattern*)	Modify _pattern of _data.
void initIteratorDEP(void)	Call initIterator method of _DEPList of _data.
char* getValueDEP(void)	Call getValueDEP method of _DEPList of _data.

CLASS: CIEnv
SYNOPSIS: The wrapping class for the identifier search stack.

Data Member	**Description**
CIEnvNodeType *_head	
CIEnvNodeType *_tail	

Methods	**Description**
CIEnvNodeType *_findNode(const char *key)	Internal search function for modify function
CIdent * find(const char *key)	Search function for a given key
CIEnv(void)	Parameterless constructor
~CIEnv(void)	Destructor
CIEnvValueType * first(void)	Returns the top node value

void insert(CIEnvValueType * data)	Inserts a given node value into the stack
void erase(const char * key)	Find a node with a given key then delete it from the stack.
void modify(char*, CIdent*)	Modifies a given node value in the stack.

CLASS: CIEnvStackDataType
SYNOPSIS: A class for definition of each element in the stack. (Stack is implemented as a linked list.)

Data Member	Description
CIEnv *_item	_item is also another stack structure
CIEnvStackDataType *_next	Next stack element

Methods	Description
CIEnvStackDataType(CIEnv *)	Constructor
~CIEnvStackDataType(void)	Destructor

CLASS: CIEnvStack
SYNOPSIS: Stack class

Data Member	Description
CIEnvStackDataType *_data	Pointer to the top element in the stack

Methods	Description
CIEnvStack(void)	Constructor
~CIEnvStack(void)	Destructor
void push(CIEnv *)	The "Push" operation on the stack.
void pop(void)	The "Pop" operation on the stack.
CIEnv * top(void)	Returns the top element.
CIEnvStackDataType * getData()	Returns _data.
CIEnvNodeType* findNodeDeep(const char*)	Search for the entire stack starting from the top.
CIEnvNodeType* findNodeFromHere(const char*, CIEnvStackDataType*)	Search the stack from the given starting point.
void deleteNodeDeep(const char*)	Find a node and delete it. Search for entire stack starting from the top.

CLASS: CState
SYNOPSIS: CState is a class for abstraction of the state of program. The result of the statement executions is recorded by passing and receiving CState.

Data Member	Description
NonType *_Store	A pointer to a current KOLA tree (local root)
CIEnvStack * _IEnv	Identifier search stack
DEPList * _InvDEPList	Inverse dependency list for current. Current only needs inverse dependency list. Dependency list for current is meaningless. (current has no ancestor)
CPattern *_CurrPattern	Pattern in the last matching statement
int _Success	Return flag

Methods	Description
CState(NonType *, CIEnvStack *, int)	Constructor
~CState(void)	Destructor
NonType *Store(void)	Returns _Store
CIEnvStack *IEnv(void)	Returns _IEnv

int ISuccess(void)	Returns _Success
void SetStore(NonType *)	Modifier for _Store
void SetSuccess(int)	Modifier for _Success
CIdent * IFind(const char *)	Search only for the top frame (another stack type) of the stack
int IInsert(const char*, CIdent*)	Identifier inserter (inserts into the top frame of the stack.
void INew(void)	Pushes an empty frame (another stack type) into the stack
void IOld(void)	Pop up the top element of the stack
void IModify(char *, NonType *, int)	Modifier for a particular identifier
void IDeleteDeep(char *)	Delete identifier from the stack. Search for the entire stack.
CIdent * IFindDeep(char *)	Search for the entire stack.
CIdent * IFindFromHere(char *, CIEnvStackDataType*)	Search for entire stack starting from the indicated point to down.
CIEnvValueType * IFirst(void);	Returns the root node value of identifier search tree.
void IModifyFromHere(char*, NonType*, int, CIEnvStackDataType*)	Same as IModify but this one searches the entire stack starting from the indicated point to down.
void SetCurrentPattern(CPattern *)	Modifier for _CurrPattern
CPattern * CurrentPattern(void)	Returns _CurrPattern
int IInsertDep(const char* key, char* a)	Find a node with key then insert a to its dependency list
int IInsertInvDep(const char*, char*)	Find a node with key then insert a to its inverse dependency list
void IDeleteInvDep(const char*)	Find a node with key then retrieve the entire key values in its inverse dependency list. With all the retrieved key values, search and delete all the entries from the stack.
int IEmptyInvDep(const char*)	First search the stack with a given key. Is an inverse dependancy list stored in the found node empty?
void SetPatternDeep(const char * key, CPattern * p)	First search the stack with a given key. Modify a pattern stored in the found node.
CPattern* GetPatternFromHere(const char*, CIEnvStackDataType*)	First search the stack with a given key. Start searching from the startingPt. Searching from the starting point instead of the top of the stack makes code more efficient. Return a pattern stored in the found node.
void CInsertInvDep(char*)	Insert a char* entry into the _InvDEPList
void CDeleteInvDep(void)	Retrieve all the key values from _InvDEPList. With all the retrieved key values, search and delete all the entries from the stack.
int CEmptyInvDep(void)	Is _InvDEPList empty?
void CCleanInvDep(void)	Clean _InvDEPList. Make _InvDEPList as it was first created. To do so, just delete old _InvDEPList and assign new DEPList to _InvDEPList.
DEPList* GetInvDep(void)	Returns _InvDEPList
void SetInvDep(DEPList* a)	Modify _InvDEPList
void DelInvDep(void)	Delete _InvDEPList

4. COKO with Semantics

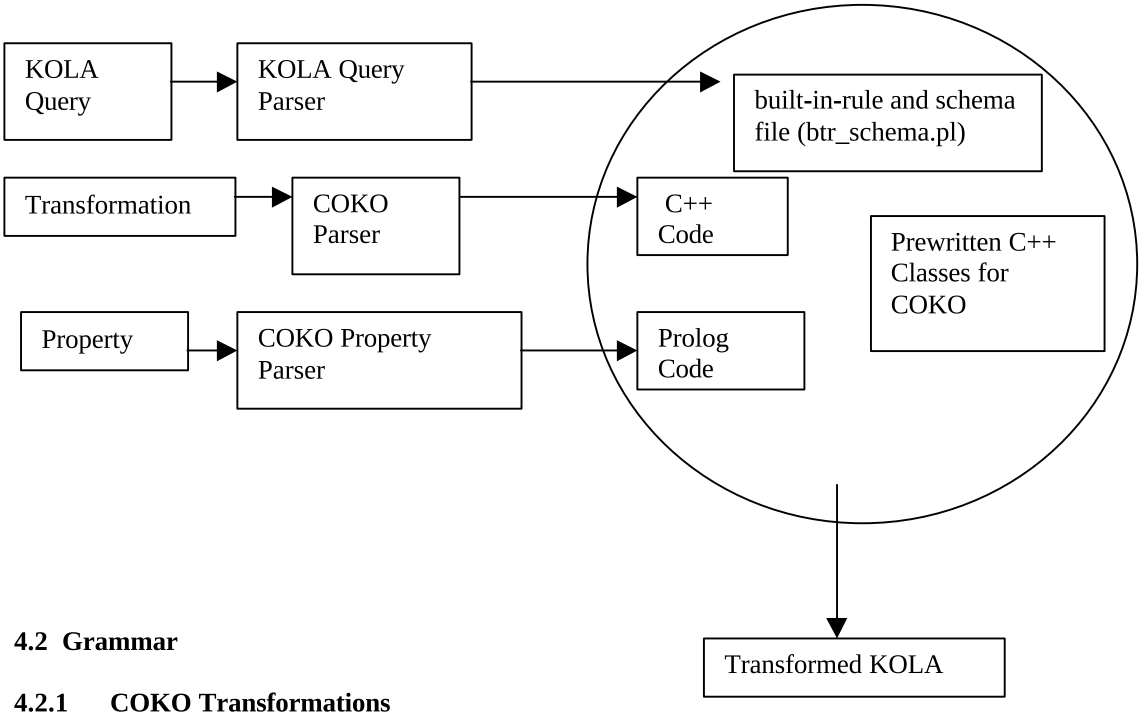
This chapter describes semantic-based extensions to COKO. Our goal is to permit algebraic optimization transformation to be conditioned on the semantics of the data and queries to which they might be applied. While this is possible by adding supplemental rule conditions (properties) expressed in code as in Starburst or Exodus, the intention of this work is to avoid compromising the theorem prover verifiability of rules that KOLA supports. This is accomplished with extensions to COKO that:

- permit firing of conditioned rewrite rules that are predicated on declarative expressed properties of queries and data, and
- permit definitions of these properties by way of declarative inference rules.

Thus, instead of using code, COKO properties are expressed with Prolog-like terms that identify relationships between identified KOLA subtrees. More detailed explanation about the semantic optimization in COKO can be found in chapter 5 of [Che97].

4.1 Design Overview

The following diagram shows how the extended COKO transformation works. Three different parsers are used for the extended COKO compiler. First one is a KOLA parser that parses KOLA query in text representation form into KOLA object tree. Second one is a COKO transformation parser that parses COKO transformations into C++ classes. Last one is a COKO property parser that parses COKO properties into series of Prolog rules and facts. Those Prolog rules and facts as well as generated C++ classes together with prewritten C++ classes and a built-in-rule and schema file are used on KOLA query tree to invoke and execute KOLA tree transformation.



4.2 Grammar

4.2.1 COKO Transformations

4.2.1.1 OptInfer

This is a section in which one can list COKO properties. This section is determined by a keyword, "**INFERS**" followed by one or more COKO property names separated by comma. Every COKO properties listed in this section is first compiled with COKO property parser to generate Prolog source codes and then compiled with Prolog engine to produce **.ql** (extension for Prolog object file) files.

4.2.1.2 Preconditioned Rule

A COKO rule is extended to include preconditioned rules. A preconditioned rule is a form of “precondition list :: E1 → E2,”where E1 and E2 are COKO expressions. For example, “`rd`” in line 3 of Figure 4.2.1.1 is a rule name and “`type(A, {_T}), injective(f)`” is a precondition list.

```
1. Transformation Injective
2. Uses
3.   rd: type(A, {_T}), injective(f) ::
4.       set o iterate(p, f) ! A  → iterate(p,f) ! A
5. Infers
6.   Injective
7. Begin
8.   rd
9. End
```

Figure 4.2.1.1 Transformation Injective

```
1. Transformation Injective2
2. Uses
3.  rd: type(A, {_T}), injective(f) ::
4.      set o iterate(p, g) ! A  → iterate(p,f) ! A
5. Infers
6.   Injective
7. Begin
8.   rd
9. End
```

Figure 4.2.1.2 Transformation Injective2

A precondition list consists of one or more preconditions separated by comma. Each precondition is a form of “IDENT (Z_1, … , Z_n)” where Z_i is a COKO expression . Those preconditions are used with Prolog engine to carry out extra unification needed for constructing right-hand side of the preconditioned rules.

The invocation of a preconditioned rule can be subdivided into three parts. The left-hand side of a rule (E1) is a matching part where variable-to-KOLA tree bindings are built and stored in COKO environment. Those bindings stored in the environment are used to construct the right-hand side of the rule later on. The precondition list is a querying part. Each precondition is used as a Prolog query on a built-in-rule and schema file together with generated Prolog rules and facts from properties in **Infers** section. The result of the unification of preconditions is used for providing extra variable binding which can be used for construction of the right-hand side of the rule. The right-hand side of a rule (E2) is a building part where a transformed KOLA query is built using the bindings stored in COKO environment and the bindings returned from the unification process of precondition list.

For example, processing the left-hand side of the rule rd of Figure 4.2.1.2 will add binding p, binding g and binding A to the environment. However, we need one more variable binding (binding f) to construct right-hand side of the rule. We can get this extra variable binding from the result of processing preconditioned rules.

Not only the preconditions return extra variable bindings but they also have important role deciding the success value of the rule. For example, all the necessary variable bindings for constructing right-hand side of the rule “rd” in Figure 4.2.1.1 are built and stored in left-hand side of the rule. Construction of the right-hand side of “rd” does not require any extra variable bindings from the preconditions. However, unsuccessful unification of the preconditions sets success value of the rule as false and prevents KOLA tree transformation from occurring.

4.2.2 COKO Properties

4.2.2.1 Property

A property consists of a set of inference rules. A COKO property is made up of a word, “**PROPERTY**” followed by a name for the property, optional **OptInfer** declaration section and a main body.

In general, a COKO property has the following form:

```
PROPERTY property-name
       OptInfer declaration
BEGIN
       PStmt.  PStmt. …
END
```

Figure 4.2.2.1: General COKO Property Structure
(Italics indicate optional parts.)

4.2.2.2 OptInfer

This is a section in which one can list other external properties. This section is made up of word, “**INFERS**” followed by one or more COKO property names separated by comma. Every COKO properties listed in this section is first compiled with COKO property parser to generate Prolog source codes and then compiled with Prolog engine to produce **.ql** (extension for Prolog object file) files.

4.2.2.3 Main Body

A main body of a property consists of one or more property statements (inference rules) separated by period.

4.2.2.4 Property Statement (Inference Rule)

COKO property inference rules are either the form “p.” or “E ==> p.” such that p is a property term (e.g., is_injective (f o g)) and E is a logical expression of property terms (e.g., is_injective(f) $\wedge$ is_injective(g)). Compilation of the first form of the rule generates the Prolog fact, “$\bar{p}$.” such that $\bar{p}$ is the Prolog translation of the property term p. Compilation of the second form of the rule generates the Prolog rule, “$\bar{p}$:- $\bar{E}$.” such that $\bar{E}$ is the Prolog translation of the logical expression E.

Property terms, which generally have the form,

$$\text{ident}\ (Z_1,\ldots Z_n)$$

such that each k_i is a KOLA pattern, are translated into Prolog terms,

$$\text{ident}(\bar{Z}_1,\ldots.\ \bar{Z}_n)$$

where the translation, $\bar{Z}_i$ of KOLA pattern Z_i

- prepends KOLA’s unification variables with an upper case “**V**” (this is required as Prolog requires all variables to begin with a capital letter(, and
- translates KOLA’s formers into prefix notation. For example, the function pattern, **f o g** is translated into the string **compose (Vf, Vg)** while invocation **(f ! A)** is translated into the string **invoke (Vf, VA)**.

Translation of logical expressions into Prolog expressions translates property terms as described above, and maps:

- conjunctive expressions “$p_1 \wedge p_2$” to “$\bar{p}_1, \bar{p}_2$”
- disjunctive expressions “$p_1 \vee p_2$” to “$\bar{p}_1; \bar{p}_2$”
- negation expressions “not (p_1)” to “not ($\bar{p}_1$)”
- equations “$p_1 = p_2$” to “$\bar{p}_1 = \bar{p}_2$”

```
PROPERTY Injective
BEGIN
  injective(ID).
  key(f) ==> injective (f).
  injective(f) /\ injective(g) ==> injective(f o g).
  injective(f) \/ injective(g) ==> injective(<f, g>).
  injective(f) ==> injective(iterate (_P, f)).
END
```

Figure 4.2.2.2: Property Injective

For example, the result of compiling property Injective is the set of Prolog rules and facts shown below:

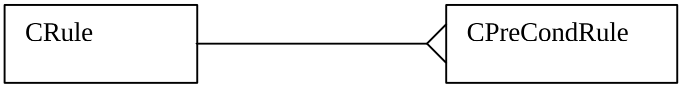
```
injective(id).
injective(Vf) :- key(Vf).
injective(compose(Vf, Vg)) :- injective(Vf), injective(Vg).
injective(plus(Vf, Vg)) :- injective(Vf); injective(Vg).
injective(iterate(_, Vf)) :- injective(Vf).
```

4.3 Architecture

4.3.1 Built-in-rule and Schema file

Every transformation that uses preconditioned rule requires a Prolog file, which can act as a database for Prolog query. This file is called “built-in-rule and schema” file.

4.3.2 CPreCondRule



This is a subclass of CRule class. This class includes a precondition list (**CPreCond**) in addition to two CPattern classes, which represent left and right-hand side of the rule. Its **Exec** method first calls **match** method of the left-hand side of the rule and then calls **Exec** method of **CPreCond** member. If the matching process of the left-hand side of the rule and Prolog unification process of the precondition list succeed then it calls **build** method on right-hand side of the rule.

4.3.3 CPreCond

This is a class in which Prolog querying process occurs. First we convert all the preconditions stored in precondition list to an appropriate Prolog query commend. Then we issue the query on a built-in-rule and schema file and Prolog rules and facts generated by property parser. The unification result from issuing the query will be parsed and selectively stored into the environment. For example, processing preconditions in Figure 4.2.1.1 will return binding A and binding f. However, those binding are ignored and not stored into the environment since there are already binding A and binding f stored in the environment as a result of matching process of the left-hand side of the rule. Similarly, processing preconditions in Figure 4.2.1.2 will return binding A and binding f. In this case, binding f will be stored into the environment since it is a new binding and will be used to construct right-hand side of the rule.

4.4 Interface

CLASS: CPreCondRule
SYNOPSIS: This is a subclass of CRule class. This class represents preconditioned rules.

Data Member	Description
CPreCond* _precond	precondition list class

Methods	**Description**
CPreCondRule (CPreCond*, CPattern*, CPattern*)	Constructor
~CPreCondRule(void)	Destructor
CState *Exec(CState*)	What to do when executed

CLASS: CPreCond
SYNOPSIS: This class is for precondition list. Actual Prolog querying process and result parsing process occurs in this class.

Data Member	**Description**
char* _RuleBlockName	Transformation name in which this precondition list is defined.
int _id	An integer used for differentiating each Prolog query. For example, a transformation can have more than one preconditioned rule defined in its USES section. In this case, _id is used as index number for different queries created by different preconditioned rules. The query from the first preconditioned rule will be named query0. The query from the second preconditioned rule will be named query1, and so on.
CondList* _condList	precondition class list

Methods	**Description**
CPreCond(char* a ...)	Constructor
~CPreCond(void)	Destructor
CState *Exec(CState *, NonType*)	What to do when executed

CLASS: CCond
SYNOPSIS: This class is for a precondition.

Data Member	**Description**
char* _condName	Stores precondition name
NonTypeList* _varList	Variable list
int _varcount	# of variables in this precondition
int _built_varcount	# of processed variables

Methods	**Description**
CCond(char * a ...)	Constructor
~CCond (void)	Destructor
int allDone(void)	Are all variables built yet?
CState* build(CState*,NonType*)	build environment
CState *Exec(CState *)	What to do when executed

CLASS: CondNode
SYNOPSIS: Class for Condition List

Data Member	**Description**
CCond * _value	value
CondNode * _next	

Methods	**Description**
CondNode(CCond * c)	Constructor
~CondNode(void)	Destructor

CLASS: CondList
SYNOPSIS: Class for Condition List

Data Member	**Description**
CondNode * _iterator	An iterator used for traversing the list.
CondNode * _head	head of the list
CondNode * _tail	tail of the list

Methods	**Description**
CondList(void)	Constructor
~CondList(void)	Destructor
void insert(CCond* a)	Insert a new node with _value = a to the list
int pop(void)	Returns _value of the top node.
CCond* top(void)	Returns _value of the top node.
int empty(void)	Is the list empty?
void initIterator(void)	Initialize _iterator. This method is used with getValue method.
CCond* getValue(void)	Returns _value of the node that _iterator is pointing to. After a call to this method, _iterator will point to the next node. One can call this method repeatly to get the _value of all the nodes. However, initIterator has to be called before the initial call to this method.

CLASS: NonTypeNode
SYNOPSIS: Class for NonType List

Data Member	**Description**
NonType * _value	value
NonTypeNode * _next	

Methods	**Description**
NonTypeNode(NonType * c)	Constructor
~NonTypeNode(void)	Destructor

CLASS: NonTypeList
SYNOPSIS: Class for NonType List

Data Member	**Description**
NonTypeNode * _iterator	An iterator used for traversing the list.
NonTypeNode * _head	head of the list
NonTypeNode * _tail	tail of the list

Methods	**Description**
NonTypeList(void)	Constructor
~NonTypeList(void)	Destructor
void insert(NonType* a)	Insert a new node with _value = a to the list
int pop(void)	Returns _value of the top node.
NonType* top(void)	Returns _value of the top node.
int empty(void)	Is the list empty?
void initIterator(void)	Initialize _iterator. This method is used with getValue method.
NonType* iteratorValue(void)	Returns _value of the node that _iterator is pointing to.

NonType* getValue(void)	Returns _value of the node that _iterator is pointing to. After a call to this method, _iterator will point to the next node. One can call this method repeatly to get the _value of all the nodes. However, initIterator has to be called before the initial call to this method.

5. Tdraw

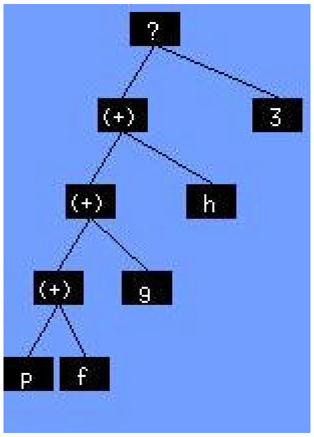
Tdraw is a graphical user interface for KOLA query transformation. Even thought there is no written documentation about this program, any programmer with fair knowledge about Motif and UNIX programming can easily understand tdraw program. (It is written by a Ph.D. student in Computer Science Department at Brown University.)

Assume that the compilation of the following transformation results in transformation executable named testRepeat. One can run this executable with a query file to see how the query is transformed as a result of invocation of *transformation repeat-test* on input query. However, since COKO is a command line based program, it is hard for a user to visualize input query tree as well as transformed result query tree by looking at the text representations of the KOLA trees.

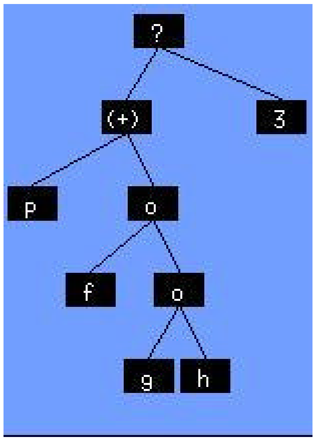
```
Transformation repeat-test
Uses
  foo: (p oplus f) oplus g → p oplus (f o g)
Begin
  GIVEN p ? _O DO * foo (p)
End
```

Figure 5.1.1.1 Transformation repeat-test

Tdraw graphically displays KOLA query trees and helps user to visualize actual KOLA tree transformation. It first displays input query tree and then displays transformed query tree as the COKO transformation occurs. To run tdraw program, type tdraw followed by a transformation executable name and a query file name at the commend line. For example, by running tdraw program with testRepeat on a query input “`((pred(p, NULL) oplus Fun(f, NULL, NULL)) oplus Fun(g, NULL, NULL)) oplus Fun(h, NULL, NULL) ? 3`”, one can see the input query tree graph and result query tree graph. The followings are screen-captured pictures for the repeat-test example.



Input KOLA query tree



Transformed KOLA query tree

6. Directory

Since there are a number of files involving the COKO compiler project and they are spread over a number of directories, it is advised to anyone who deals with this project to become familiar with the directory hierarchy and all the related files before starting to work on the project.

When one first starts to work with KOLA/COKO related project, there are one main directory one might want to take a very careful look at. /pro/oodb/cokokola/ directory and its all sub-directories have all the project related files. These directories have a stable version of the project and should be remained stable all the time. One should not make any changes in those directories unless every source code modification made to the files in these directory is bug free and does not conflict with existing projects.

All the KOLA/COKO files have RCS directories and RCS files related to them. Using RCS is a way of preventing possible conflict among a group of people working on the same file. First thing needs to be done, when one begins to modify KOLA/COKO files, is making one's own working directory that have same directory hierarchy as /pro/oodb/cokokola/src. Then one have to make RCS links in the working directory referring to the original RCS directories.

After modifying working files, one should make sure that the codes are working and bug free. Any files checked back in original RCS directories must be a final bug free version.

Followings are short description of the directories.

- /pro/oodb/cokokola – COKO/KOLA project main directory
- /pro/oodb/cokokola/bin – COKO/KOLA related Programs (executable files)
- /pro/oodb/cokokola/src – Source codes
- /pro/oodb/cokokola/released – Released version of programs (compressed and tar'd files)
- /pro/oodb/cokokola/doc – documentation
- /pro/oodb/cokokola/data – transformation examples

7. References

[Che97] Mitch Cherniack. Building Query Optimizers With Combinators. Dissertation Proposal, Brown University Department of Computer science, December 1997.

[CZ96] Mitch Cherniack and Stan Zdonik. Rule Languages and Internal Algebras for Rule-Based Optimizers. Proceedings of the ACM SIGMOD International Conference on Management of Data, Montreal, Quebec, June 1996.

[CZ98a] Mitch Cherniack and Stan Zdonik. Inferring Function Semantics to Optimize Queries. Submitted to VLDB '98.

[CZ98b] Mitch Cherniack and Stan Zdonik. Changing the Rules: Transformations for Rule-Based Optimizers. Proceedings of the ACM SIGMOD International Conference on Management of Data, Seattle, WA, June 1998. (To Appear)