Implementation of Attribute Metadata with Application to Data Mining

Andrey Rozenberg

School of Computer Science

McGill University, Montreal

January 2002

A Project Report Submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements of the degree of Master of Science in Computer Science

T. H. Merrett, Advisor

Contents

List of Figures

Chapter 1

Introduction

1.1 Background

The implementation of Attribute metadata was built on an existing database system called jRelix, a Relational database programming language in Java. It is based on an algebraic data manipulation language called Aldat [Mer77] and provides an interactive environment for exploring the concepts of relational database model described in [Mer84]. The jRelix system supports the relational algebra as well as the domain algebra for both flat and nested relations. Since its original inception in 1997, more functionalities such as computations, event handling and even a geo-spatial data editor were added to the jRelix system.

1.2 Motivation

As described in [Mer01], the classical relational algebra shows limitations when it comes to multi-dimensional data analysis. In case of an on-line analytical processing (OLAP), a d-dimensional datacube must be built using relational algebra. The previous implementation of jRelix allows to build a specific datacube using relational and domain algebra. However, it fails when it comes to generalize this relational implementation to a datacube of arbitrary dimension: it has no support for a loop of d steps where each step of this loop involves a different attribute of the data relation that is being analyzed. As a result, a relational database programming language such as jRelix must support attribute metadata together with the other allowed attribute data types such as `integer, string, boolean, double`, etc... in order to be able to perform generalized data analysis. For a more thorough treatment of the subject please refer to [Mer01].

Chapter 2

User's Manual on Attribute Metadata

In this chapter, the two relations of Figures 2.1 and 2.2 are used for illustrative purposes. Relation "Training" is a flat relation whose attributes are all of the same type `String`. On the other hand, the attributes of relation "TestRel" are all of different types.

2.1 Getting Started

The jRelix software can run on any platform with Java Runtime Environment installed. To start jRelix the user must enter the following command at the command prompt of the operating system:

```
>java jRelix
```

After performing internal system initialization, jRelix displays its copyright information and a command prompt sign ">". At this point jRelix system is ready to accept user commands. Each user command or a statement must terminate with a semicolon (';').

Outlook	Temperature	Humidity	Windy	Class
overcast	cool	normal	t	P
overcast	hot	high	f	P
overcast	hot	normal	f	P
overcast	mild	high	t	P
rain	cool	normal	f	P
rain	cool	normal	t	N
rain	mild	high	f	P
rain	mild	high	t	N
rain	mild	normal	f	P
sunny	cool	normal	f	P
sunny	hot	high	f	N
sunny	hot	high	t	N
sunny	mild	high	f	N
sunny	mild	normal	t	P

Figure 2.1: Training relation

A	B	C	D
A1	1	true	1.1
A2	2	false	1.2
A3	3	true	1.3
A4	4	false	1.4
A5	5	true	1.5
A6	6	false	1.6

Figure 2.2: TestRel relation

To exit the jRelix program, the command **quit**, terminated by a semicolon, must be entered after the command prompt sign.

2.2 System Commands

2.2.1 *pr* Command

One of the most widely used commands in the jRelix is the **pr** command which is used to print the contents of a relation. The general syntax of the command is the following:

>**pr** *rel_name*;

where *rel_name* is the name of the relation the contents of which will be displayed on the screen.

However, in case of a nested relation, the data of a nested domain is not printed along the data of the rest of the relation. Instead, the nested domain of the printed relation will contain the surrogates or pointers to the actual values of this nested attribute which are stored in a separate relation. The name of the relation which stores the actual values of the nested attribute is the same as that of the nested domain preceded by a dot.

Figure 2.3 shows an example of a nested relation and its contents when printed. Relation R is a nested relation where attribute `xpose` is a nested domain. When relation R is printed the data of the nested attribute `xpose` is replaced with surrogates (1 and 2). The actual data values of the nested attribute `xpose` are stores in the relation `.xpose`.

```
 R (  A     |     B     |    C    |   xpose                      )
                                   ( Attr  |       val          )
-----------+----------+--------+----------+-------------------
    ``a''  |     1    |  true  |   ``A''  | ``string;a''
           |          |        |   ``B''  | ``integer;1''
           |          |        |   ``C''  | ``boolean;true''
-----------+----------+--------+----------+-------------------
    ``b''  |     1    | false  |   ``A''  | ``string;b''
           |          |        |   ``B''  | ``integer;1''
           |          |        |   ``C''  | ``boolean;false''
-----------+----------+--------+----------+-------------------


>pr R;
   A    |     B     |    C    |   xpose
--------+----------+--------+-----------
   a    |     1    |  true  |     1
   b    |     1    |  false |     2
--------+----------+--------+-----------


>pr .xpose;
   .id  |   Attr   |     val
--------+----------+------------------
    1   |     A    |   string;a
    1   |     B    |   integer;1
    1   |     C    |   boolean;true
    2   |     A    |   string;b
    2   |     B    |   integer;1
    2   |     C    |   boolean;false
```

Figure 2.3: Output of the **pr** command on a nested relation

2.2.2 *sd* and *sr* Commands

Two particularly useful commands are **sd** and **sr** which are used to display information about domains and relations respectively which have been declared in the jRelix system.

More specifically, the **sd** command displays information about all domains declared in the database system. The shown information consists of every domain's name, its type, how many times this domain is used by relations or other domains, and in case of a nested domain a list of domains on which the current nested domain is defined. In case of a virtual domain, a corresponding expression tree is shown.

```
>sd;
------------------------------- Domain Table -------------------------------
Name        Type            NumRef       Dom_List
----------------------------------------------------------------------------
Attr        string              1
val         string              1
xpose       idlist              0          .id, Attr, val,
x           integer             0
                              Vertical:308:332:null:0
                               Literal:470:453:null:0
                               ExpressionList:592:592:null:0
                                Identifier:230:230:Attr:0
                                Identifier:230:230:val:0
```

Figure 2.4: Structure of a Domain Table

An example of the structure of a domain table is shown in Figure 2.4. As can be seen, domains *Attr* and *val* are both of type `string` and are referenced once by the *xpose* domain. The *xpose* domain, on the other hand, is a nested domain defined on *Attr* and *val* attributes which are listed in the `Dom_List` column. Attribute *x* is a virtual attribute and its expression tree is shown in the domain table.

In addition, if the **sd** command is followed by an optional domain name, then only this particular domain's information will be displayed. The command syntax is as follows:

>**sd** [*dom_name*];

where [*dom_name*] indicates an optional domain name.

Similarly, the **sr** command displays information about all relations declared in the jRelix system. The shown information consists of relation's name, its type which can be a `relation`, a `computation` or a `view`, its number of attributes, its number of tuples, and the number of attributes this current relation is sorted on.

Similarly to the **sd** command, the **sr** command can be followed by an optional relation name. As a result, only information regarding this particular relation will be displayed. The command syntax is as follows:

>**sr** [*rel_name*];

where [*rel_name*] indicates an optional relation name.

However, both **sd** and **sr** commands only display user-defined domains and relations. Two additional commands, **ssd** and **ssr**, allow us to display information on the system-defined domains and relations respectively whose names are preceded with a dot ('.') and are hidden from the user. Both **ssd** and **ssr** commands do this by acting as toggles for the **sd** and **sr** commands respectively.

2.2.3 *srd* Command

The **srd** command is used to display information relating relations to domains on which these relations are defined. the shown information consists of a relation name, domain names on which this relation is defined, and the index (position) of a domain in its relation. The command syntax is simply:

>**srd**;

2.2.4 *dd* and *dr* Commands

The **dd** and **dr** commands are used to remove domains and relations respectively from the jRelix system. The syntax for both commands is as follows:

>**dd** *dom_list*;

and

>**dr** *rel_list*;

where *dom_list* and *rel_list* are lists of domains and relations respectively.

2.2.5 *debug* Command

This command turns the debugging mode on/off in the jRelix system. When the debugging mode is turned `on`, a syntax tree corresponding to a user command or a statement is printed, It is particularly useful for debugging purposes to see how the jRelix system processes different user commands and statements. The command syntax is simply:

>**debug**;

2.2.6 *input* Command

This particular command allows the user to input a file containing user commands and statements to the jRelix program. This input file can be generated using any text editor. The syntax for this command is as follows:

>**input** *"file_name"*;

where *"file_name"* is the name of the input file. However, this input file must be in the current directory. Otherwise, the path to the file must be specified as part of *"file_name"*.

2.3 Declaration of Attribute and Universal Data Types

All data types available in jRelix system can be divided into two categories: atomic and complex data types. The atomic data types are primitive types such as `double`, `integer`, `string`, ... On the other hand, nested relations and computations are examples of complex data types.

```
Data Type |  Short Form |       Domain
----------+-------------+----------------------
integer   |     intg    | signed integer
short     |     short   | signed short integer
long      |     long    | signed long integer
float     |     float   | signed floating point
double    |     double  | signed double point
boolean   |     bool    | true or false
string    |     strg    | sequence of characters
attribute |     attr    | sequence of characters
universal |     univ    | sequence of characters
```

Figure 2.5: Atomic Data Types in jRelix

We added two more atomic data types to the jRelix system. These are **Attribute** and **Universal** data types. The **Attribute** data type is used to store the name of an attribute while the **Universal** data type is capable of holding any type of data. Figure 2.5 shows all nine atomic data types available in jRelix.

The syntax used to declare a domain of type **Attribute** or of type **Universal** is the same as that used for any atomic data type:

>**domain** *dom_name dataType*;

where *dom_name* is the name of the domain that is being declared and *dataType* is an atomic data type of Figure 2.5 that is assigned to this newly declared domain. Here are two examples of domain declaration:

>**domain** Attr attribute;

>**domain** val univ;

Figure 2.6 displays all domains declared in the database. Domain *Attr* is of type `attribute` and domain *val* is of type `universal`. Both domains are not referenced by any other domain or a relation: their `NumRef` value is 0. All domains declared are actual domains and are not nested.

The values of both data types are represented and stored as strings, although their format differs somewhat. In comparison with the data values of type Attribute, which can be stored as simple strings, the Universal data type is capable of holding any type of data. Thus, to store a data value of type Universal, it is necessary to store its type together with its value. As a result, the data value of type Universal is stored as a concatenation of two strings: a string representing the type of data and the data value itself casted as a string. Both strings are separated by a semicolon. For example, if 'a' is of type Universal and its value is 1.1 (float) then it will be stored as "float;1.1" string.

```
>sd;
------------------------------- Domain Table -------
Name        Type            NumRef       Dom_List
----------------------------------------------------
Temperature string              1
Class       string              1
Outlook     string              1
D           float               1
C           boolean             1
B           integer             1
A           string              1
Attr        attribute           0
Humidity    string              1
Windy       string              1
val         universal           0
```

Figure 2.6: A List of Declared Domains

2.4 *AttribsOf* Operator

AttribsOf is a jRelix operator which creates a relation of all the attributes of the operand. The general syntax of the **AttribsOf** operator is as follows:

AttribsOf *operand*;

where *operand* is a relation or an expression which results in a relation. Thus, **AttribsOf** is a unary operator. The output relation is defined on a single attribute of type `Attribute` and its values are the names of all the attributes of the operand relation.The name of the attribute of this unary relation is "Attr" by default. Figures 2.7 to 2.9 illustrate some examples of the **AttribsOf** operator.

```
>AttribsTraining <- AttribsOf Training;
>pr AttribsTraining;
+----------------------+
|  Attr                |
+----------------------+
| Outlook              |
| Temperature          |
| Humidity             |
| Windy                |
| Class                |
```

Figure 2.7: Attributes of "Training" relation

The parenthesis are used to indicate the precedence of operations: unary operators such as **AttribsOf** have precedence over projection and join expression. For more details on precedence of operations other than **AttribsOf** please refer to [Hao98].

```
>AttribsTest <- AttribsOf ([A,B,C] in TestRel);
>pr AttribsTest;
+----------------------+
|  Attr                |
+----------------------+
| A                    |
| B                    |
| C                    |
```

Figure 2.8: Projected Attributes of “TestRel” relation

```
>rel1 <- [A,B] in TestRel;
>rel2 <- [B,C,D] in TestRel;
>AttribsTest <- AttribsOf (rel1 ujoin rel2);
>pr AttribsTest;
+----------------------+
|  Attr                |
+----------------------+
| B                    |
| A                    |
| C                    |
| D                    |
```

Figure 2.9: Attributes of a relation resulting from a join expression

2.5 *Transpose* Operator

The AttribsOf operator can be replaced by a more general **Transpose** operator. Given some flat relation, the **Transpose** operator creates a nested relation of two attributes. One is of type Attribute and holds the list of attributes of this flat relation. The other attribute is of type Universal and holds the values for each attribute of the flat relation. In effect, the **Transpose** operator converts each tuple of an ordinary relation into a set of attribute-value pairs. In addition, the **Transpose** operator can transpose all or any subset of attributes of a given relation. While AttribsOf is a relational operator, the **Transpose** operator is part of domain algebra. The general syntax of the **Transpose** operator is as follows:

[*projected attributes*] **transpose** *dom_list*

where *projected attributes* is a list of attributes of the nested relation created by the **Transpose** operator. This list of attributes must contain one attribute of type Attribute or one of type Universal or both. The order in which these attributes are projected is not important. The *dom_list* parameter specifies a list of domains to transpose.

Figure 2.10 shows a simple example of applying the **Transpose** operator on relation "TestRel".

The resulting relation *TransposeTest* is a nested relation where *xpose* is a nested domain defined on two attributes: *Attr* which is of type Attribute and contains the names of attributes of the "TestRel" relation, and *val* which is of type Universal and holds the data values of "TestRel" relation, including its corresponding data type. That is, for each tuple and each attribute of the "TestRel" relation, the **Transpose** operator generates an attribute-value pair. This will allow to examine both attributes and its values at the same time.

In addition, the **Transpose** operator can transpose any subset of all the attributes of a relation. Figure 2.11 shows the result of transposing a subset of attributes of the "TestRel"

```
>domain Attr attribute;
>domain val universal;
>let xpose be [Attr,val] transpose A,B,C,D;
>TransposeTest <- [A,B,C,D,xpose] in TestRel;
>pr TransposeTest;
----------------------+-------------+--------+----------------+----------
    A                 | B           | C      | D              | xpose
----------------------+-------------+--------+----------------+----------
    A1                | 1           | true   | 1.1            | 7
    A2                | 2           | false  | 2.1            | 8
    A3                | 3           | true   | 3.1            | 9
    A4                | 4           | false  | 4.1            | 10
    A5                | 5           | true   | 5.1            | 11
    A6                | 6           | false  | 6.1            | 12
----------------------+-------------+--------+----------------+----------
>pr .xpose;
-----------------+----------------
  .id    |  Attr |      val
---------+-------+----------------
   7     |   A   |  string;A1
   7     |   B   |  integer;1
   7     |   C   |  boolean;true
   7     |   D   |  float;1.1
   8     |   A   |  string;A2
   8     |   B   |  integer;2
   8     |   C   |  boolean;false
   8     |   D   |  float;2.1
   9     |   A   |  string;A3
   9     |   B   |  integer;3
   9     |   C   |  boolean;true
   9     |   D   |  float;3.1
   10    |   A   |  string;A4
   10    |   B   |  integer;4
   10    |   C   |  boolean;false
   10    |   D   |  float;4.1
   11    |   A   |  string;A5
   11    |   B   |  integer;5
   11    |   C   |  boolean;true
   11    |   D   |  float;5.1
   12    |   A   |  string;A6
   12    |   B   |  integer;6
   12    |   C   |  boolean;false
   12    |   D   |  float;6.1
```

Figure 2.10: A "Transposed" nested relation

```
>domain Attr attribute;
>let xpose be [Attr] transpose A,B,D;
>TransposeTest <- [A,B,D,xpose] in TestRel;
>pr TransposeTest;
----------------------+-------------+---------------+----------
    A                 | B           | D             | xpose
----------------------+-------------+---------------+----------
    A1                | 1           | 1.1           | 7
    A2                | 2           | 2.1           | 8
    A3                | 3           | 3.1           | 9
    A4                | 4           | 4.1           | 10
    A5                | 5           | 5.1           | 11
    A6                | 6           | 6.1           | 12
----------------------+-------------+---------------+----------
>pr .xpose;
--------------------
  .id    |   Attr
---------+----------
   7     |    A
   7     |    B
   7     |    D
   8     |    A
   8     |    B
   8     |    D
   9     |    A
   9     |    B
   9     |    D
   10    |    A
   10    |    B
   10    |    D
   11    |    A
   11    |    B
   11    |    D
   12    |    A
   12    |    B
   12    |    D
```

Figure 2.11: A nested relation "transposed" on a subset of attributes

relation and projecting a single attribute, in this case *Attr* which is of type Attribute. The transposed attributes are A, B and D.

```
>domain Attr attribute;
>let xpose be red ujoin of [Attr] transpose A,B,C,D;
>TransposeTest <- [A,B,C,D,xpose] in TestRel;
>pr TransposeTest;
+----------------------+-------------+--------+---------------+----------
|    A                 | B           | C      | D             | xpose
+----------------------+-------------+--------+---------------+----------
|    A1                | 1           | true   | 1.1           | 7
|    A2                | 2           | false  | 2.1           | 7
|    A3                | 3           | true   | 3.1           | 7
|    A4                | 4           | false  | 4.1           | 7
|    A5                | 5           | true   | 5.1           | 7
|    A6                | 6           | false  | 6.1           | 7

>pr .xpose;
+------------------
|  .id    |   Attr
+---------+--------
|   7     |    A
|   7     |    B
|   7     |    C
|   7     |    D
```

Figure 2.12: Attributes of the "TestRel" relation obtained using Transpose operator

The example of Figure 2.12 shows how to find all attributes of a given relation using the **Transpose** operator. In this case, the values of the nested domain *xpose* are all the attributes of the "TestRel" relation.

2.6 Selection Based on Attribute of Type Universal

The Selection operation creates a new relation by limiting the set of tuples of a given relation according to a specific criterion or a "selection condition". This selection condition determines, for each given tuple of the relation, whether that tuple is qualified to be selected. The condition for selection can take form of any logical expression that can be evaluated to `true` or `false`.

In comparison to other data types in jRelix, the Universal data type is capable of holding any type of data: strings, integers, reals, etc... In case of a selection operation based on attributes of type Universal, the operands of the logical expression must be at least of the same data type in order to be able to satisfy the selection criterion. If the data types of the operands do not agree, the logical expression automatically returns `false`. The general syntax of the Selection operation is as follows:

where *selection_condition* **in** *relation*

The query in the example of Figure 2.13 selects all tuples where the data value of the attribute *val* is bigger than 1.1, a constant value of type `float`. The relation *.xpose* used is the same as that shown in Figure 2.10. Even though some of the data values of the attribute *val* which are of type `integer` may satisfy the selection criterion, the corresponding tuples are not selected because the data type `integer` does not agree with the data type of the constant 1.1 which is of type float. As a result, only those tuples whose value data type of the attribute *val* is of type `float` and satisfy the selection condition are selected.

The example of Figure 2.14 shows how to select tuples where data values of an attribute of type Universal in the resulting relation are of different data types. In this example, all tuples where the data value of the attribute *val* is bigger than 1.1 (a constant value of type

```
>UnivTest<- where val > 1.1 in .xpose;

>pr UnivTest;

+----------------------+----------------------+----------------------+
| .id                  | Attr                 | val                  |
+----------------------+----------------------+----------------------+
| 8                    | D                    | float;2.1            |
| 9                    | D                    | float;3.1            |
| 10                   | D                    | float;4.1            |
| 11                   | D                    | float;5.1            |
| 12                   | D                    | float;6.1            |
```

Figure 2.13: Selection based on attribute of type Universal
(relation *.xpose* is shown in Figure 2.10)

```
>UnivTest<- where val > 1.1 or val = ‘‘A1’’ in .xpose;

>pr UnivTest;

+----------------------+----------------------+----------------------+
| .id                  | Attr                 | val                  |
+----------------------+----------------------+----------------------+
| 7                    | A                    | string;A1            |
| 8                    | D                    | float;2.1            |
| 9                    | D                    | float;3.1            |
| 10                   | D                    | float;4.1            |
| 11                   | D                    | float;5.1            |
| 12                   | D                    | float;6.1            |
```

Figure 2.14: Selection based on attribute of type Universal with multiple data types
(relation *.xpose* is shown in Figure 2.10)

`float`) or where data value of the attribute *val* is equal to "A1" (a constant value of type `string`) are selected.

2.7 *Eval* Operator

The **Eval** operator applies to a unary relation the attribute of which is of type Attribute and replaces the variables by their value. The operand relation can have more than one tuple, but all its variables must be of the same type as the value that replaces them. Otherwise, an exception of type-incompatibility is raised. **Eval** operator is part of the domain algebra. However it must be statically evaluated during its declaration. The general syntax of the **Eval** operator is as follows:

let eval *relation* **be** *constant*;

where *relation* is a unary relation the attribute of which is of type Attribute. The example of Figure 2.15 shows the effect of evaluating the *AttribsTraining* relation of Figure 2.7, which contains all attributes of *Training* relation of Figure 2.1. The relation *TestRel* used in the example is that of Figure 2.2.

Every value of *AttribsTraining* relation becomes a virtual attribute. The **sd** command can be used to see the correspoding expression tree stored in the domain table. The value of both newly actualized virtual attributes *Windy* and *Class* is a constant string "ANY" defined in the **Eval** expression by *constant*. In other words, the **Eval** operator replaces the values of attributes *Windy* and *Class* with a constant value "ANY". It has a similar effect as declaring:

let Windy be "ANY";

let Class be "ANY";

```
>let eval AttribsTraining be ``ANY'';
>EvalAttribsTraining <- [A,B,C,D, Windy, Class] in TestRel;
>pr EvalAttribsTraining;
--------+----+--------+------+----------+------------------
   A    | B  | C      | D    | Windy    | Class
--------+----+--------+------+----------+------------------
   A1   | 1  | true   | 1.1  | ANY      | ANY
   A2   | 2  | false  | 2.1  | ANY      | ANY
   A3   | 3  | true   | 3.1  | ANY      | ANY
   A4   | 4  | false  | 4.1  | ANY      | ANY
   A5   | 5  | true   | 5.1  | ANY      | ANY
   A6   | 6  | false  | 6.1  | ANY      | ANY
```

Figure 2.15: Projection of virtual domains resulting from the Eval operator

On the other hand, Figure 2.16 shows the result of the **Eval** operator when the values of the operand are not of the same type. The relation *AttribsTest* used in the example is that of Figure 2.8.

```
>let eval AttribsTest be "ANY";
InterpretError: error: new type of 'B' is 'string',
                but the old type is 'integer'.
```

Figure 2.16: Type-incompatibility exception raised by the Eval operator

2.8 *Quote* Operator

The **Quote** operator converts any attribute name to Attribute metadata. It is an operator of the domain algebra of the jRelix database system. The general syntax of the **Quote** operator is as follows:

let *virt_dom_name* **be quote** *act_dom_name*;

where *virt_dom_name* is the name of a new virtual domain which is of type Attribute. The value of this virtual domain is the name of the actual domain defined by *act_dom_name*.

```
>let accum be quote Windy;
>QuoteWindy <- [Outlook,Humidity, Windy, accum] in Training;
>pr QuoteWindy;
-----------------------+----------------------+---------+-----------
  Outlook              | Humidity             | Windy   | accum
-----------------------+----------------------+---------+-----------
  overcast             | high                 | f       | Windy
  overcast             | high                 | t       | Windy
  overcast             | normal               | f       | Windy
  overcast             | normal               | t       | Windy
  rain                 | high                 | f       | Windy
  rain                 | high                 | t       | Windy
  rain                 | normal               | f       | Windy
  rain                 | normal               | t       | Windy
  sunny                | high                 | f       | Windy
  sunny                | high                 | t       | Windy
  sunny                | normal               | f       | Windy
  sunny                | normal               | t       | Windy
```

Figure 2.17: Projection of a virtual attribute defined by the Quote operator

The example of Figure 2.17 illustrates the result of a **Quote** operator on attribute *Windy*. The virtual attribute *accum* defined by the **Quote** operator is of type Attribute. This can be checked by using the **sd** command. The value of the actualized domain *accum* in the *QuoteWindy* relation is the name of the actual domain of the source relation on which the *accum* was actualized. In this case, the value of *accum* attribute is "Windy", the name of an attribute refered by *act_dom_name* in the **Quote** expression.

2.9 Relational Expression in Projection List

Projection specifies an unordered subset of attributes of a relation and creates a new relation which is defined on this subset of attributes.

The addition of the Attribute data type to the jRelix system allows to have unary relations which are defined on an attribute of type Attribute and which may result from a relational expression.

As a consequence, the relational expressions resulting in this type of unary relations can replace an unordered subset of attributes in projection lists. The new syntax for projection is the following:

[*dom_list, rel_list*, RelationalExpression] **in** *relation*;

where *dom_list* is a sequence of domains, while *rel_list* is a sequence of unary relations defined on an attribute of type Attribute.

In addition, any relational expression is allowed as long as the result is a unary relations defined on an attribute of type Attribute. Figures 2.18 and 2.19 show an example of a projection with a join expression in a projection list.

```
>Rel1 <- AttribsOf ([Humidity] in Training);
>pr Rel1;
+----------------------+
| .attr                |
+----------------------+
| Humidity             |
+----------------------+

>Rel2 <- AttribsOf ([Temperature,Windy] in Training);
>pr Rel2;
+----------------------+
| .attr                |
+----------------------+
| Temperature          |
| Windy                |
+----------------------+

>Rel3 <- AttribsOf ([Windy] in Training);
>pr Rel3;
+----------------------+
| .attr                |
+----------------------+
| Windy                |
```

Figure 2.18: Unary relations

```
>ProjectTraining <- [Outlook, Rel1, Rel2 ujoin Rel3] in Training;

>pr ProjectTraining;

-----------------------+----------------------+--------------+-----------

 Outlook               | Humidity             | Temperature  | Windy

-----------------------+----------------------+--------------+-----------

 overcast              | high                 | hot          | f

 overcast              | high                 | mild         | t

 overcast              | normal               | cool         | t

 overcast              | normal               | hot          | f

 rain                  | high                 | mild         | f

 rain                  | high                 | mild         | t

 rain                  | normal               | cool         | f

 rain                  | normal               | cool         | t

 rain                  | normal               | mild         | f

 sunny                 | high                 | hot          | f

 sunny                 | high                 | hot          | t

 sunny                 | high                 | mild         | f

 sunny                 | normal               | cool         | f

 sunny                 | normal               | mild         | t
```

Figure 2.19: Relational expressions in a projection list

2.10 Relational Expression after the "by" clause of an Equivalence Reduction

Another application where a relational expression, resulting in a unary relation of an attribute of type Attribute, can replace an unordered set of attributes is after the "by" clause of an equivalence reduction. The new syntax for declaring a virtual attribute that performs an equivalence reduction operation is as follows:

let *virt_dom* **be equiv** *operator* **of** *domain* **by** *expr_list*;

where

- *virt_dom* is the name of a new virtual domain declared
- *operator* can be one of:

  a) arithmetic operators such as addition (+) and multiplication (*);

  b) numeric operators such as `max` and `min`;

  c) boolean operators such as `and`, `or`, etc ...;

  d) relational operators such as `ijoin`, `ujoin`, etc ...;

- *domain* is the name of an actual domain the values of which are used in the reduction
- *expr_list* can be one of:

  a) a unary relation defined on an attribute of type Attribute;

  b) any relational expression which results in such a unary relation;

  c) or just a list of domains;

```
>let count be equiv + of 1 by [Attr] in .xpose;

>EquivTest <- [A,B,C,D, count] in TestRel;

>pr EquivTest;

-----------------------+-------------+--------+---------------+-------------
      A                | B           | C      | D             | count
-----------------------+-------------+--------+---------------+-------------
      A1               | 1           | true   | 1.1           | 1
      A2               | 2           | false  | 2.1           | 1
      A3               | 3           | true   | 3.1           | 1
      A4               | 4           | false  | 4.1           | 1
      A5               | 5           | true   | 5.1           | 1
      A6               | 6           | false  | 6.1           | 1
```

Figure 2.20: Projection in an equivalence reduction statement

The example of Figure 2.20 shows how a projection, which results in a relation of a single attribute of type Attribute, can replace an unordered set of attributes following the "by" clause of an equivalence reduction. The relation *.xpose* used in the declaration of virtual domain *count* is the same as that shown in Figure 2.11. The unordered set of attributes which results from the projection is A, B and D. The same result can be obtained by using the following declaration:

```
>let count be equiv + of 1 by A,B,D;
>EquivTest <- [A,B,C,D, count] in TestRel;
```

The example of Figure 2.21 is similar to the previous example except the Projection is replaced by a Selection in an equivalence reduction statement.

Once again, the relation *.xpose* used in the declaration of virtual domain *count* is the same as that of the previous example. The unordered set of attributes which results from

```
>let count be equiv + of D by [Attr] where val < 3 in .xpose;
>EquivTest <- [A,B,C,D, count] in TestRel;
>pr EquivTest;
-----------------------+-------------+--------+---------------+---------------
     A                 | B           | C      | D             | count
-----------------------+-------------+--------+---------------+---------------
     A1                | 1           | true   | 1.1           | 1.1
     A2                | 2           | false  | 2.1           | 2.1
     A3                | 3           | true   | 3.1           | 3.1
     A4                | 4           | false  | 4.1           | 4.1
     A5                | 5           | true   | 5.1           | 5.1
     A6                | 6           | false  | 6.1           | 6.1
```

Figure 2.21: Selection in an equivalence reduction statement

the selection is a single attribute B. The same result can be obtained by using the following declaration:

```
>let count be equiv + of D by B;
>EquivTest <- [A,B,C,D, count] in TestRel;
```

Chapter 3

Implementation of Attribute Metadata

3.1 Development Environment

3.1.1 Java Programming Language

The implementation of jRelix system is done using a Java programming language, more specifically JDK 1.2.2 programming environment.

3.1.2 Other Tools

Other tools that are used in jRelix development are JavaCC and JJTree. JavaCC, a java compiler compiler, is an utility written in Java which automatically generates a parser by compiling a high-level grammar specification stored in a text file and converts it to a Java program that can recognize matches in the grammar. JJTree, on the other hand, is a preprocessor for JavaCC utility that inserts parse tree building actions at various places in the JavaCC source. Please refer to the JavaCC and JJTree documentation in [Jav01] for more details.

3.2 System Overview

Conceptually, the jRelix database system is composed of three functional modules: a *front-end interface*, a *central database engine* and a *system maintainer*. The overall structure of the jRelix system is shown in Figure 3.1. As its name implies, the front-end module serves as an interface between the user and the central database engine. First, it accepts a user command and performs syntax analysis. It then converts these commands into a tree-like structure called a syntax tree, performs error checking and finally calls appropriate functions in the central database engine to execute the necessary operations. On the other hand, the central database engine in jRelix is responsible for implementation of relational and domain algebras as well as computation. The output of the database engine is then passed to the system maintainer which updates the database system appropriately.

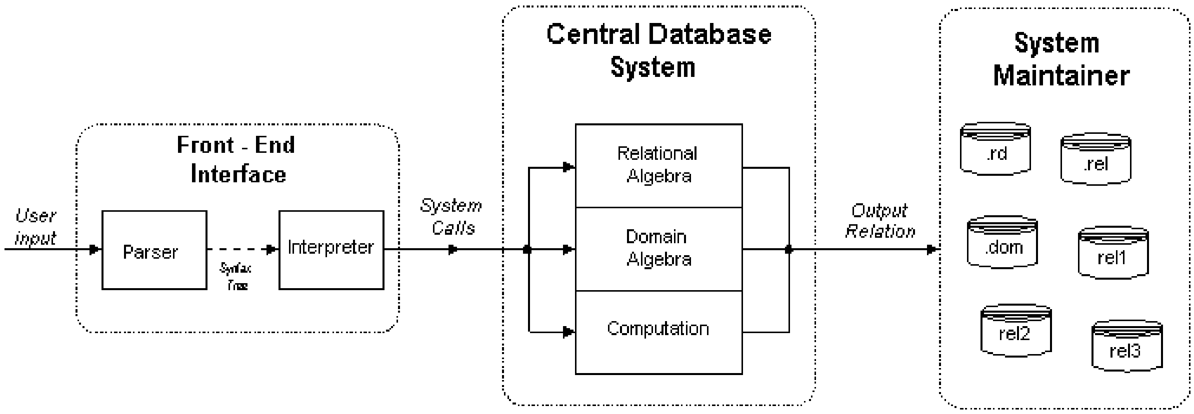

Figure 3.1: jRelix overall structure

The implementation of Attribute metadata was built on an existing system. As a result, the front-end module as well as the central database engine were augmented to support Attribute metadata as well as additional domain and relational algebra operators.

3.2.1 Front-End Interface

As mentioned in the previous section, the front-end module serves as an interface between the user and the central database engine module. It consists of two parts: a *parser* and an *interpreter*. These are discussed in the two sections below.

Parser

The parser is the part of the jRelix system which accepts user command input, analyzes the command syntax and converts this user command into an intermediate code which is understood by the interpreter. This intermediate code has a tree-like structure and is called a command syntax tree. The source code for the parser is located in *Parser.java* file. The parser file is created by using two utilities mentioned in section 3.1.2: JavaCC and JJTree. The procedure for creating the *Parser.java* file is as follows. First, a source file which defines the jRelix grammar and actions for a match of the grammar and which conforms to the JJTree syntax (see JJTree documentation in [Jav01]) is input to the JJTree utility. JJTree in return generates code to construct parse tree nodes. The output of JJTree is then run through JavaCC to create the parser. Figure 3.2 illustrates the procedure of creating a parser for the jRelix system.

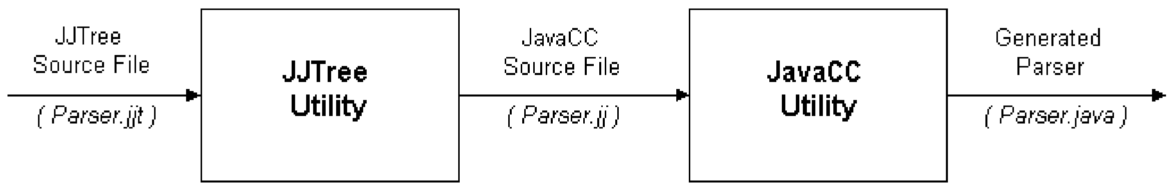

Figure 3.2: Procedure for creating a jRelix parser

Interpreter

The command syntax tree generated by the parser is passed to the interpreter. At this point, the interpreter performs error checking, traverses the syntax tree and converts it into a set of method calls which are then passed to the jRelix database execution engine. The interpreter is implemented in the *Interpreter.java* file.

3.2.2 Central Database Engine

As described previously, the central execution engine of jRelix system implements relational algebra, domain algebra and computation. The engine consists mainly of three java files: *Relation.java*, *Actualizer.java* and *Computation.java*. The Relation class implements the "relation" data structure, a primitive in unit of data in jRelix, as well as the relational unary and binary operators that operate on entire relations. These operators include assignment, projection, selection and different joins. For more details please refer to [Hao98]. The class Actualizer, on the other hand, implements the domain algebra in jRelix database system. In particular, it implements horizontal and vertical operations on flat as well as nested relations. For more details see section 4.5 in [Yua98]. The computation class extends the Relation data structure and implements computation for each tuple of a relation. Please refer to [Bak98] for more details.

3.2.3 System Maintainer

Every jRelix database maintains file system files which store information about relations, attributes, views and computations that are defined in a database. These five system files are ".rel", ".dom", ".rd", ".expr", and ".comp". System files ".rel" and ".dom" store information about all relations and domains that are defined in the database. System file ".rd", on the other hand, maintains all information that links relations with the domains on which they are defined. In addition, file ".expr" stores the syntax trees for virtual domains and views, while file ".comp" stores syntax trees for computations. When a new relation, domain, computation or a virtual attribute is added to the database, these system files must be updated correspondingly. For a more detailed treatment see section 4.4 in [Yua98].

3.3 JRelix Functional Map

As mentioned in section 3.2, the implementation of Attribute metadata was built on an existing system. That is, the classes that implement the front-end module as well as the central database engine were augmented to support Attribute metadata and its corresponding domain and relational algebra operators. Even though no new classes were created in the implementation of Attribute metadata, a significant number of new or existing methods were created or modified respectively. Figure 3.3 presents a map of all the classes of the jRelix system while Figure 3.4 shows all the classes and functions that were created or modified.

3.3.1 SimpleNode Class

One of the classes used extensively in the implementation of Attribute metadata and of its operators is the SimpleNode class. This class is used to represent a node of an expression syntax tree produced by the parser. In addition to the class methods mentioned in Figure 3.4, each node of the expression syntax tree carries the following information:

- *Identifier* identifies the type of node that it represents
- *Info* stores the type and value of a constant in case the node represents a constant (null otherwise)
- *Type* stores the type of operation of the node
- *Opcode* stores the operation code of a given operation
- *Name* stores the name of a variable in case a node represents a variable (null otherwise)

Class Name	Description	Reference
JRelix	Main Program Files	[Yua98]
JRelixInputStream		
JRelixParser		
Relation	Relational Algebra Classes	
RelTable		
RelInfo		
Domain	Domain Algebra Classes	
DomTable		
DomInfo		
Actualizer		
Environment	jRelix System Environment	
Global	Global Attributes/Methods	
Parser	Parser Grammar Classes	[Jav01]
ParserTokenManager		
ParserConstants		
ParserTreeConstants		
JJParserCalls		
JJTParserState		
ASCII_CharStream		
Token		
TokenMgrError		
Node	Syntax Tree Structure	
SimpleNode		
BlockInputStream	Disk I/O Classes	[Hao98]
BlockOutputStream		
SemanticChecker	Semantic Check of User Input	
SemanticCheckError		
Constants	Operation Constants	
TypeConstants	Data Type Constants	
Surrogate	Surrogate Values	
Interpreter	Interpreter Classes	
InterpretError		
PrettyPrint	Formats Display Output	
Utility	Data Type Utilities	
NREnvironment	Nested Relations Environment	
NRInfo		
Computation	Computation Classes	[Bak98]
CompBlock		
CompTable		
IDInfo		
LocalInfo		
ParamInfo		
StateInfo		
EvalExpr	Evaluation of Expressions	
ExprEntry		
ExprTable		
Gedit*	GIS Editor	[Che01]
AddLayersDialog		
CvDraw		
Legend		
TriggerNode	Active Database Classes	[Sun00]
CTrigger		

Note: Gedit* are all the classes the name of which begin with ''Gedit''

Figure 3.3: Map of jRelix classes

Class Name	Method Name	Status
SimpleNode	jjtAddChild	Untouched
	jjtCreateNode	Untouched
	jjtGetChild	Untouched
	jjtGetNumChildren	Untouched
	jjtRemoveChild	Untouched
	jjtReplace	Untouched
	jjtSetParent	Untouched
Constants		Modified
TypeConstants		Modified
Domain	Domain	Untouched
Relation	AttribsOf	New
	Relation	Untouched
	createDataArray	Untouched
	append	Untouched
	duplicateLoadAll	Untouched
	project	Untouched
Actualizer	Actualizer	Modified (shell)
	Actualizer_old	Original ‘‘Actualizer’’
	actBoolCell	Modified (shell)
	actBoolCell_old	Original ‘‘actBoolCell’’
	IsInTree	New
Interpreter	ExpressionListToDomains	New
	AddVirtDom2DomTable	New
	AttribsRel2DomList	New
	traverseType	Modified
	executeDeclaration	Modified
	evaluateTLexpression	Untouched
	evaluateProject	Untouched
	nextTempName	Untouched
Environment	put	Untouched
	lookupRel	Untouched
	lookupDom	Untouched
Surrogate	next	Untouched
Parser		Modified
Utility	isNumeric	Modified
	isText	Modified
	type2String	Modified
	typeConvert	Modified
PrettyPrint	PrettyPrint	Modified

Figure 3.4: Classes and functions that were modified and are referenced in this chapter

3.4 Implementation of Attribute Metadata

3.4.1 Attribute and Universal Data Types

Two new data types were added to the jRelix system. The Attribute data type is used to store the name of an attribute while the Universal data type is capable of holding any conventional (i.e. non-Attribute) type of data. The Attribute data type was implemented as an extension of the String data type defined in jRelix. The Attribute data type supports AttribsOf, Transpose, Eval and Quote operators. On the other hand, the Universal data type supports all operations defined on numeric as well as text data types.

Like all other data types in jRelix, the Attribute and Universal data types are referenced via constants defined in the file *TypeConstants.java*. In addition, every operation in jRelix has a corresponding operation code or "opcode" constant assigned to it. These constants are defined in the *Constants.java* file. The opcodes corresponding to Attribute and Universal data types are OP_ATTRIBUTE and OP_UNIVERSAL respectively. The type constants assigned to Attribute and Universal types are ATTRIBUTE = 18 and UNIVERSAL = 25. These constant values were chosen such that they would not conflict with any other data type and such that OP_x − OP_BOOLEAN = x. This is summarized in the figure below.

```
     Name       |  Value  | Class of definition
----------------+---------+----------------------
 ATTRIBUTE      |    18   |  TypeConstants.java
 UNIVERSAL      |    25   |  TypeConstants.java
 OP_ATTRIBUTE   |   469   |  Constants.java
 OP_UNIVERSAL   |   476   |  Constants.java
 OP_BOOLEAN     |   451   |  Constants.java
```

Figure 3.5: Definition of Attribute and Universal data types

The values of both data types are represented and stored as strings, although their format differs somewhat. In comparison with the data values of type Attribute, which can be stored as simple strings, the Universal data type is capable of holding any type of data. Thus, to store a data value of type Universal, it is necessary to store its type together with its value. As a result, the data value of type Universal is stored as a concatenation of two strings: a string representing the type of data and the data value itself, cast to a string. The strings are separated by a semicolon. For example, if ‘a’ is of type Universal and its value is 1.1 (`float`) then it will be stored as the string “float;1.1”.

3.4.2 Implementation of *AttribsOf* Operator

AttribsOf is a jRelix operator which creates a relation of all the attributes of the operand. As with all other jRelix commands, the interpreter traverses the syntax tree, which corresponds to the AttribsOf command and is generated by the parser, and calls out the *AttribsOf* method of the database execution engine. (See *evaluateTLexpression* function in *Interpreter.java* file.)

The operand of the AttribsOf operator is a relation. As a result, AttribsOf is a unary operator and is part of the relational algebra of the jRelix database execution engine. The implementation of the *AttribsOf* method can be found in *Relation.java* file.

The *AttribsOf* method accepts one parameter of type String (the name of the output relation) and generates a new unary relation, with an attribute of type Attribute, of all the attributes of the operand relation.

The attribute of the output relation is created by using the *Domain* constructor. The new domain is of type ATTRIBUTE and it is named “Attr” by default. The output relation is created by using the *Relation* constructor. It is a relation defined on a single attribute, created in the previous step, and at this point does not contain any tuples. It is of size zero.

```
Relation AttribsOf(String OutputRelationName)
{
    1. Create a new domain of type Attribute called ``Attr''
    2. Create a new unary relation whose attribute is obtained in step 1.
    3. Get all attributes of the operand relation and store these
       in a data array
    4. Append the data array of step 3. to the output relation
       created in step 2.
    5. Return this output relation
}
```

Figure 3.6: AttribsOf Algorithm

In order to append data to this new relation, first an array of objects, that will store the data of the output relation, is created using the *createDataArray* function of the *Relation* class passing the size specification as its parameters. The size is specified by the size of the domain array of the relation (in this case the size of the domain array is 1) and by the number of rows. In this case the number of rows correspond to the number of attributes of the operand relation. Then, the names of all the attributes of the operand relation are extracted, by accessing the name field of the domains structure of this relation, and are stored in the data array as strings.

Finally, the resulting data array from the previous step is appended to the new relation by calling the *append* method of the *Relation* class.

3.4.3 Implementation of *Transpose* Operator

On the other hand, the AttribsOf operator can be replaced by a more general Transpose operator. Given some flat relation, the Transpose operator creates a nested relation of two attributes. One is of type Attribute and holds the list of attributes of this flat relation.

The other attribute is of type Universal and holds the values for each attribute of the flat relation. In effect, the Transpose operator converts each tuple of an ordinary relation into a set of attribute-value pairs. In addition, the Transpose operator can transpose all or any subset of attributes of a given relation.

Unlike AttribsOf, the Transpose operator is part of domain algebra and therefore is implemented in *Actualizer.java* file.

When a new virtual domain is declared, a function *traverseType* of the *Interpreter* class is called. This function traverses the expression tree of the domain algebra statement and assigns a data type to this new virtual domain based on the data type of the result of the domain algebra expression. The *traverseType* function was modified to include the Transpose operator. Since the result of the Transpose operation is a nested relation, then the data type assigned to the new virtual domain is of type IDLIST.

As mentioned previously, the implementation of the Transpose operator should be included in the *Actualizer* class. This would require significant modifications to the existing implementation of the Actualizer. One of the objectives of the implementation of Transpose and of other operators was to add more functionality to jRelix with as little modification as possible to the existing system.

In order to minimize changes to the existing Actualizer, the domain algebra expression tree of the Transpose expression is preprocessed and converted to a corresponding expression tree that the existing Actualizer understands and is able to process. This preprocessing step is implemented in the Actualizer "shell", a wrapper around the original Actualizer. The Actualizer "shell" is implemented in the *Actualizer* constructor while the original Actualizer function is renamed to *Actualizer_old*.

The Actualizer "shell" accepts the same number and type of parameters as the old Actualizer constructor and is thus transparent to the rest of the jRelix system. The *Actualizer* constructor serves as a preprocessor for the Transpose and Quote expressions. In addition, it preprocesses all equivalence reduction statements where the "by" clause is followed by a relational expression. The Actualizer "shell" accepts the following parameters: a list of domains to actualize, a relation in the context of which to actualize the domains, and the environment of the database.

```
Actualizer(Domain[] DomainsArray, Relation SourceRelation, Environment env) {
   For all virtual domains in the DomainsArray do
   {
       if (DomainsArray[i].tree == ``Transpose'' expression)
           DomainsArray[i].tree = ``Transpose'' type Preprocessing)

       else if (DomainsArray[i].tree == ``Quote'' expression)
           DomainsArray[i].tree = ``Quote'' type Preprocessing)

       else if (DomainsArray[i].tree == Relational expression in Equiv red.)
           DomainsArray[i].tree = ``Equiv'' type Preprocessing)
   }
  Actualizer_old(DomainsArray, SourceRelation, env)
}
```

Figure 3.7: Actualizer "shell" Algorithm

The "shell" traverses the domain list in a loop and checks whether a given domain is virtual or not. The distinction is made by checking the tree field of the domain structure where the expression tree of a virtual domain is stored. If the value of the "tree" field is null then the domain is not virtual and nothing is done. In order to determine whether an expression tree of a given virtual domain contains "Transpose", "Quote" or an equivalence reduction expressions the function *IsInTree* of the *Actualizer* class is called .

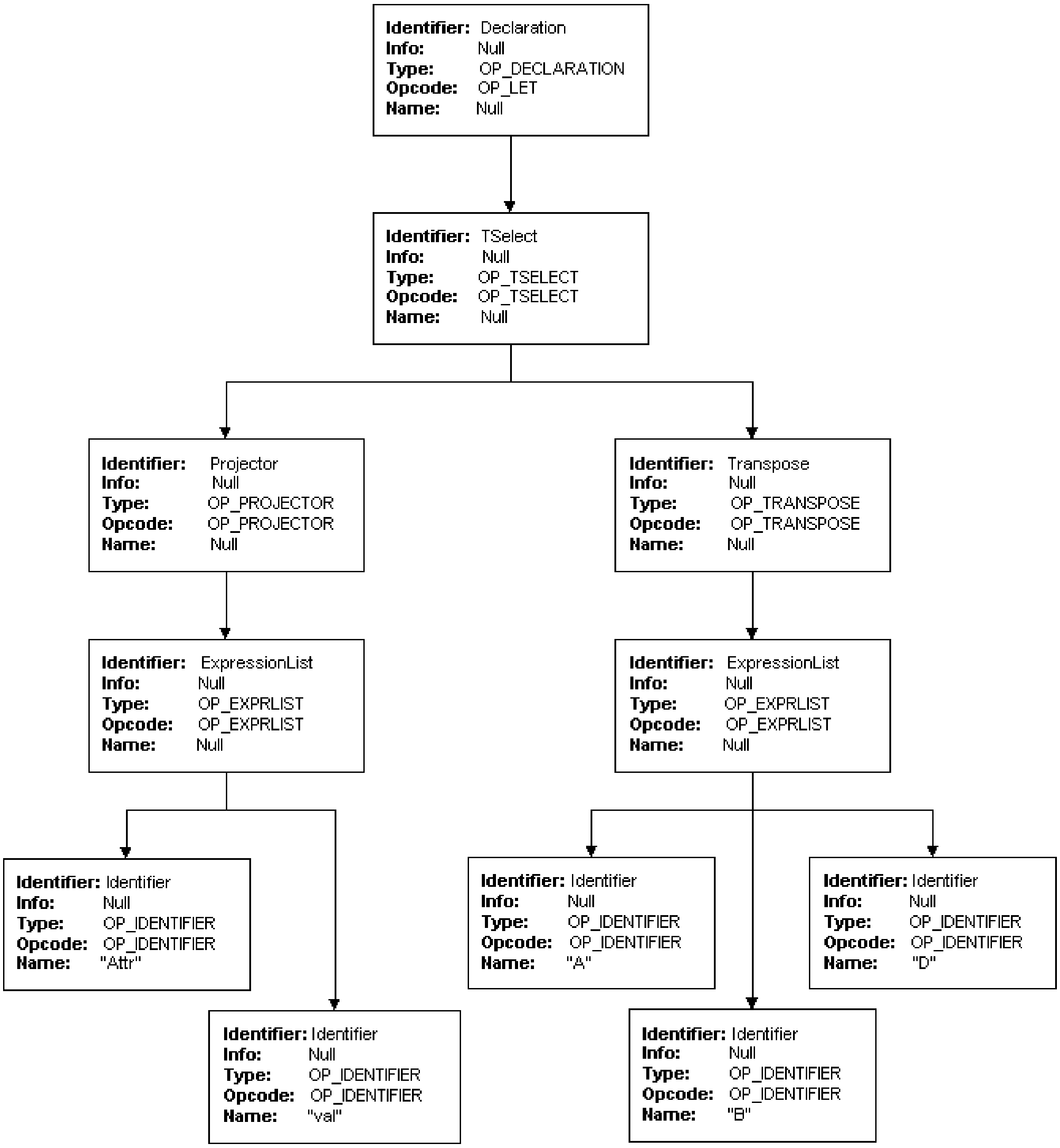

Figure 3.8: A "Transpose" expression tree which corresponds to the following declaration: *let xpose be [Attr,val] transpose A,B,D*

It returns a node corresponding to the operator if it is found in the expression tree and null otherwise. In case the expression tree of a given virtual domain contains "Transpose", "Quote" or an equivalence reduction expressions, the "shell" preprocesses (modifies) the expression trees to the syntax and format that the original Actualizer is able to process. Finally, the *Actualizer_old* function is called.

The first step in the implementation of the Transpose operator is to project the subset of attributes, on which to apply the transpose operator, from the source relation.

A	B	C	D
A1	1	true	1.1
A2	2	false	1.2
A3	3	true	1.3
A4	4	false	1.4
A5	5	true	1.5
A6	6	false	1.6

Figure 3.9: "Source" relation from the example of Figure 2.11:
TransposeTest $\leftarrow$ *[A,B,D,xpose] in TestRel*

This is done by using the "Transpose" node returned by the *IsInTree* function. The subtree of the "Transpose" node stores a subset of attributes to transpose. The function *jjtGetChild* defined in class *SimpleNode* extracts this subtree which is then passed to the *ExpressionListToDomains* function of the *Interpreter* class as its parameter. The latter function converts this expression subtree of attributes to transpose to a corresponding array of domains.

The source relation is then copied to a new relation with all data loaded into it. This is done using the *duplicateLoadAll* function of the *Relation* class. The projection on the

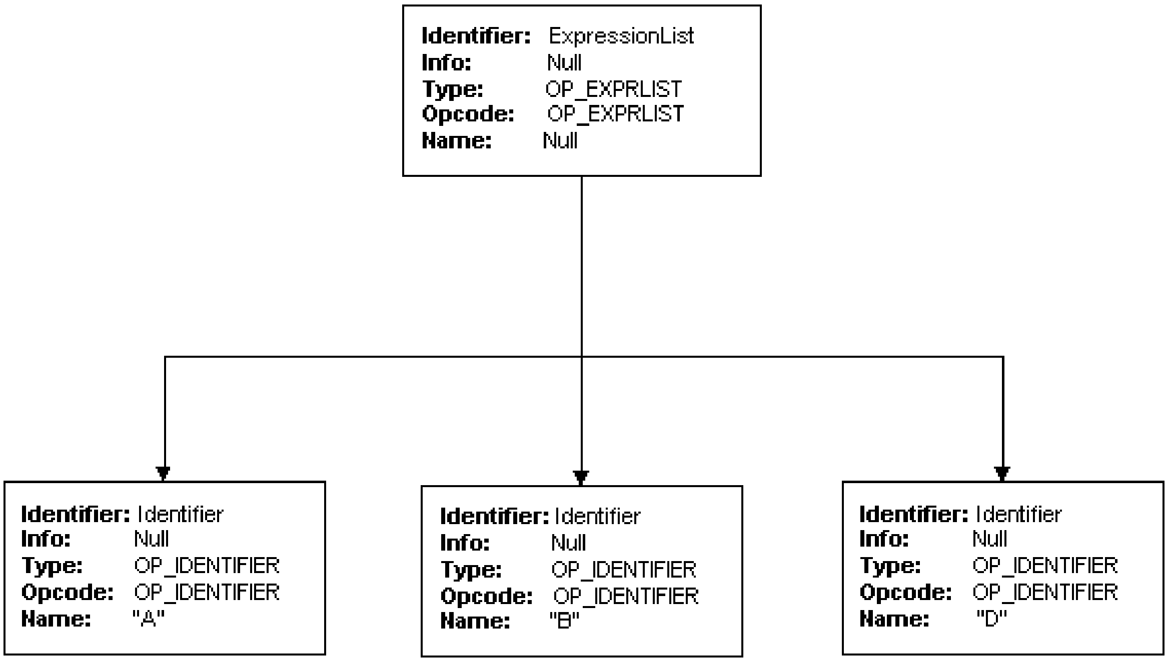

Figure 3.10: Subtree of the expression tree of Figure 3.8 which stores subset of attributes to transpose from the source relation

A	B	D
A1	1	1.1
A2	2	1.2
A3	3	1.3
A4	4	1.4
A5	5	1.5
A6	6	1.6

Figure 3.11: Projection of the "source" relation on which Transpose operator is applied (*tempName1* $\leftarrow$ *[A,B,D] in TestRel*)

subset of domains to transpose is obtained by calling the function *project* of the *Relation* class and passing an array of attributes to project as one of its parameters. The second parameter is used to specify the name of the output relation. It is generated by the jRelix system using the *nextTempName* function of the *Interpreter* class and is added to the "tempnames" vector for maintenance purposes: during the clean-up phase of the execution, all relations (files) whose name is stored in this vector will be removed (deleted) from the database.

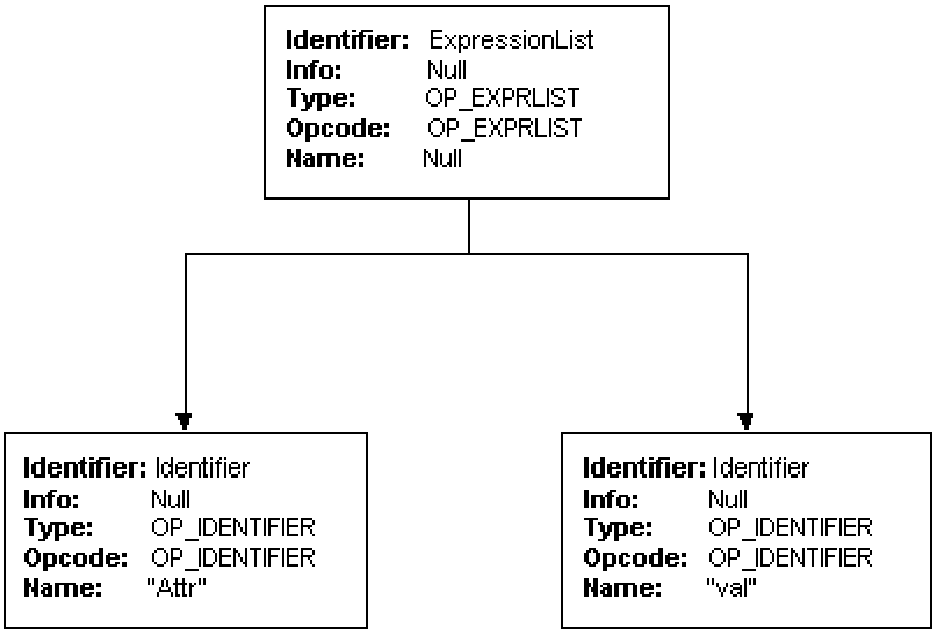

Figure 3.12: Subtree of the expression tree of Figure 3.8 which stores subset of attributes to project from the "transposed" relation

The second step in the implementation is to get the list of domains (attribute and/or value) that the resulting nested relation will have. Once again, the subtree of the syntax tree that contains the list of attributes to project from the "transposed" relation is passed to the *ExpressionListToDomains* function and the resulting set of domains is stored in the "domList" array.

The next step is to modify the syntax tree stored in the tree field of the virtual domain so that the original Actualizer will process it as was intended by the "Transpose" expression. The modification of the syntax tree is as follows.

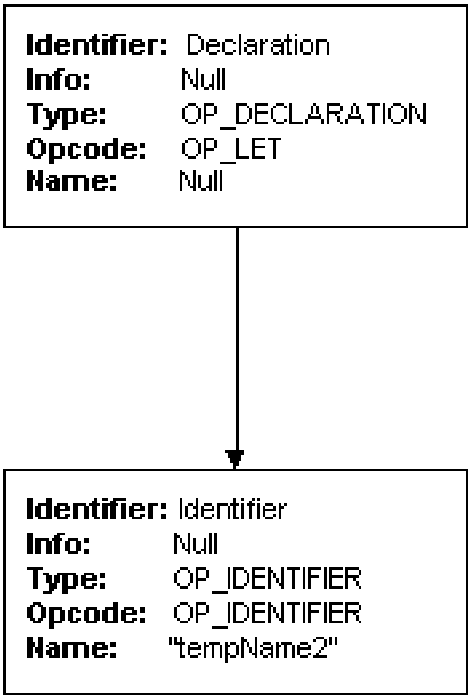

Figure 3.13: Modified "Transpose" expression tree

(Equivalent to : *let xpose be tempName2*)

The "transpose" part of the overall expression (this includes attributes on which to transpose a relation and attributes to project from the resulting nested relation) is replaced by a single identifier which identifies the nested relation resulting from the transpose operation. The name of the identifier is generated using the *nextTempName* function. The corresponding node is created using the *jjtCreateNode* function of the *SimpleNode* class and is added to the syntax tree using the *jjtReplace* function. The newly created node replaces the subtree containing the "transpose" part of the overall expression. The new node is a leaf of the syntax tree. In other words, this new identifier, which represents the

.id	Attr	val
7	A	string;A1
7	B	integer;1
7	D	float;1.1
8	A	string;A2
8	B	integer;2
8	D	float;2.1
9	A	string;A3
9	B	integer;3
9	D	float;3.1
10	A	string;A4
10	B	integer;4
10	D	float;4.1
11	A	string;A5
11	B	integer;5
11	D	float;5.1
12	A	string;A6
12	B	integer;6
12	D	float;6.1

Figure 3.14: .tempName2: “Transposed” nested relation

nested relation resulting from a Transpose operation, will define the virtual attribute.

Since a virtual attribute must be defined on something that already exists, the new identifier must be registered in the system as a domain of type `IDLIST`, a nested relation. First, the domain corresponding to the new identifier is created. It is of type `IDLIST` and is defined on attributes determined in step two of the implementation. It is then registered in the environment of the database using the *put* function of the *Environment* class.

In addition, a virtual domain must be defined on actual domains of a source relation. That is, the newly created domain that references the “transposed” nested relation and on which the virtual domain is defined must be an actual domain of source relation. The final step of the implementation is the generation of data for the “transposed” nested relation and the addition of this newly created domain to the source relation which is to be transposed.

```
   A    |   B    |  C    |   D  | tempName2
--------+--------+-------+------+-----------
   A1   |   1    | true  |  1.1 |     7
   A2   |   2    | false |  1.2 |     8
   A3   |   3    | true  |  1.3 |     9
   A4   |   4    | false |  1.4 |    10
   A5   |   5    | true  |  1.5 |    11
   A6   |   6    | false |  1.6 |    12
```

Figure 3.15: New source relation that is passed to the original Actualizer

The resulting “augmented” relation will become the new source relation and will be passed to the original Actualizer for further processing. The name of this new source relation is generated once again by using *nextTempName* function. It will have the same attributes as the original relation to transpose plus the attribute which references the "transposed" nested relation. The new source relation is created using the *Relation* constructor and is of size zero. Its data will be temporarily stored in a two dimensional array called "dArray" and will be appended to the relation later on.

The format of this relation corresponds to the nested relation format. See [Hao98] for more details. The nested attribute will contain the surrogates or pointers to the actual values of this nested attribute which are stored in a separate relation. The surrogates are generated using the *next* function of the *Surrogate* class. The rest of the attributes of this new source relation have the same data as their counterparts in the original relation that is transposed.

When the nested domain (type IDLIST) was created, the *Domain* constructor created another relation by the same name as the nested domain prefixed with a dot. This relation stores the values of this nested attribute. It has an additional attribute ".id" which is used to link the relation data back to the nested attribute and its relation. This dot relation is referenced using the *lookupRel* function of the *Environment* class. The relation is originally empty and data will be appended to it. The values of the nested attribute are temporarily stored in the two dimensional array called "dataArray" the size of which is defined by the number of tuples multiplied by the number of attributes that are in the relation that must be transposed.

The heart of the implementation of the Transpose operator is a double loop which traverses the relation to be transposed and for each tuple and attribute of this relation generates a new tuple of two or three attributes: one is the ".id" attribute and the other attributes are those determined in step two. The attributes determined in step two are of type Attribute and/or Universal. The attribute of type Attribute stores the names of the attributes of the relation to be transposed. On the other hand, the attribute of type Universal stores the corresponding value and its type. Please refer to section 3.4.1 for more details on how this data is represented and is stored in the jRelix system.

3.4.4 Implementation of Selection Based on Attribute of Type Universal

Selection operation creates a new relation by including those tuples of the source relation which satisfy a "selection condition". This selection condition can be any logical expression which is evaluated to a boolean, `true` or `false`, on any tuple of the source relation. The conditional expression of the selection is represented in jRelix system by a virtual attribute which is constructed on the fly and is actualized on the source relation. The tuples in which the actualized conditional attribute is `true` are selected.

In comparison to other data types in jRelix, the Universal data type is capable of holding any type of data such as integers, reals, strings, booleans, etc ... . As a result, some preprocessing is required so that the original implementation of the selection operation is able to process logical expressions where one of the two operands is of type Universal. This preprocessing is implemented in the *actBoolCell* function of the *Actualizer* class.

```
boolean actBoolCell(SimpleNode SelectionTree)
{
   if left operand of Selection Expression is not of type Universal then
      Process this expression as usual;
   else {
           if the values of the operands of the selection expression are not
           of the same data type then
              return False;
           else { Preprocess the expression tree;
                  Process the new expression tree as usual;}
       }
}
```

Figure 3.16: Selection "shell" Algorithm

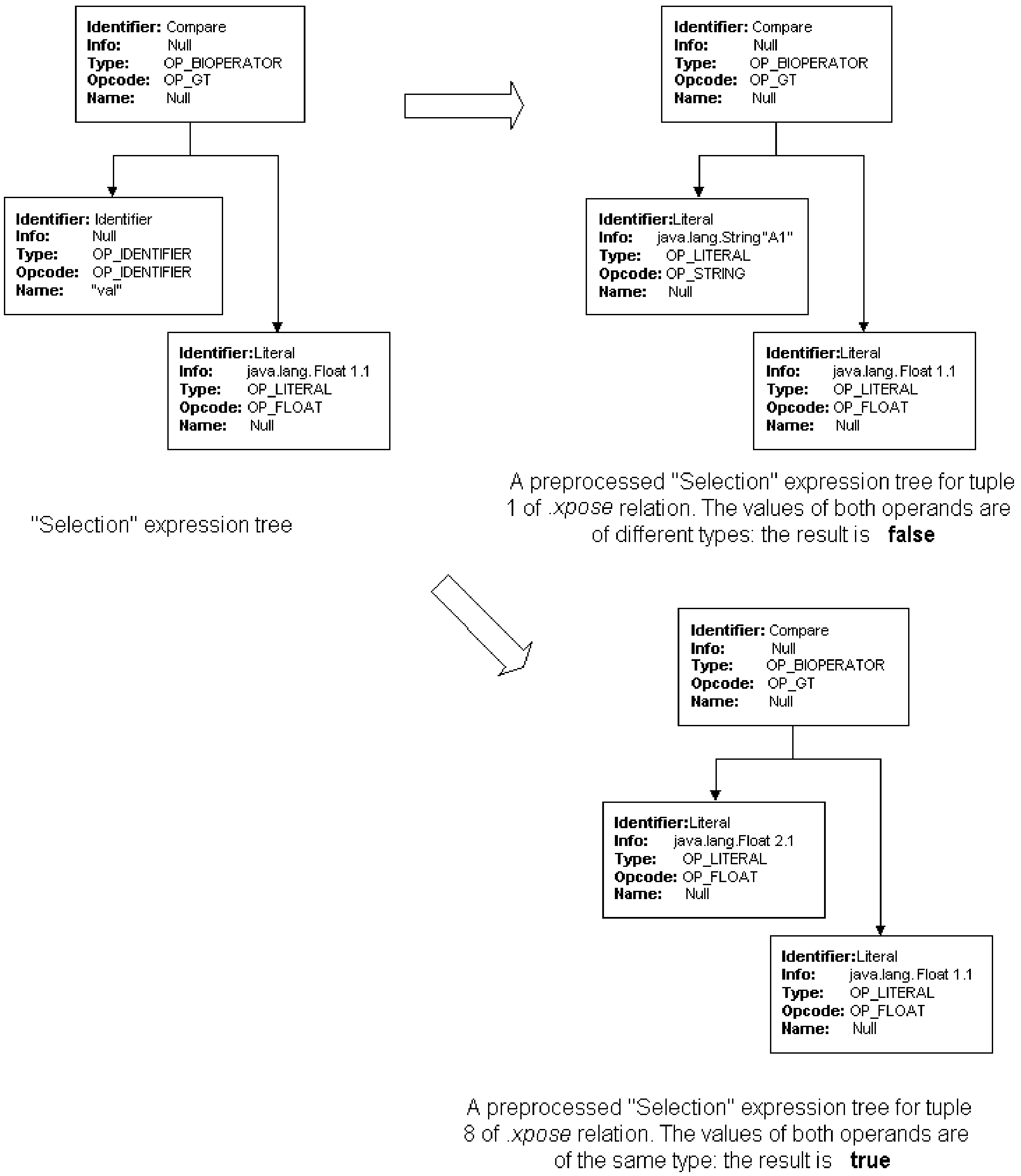

Figure 3.17: An example of a Selection expression tree corresponding to Figure 2.13

The first step of the implementation is to check whether the left operand of the logical expression corresponding to the selection criterion is of type Universal. First, the node corresponding to the left operand is extracted from the expression tree using the *jjtGetChild* function of the *SimpleNode* class. This node is the first child of the root node which is pased to the *actBoolCell* function as its parameter. The type of the left operand is obtained by first retrieving the corresponding Domain structure using the *lookupDom* function with passing the name of the left operand as its parameter, and then reading the value of its "type" field.

If the left operand of the logical expression is not of type Universal, then no preprocessing is required and the execution is passed to the *actBoolCell_old* function which is the original implementation of the Selection operation.

On the other hand, if the left operand is of type Universal, then the corresponding domain can hold data of different types. However, in order for the logical expression to be satisfied the data types of its operands must be the same. The second step of the implementation is to check whether the values of the operands of the logical expression are of the same data type. As suggested in section 3.4.1 in case of an attribute of type Universal, the data type of the value stored is part of the value of the attribute. The retrieving and comparison of the data types of the values of the operands of the logical expression are achieved by using simple string manipulation functions inherent in Java String class. As a result, if the data types of the values of the operands of the logical expression are not of the same type then the selection criterion is not satisfied and the *actBoolCell* function returns `false`.

Otherwise, if the data types of the values of the operands agree then the node of the "selection" expression tree corresponding to the left operand is replaced with a new node which is of type "Literal". The value of this new literal node is the value of the domain of type Universal on which the selection is based, for a current tuple.In fact, from the point of view of the original implementation, the selection criterion becomes a logical expression whose operands are both constants and are of the same data type.

Finally, the new "selection" expression tree is passed to the *actBollCell_old* function for further processing.

3.4.5 Implementation of the *Eval* Operator

The Eval operator applies to a unary relation the attribute of which is of type Attribute and replaces the variables by their value. The operand relation can have more than one tuple; however, all its variables must be of the same type as the value that replaces them. Otherwise, an exception of "type incompatibility" is raised.

The Eval operator is part of the domain algebra and as a result must be implemented in the *Actualizer* class. However, the Eval operator must be evaluated statically during its declaration and as a result its implementation is found in the *executeDeclaration* function of the *Interpreter* class. Only the part of the function that processes the "let"-type declaration is modified.

The first step of the implementation is to determine whether this "let" declaration contains the Eval operator. This is accomplished by extracting the first child of the root of the syntax tree and verifying its "opcode" field. If the "opcode" of the node does not correspond to the Eval operator then the execution is passed to the *AddVirtDom2DomTable* function of the *Interpreter* class. This function processes all domain algebra declarations. On the other hand, if the "opcode" of the node corresponds to that of the Eval operator, then some preprocessing is required.

The second step is to extract the values of the operand relation and to convert these into an array of domains. This is done by using the *ExpressionListToDomains* function passing as its parameter the syntax subtree the root node of which corresponds to the Eval operator.

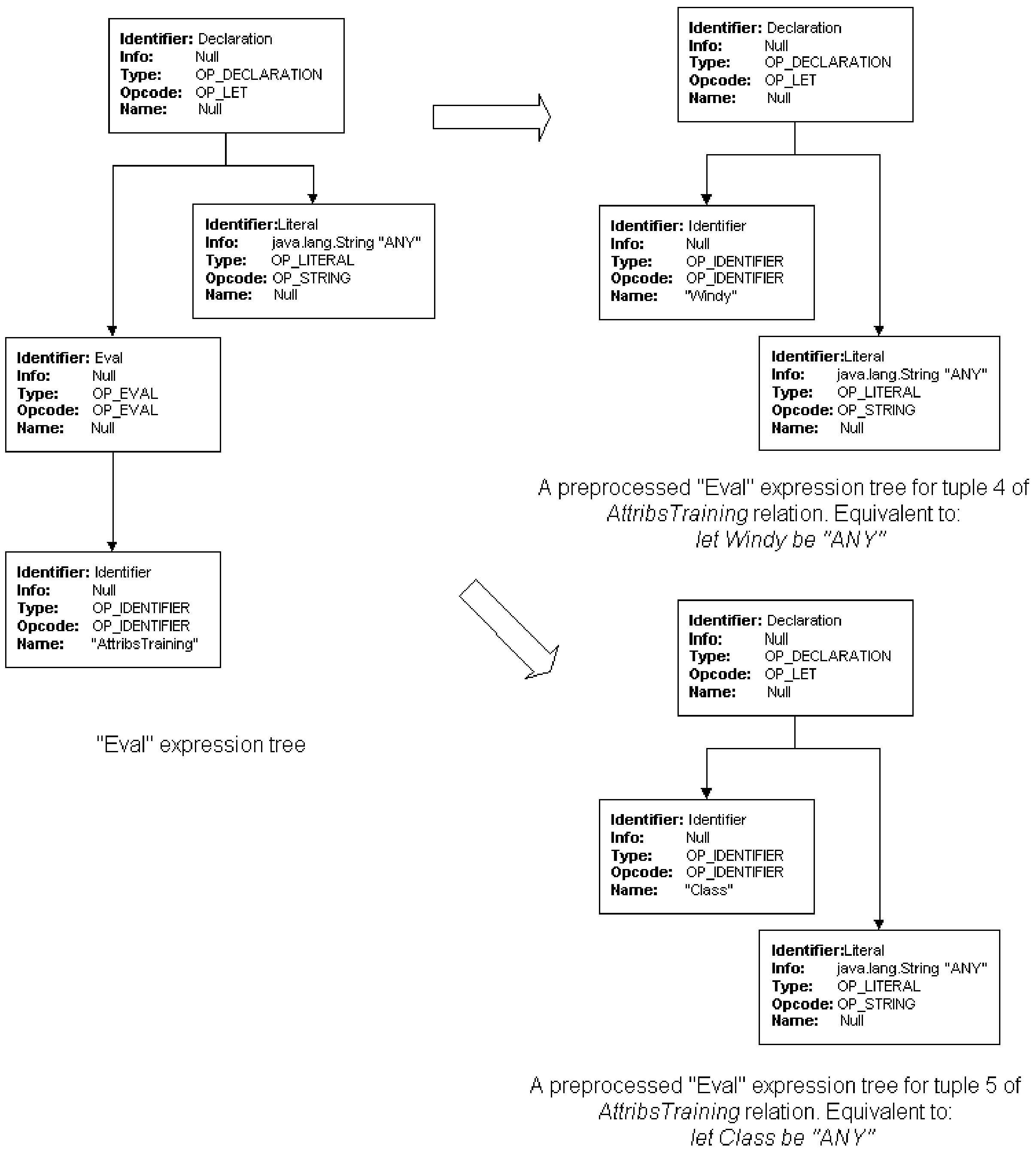

Figure 3.18: An example of an “Eval” expression tree corresponding to Figure 2.15

The final step of the implementation is to create a syntax tree for each extracted variable of the operand relation such that the existing mechanism for processing a domain algebra declaration, implemented in *AddVirtDom2DomTable* function, can be used. First, the subtree corresponding to the Eval operator is removed from the original syntax tree, stored in the "node" variable, using the *jjtRemoveChild* function of the *SimpleNode* class. Then, for each extracted variable, a new node is created which has the same name as the variable and is of type "Identifier". The *jjtCreateNode* function is used for this operation. This new node corresponds to the "virtual domain" node of the syntax tree for a regular "let"-type declaration. The next step is to insert this new node in the original syntax tree in the first child position of the root thus replacing the Eval operator subtree. The last step is to call *AddVirtDom2DomTable* function passing the preprocessed syntax tree as its parameter.

3.4.6 Implementation of the *Quote* Operator

The Quote operator converts any attribute name to Attribute metadata. It is an operator of the domain algebra of the jRelix database system and is implemented in the Actualizer class. The goal in the implementation of the Quote operator was similar to that of the Transpose operator implementation. That is, to implement the Quote operator with as little modifications as possible to the existing code in the Actualizer class. As a result, the Quote operator is implemented in the Actualizer "shell" (*Actualizer* constructor).

Similarly to the Transpose operator case, the *traverseType* function was modified to include the Quote operator. The data type returned by the *traverseType* function for the "Quote" case and which is assigned to the new virtual attribute is type Attribute.

The effect of the Quote operator is very similar to that of assigning a literal of type String to a virtual attribute. The exception is that this virtual attribute must be of type Attribute and must have the name of the operand as its constant value.

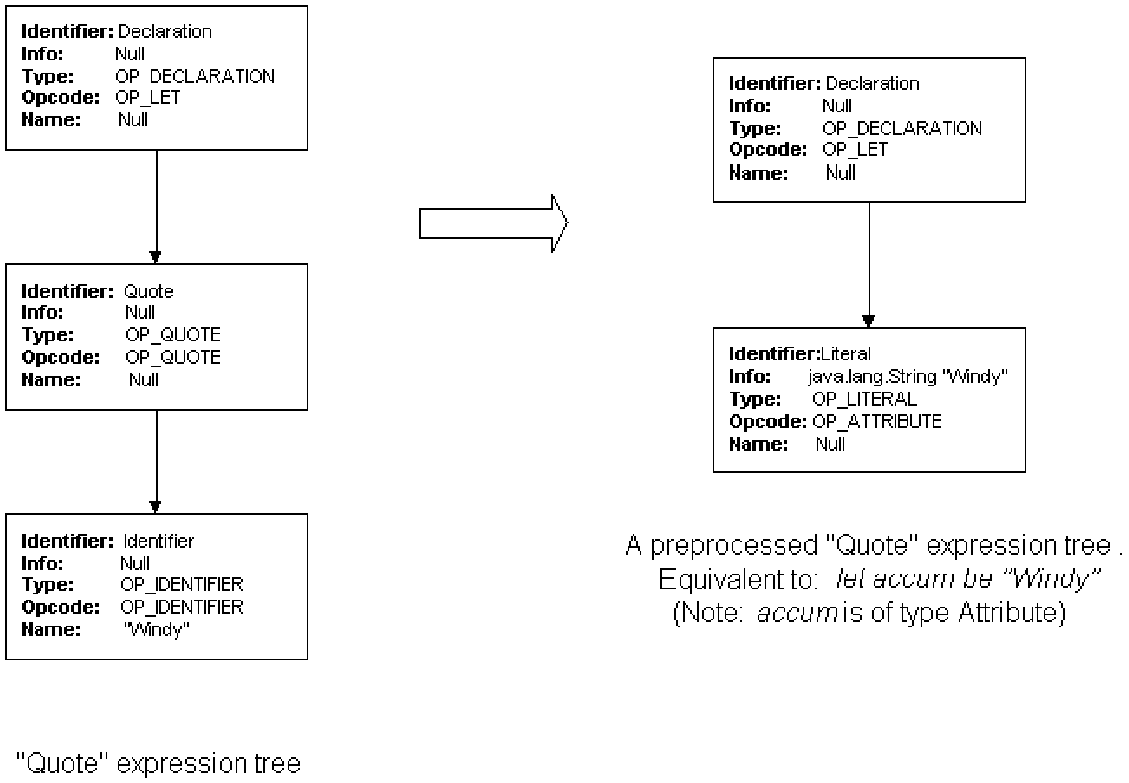

Figure 3.19: An example of an “Quote” expression tree corresponding to Figure 2.17

The Actualizer "shell" will first determine whether the expression tree stored in the "tree" field of the virtual domain is of type "Quote". It suffices to check the node type of the first child of the root in the expression tree. The node is obtained by using the *jjtGetChild* method and type of the node, stored in "type" variable of the *SimpleNode* class is compared with the Quote operation type.

If the two types agree, then the second step is to create an expression tree that is similar to the one resulting from a declaration of a constant attribute which the original Actualizer is able to process correctly. First, a node corresponding to a constant value is created using *jjtCreateNode* function. The node corresponds to a literal (constant) and is of type Attribute. The "info" variable of the node structure stores the literal's value. In this case the value is the name of the operand attribute of the Quote operator. The final step is to replace the subtree, which consists of a "quote" node and of its operand node, with the newly created "literal" node. The function *jjtRemoveChild* removes the "quote" subtree. the new "literal" node is added to the expression tree using the *jjtAddChild* function.

3.4.7 Implementation of Relational Expression in Projection List

The addition of the Attribute data type to the jRelix system allows it to have unary relations which are defined on an attribute of type Attribute and which may result from a relational expression. As a consequence, the relational expressions resulting in this type of unary relations can replace an (unordered) subset of attributes in projection lists.

As was described in section 3.2.1 the interpreter traverses the syntax tree, generated by the parser, and converts it into a set of method calls which are then passed to the jRelix database execution engine. The part of the *Interpreter* class that deals with the processing of the syntax tree corresponding to the projection expression is the *evaluateProject* function. One of the steps it has to perform before calling the *project* function of the *Relation* class, which performs the actual projection, is to convert the part of the syntax tree that represents the projection list to an array of domains to project. This is accomplished by the *ExpressionListToDomains* function which can be found in the *Interpreter* class. This function accepts a syntax subtree whose root node is of type *ExpressionList* and converts it to an array of domains. The description of the implementation of this function is presented below.

The first step is to obtain the number of children of the root node using the *jjtGetNumChildren* function of the *SimpleNode* class. In addition, a vector "domList" is instantiated. Each entry of this vector will hold an array of domains corresponding to each child.

The next step is to traverse each child node and to convert it to an array of domains it represents. The child of the root node is obtained by using the *jjtGetChild* function. This child node can be of either an "Identifier" or a "JoinExpression" type node.

If a child node is of type "Identifier" then it can represent either a domain or a unary relation defined on an attribute of type Attribute. The distinction is made simply by checking whether the name of the "Identifier" node belongs to a relation or to a domain. The name of the "Identifier" node is obtained by accessing the "name" field of the child node. Then the name of the "Identifier" is passed as a parameter to the *lookupRel* function of the *Environment* class to check whether it belongs to a relation. The *lookupRel* function returns a matching relation if it is present in the database or a null value otherwise.

In case a relation is returned, it is passed to *AttribsRel2DomList* function as its parameter. This function, found in the *Interpreter* class, first checks whether its parameter relation is defined on a single attribute of type Attribute. If the answer is yes then the values of this relation are extracted using the *lookupDom* function of the *Environment* class and stored in an array of domains, which is returned by the function. Otherwise, if the unary relation is not defined on an attribute of type Attribute, an exception is thrown. The array of domains returned by the *AttribsRel2DomList* function is stored in the "domList" vector.

If the name of the Identifier does not belong to any relation in the database (*lookupRel* function returns a null value) then the child node might correspond to a domain. The check is performed using the *lookupDom* function of the *Environment* class. If a domain by this name does not exist in the database, then an exception is thrown. Otherwise, the domain returned by the *lookupDom* function is assigned to an array of size one which is then stored in the "domList" vector.

On the other hand, if a child is of type "JoinExpression" then it must correspond to a relational expression whose result must be a unary relation defined on an attribute of type Attribute. First, this relational expression is evaluated using the *evaluateTLexpression* function of the *Interpreter* class. The resulting relation is then passed to the *AttribsRel2DomList* function. The obtained array of domains is added to the "domList" vector.

The final step is to combine all these different domain arrays of the "domList" vector into one big array which will hold all domains of the projection list. First, the size of the resulting array is calculated by summing up the sizes of all the arrays of the "domList" vector. Then the domains stored in the "domList" vector are copied one by one into the resulting array which is returned by the *ExpressionListToDomains* function.

3.4.8 Implementation of Relational Expression after the "by" clause of an equivalence reduction

Another application where a relational expression, resulting in a unary relation of an attribute of type Attribute, can replace an (unordered) subset of attributes is after the "by" clause of an equivalence reduction. Since equivalence reduction is part of the domain algebra, it is implemented in the *Actualizer* class.

However, in order for the Actualizer to be able to process an equivalence reduction with a relational expression after the "by" clause, additional preprocessing is required. This preprocessing is implemented once again in the Actualizer "shell". The goal of the implementation is to convert an expression subtree, corresponding to a relational expression, to a subtree representing an equivalent unordered set of attributes.

The first step is to convert an expression subtree with a root node of type ExpressionList to an array of domains. This is done using the *ExpressionListToDomains* function of the *Interpreter* class. Similarly to the projection list case, each child may represent a domain, a unary relation whose attribute is of type Attribute or a relational expression which results in a relation with a single attribute of type Attribute.

The final step of the implementation is to remove all children of the ExpressionList node and for every domain of the array, returned by the *ExpressionListToDomains* function, create a new node corresponding to that domain and add it to the expression tree as a child of the ExpressionList node. The removal of children of the ExpressionList node is done in a loop using the *jjtRemoveChild* function of the *SimpleNode* class. Each child node is indexed using the loop counter. Each newly created node is of type "Identifier" and has the same name as its corresponding domain. The new node is added to the expression tree using the *jjtAddChild* function. The ExpressionList node is assigned to each added node as its parent using the *jjtSetParent* function.

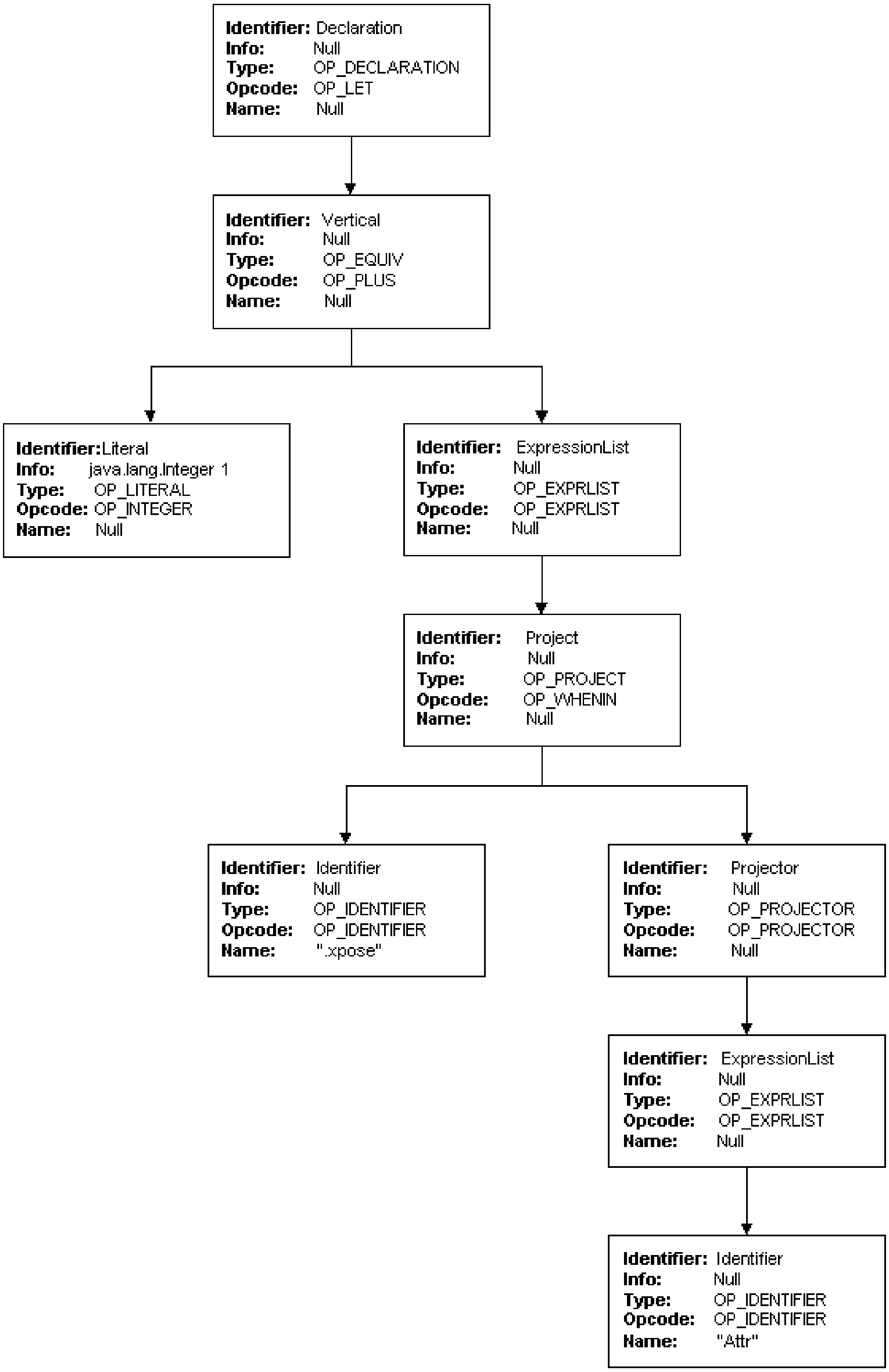

Figure 3.20: An example of an “Equiv” expression tree corresponding to Figure 2.20

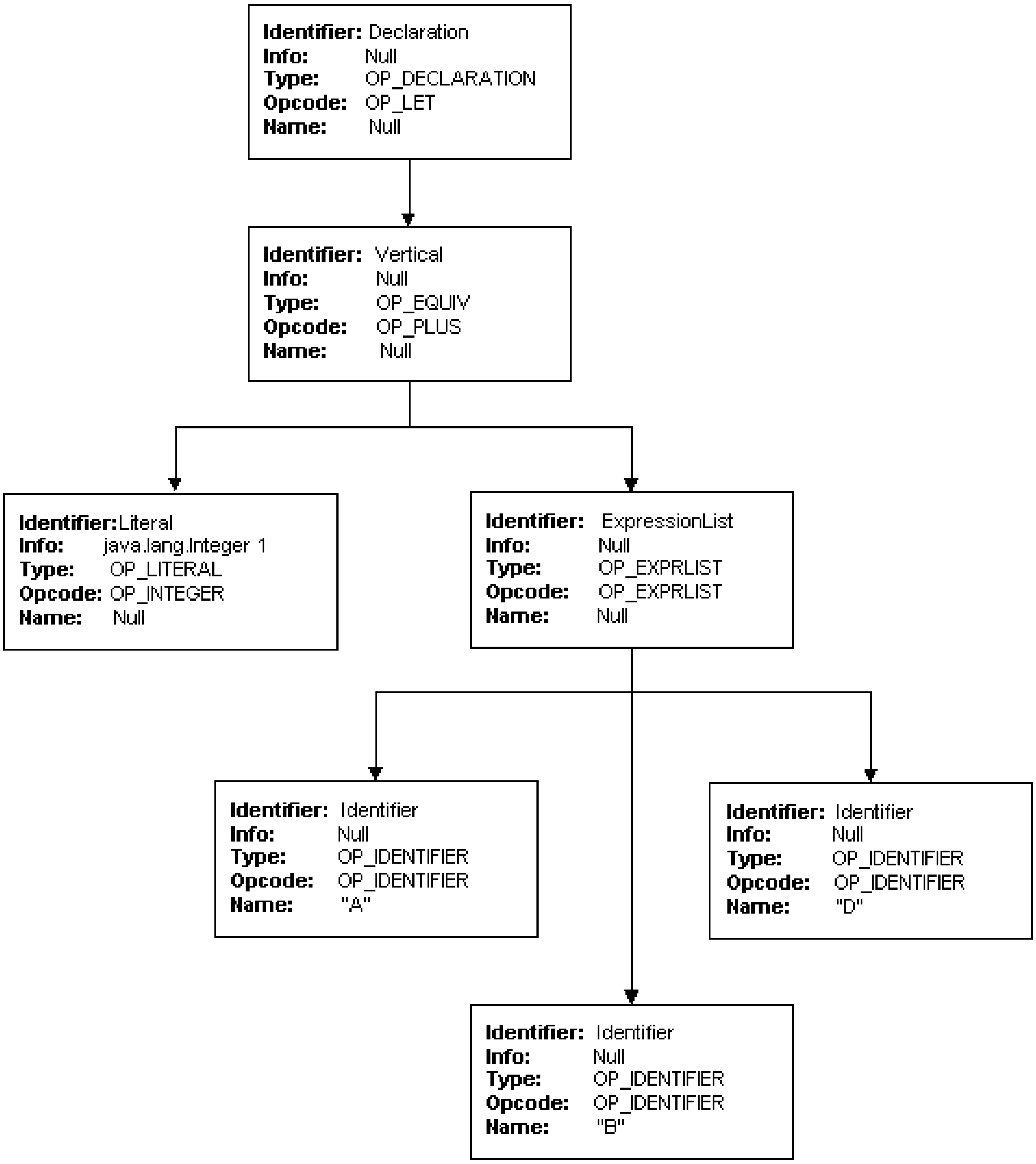

Figure 3.21: A preprocessed "Equiv" expression tree corresponding to Figure 2.20

Chapter 4

Conclusion

The objective of this project was the implementation of Attribute metadata and of its associated operators:

- AttribsOf and Transpose used to create a relation of the set of attributes and their values from an operand relation
- Eval and Quote - special operators for any metadata:

  Eval - replaces any metadata variable by its value

  Quote - converts any attribute name to attribute metadata

In addition, support was added to the jRelix system to allow relational expressions, which result in a unary relation whose attribute is of type Attribute, to be used anywhere where an unordered set of attributes was previously allowed. Furthermore, a new data type, called Universal, was added to the jRelix system as well as support for Selection operation based on a variable of Universal type.

Future Work

However, at the present time the domain algebra of the jRelix system does not support operations on data of type Universal. These unsupported operations would include horizontal operations as well as reduction and functional mapping operations. One of the things to consider in the implementation of domain algebra support for data of Universal type would be how to deal with arithmetic operations when the data type of the value of an operand is not numeric.

The current implementation of the Transpose operator works only on flat relations. This implementation of the Transpose operator should be extended to support nested relations as well.

Another possible topic for future implementation might be type casting in jRelix. A type cast of Type A to Type B can be done in two steps: first, convert data of Type A to a Universal data type and then convert this intermediate Universal type data to data of Type B. The number of functions to implement would be two times the number of data types in the system as opposed to this number squared in case the casting is implemented without an intermediate Universal data type.

Bibliography

[Bak98] Patrick Baker. Design and implementation of database computations in java. Master's thesis, McGill University, Montreal, 1998.

[Che01] YuLing Chen. A GIS editor for a database programming language. Master's thesis, McGill University, Montreal, 2001.

[Hao98] Biao Hao. Implementation of the nested relational algebra in java. Master's thesis, McGill University, Montreal, 1998.

[Jav01] Java Compiler Compiler (JavaCC)-The Java Parser Generator. JavaCC web site at: www.webgain.com/products/javacc/documentation.html, 2001. The web site contains documentation softwares for JavaCC and JJTree.

[Mer77] T. H. Merrett. Relations as programming language elements. *Information Processing Letters*, 6(1):29–33, 1977.

[Mer84] T.H. Merrett. *Relational Information Systems.* Reston Publishing Co., Reston, VA, 1984.

[Mer01] T. H. Merrett. Attribute metadata for relational olap and data mining. In *Proceedings, 8th Biennial Workshop on Data Bases and Programming Languages*, pages 65–76, Rome, Italy, September 2001. Web link: http://www.cs.mcgill.ca/~tim/dbpl/attrMeta.ps.

[Sun00] Weizhong Sun. Updates and events in a nested relation programming language. Master's thesis, McGill University, Montreal, 2000.

[Yua98] Zhongxia Yuan. Java implementation of the domain algebra for nested relations. Master's thesis, McGill University, Montreal, 1998.