BROWN UNIVERSITY
Department of Computer Science
Master's Project

CS-95-M8

"Implementation of the EAT-Based Control Strategy for the EPOQ Optimizer"

by

Weining Lu

Implementation of the EAT-Based Control Strategy for the EPOQ Optimizer

By

Weining Lu
Department of Computer Science
Brown University

May 1995

This research project by Weining Lu is accepted in its present form by the Department of Computer Science as satisfying the partial requirement for the degree of Master of Science

Date 5/19/95 Stanley B Zdonik

Professor Stanley B. Zdonik
Advisor

Date 5/19/95 Marian H. Nodine

Dr. Marian H. Nodine
Advisor

Contents

1 Introduction **1**

2 Overview of EPOQ Optimizer **2**

2.1 Region Architecture 2

2.2 Interface of a Region 2

2.3 Attain of a Region 3

2.4 Interactions Between Regions 3

2.5 Architecture of EPOQ Optimizer 4

2.6 The AQUA Query Algebra and its EAT Representation 5

3 The EAT-Based Regions **7**

3.1 Implementation of the EAT-Based Control Region 7

3.2 Class of the EAT-Based Control Region 18

3.2.1 REControlRegionEATBased::public REControlRegion 18

3.2.2 SR_Optimizer::public REControlRegionEATBased 19

3.3 Testing Strategy 20

3.3.1 Test Regions 20

3.3.2 Test Region Classes 23

4 The Query Transformation **26**

4.1 Implementation of Query Transformation 26

4.1.1 The First Transformation of the Query Containing the Root Function apply . 27

4.1.2 The Second Transformation of the Query Containing the Root Function apply 27

4.1.3 The Third Transformation of the Query Containing the Root Function apply 28

4.1.4 The Fourth Transformation of the Query Containing the Root Function apply 28

4.1.5 The Fifth Transformation of the Query Containing the Root Function apply 29

4.2 Class of Leaf Region 32

5 Conclusion **34**

List of Figures

1 The Region Architecture 2

2 The EPOQ Architecture 4

3 The EAT Representation of an AQUA query, where A is a set, x is a variable and set is a function. 5

4 The Working.Mechanism of the EAT-Control Region (1) 8

5 The Working Mechanism of the EAT-Control Region (2), where arrow in (a) indicates the location where we get the subquery, and the query in (b) is the one we send to the child region. 9

6 The Working Mechanism of the EAT-Control Region (3), where arrow in (a) indicates the location where we get the subquery, and the query in (b) is the one we send to the child region. 10

7 The Working Mechanism of the EAT-Control Region (4), where arrow in (a) indicates the location where we get the subquery, and the query in (b) is the one we send to the child region. 11

8 The Working Mechanism of the EAT-Control Region (5), where arrow in (a) indicates the location where we get the subquery, and the query in (b) is the one we send to the child region. 12

9 The Working Mechanism of the EAT-Control Region (6), where arrow in (a) indicates the location where we get the subquery, and the query in (b) is the one we send to the child region. 13

10 The Working Mechanism of the EAT-Control Region (7), where arrow in (a) indicates the location where we get the subquery, and the query in (b) is the one we send to the child region. 14

11 The Working Mechanism of the EAT-Control Region (8) 15

12 The Working Mechanism of the EAT-Control Region (9) 16

13 The Working Mechanism of the EAT-Control Region (10) 17

14 Architecture of Test Regions 20

15 Query Transformation in the Test Regions 23

16 Query Transformation 1 27

17 Query Transformation 2 27

18 Query Transformation 3 28

19 Query Transformation 4 29
20 The Example of Query Transformation 4 30
21 The Example of Query Transformation 5 30

Abstract

We present an EAT-based control strategy for the EPOQ optimizer. The EPOQ optimzer is designed not only to have an extensible achitecture, but also to allow the extension of the control strategy. Unlike other extensible optimizers which have a fixed control strategy, the EPOQ optimizer with the EAT-based control strategy provides an efficient means to search the space of query transfomation rules by systematically decomposing the query into subqueries, optimizing those subqueries, and combining the results.

The EAT-based control region implemented in this paper is responsible for carrying out the EAT-based control strategy. It determines the goal based on the given EAT, requests the bids from the child regions on the EAT and sends the EAT to the regions which satisfy the applicabilty on the goal.

The leaf region **apply** has been developed as a child region of the EAT-based control region to perform the transformation of the EATs associated with the function apply. We have used this region, along with an existing select region, to test the EAT-based control strategy.

1 Introduction

Query optimization remains one of the most important challenges to researchers and developers of Object-Oriented database systems[1]. Query optimization is a process of searching, given an input query, for equivalent queries that are efficient to execute. Thus, the goal of the optimizer is to examine as many equivalent queries as possible. However, an optimizer can only visit some portion of the space of equivalent queries because of the availability of transformation rules, the bound of cost for the optimization, and how long the optimization takes. Therefore, the query optimization is a crucial factor that determines performance of query processing.

Many efforts have been made to implement the object-oriented query optimizers. The extensibility of object-oriented database systems requires that an optimizer be extensible in response to new types of expressions. Because new optimization strategies continue to be developed, the optimizer needs to be extensible so it can add these strategies. Most extensible optimizers have focused on the extensibility in terms of adding rules, new data access strategies, and new algebraic operators[2][3][4]. These optimizers are usually based on the rewrite rules for a set of operators defined on the bulk types. These rules are applied to a query expression to generate equivalent, but hopefully more efficient, forms of expressions. The transformation expressions can then be evaluated based on the cost model to select an optimized plan for execution. The extensibility of those optimizer depends upon the ability to define a set of algebraic rewrite rules. The disadvantage of these optimizers are that the optimizers have to search the whole space which is defined by the rewrite rules in order to generate an optimized plan for selecting which rule to apply and when to apply. Accordingly, the efficiency of the optimizers is affected by this strategy. These optimizers also only provide a fixed control strategy to manipulate queries, which makes it difficult for these optimizers to adapt to a new control strategy. At Brown, we are developing a new optimizer, EPOQ [5][6], with an extensible architecture has been designed to improve the efficiency of searching the space for the transformation rules and to allow extension of the control strategy.

In this paper, we present an overview of the architecture of the EPOQ optimizer, develop an EAT-based control strategy for the EPOQ optimizer, perform the query transformation of the apply-associated EAT and finally implement a test region to verify both the EAT-based control strategy and query transformation.

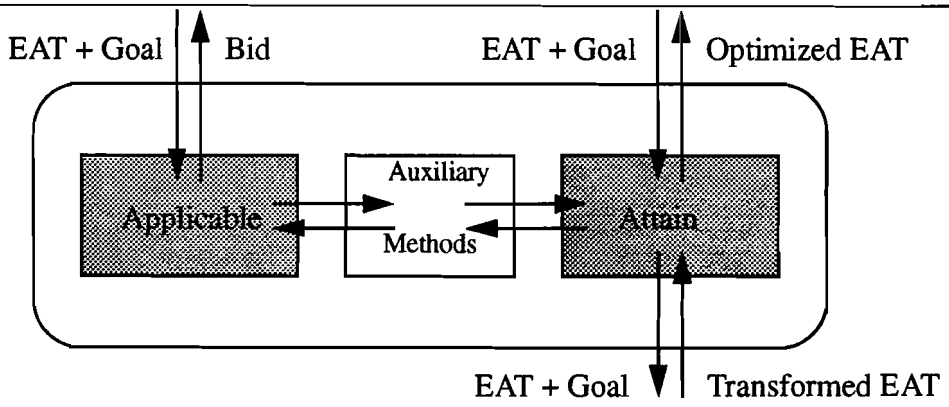


Figure 1: The Region Architecture

2 Overview of EPOQ Optimizer

2.1 Region Architecture

A region is a module, a basic element in the EPOQ optimizer. A region has interfaces for the communication between regions, execution methods for control strategy, applicable and attain, and auxiliary methods, as shown in Figure 1. The auxiliary methods provide the support to the two methods applicable and attain, for example, to get a goal for the region, to store the goal, and to traverse the EAT.

Regions have control over query transformation, and have goals to be achieved. The region control is carried out based on the control strategy of the region. The way to achieve some goal, for example, to transform a query or to lower the cost, is also embedded in the control strategy. A goal may have subgoals to be accomplished by child regions. For each subgoal, the region allocates a goal slot and expects a child region to plug in it. If there is no child region residing at the goal slot, the corresponding goal can not be achieved.

2.2 Interface of a Region

A region provides an interface to support communication between the control of a region and its parent or its child regions. The interface to a region's child allows the region to use the child region to achieve a goal or to perform a specific task. A goal is referred to as a desired result that has to be achieved, for example, query transformation or lower cost.

EPOQ defines a common structure for the interface to ensure compatibility in structure. This approach supports communication between regions as well as the addition of new regions to the

optimizer. The method applicable is implemented to evaluate the applicability of the region on a given goal. Its performance affects the efficiency of the optimizer.

An interface supports the implementation of both a goal and an applicability. Applicability refers to the ability of a region to attain a goal. Thus, applicability is directly related to the control strategy of a region. The applicability is used to find the regions that are able to execute the assigned goal. Thus, the efficiency of the optimizer has been improved by encapsulating the applicability into the control strategy.

However, applicability does not guarantee that a region can achieve a goal, but only indicates a probability that the region can achieve the goal. If a region sends a request to a child region based upon the applicability of the child region to achieve a particular goal on a query, the child region may fail to achieve the goal. Applicability can be represented by a bid, which estimates the probability that a region can achieve a given goal on a query. Implementation of Bids may depend upon given queries and goals.

The messages passed through interfaces between regions are goals and bids along with queries to be optimized.

2.3 Attain of a Region

Attain is an execution unit for a region. Once a region is selected to achieve a goal, the attain method is called to process the query. In a control region, attain may involve some complex algorithm to pass a query back and forth to its child regions to collect bids on the query or to transform the query, to sort the transformed queries in order of cost, and to prune the transformed queries. In leaf region, attain may only perform specific query transformations.

2.4 Interactions Between Regions

The interactions between regions are carried out by means of bidding and branching[7].

Bidding is an action that a region takes to request bids from its child regions. Branching represents the decision to send a query selectively to child regions which satisfy the requirement for the applicability on the query to attain a goal. Bidding is always performed first, and then Branching.

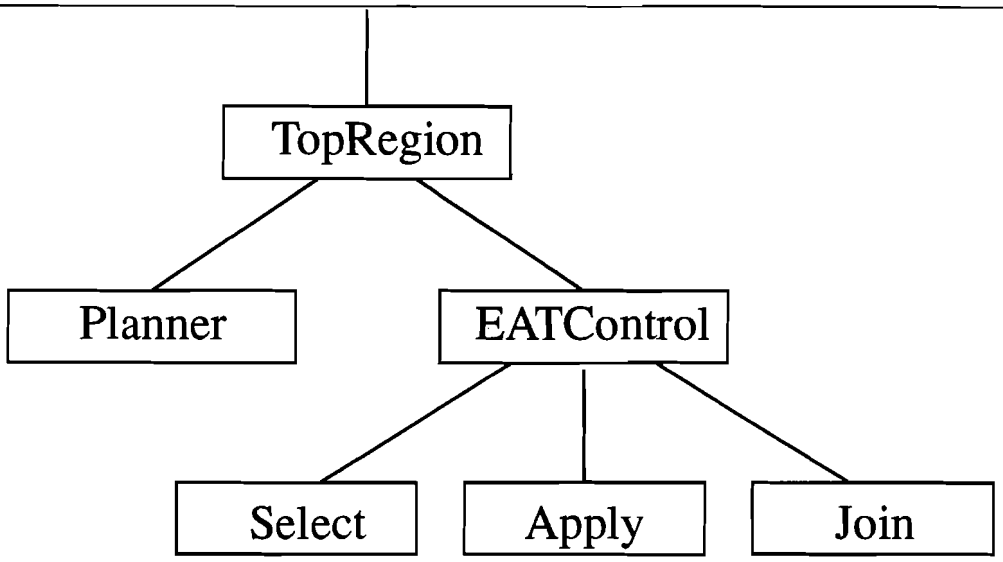


Figure 2: The EPOQ Architecture

2.5 Architecture of EPOQ Optimizer

An EPOQ optimizer is a collection of regions, each of which embodies one strategy for the optimization of the queries, as shown in Figure 2. The EPOQ architecture integrates the regions through a common interface and a global control that combines the actions of subordinate regions to process a given query. The way to achieve some goal resides in the control strategy.

There are two kinds of regions in an EPOQ optimizer: interior regions (including the top region) and leaf regions. The difference between the two kinds of regions is that an interior region can manipulate queries by passing them to other regions, whereas the action of a leaf region is accomplished within the region.

In an EPOQ optimizer, query transformation is performed by a set of regions, each with its own strategy for manipulating query expressions. Different regions will perform different tasks, such as bidding and branching (in control regions) and query transformation (in leaf regions).

The regions are organized hierarchically, with a parent region controlling its child regions. The top region of the optimizer communicates with the query processing system. It receives a query to optimize, and produces a set of optimized queries. These queries are obtained with the assistance of its child regions. The child regions may also act as parents by using their child regions to assist with this transformation.

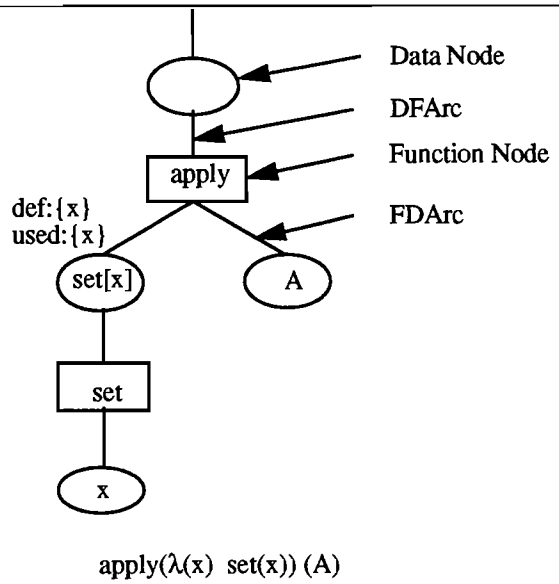


Figure 3: The EAT Representation of an AQUA query, where A is a set, x is a variable and set is a function.

2.6 The AQUA Query Algebra and its EAT Representation

The query language used in the EPOQ optimizer is the AQUA (A QUery Algebra) query algebra. AQUA is designed to be an input language to a broad class of object-oriented query optimizer[]. AQUA is intended to be an intermediate language between the user's query and the query optimizers. as shown in Figure 2. A query in the EPOQ optimizer is represented as an Extensible Annotated Tree (an EAT) which is similar to a parse tree. An EAT consists of alternating layers of function nodes and data nodes, connected by labeled arcs. Arcs in the EAT represent the relationship between functions and data. The EAT representation for the AQUA algebra

$$\textbf{apply}(f)(A) = \{f(a) | a \in A\}, \text{ where A is the input set,}$$

is shown in Figure. In the figure, the data nodes are represented by the ovals, the function node is represented by the rectangle, and the arcs by the lines. The arcs are classified into two kinds, FDArc and DFArc, which are directional. The FDArc stands for a arc which connects the parent function node with the child data node, and the DFARc stands for a arc which which connects the parent data node with the child function node. A FDArc indicates that an argument of the parent function node is provided by its child data node. A DFArc indicates that the parent data node is generated by its child function node.

The data nodes represent either an object (set A) in the database or an object built by a query, subquery, or functions, (e.g., the data node labeled by "set[x]"). The function nodes represent the execution that is performed on the data, such as "apply" and "select", and usually return some values that are stored in the data nodes, for example in this case, the function "select" returns a value "set[x]" and put it in the data node. Thus, A function node always has a parent data node for storing the returned value of the function.

The annotation about information on the scope and use of lambda variables within the query is represented in a way such that the definition def:x represents the variable x is defined for the entire subquery, and used:x represents the variable is actually used in the subquery. This is shown in Figure 3.

For more details about AQUA and EAT, see [8].

3 The EAT-Based Regions

3.1 Implementation of the EAT-Based Control Region

The EAT-based control strategy is implemented in a way such that the control strategy depends upon the given EAT. The EAT is traversed bottom up, optimizing the subqueries rooted at different FDArcs. For each subquery, the goal chosen depends on the name of the top function in the EAT. For instance, an optimizer has a "select" and "join" leaf regions which are responsible for transforming the select-associated and join-associated queries respectively. The goals from the two regions are "select" and "join" respectively. Thus, it is reasonable to choose the top function name as a goal.

The responsibility of the EAT-based control region is to determine the goal based on the given EAT, to request the bids from the child regions on the EAT and to branch the EAT to the regions which return the required bids for further processing. The transformation of an EAT is carried out from bottom to top, i.e. the entire EAT is optimized only after all the subtrees have been optimized. The control strategy for optimization is executed in the following sequence:

1. Traverse the EAT to a leaf and get a copy of the leaf.

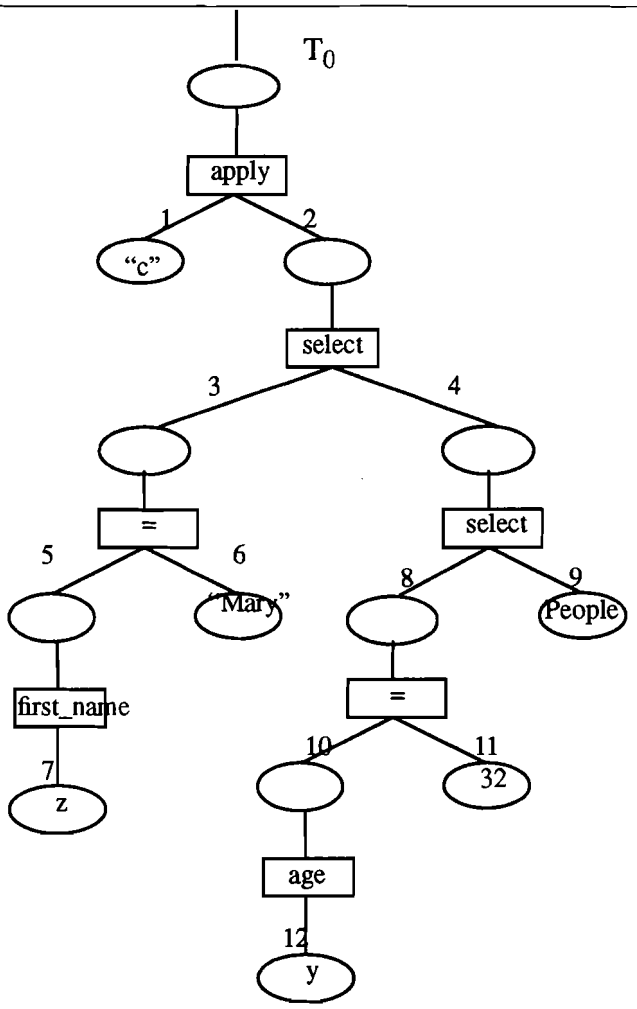
2. Derive a goal from the leaf tree, which is the name of the top function of the leaf.

3. Send the leaf tree for bidding to the leaf regions whose in-goal match the goal of the control region for bidding.

4. Branch the leaf tree to the child regions which satisfy the applicability on a goal. If several region return the same bid, branch the leaf tree to those regions.

5. If the returned leaf EAT from the child region is not NULL, we know that more efficient subqueries were found. Truncate the old leaf and insert the new leaf at same location. If more than one EAT are returned, make a number of copies of the original EAT, delete the old arcs and insert new arcs at same location on the separate copied EATs respectively. All of the modified EATs are stored in a query set.

6. Get a copy of another leaf from the EAT, and repeat the steps from 1 to 5 until all the leaves of the EAT are optimized.



apply (λ (x) "c") (select (λ (z) (z.first_name = "Mary")) (select (λ (y) (y.age =32)) (People)))

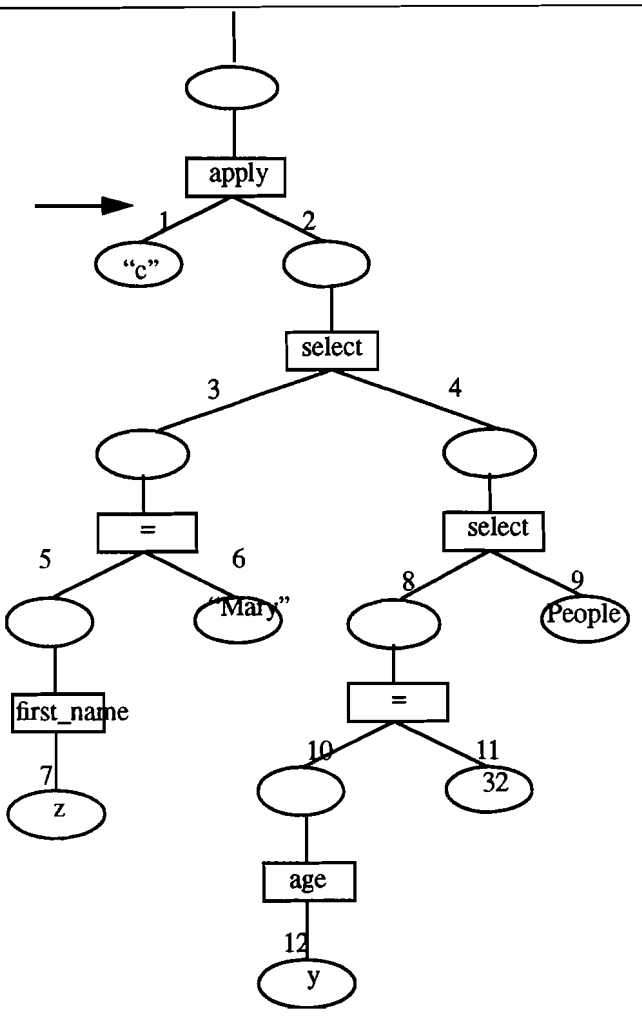
Figure 4: The Working Mechanism of the EAT-Control Region (1)

7. Now all of the transformed subqueries are in the query set. compare the queries in the set with the original query, and return the set that contains the queries with the lower cost than the original one.

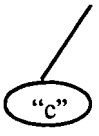
8. The control goes to the higher level of the EAT, and repeat the procedure from 1 to 7.

9. When the entire EAT is optimized, the set which contains these optimized queries is sorted by cost (lowest to highest) and pruned. The pruned set is eventually returned.

The Figures 4 to 14 have illustrated how the control region manipulates the query transformation. It is assumed that the optimizer has the leaf regions, "select" and "apply" where the region



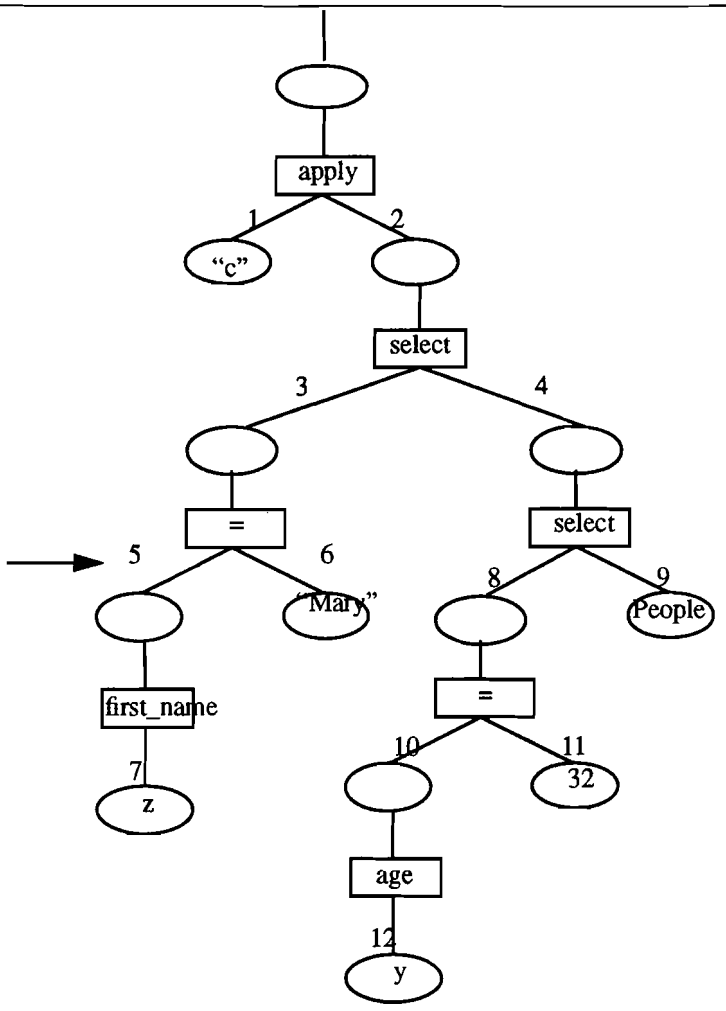
(a)



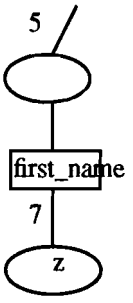
(b)

apply(λ(x) "c") (select (λ(z) (z.first_name = " Mary")) (select (λ(y) (y.age = 32)) (People)))

Figure 5: The Working Mechanism of the EAT-Control Region (2), where arrow in (a) indicates the location where we get the subquery, and the query in (b) is the one we send to the child region.



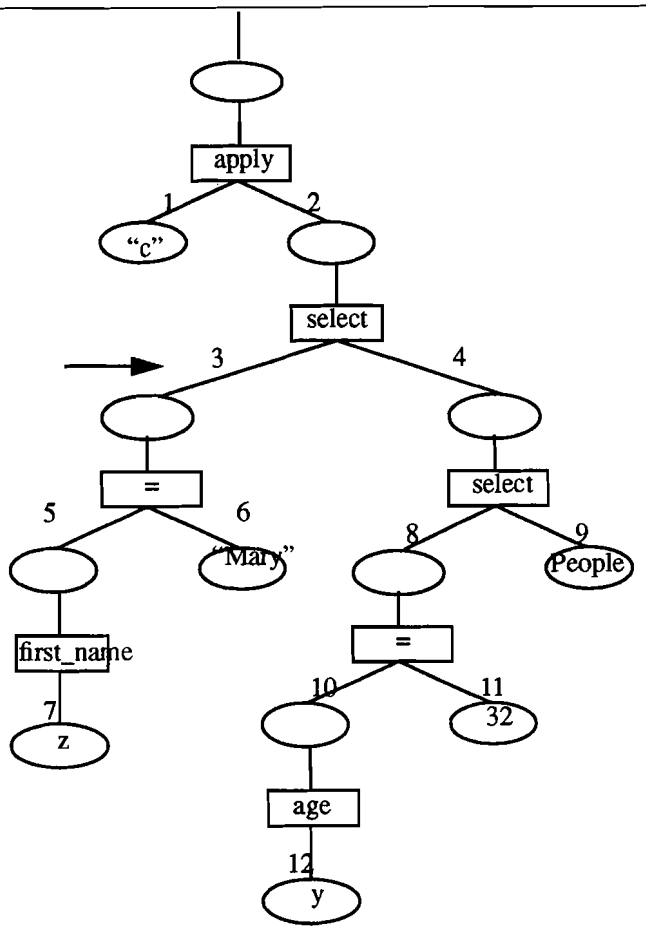
(a)



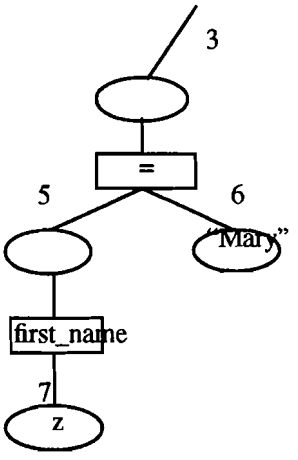
(b)

apply(λ(x) "c") (select (λ(z) (z.first_name = " Mary")) (select (λ(y) (y.age = 32)) (People)))

Figure 6: The Working Mechanism of the EAT-Control Region (3), where arrow in (a) indicates the location where we get the subquery, and the query in (b) is the one we send to the child region.



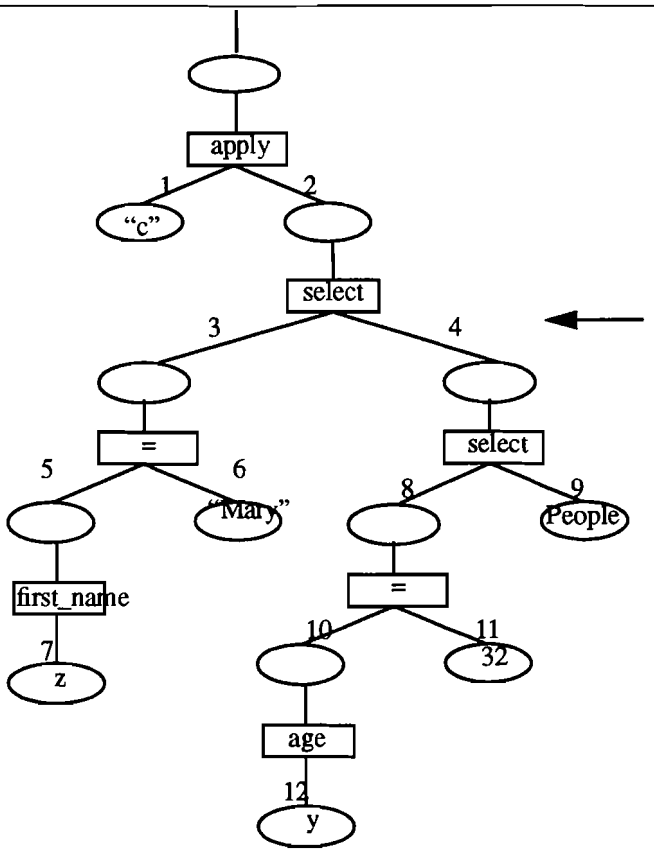
(a)



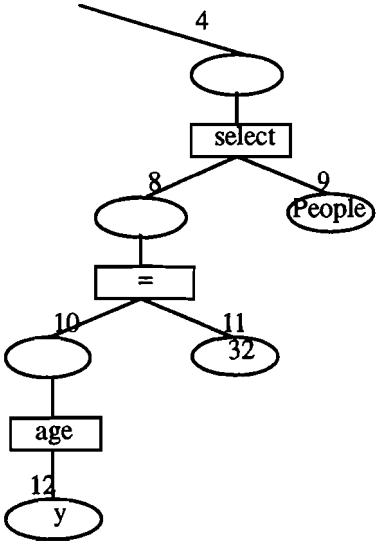
(b)

apply(λ(x) "c") (select (λ(z) (z.first_name = " Mary")) (select (λ(y) (y.age = 32)) (People)))

Figure 7: The Working Mechanism of the EAT-Control Region (4), where arrow in (a) indicates the location where we get the subquery, and the query in (b) is the one we send to the child region.



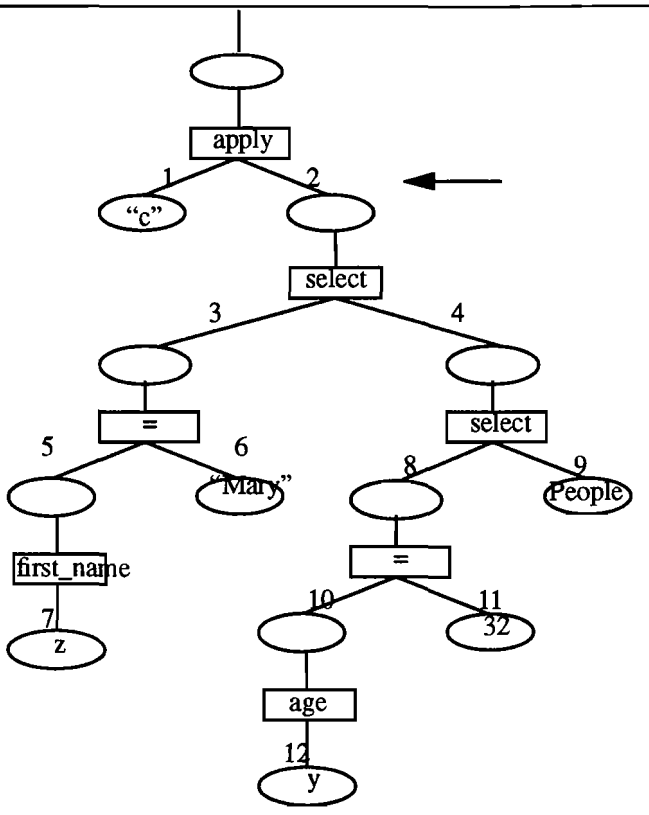
(a)



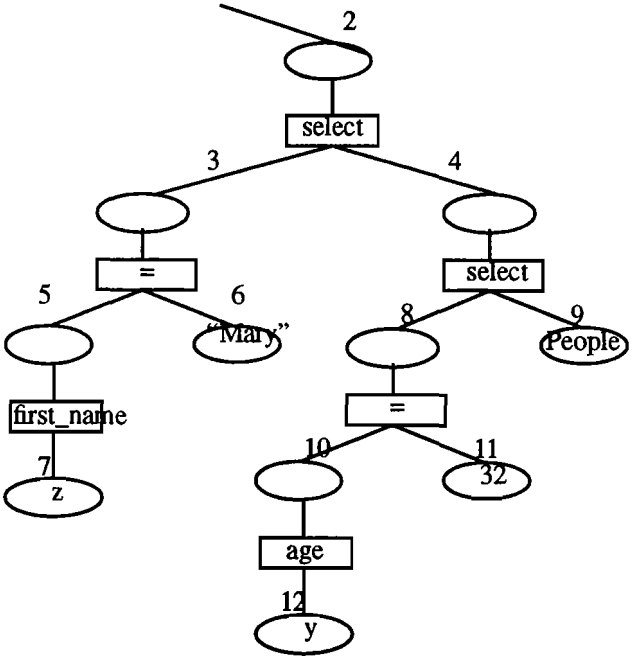
(b)

apply(λ(x) "c") (select (λ(z) (z.first_name = " Mary")) (select (λ(y) (y.age = 32)) (People)))

Figure 8: The Working Mechanism of the EAT-Control Region (5), where arrow in (a) indicates the location where we get the subquery, and the query in (b) is the one we send to the child region.



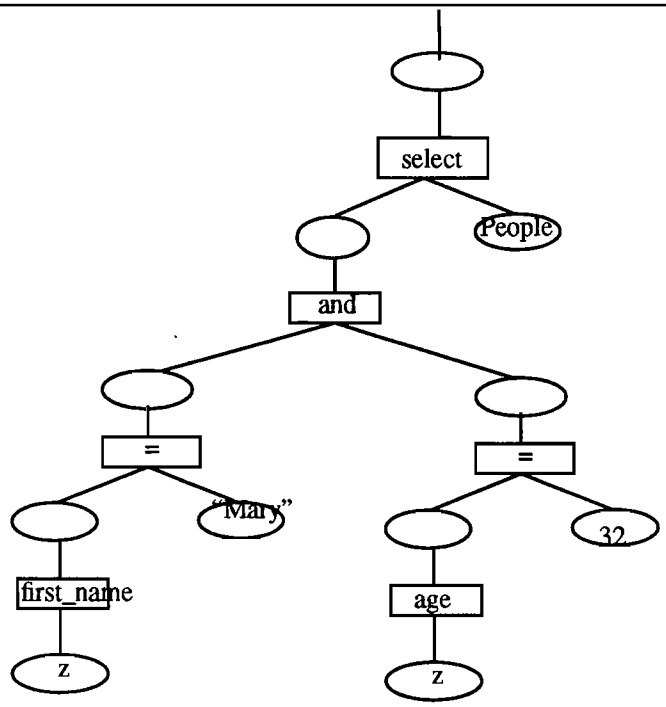
(a)



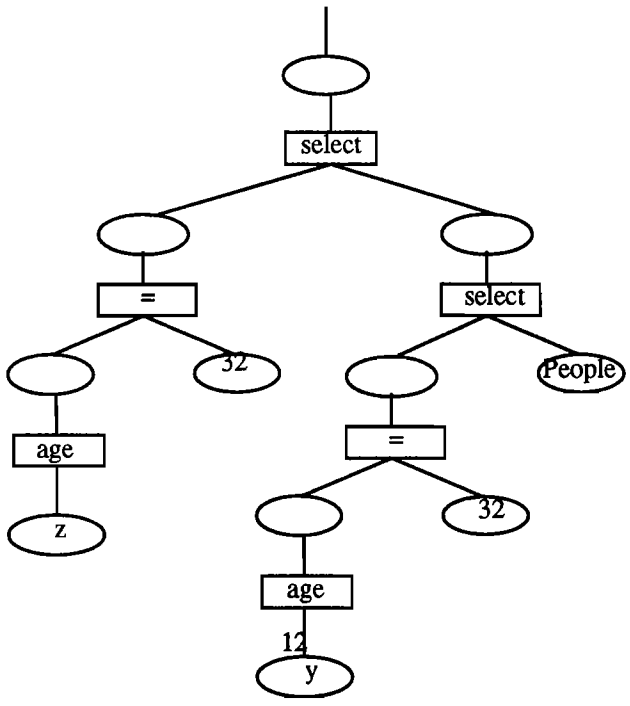
(b)

apply(λ(x) "c") (select (λ(z) (z.first_name = " Mary")) (select (λ(y) (y.age = 32)) (People)))

Figure 9: The Working Mechanism of the EAT-Control Region (6), where arrow in (a) indicates the location where we get the subquery, and the query in (b) is the one we send to the child region.



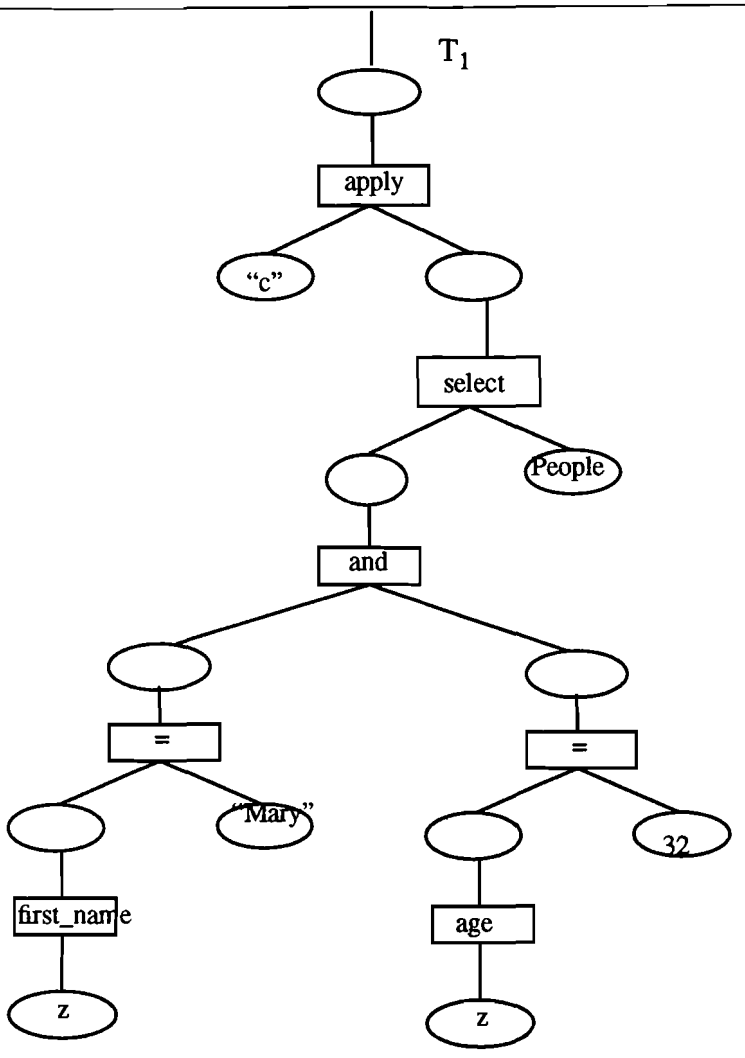
(c)



(d)

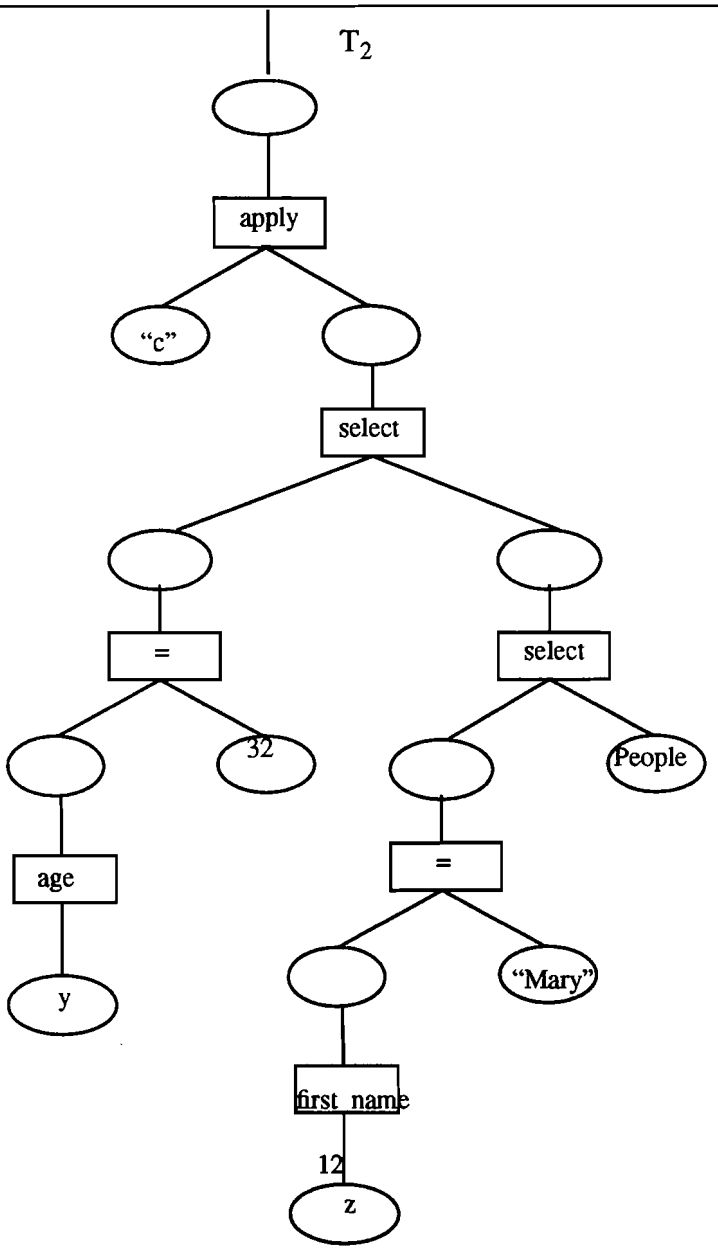
apply(λ(x) "c") (select (λ(z) (z.first_name = " Mary")) (select (λ(y) (y.age = 32)) (People)))

Figure 10: The Working Mechanism of the EAT-Control Region (7), where arrow in (a) indicates the location where we get the subquery, and the query in (b) is the one we send to the child region.



applt(λ(x) "c") (select (λ(z) (z.first_name ="Mary") and (z.age = 32)))(People))

Figure 11: The Working Mechanism of the EAT-Control Region (8)



apply(λ(x) "c") (select (λ(z) (z.first_name) = "Mary") and (z.age =32))) (People))

Figure 12: The Working Mechanism of the EAT-Control Region (9)

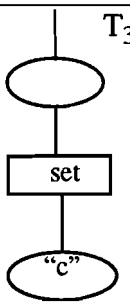


Figure 13: The Working Mechanism of the EAT-Control Region (10)

"select" and "apply" have goals "select" and "apply" respectively. The sequence of traversing the EAT is other arc first, and then input arc at the same level. The EAT is traversed bottom up. For the clarity of the presentation, all the FDArcs are labeled with numbers.

1. At first, we reach the leaf arc, arc 1, as shown Figure 4. we know a leaf arc is reached because the arc does not have any function node as its child. Clearly, it is not necessary to transform this arc.

2. Next we go to the arc 5 because the arc 7 is a leaf arc, as shown in Figure 5, and take the name of the function node "first_name" as a subgoal. Since no match on the subgoal can be found, no further action is taken on this arc. As the arc 6 is a leaf, we go to the arc 3, as shown in Figure 7. The subgoal for the EAT is '=', no match is found.

3. Next the arc 4 is traversed. Obviously, in this arc only the name of the top function "select" matches the goal of the child region "select". We make a copy of this arc and send the copied EAT, as shown in Figure 8, to the child regions for bids, and choose the select region for the transformation. In the child region, the pattern of the EAT is compared with the predefined patterns. However, no match is found, and a null value is returned.

4. The control returns to the arc 2. Make a copy of the arc, as shown in Figure 9, collect the bids from the child regions, and send the EAT to the select region. In the child region, the pattern of the EAT matches two predefined patterns. Thus, two equivalent queries are returned, as shown in Figures 10. In the control region, make the two copies of the original EAT T_0, truncate the old arcs from the copied trees, and insert the two queries at the same location on the separate trees, Thus, we have totally stored a set of three trees, T_0, T_1, T_2 in the control region, as shown in Figures 11 to 12.

5. Go to the root arc of the entire EAT.

6. At this moment, we have to send all the queries in the query set one by one to the apply region for the transformation. This time, the child region transforms the inputs into identical EATs for all the input EATs to the region, as shown in Figure. All the returned EAT are put into the query set, and the duplicate EATs are eliminated. Therefore, there are a total of four EATs in the query set, T_0, T_1, T_2 and T_3

7. In the control region, the returned query set is sorted and pruned based on the cost. This leaves us with the EAT representing the query "set[c]" as the best equivalent query.

3.2 Class of the EAT-Based Control Region

3.2.1 REControlRegionEATBased::public REControlRegion

This class is implemented to carry out the EAT-based control strategy.

Public Methods:

- **REControlRegionEATBased**
  Constructor.

- **~REControlRegionEATBased**
  Destructor.

Protected Methods:

- **virtual int CustomApplicable(RPFDArc* TheEat, const char* Goal) ;**
  If the given goal matches the predefined the goal of the region, return a non-zero bid, otherwise a zero bid.

- **virtual REQuerySet* CustomAttain (RPFDArc* TheEat, const char* Goal, WORD MaxReturnSize);**
  Call the EatTraverse method to optimize the EAT from bottom up, sort the transformed EAT in order of cost, prune the query set, and finally return the pruned query set.

- **REQuerySet* EatTraverse(RPFDArc *TheEat) = 0;**
  This is a pure virtual function, and the user must implement it.

3.2.2 SR_Optimizer::public REControlRegionEATBased

This class is a user-defined class to carry out the EAT-based control strategy.

Public Methods:

- **SR_Optimizer**

  Constructor.

- **~SR_Optimizer**

  Destructor.

Protected Methods:

- **REQuerySet* EatTraverse(RPFDArc *TheEat);**

  This method is called from the method CustomAttain. It is called recursively in order to traverse the entire EAT. This method slices each of the child arcs of the current function node and makes a recursive call to itself to traverse the EAT from the current function node one level down to all its child function nodes. The call to this method returns a set of transformed EATs to its parent, and the parent inserts the returned EATs into where they come from. Finally, the transformed EATs are sent to a child region for the transformation, and the transformed queries are returnd in a set.

- **REQuerySet* InputTreeTraverse(RPFDArc *TheEat,REQuerySet * QSStorage);**

  This method is called from the method EatTraverse to traverse the input arc of the current function node. In this method, the input arcs of the given query TheEAT are optimized on by one. If a non-null value is returned for each input arc, the old input arc is removed and the optimized one is inserted at the same location.

- **REQuerySet* OtherTreeTraverse(RPFDArc *TheEat, REQuerySet * QSStorage);**

  This method is called from the method EeaTraverse to traverse the other arc of the current function node.In this method, the other arcs of the given query TheEAT are optimized on by one. If a non-null value is returned for each other arc, the old other arc is removed and the optimized one is inserted at the same location.

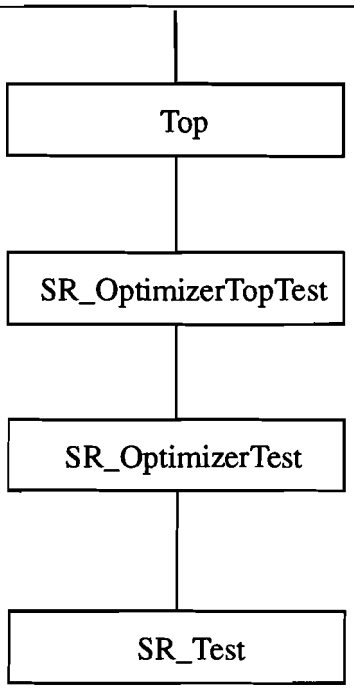


Figure 14: Architecture of Test Regions

3.3 Testing Strategy

The testing strategy in the EPOQ optimizer is developed to determine the performance of both the control region strategy and query transformation. The approach used is to have a control region slice a piece of EAT and send the EAT to a child region. The child takes in the EAT, deletes all the child nodes of the given EAT's the top data node, and returns the top data node to the control region. The control region inserts the transformed EAT back into the arc where the EAT came from. The iteration is implemented to keep slicing the EAT and sending the spliced EAT to the child region, so we can observe by means of GrOOVE [1] that the entire EAT is shrinking until its top data node is left.

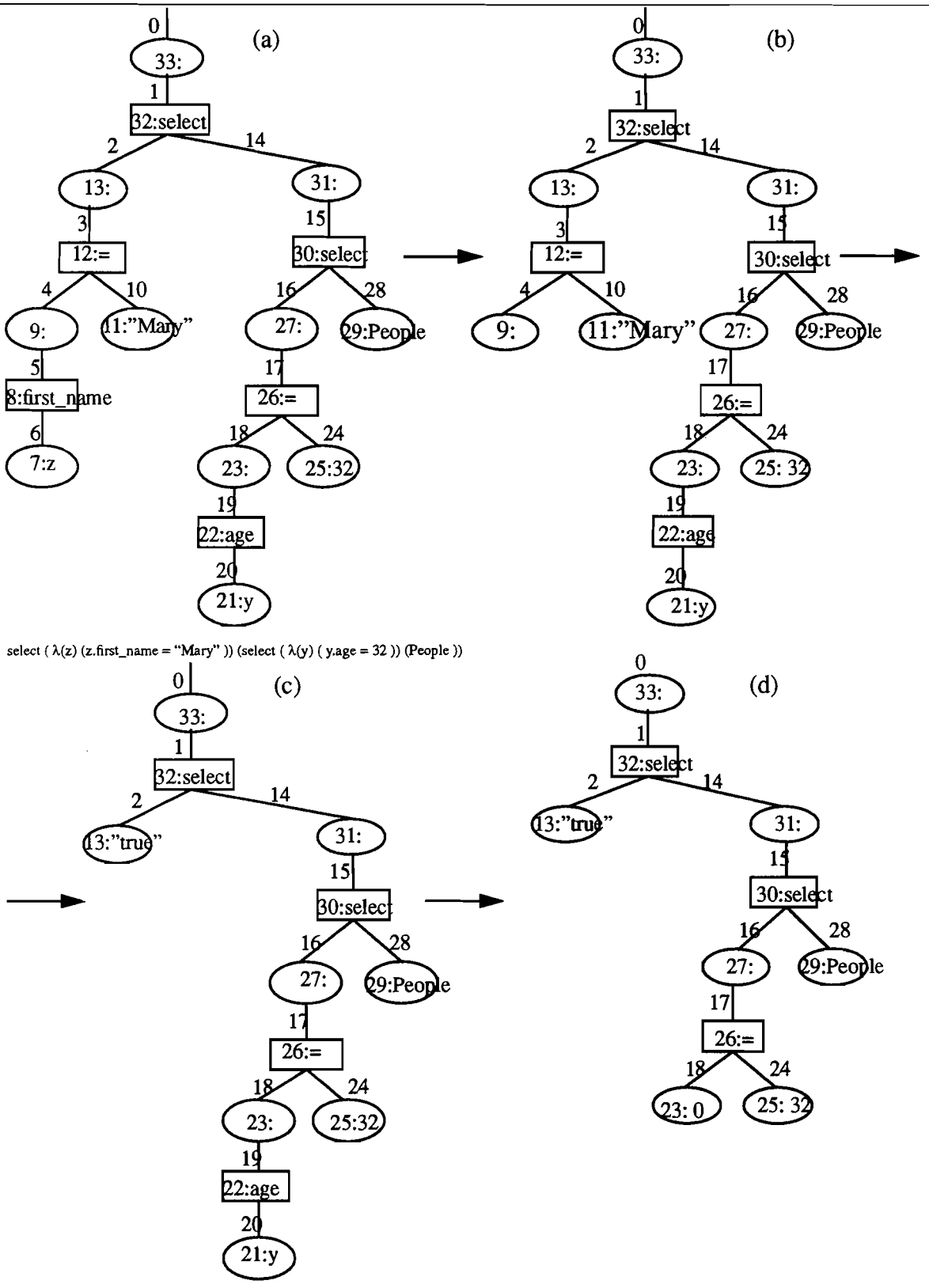
3.3.1 Test Regions

The testing strategy is implemented by a test module, a collection of test regions. The architecture of the test regions is shown in Figure 14, in which both SR_OptimizerTopTest and SR_OptimizerTest are control regions. The reason we use two control regions is that it facilitates the display of the movie in GrOOVE. GrOOVE only displays an updated EAT when the EAT enters or leaves a

[1] A movie displaying panel for the EAT[9]

region. In our test regions, the control region SR_OptimizerTest has a direct control over the leaf region, and keeps the information on the updated (transformed) EATs, but the updated EAT in this region can not be displayed without leaving the region. By using two control regions, the control strategy is performed by both of the regions, and both of the regions communicate with each other all the time, including sending the EAT back and forth between the two two regions. Thus, the EAT will be displayed as it leaves and enter either of the regions. In this approach the top control region SR_OptimizerTopTest only serves as a receiver and sender of an EAT.

Figure 8 has demonstrated the working mechanism of test regions. The order of traversing an EAT is traversing the Other arc first, and then the Input arc. A leaf arc to be sent to the leaf region is always FDArc. In this example, the control flow first reaches the arc 6 as well as node 7, and send the arc to the leaf region. Since this arc has no child function node to be truncated, the unattached arc is returned. The control will go to one level up to the arc 4, and send the arc to the leaf region. The leaf region only keeps the data node 9, deletes the arc5, and returns the arc 4. The value of the left data node is reassigned based on the type of this node. For instance, if the type is a boolean, the value of the node will be "true", and if the type is an integer, the value will be "0". After the control region receives the new arc4, it deletes the old arc 4, insert the new one, and returns to the higher level. This time the control sends the arc 2 to the leaf region, and the leaf region returns a arc with a data node whose value is "true". The old arc 2 is deleted and the new arc is inserted. The same working mechanism is applied to the transformation of the Input arc 14.


(a)
0
33:
1
32:select
2
14
13:
31:
3
15
12:=
30:select
4
10
16
28
9:
11:"Mary"
27:
29:People
5
17
8:first_name
26:=
6
18
24
7:z
23:
25:32
19
22:age
20
21:y
(b)
0
33:
1
32:select
2
14
13:
31:
3
15
12:=
30:select
4
10
16
28
9:
11:"Mary"
27:
29:People
17
26:=
18
24
23:
25: 32
19
22:age
20
21:y
select (λ(z) (z.first_name = "Mary")) (select (λ(y) (y.age = 32)) (People))
(c)
0
33:
1
32:select
2
14
13:"true"
31:
15
30:select
16
28
27:
29:People
17
26:=
18
24
23:
25:32
19
22:age
20
21:y
(d)
0
33:
1
32:select
2
14
13:"true"
31:
15
30:select
16
28
27:
29:People
17
26:=
18
24
23: 0
25: 32

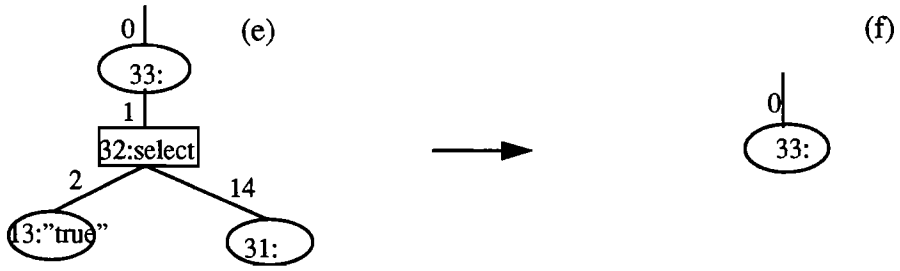


Figure 15: Query Transformation in the Test Regions

3.3.2 Test Region Classes

class SR_OptimizerTopTest : public REControlRegionEATBased

This is a class only responsible for sending and receiving an EAT.

Public Methods:

- **SR_OptimizerTopTest(char* FileName)**

  Constructor.

- **~SR_OptimizerTopTest(void)**

  Destructor.

- **int CustomApplicable(RPFDArc* TheEat, const char* Goal)**

  Return a non-zero bid, if the goal is matched, otherwise zero.

- **REQuerySet* CustomAttain (RPFDArc* TheEat, const char* Goal, WORD MaxReturnSize)**

  Keep sending the EAT to its child region for slicing the EAT until only the top node of EAT is left.

class SR_OptimizerTest : public REControlRegionEATBasedTest

This is a class responsible for branching the EAT to the test leaf region for query transformation. (slicing the EAT)

Public Methods:

- **SR_OptimizerTest(char* FileName)**

  Constructor.

- **~SR_OptimizerTest(void)**

  Destructor.;

- **int CustomApplicable(RPFDArc* TheEat, const char* Goal)**

  If the given goal matches the goal "test", return a non-zero bid.

- **REQuerySet* CustomAttain (RPFDArc* TheEat, const char* Goal, WORD MaxReturnSize)**

  Keep sending the EAT back and forth to the other control region until the EAT shrinks to a arc with a single data node.

class REControlRegionEATBasedTest: public REControlRegionEATBased

This class is responsible to test the performance of control region strategy and query transformation.

Public Methods:

- **REControlRegionEATBasedTest();**

  Constructor.

- **REControlRegionEATBasedTest(char * FileName);**

  Constructor. It opens a file which specifies the goal which the child region has to achieve.

- **REControlRegionEATBasedTest();**

  Destructor.

- **RPFDArc* TestTraverse(RPFDArc *TheEat, int &flag);**

  This method is called recursively in order to traverse an entire EAT. This method is the same as the method EatTraverse() in the class REControlRegionEATBased, except that one more parameter, flag, is added to the TestTraverse. Flag indicates an action whether or not to transform a query. In the recursive call to this method, only a leaf of the given EAT is transformed. Once the transformation is done on the leaf, no further transformation is performed, and flag is set to zero (the initial value of flag is 1). Finally, the input EAT returns.

class SR_Test : public RELeafRegion

This is a class responsible for branching the EAT to the test leaf region for query transformation (truncating the EAT).

Public Methods:

- **SR_Test(char* FileName)**

  Constructor.

- **~SR_Test(void)**

  Destructor.;

- **int CustomApplicable(RPFDArc* TheEat, const char* Goal)**

  If the given EAT contains a function node, return a non-zero bid because this EAT should be sent to the child region for transformation (truncation).

- **REQuerySet* CustomAttain (RPFDArc* TheEat, const char* Goal, WORD MaxReturnSize)**

  This is a method to do the actual transformation (truncation). The given EAT is truncated into a smaller size, that is, delete the child nodes of the top data node. The value of the data node is re-set based on the type of the data. For example, if the data type is a boolean, the value is set to "true", and if the data type is a integer, the value is set to "0", etc. Finally we return the new EAT.

4 The Query Transformation

4.1 Implementation of Query Transformation

Query transformation is the responsibility of child regions. Each child region is assigned a task to transform the queries which match certain patterns. In this paper, pattern refers to as a sequence of function nodes that appear in a EAT, with specification of the location for each function node. In the EPOQ optimizer, the pattern is hard-coded into the application code.

As stated in previous chapter, once a query matches a pattern, the query will be sent to a child region for transformation. The transformation is based on reordering of the input query. In the child region, the given query is reorganized into an equivalent query but that is expected to execute efficiently. In this chapter, we will focus on the transformation of the queries that contain the top function node **apply**.

In AQUA query algebra, the definition of function **apply** is:

apply(f)(A) = {f(a)|a ∈ A}.

where A is the input set.

Apply applies the function f to each element in the set A. The result is a set with one element corresponding to each element in set A, derived from applying f to the element (with duplicates eliminated).

For the definitions of other operators in the AQUA, see [8].

The query transformation associated with the operator **apply** is classified into the following:

1. **apply**($\lambda(x)$**x**)(**A**) ⇒ **A**

   where A is a set.

2. **apply**($\lambda(x)$ **const**)(**A**) ⇒ **set (const)**

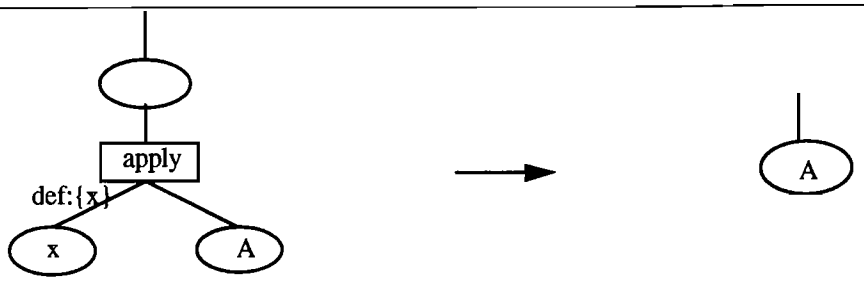
   where A is a set, and 'const' is a constant function that does not depend on the the input..

3. **apply**($\lambda(x)$ **f(x)**)(**union** (≡)(**A, B**)) ⇒ **union**(≡)(**apply**($\lambda(x)$**f(x)**)(**A**), **apply**($\lambda(x)$ **f(x)**)(**B**))

   where A and B are sets.

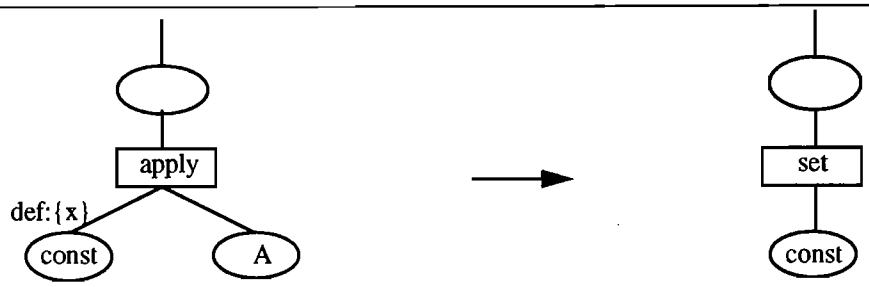
4. **apply**($\lambda(x)$ **f(x)**)(**apply**($\lambda(x)$ **g(x)**)(**A**)) ⇒ **apply**($\lambda(x)$ **f(x).g(x)**)(**A**)

   where f(x) and g(x) are unary functions (path expression), · is the composition operator and A is a set.



apply(λ(x) x) (A) -> A
where A is a set.

Figure 16: Query Transformation 1



apply(λ(x) const)(A) -> set (const)
where A is a set, and const is a constant function that does not depend on the input.

Figure 17: Query Transformation 2

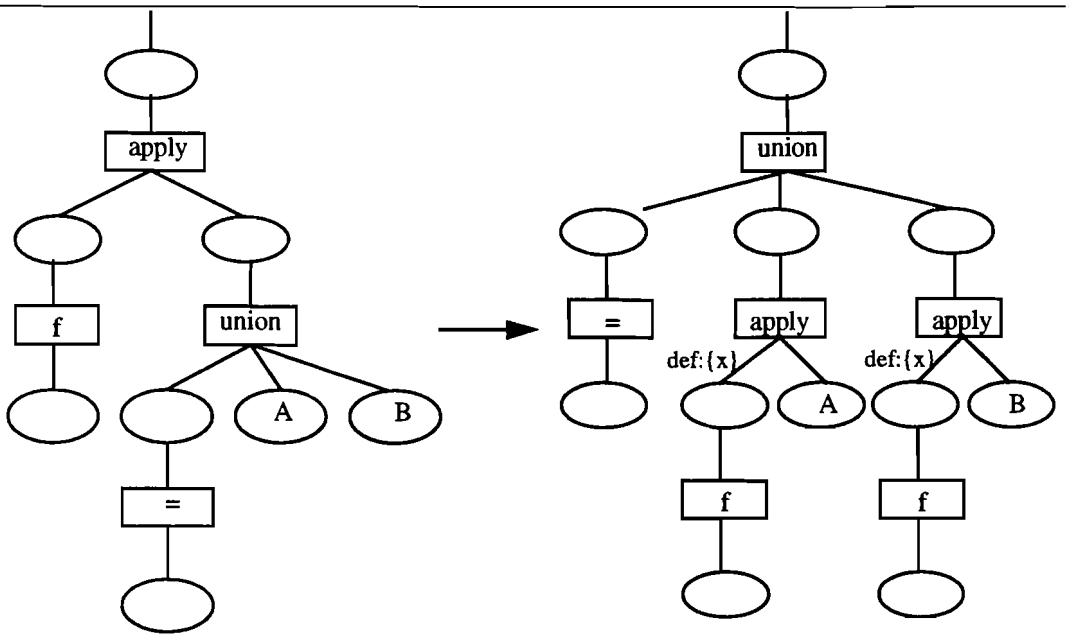
5. **apply**($\lambda(x)$ **f(x))(apply**($\lambda(x)$ **g(x))(A))** $\Rightarrow$ **apply**($\lambda(x)$ **f(g(x))(A)**
   where f(x) and g(x) are any functions, and A is a set.

4.1.1 The First Transformation of the Query Containing the Root Function apply

In this transformation, as shown in Figure 9, We simply make a copy of the FDArc which is connected with the top data node of set[A], and return that arc.

4.1.2 The Second Transformation of the Query Containing the Root Function apply

In this transformation, as shown in Figure 10, we make a copy of the top arc to the constant data node, delete the annotations for the lambda variable "x', and use it as the bottom FDArc of the constant set. Next thing to do is to build a new set. Since the EAT is a double-linked tree, the algorithm to build a relationship between two nodes A, B is as follows.



apply(λ(y) f)(union(=) (A, B)) -> union((=) (apply(λ(x) f)(A), apply(λ(x) f)(B))
where f is a function, and A, B are sets.

Figure 18: Query Transformation 3

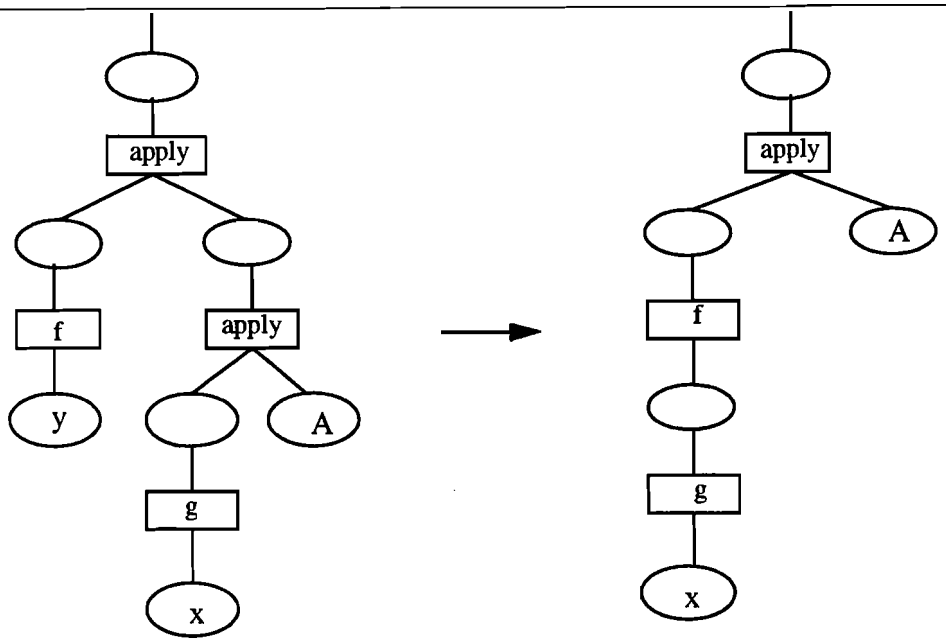
- Set B as the A's child.
- Set A as the B's parent.

4.1.3 The Third Transformation of the Query Containing the Root Function apply

Figure 11 demonstrates the transformation 3. In this case, we make a copy of both **set A** arc and **set B** arc, a copy of **union** arc, and delete the two input arcs of **union** function. Subsequently, we make two copies of **apply** arcs, delete their input arcs, and insert the **set A** arc and **set B** arc in the input arcs of the two **apply** arcs respectively. Finally, we insert the two new **apply** arcs in the input arcs of the new **union** arc. When building the new EAT, we ensure that both parent and child relationship are set up, annotations are adjusted, and types are re-inferred.

4.1.4 The Fourth Transformation of the Query Containing the Root Function apply

In this transformation, as shown in Figure 12, we make a copy of **set A** arc, and a copy of the other arc of the second apply function, which contains function g. Next we delete the input arc of



apply(λ(y) f)(apply(λ(x) g)(A) = apply(λ(x) f.g)(A)
where f and g are unary functions (path expression)

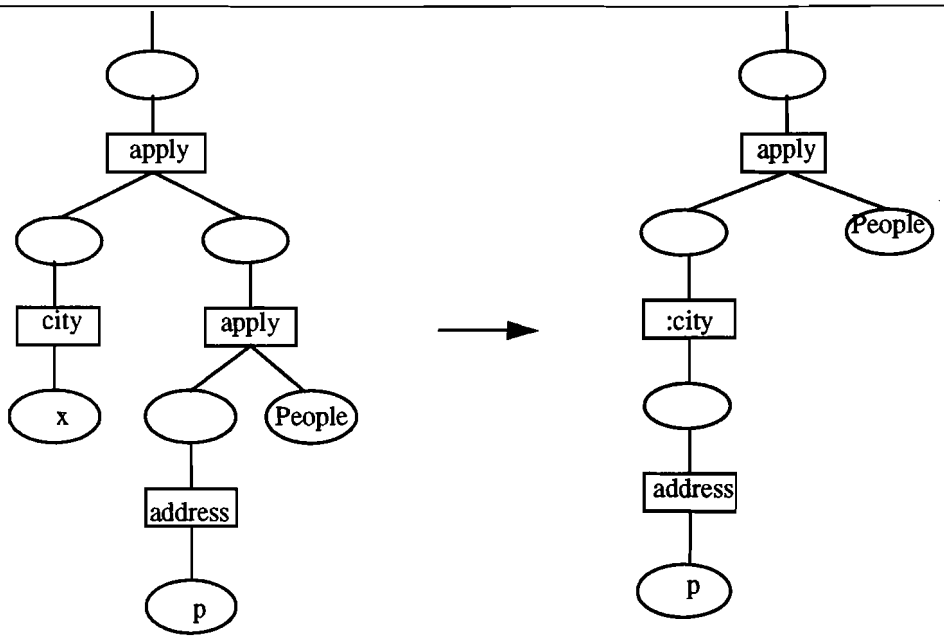
Figure 19: Query Transformation 4

the top function **apply**, and insert the **set A** arc. Finally, we insert the function g arc as the child arc of the function f. An example is shown in Figure 13.

4.1.5 The Fifth Transformation of the Query Containing the Root Function apply

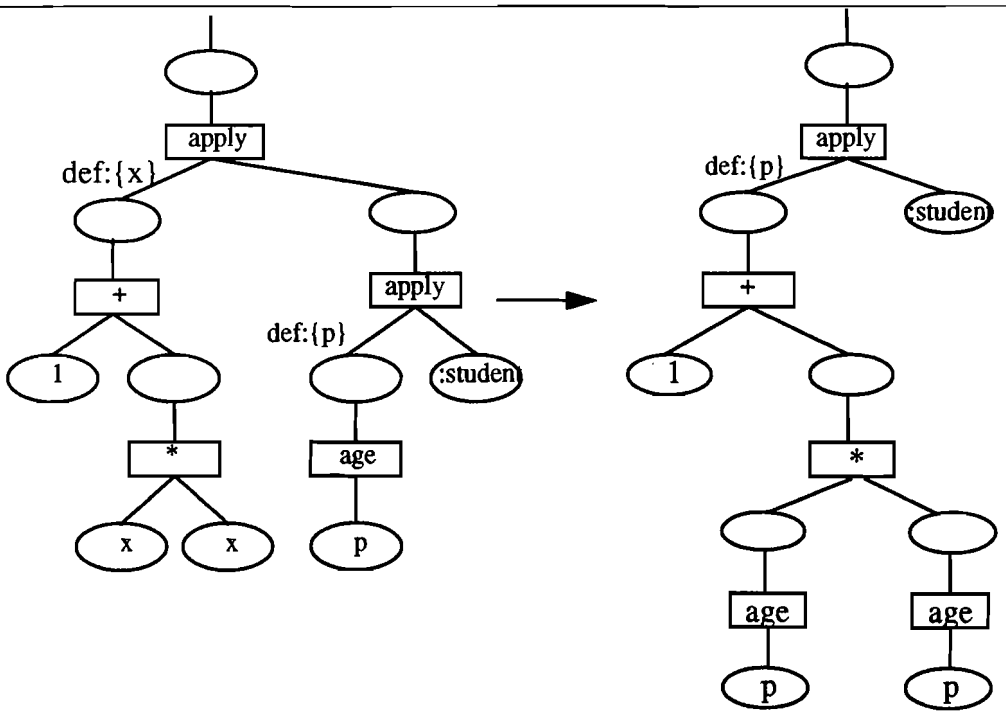
In this transformation, as shown in Figure 14, the variable **x** in the entire other Arc of the top **apply** function in the EAT is required to be replaced with a new function value, i.e. the variable x in the function f(x) is required to be replaced with the function g(y) in the AQUA query algebra. Thus, a method, ReplaceDataNode, is implemented to deal with the replacing variable with the function and adujsting the annotations.

The variable replacement is performed from the bottom up. The way to deal with variable replacement is to make a copy of the other Arc (a tree), to recursively traverse down the copied tree to the leaf, and to start the variable replacement by determining whether the defined variable on the top arc of the copied tree matches the name of the current data node. If a match is found, the current data node will be replaced with the new data node which contains the new function value (here the new data node may have its child function node whose value is kept at the new data node). Once the replacement has finished at this level, the recursive call will return to an upper



apply(λ(x) (x.city)) (apply (λ(p) (p.address) (People)) -> apply(λ(p) (p.city.address) (People)

Figure 20: The Example of Query Transformation 4



apply(λ(x) 1 + x * x)(apply(p) p.age) (Students)) -> apply(λ(p) 1 + p.age * p.age) (Students)

Figure 21: The Example of Query Transformation 5

level, and the replacement will continue all the way up to the top arc of the tree to be replaced. After the variable replacement has been accomplished, the old other Arc will be deleted and the replaced arc will be inserted at the same location. The algorithm is given as follows.
In the following function,

- Var stands for the value of a data node to be replaced, which is defined on the top FDArc of Tree.
- NewVar for a arc that contains the new value to replace.
- Tree for the EAT whose data node's value matching Var will be replaced by the newvar.

ReplaceDataNode(Tree, Var, NewVar)

1. *Get a copy of Tree, RetTree.*
2. *If the Top data value = Var,*
   return NewVar;
3. *For each of other Arc of the RetTree's top function, OtherArc,*
   (a) Traverse the Other Arc and replace variables.
   ReplaceDataNode(OtherArc, Var, NewVar);
   (b) Delete the old Other Arc, and insert the new Other Arc.
4. *For each of input Arc of the RetTree's top function, InputArc,*
   (a) Traverse the Input Arc and replace variables.
   ReplaceDataNode(InputArc, Var, NewVar);
   (b) Delete the old Input Arc, and insert the new Input Arc.
5. *Return RetTree.*

The algorithm for the transformation 5 is as follows.

1. *Make a copy of the EAT for the given EAT, RetEAT.*
2. *Make a copy of the other arc of RetEAT, OtherArc, in which the values of the variables are to be replaced with a new value.*

3. *Get the value of the defined variable on the OtherArc, Var, with which all of the variables are to be replaced.*

4. *Get the arc which contains the new value. Here, the other arc of the second apply function.*

5. *ReplaceDataNode(OtherArc, Var, NewVar).*

6. *delete the old other arc of the RetEAT, and insert the new one at the same location.*

7. *Make a copy of the* **set[A]** *arc.*

8. *Delete the input arc of the top* **apply** *function, and insert the* **set[A]** *arc at the same place.*

9. *Return RetEAT.*

4.2 Class of Leaf Region

class SR_ApplyLeaf : public RELeafRegion

This class is responsible for the transformation of queries which are associated with the operator apply.

Public Methods:

- **SR_ApplyLeaf();**
  Default constructor.

- **SR_ApplyLeaf(char * FileName);**
  Constructor. It opens a file which specifies the goal which the child region has to achieve.

- **~SR_ApplyLeaf();**
  Destructor.

- **int CustomApplicable (RPFDArc* TheEat, const char * Goal);**
  If the top function of the given EAT is apply, return a non-zero bid. Otherwise, return zero.

- **REQuerySet* CustomAttain (RPFDArc* TheEat, const char* Goal, WORD MaxReturnSize);**
  In this method, a given query is transformed based the five predefined transformation rules. One transformation is done for each pattern that the query matches. The transformed queries are stored in a query set. Finally, all the transformed queries in the query set are returned to the parent region.

Private Methods

- **RPFDArc DoTransform1(RPFDArc TheEat);**

  Transform the query which matches the transformation rule 1.

- **RPFDArc DoTransform2(RPFDArc TheEat);**

  Transform the query which matches the transformation rule 2.

- **RPFDArc DoTransform3(RPFDArc TheEat);**

  Transform the query which matches the transformation rule 3.

- **RPFDArc DoTransform4(RPFDArc TheEat);**

  Transform the query which matches the transformation rule 4.

- **RPFDArc DoTransform5(RPFDArc TheEat);**

  Transform the query which matches the transformation rule 5.

- **RPFDArc *ReplaceDataNode(RPFDArc *TheEat, char *defvar, RPFDArc *Arc);**

  In the given tree TheEat, replace the arc whose data nodes' values match **defvar** with **Arc**. For details, see the algorithm in section 4.1.5.

5 Conclusion

In this project, the EAT-based control region was implemented to carry out a bottom-up EAT-based control strategy. The EAT-base control strategy is the strategy that the control decision is made based on the given EAT, i.e. the goal chosen depends on the name of the top function of the current EAT subquery. Unlike other extensible optimizers which have a fixed control strategy, the EPOQ optimizer using the EAT-based control strategy provides the efficient means in searching the space of transformation rules by modularizing the rule base by the top AQUA operation in the EAT, and the flexibility in making a decision based on the given query.

The responsibility of the EAT-based control region is to determine the goal based on the given EAT, to request the bids from the child regions on the EAT and to branch the EAT to the regions which return the required bids for further processing.

The leaf region **apply** has been developed, which is responsible for the query transformation of the EATs associated with the function apply. The Test regions are also built to determine the performance of both the control region strategy and query transformation.

REFERENCES

[1] F. Bancilhon and W. Kim. *Object-Oriented Database Systems: In Transition. ACM SIGMOD Record Special Issue on Directions for Future Database Research and Development*, 19, 4 (December 1990), 49.

[2] J. C. Freytag, *A Rule-Based View of Query Optimization*, in SIGMOD Proceedings, pp. 173-180, May 1987.

[3] Goetz Graefe and David J. DeWitt. *The EXODUS Optimizer Generator*, pp. 160-172, 1987 ACM.

[4] Jose A. Blakeley, William J. McKenna and Goetz Graefe. *Experiences Building the Open OODB Query Optimizer*, SIGMOD, pp. 287-296, May 1993.

[5] Gail Mitchell. *Extensible Query Processing in an Object- Oriented Database.* Ph.D thesis. Department of Computer Science, Brown University, May 1993. Technical Report CS-93-16.

[6] Gail Mitchell, Umeshwar Dayal and Stanley B. Zdonik. *Control of Extensible Query Optimizer: A Planning-Based Approach*, Proceedings of the 19th VLDB Conference, Dublin, Ireland 1993.

[7] Laurent-David Hasson. EPOQ Regions: New Design Specifications, Brown University Department of Computer Science, December, 1994.

[8] Marian H. Nodine, Farah B. Abbas, Mitch Cherniack. *The EPOQ Query Optimizer for Object-Oriented Database Design Specification*, Brown University Department of Computer Science, June, 1994.

[9] Curtis Bragdon etc. *GrOOVE - User and Programmer Notes*, Department of Computer Science, Brown University, May, 1994.