# A GRAPH REWRITING VISUAL LANGUAGE
# FOR DATABASE PROGRAMMING

PJ Rodgers
University of Sheffield

PJH King
Birkbeck College, University of London

contact: Peter Rodgers, Department of Computer Science, University of Sheffield,
Regent Court, 211 Portobello Street, Sheffield, S1 4DP, UK.

email: P.Rodgers@dcs.shef.ac.uk

## Abstract

*Textual database programming languages are computationally complete, but have the disadvantage of giving the user a non-intuitive view of the database information that is being manipulated. Visual languages developed in recent years have allowed naive users access to a direct representation of data, often in a graph form, but have concentrated on user interface rather than complex programming tasks. There is a need for a system which combines the advantages of both these programming methods.*

*We describe an implementation of Spider, an experimental visual database programming language aimed at programmers. It uses a graph rewriting paradigm as a basis for a fully visual, computationally complete language. The graphs it rewrites represent the schema and instances of a database.*

*The unique graph rewriting method used by Spider has syntactic and semantic simplicity. Its form of algorithmic expression allows complex computation to be easily represented in short programs. Furthermore, Spider has greater power than normally provided in textual systems, and we show that queries on the schema and associative queries can be performed easily and without requiring any additions to the language.*

## 1: Introduction

We believe we have identified a need for a visual language for programmers who use complex data structures, such as those found in databases. Graphs, when used wisely, can aid experienced users to comprehend complex concepts [1]. Most previous work in the area of graph based visual languages has concentrated on allowing less expert users access to low complexity programming facilities [2, 3, 4, 5] and these systems have met their aims with some success. However, attempts to allow expert users access to the advantages of visualisation of data and programs has been less fruitful. This paper describes Spider, a visual database language based on graph rewriting. It integrates complex programming facilities with a graph based data representation in a seamless manner. We also discuss some types of query that show this paradigm has benefits that are not available in most textual query languages.

Graph rewriting derives from the mathematical area of graph grammars [6]. Dactyl [7], is a textual programming language that uses graph rewriting. Derived from graph grammars are $\Delta$ grammars, which have been used as a basis for a powerful visual graph rewriting language by Loyall and Kaplan [8]. They give examples based both on state oriented and data oriented applications. The definition of the rewrites suffer from restrictions on their appearance to a triangular form. The rewrites work on a parallel basis, where the largest number of non conflicting subgraphs are rewritten.

PROGRES [9] and GOOD [10] are systems that have been developed for graph rewriting with databases. PROGRES combines visual rewriting with a textual programming language. It has a non-deterministic approach to subgraph matching. GOOD has a visual graph rewriting system using an object oriented database, where the primitive changes are specified explicitly on one diagram. In this system rewrites operate in parallel on all the matching subgraphs.

Spider is an implemented database programming language. It is fully visual, using graphs to display both data and programs. The language works by rewriting a graph containing the database and nodes initiating computation. This produces a computationally complete, conceptually simple visual programming language that allows programmers direct access to a representation of the data structure they are modifying. A program contains transformations similar in structure to the functions or predicates of declarative textual languages. Transformations are composed of a series of graph rewrites. The rewrites are formed in such a way as to maintain the visual structure of the data, with each rewrite having two graphs, one to represent the graph to be matched, and the other to represent the changes to be made. Transformations are started simply by the presence of application nodes in the host graph, called here the ‘application graph’. Application nodes do not represent data, but are introduced into the application graph initially by the user, and then by the transformations that are executed as a consequence. Execution halts when no application nodes remain in the application graph. This allows Spider to be syntactically simple and flexible in the types of transformation that can be used.

Spider has a deterministic approach to rewriting a single subgraph of the application graph. This is achieved by sorting the graph and then matching the highest ordered of the subgraphs. Rewriting only one of many potential subgraphs allows the formation of more classes of non conflicting transformation.

Functionality not found in textual languages is present in Spider as a natural consequence of its paradigm. Programs that query and alter the schema are possible. Such queries allow access to the meta data that forms the schema. Spider also has the capability to perform associative queries. These examine the structure of the data and its connections, so that the database can be treated as a graph and paths connecting specified nodes are the results.

The database integrated into Spider uses the binary-relational model [11]. Relational databases contain relations that may have any number of entities, whereas a binary-relational database can only contain relations that have two entities. This allows the database to be easily represented using a graph, as each relation can represent an arc between two set entities at the schema level and represent a set of arcs that connect instance values at the instance level.

Section two of this paper contains an overview of Spiders’ user interface and programming method, using an update on the schema and instances of a database as an example. Section three has a detailed description of the graph matching strategy used by Spider. The associative queries are given in section four. We give our conclusions and further work in section five. Appendix A contains a formal definition of spiders rewriting strategy. Appendix B has a demonstration of the computational completeness of Spider by implementing normal order reduction of the lambda calculus, this program is then used to compare graph rewriting with the functional paradigm.

## 2: Informal overview of Spider

In this section we introduce Spiders’ syntax and semantics in an informal manner, using a database update command as a running example. The database is a small section of a London local election database. The update will find the people who have at some time been a winner in a ward election. The method used is to first add the required meta data to the schema and then update the instance data. A more formal approach to the semantics of Spider is given in appendix A.

## 2.1: Graph primitives

Spiders data and programs are formed from various kinds of graph, which are constructed from two types of primitives, nodes and arcs. The primitive *characteristics* described here are used when sorting the graph. The three possible node types are: *set nodes* (circular border), *instance nodes* (oval), and *application nodes* (rectangular). All nodes must be labelled, and may have *duplicate identifiers* in some graphs (a superscript to the node label) to distinguish nodes with the same label. Set and instance nodes describe data, whereas the application nodes are for performing graph transformations. The two arc types are *function arcs* (black solid lines with labels) and *instance arcs* (grey dotted lines without labels). Both types of arc must be directional.

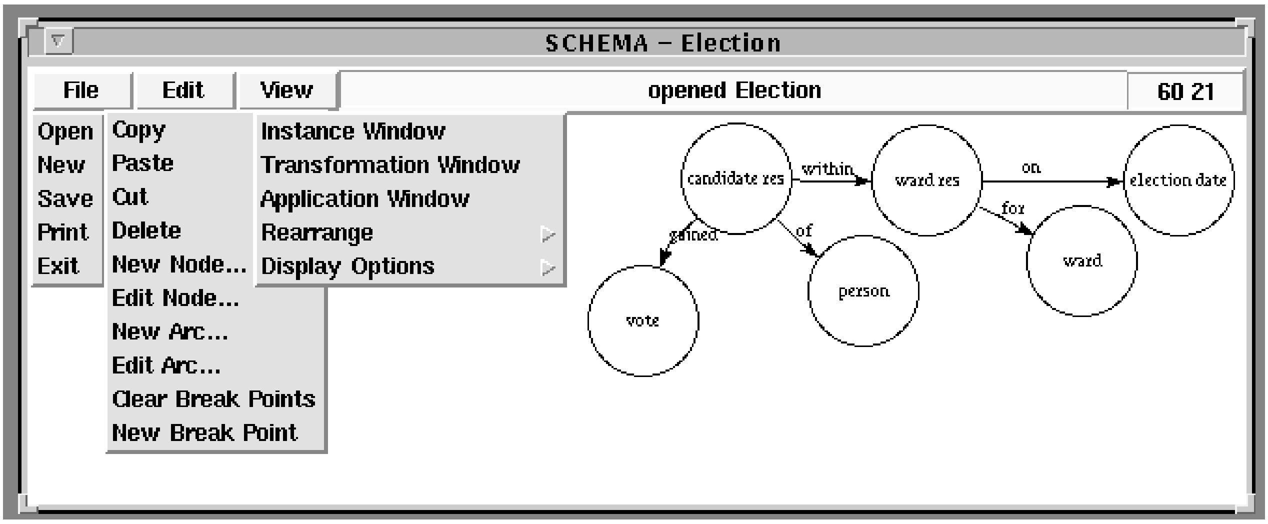


**Figure 2.1: The Schema Window with its Menu Options Displayed**

## 2.2: The user interface

The Spider system has four main windows. The *schema window* (figure 2.1) allows the user to define, save and load a binary-relational database schema. Data is entered and deleted in the *instance window*, where the schema also appears, but which cannot be changed in this window. The *transformations*, which are used to manipulate the database, and constitute a visual program, are defined in the *transformation window*. Execution of the transformations is displayed in the *application window.*

Every window has menus to access the functionality provided. User feedback is either written in the free space in the menu bar of the windows or output via a dialogue box in the case of important messages. The position of the mouse pointer is given as x y coordinates on the right of the menu bar.

All windows (including the application window) have facilities to edit the graphs in them. The basic operations described here can be seen in the ‘Edit’ menu of the schema window (figure 2.1). A mouse button one click over a primitive, or a drag over an area using button two, selects primitives for the editing operations that require it. Most windows have the following editing options: adding and editing a node; moving nodes (by a mouse button three drag); adding an arc between two selected nodes; editing the type and name of an arc; copying, cutting and deleting selected subgraphs (unconnected arcs are not left in the graph, nor added to the clipboard); pasting copied or cut subgraphs to a graph from the clipboard. Pasted subgraphs can originate in any graph as long as the primitives are valid for the destination graph. To ensure that all the structures in the Spider system are valid graphs, (ie. that all the arcs have two connecting nodes) two terminal nodes must first be selected before an arc can be added, and all the arcs that connect to a node are removed when that node is removed. The restrictions on editing of specific graphs will be explained in due course. The primitive creating and editing commands bring up a dialogue box which allows the input of the characteristics, this method means that invalid characteristic options can be deactivated.

The windows always have a ‘File’ menu which, at the very least, has commands to exit the window and print the window contents to a postscript file (many of the diagrams in this paper have been produced using this).

The ‘View’ menu has two main functions: to access other windows; and redisplay the current window contents. In the case of the schema window, all the other windows can be accessed. The ‘Rearrange’ option leads to a sub-menu that has a ‘redraw’ command to redisplay the window and may in the future contain access to graph rearranging algorithms (the instance window is currently the only one with such functionality). The ‘Display Options’ sub-menu has facilities for changing the colour, size and shape of the primitives in the graph.

## 2.3: Database schema window

The schema window is shown in figure 2.1, which also includes the menu options, which are discussed above, in the section on user interface. Spider displays and manipulates the schema information in this window, allowing only set nodes and function arcs. This window is the initial one displayed and allows access (via the ‘View’ menu) to the other windows of the system. The schema consists of set nodes connected by function arcs. The schema window has in its ‘File’ menu facilities for creating, saving and loading databases.

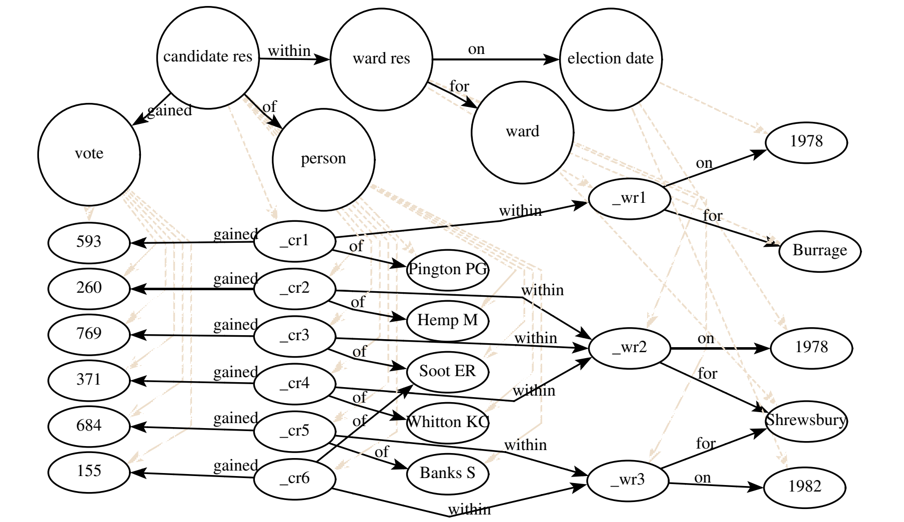


**Figure 2.2: The Instance Window Contents**

## 2.4: Instance window

Figure 2.2 shows the contents of the instance window: the example Election schema and the associated instance data. This window is entered from the schema window by first selecting a subgraph of the schema to indicate the part of the database that is to be displayed. Note that the instance nodes are connected by the same function arcs as their corresponding set nodes. Each instance is connected to one (and only one) set node by an instance arc. There can only be a function arc between instance nodes if there is one between their corresponding set nodes. An instance node or function arc is added to this graph by first selecting a ‘template’ primitive. If this is a node, then it is automatically connected to the appropriate set node. If it is a function arc, then the system ensures that the subsequently selected nodes at either end are correct. The instance arcs can only be edited indirectly by adding or deleting an instance node. Set nodes cannot be edited here. Copying, cutting and deleting of instance nodes and function arcs is allowed. Pasting of valid subgraphs is permitted. An additional menu, ‘Execute’ is unique to this window. This has the commands ‘Query’ and ‘Update’ which start a user defined program, discussed later.

The display of instances can be altered to some extent by hiding or revealing parts of the graph. The problem of displaying instance data is a non-trivial one. Systems such as SQL which have a table format allow easy access to large amounts of data, but only simple relationships can be seen. A graph representation allows an information rich display, where the connections between instances are shown, but the diagram can easily become unmanageable. The problems associated with visualising large amounts of data have been discussed by among others: Southerden [12]; Consens et al. [3]; and

Goldman et al. [13].

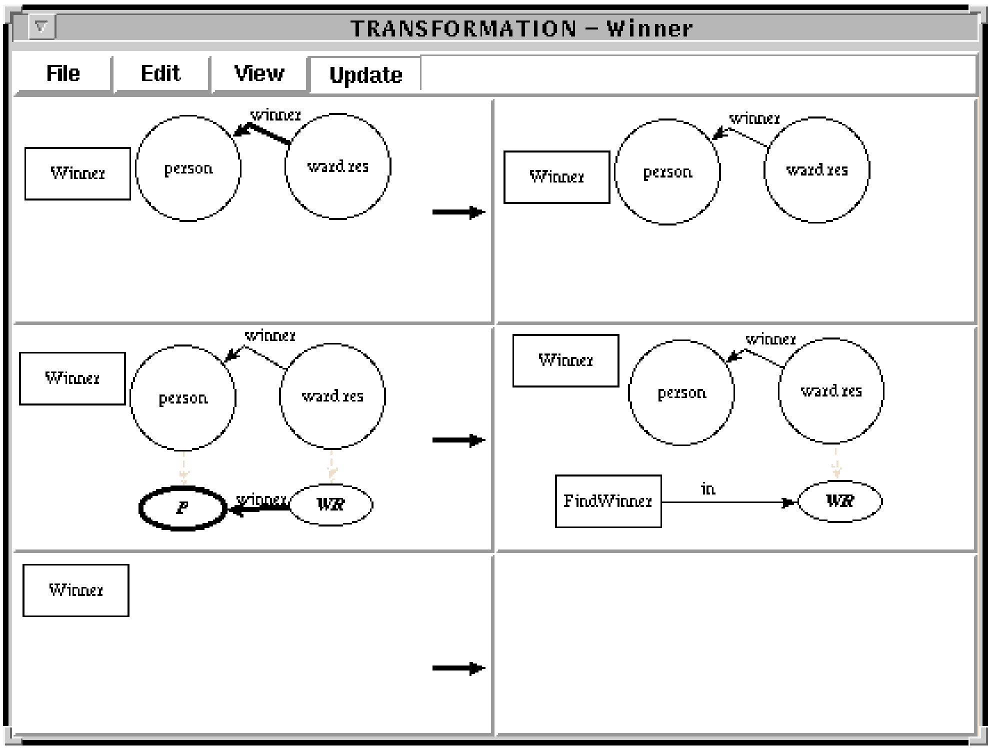


**Figure 2.3: The Transformation Window with 'Winner' Displayed**

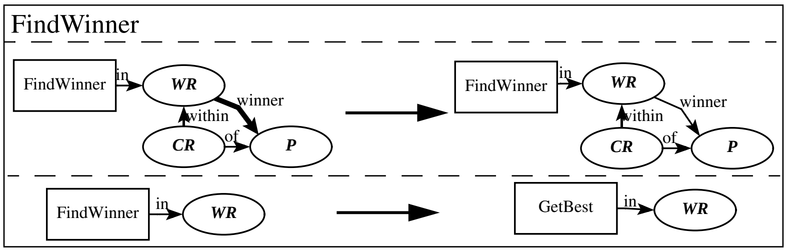


**Figure 2.4: The transformation 'FindWinner'**

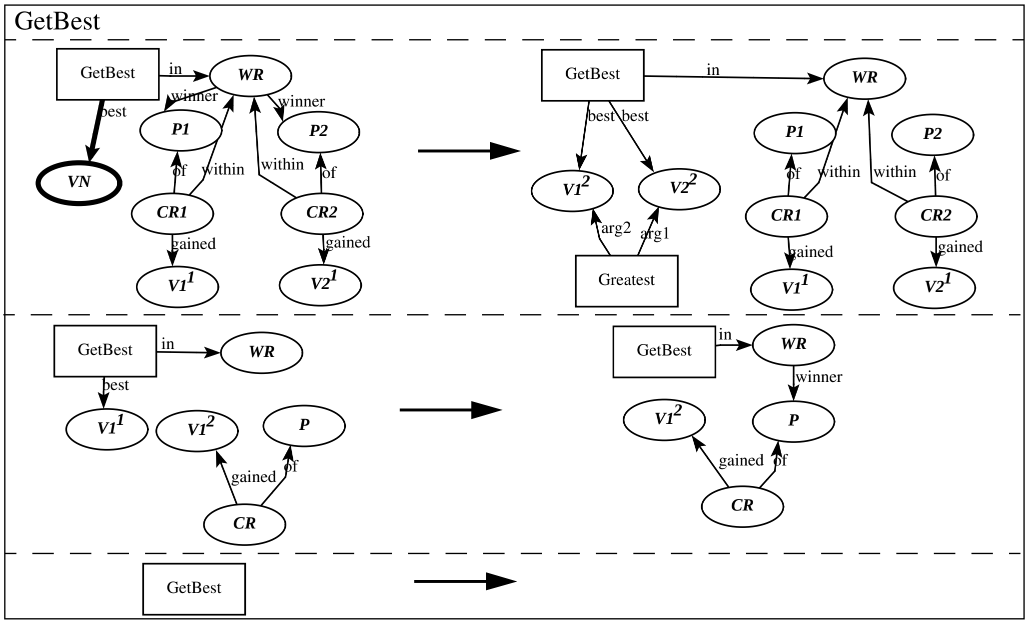


**Figure 2.5: The transformation 'GetBest'**

## 2.5: Transformation window

A visual program written using Spider is comprised of a number of *transformations*. The window for the 'Winner' transformation is shown in figure 2.3. This is the highest level transformation of the program, the rest of which are shown in figures 2.4 and 2.5. The operation of the program is discussed during the explanation of the Spider programming language that consists of the rest of this section. Transformations are defined by the user and must have a name. A transformation consists of a sequence of *graph rewrites* ('Winner' has three of these). Each graph rewrite has an *LHS graph* (on the left hand side) to specify a template of a subgraph to be found and changed, and an *RHS graph* (on the right hand side) to specify the resultant alteration.

Transformations can be saved and loaded using the 'File' menu. LHS or RHS graphs cannot be directly edited in this window, instead the 'Edit' menu has an option to create a graph editing sub-window for a particular graph. The 'Edit' menu also has options to create and delete rewrites. It also allows the user to toggle the state of transformation between 'Query', and 'Update', with the current state shown next to the 'View' menu in the window. We note that an update is usually more complex than a query as it must maintain the database data, whereas a query can destructively rewrite the database graph.

**2.5.1: LHS graph:** The LHS graph defines a template to find the subgraph to be rewritten. Special forms of primitive, whose effects are described later, are *variable* nodes and arcs (drawn with italic labels), and *negative primitives* (drawn with thick lines). The nodes in this graph may be connected in any way, with either type of arc. There can only be at most one application node in this type of graph, and it must have the same name as the transformation.

**2.5.2: RHS graph:** The new subgraph which is to replace that matched by the LHS graph is given by the RHS graph. Excepting negative primitives, all the nodes and arcs valid for the LHS graph are valid in this graph, and may be connected in any way. There may be any number of application nodes in this graph, which provides a mechanism for further calculation and recursion.

**2.5.3: Duplicate Identifiers:** If two or more nodes of the same type have the same label in an LHS or RHS graph, then duplicate identifiers must be used. These are written as a superscript to the node label. The use of duplicate identifiers can be seen in figure 2.5.

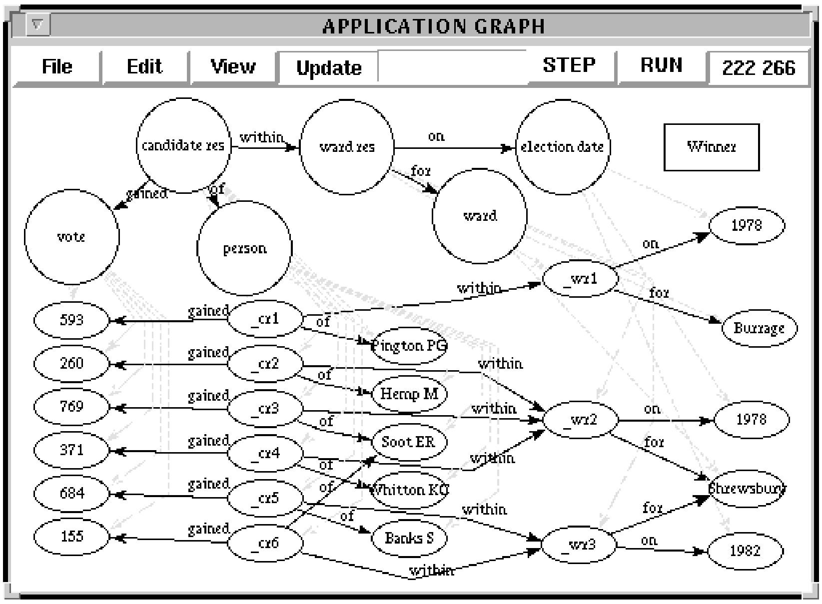


**Figure 2.6: The Application Window at the Start of Execution**

## 2.6: Application graph

The *application graph* is the host graph to be rewritten. It contains set, instance and application nodes. There is no restriction on how these may be connected by instance and function arcs (features such as loops, arcs with the same source and destination, and duplicated labels are all possible). The initial application graph is created from an instance graph by adding an application node to it, after specifying whether the execution will be in ‘Query’ or ‘Update’ mode. Queries have no effect on the database, and therefore can rewrite the application graph in a destructive manner, whereas the resultant graph of an update program is the database, so the data in it must be maintained. Note that updates which create invalid databases (due to the structure of the new graph) are not allowed to change the database.

The application graph at the start of the ‘Winner’ update is shown in figure 2.6. Note that this is the database with the simple addition of a ‘Winner’ application node. It is possible to edit application graphs in an arbitrary manner, with the addition, deletion and changing of primitives. The ‘File’ menu allow graphs to saved and loaded. These editing and file functions are not designed to be used during the evaluation of queries or updates, but mean that Spider is not constrained to work only on a database, so that general programs may be written for any data structure. This also helps with testing the operation of Spider, and there are commands in the ‘File’ menu to allow the formation and execution of a test bed. Within the ‘View’ menu are the commands ‘Hide’, giving the user the facility to hide selected parts of the application graph without affecting the rewriting process, and ‘Show All’ to reveal all the hidden primitives.

To initiate execution, two buttons are provided, ‘Step’ and ‘Run’. Step performs just the next execution step, whereas ‘Run’ performs all the execution steps until the program is completed.

## 2.7: Transformation step

A program is executed by performing a series of *transformation steps*. A transformation step is the process of applying all the most recently created (*newest*) application nodes in the application graph. These *node applications* will cause the transformation with the same name as the node label to be applied. This will rewrite the graph, creating and deleting primitives. Execution stops when there are no longer any application nodes in the graph.

If there is more than one newest application node (ie. they were created by the same transformation step), then they are considered to be applied in parallel. An *application conflict* occurs if one node deletes an area of the graph that is matched by a different node, in the same step. In this situation, execution is halted with an error message. The responsibility for avoiding application conflicts lies with the user.

When an application node is applied, the first graph rewrite in the transformation that matches with a subgraph of the application graph will be used. If no rewrite of a transformation can be used, then the application node remains in the graph.

The application graph in figure 2.6 contains one application node, 'Winner', this then is the only newest node in the graph, and so will be applied. The resultant rewrite will depend on which one of the three LHS graphs of 'Winner' (figure 2.3) can be matched in the application graph.

The *newest first* strategy for applying the application nodes creates a hierarchy of the transformations, so that the user knows that the application nodes and those that are derived from it have to be applied before the older ones are activated again. This allows the user to structure programs in a sensible and clear manner.

### 2.8: Finding a matching subgraph

*Graph matching* is the process of finding a part of the application graph that is a match of the LHS graph. The LHS graph and application subgraph must have the same logical structure, and the primitive labels must be the same, unless the LHS primitive is a variable. When there are several potential subgraphs only one of them is matched, the method for deciding which is discussed later in the more detailed section on graph matching.

A node or arc in an LHS graph that has a variable label may match with any primitive of the same type in the application graph. The variable is then *instantiated* with that label. If it occurs more than once with the same kind of primitive in an LHS graph, then it must match with the same label each time.

When negative nodes or arcs (drawn with thick lines) are in an LHS graph, it will only match if a subgraph matches with the positive primitives, and is not part of a subgraph which matches the whole LHS graph (ie. both positive and negative primitives). Any positive arcs connected to negative nodes are treated as if they were negative arcs. Note that only the positive part of the subgraph is the 'matched subgraph'.

We can see then that the first LHS graph of 'Winner' will find a match in the application graph of figure 2.6. This is because the two set nodes 'person' and 'ward res' exist in the application graph, and are not connected by an arc labelled 'winner' (this arc is drawn with a thick line and so it is a negative arc).

### 2.9: Graph rewriting

Once an appropriate subgraph has been matched it will be rewritten. The rewrite is decided by comparing the positive primitives in the LHS graph with the RHS graph. Later in this section the consequences of having an *attractor node* in the RHS graph is explained, but the following gives the simple rules for graph rewriting:

- **Leave in the application graph those primitives that are the same in both the LHS and RHS graphs. Primitives which are in the LHS graph, but not the RHS graph are deleted from the application graph. Primitives in the RHS graph, but not in the LHS graph are created in the application graph. Arcs that have a source and/or destination node deleted are deleted.**
- **Variable primitives that get created are given their instantiated label, if they have not been instantiated (as the variable name is in the RHS but not the LHS), then they are given a unique label.**
- **If a set node is deleted, then all its instances not explicitly defined (ie. in both the LHS and RHS graph), are deleted as well.**

[Note that Nodes are defined to be the same if they have the same type, label and duplicate identifier (superscript to the label). Arcs are defined to be the same if they have the same type, label, and the source and destination nodes are the same.]

Comparing the first LHS and RHS graphs of the first rewrite of 'Winner' in figure 2.3 we can see that the application node 'Winner' is in both graphs, so it will not be changed in the application graph. This is true also of the set nodes 'person' and 'ward res', but that a new arc 'winner' is created between these set nodes as it is in the RHS graph but not the LHS graph (a negative arc of the same name is in it, but has no effect on the rewriting process). The resultant application graph is then the same as that of figure 2.6, with the addition of a function arc 'winner'. This rewrite performs a query and update of the database schema.

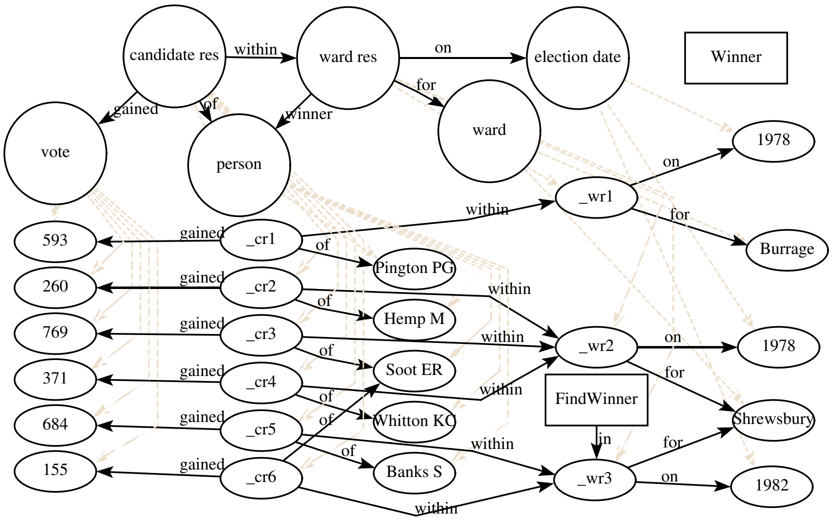


**Figure 2.7: The Application Window Contents After Step 2**

For the next transformation step the only application node in the application graph is ‘Winner’ again. The first LHS graph will not match this time as the negative arc will match with the ‘winner’ arc created in the first step, so the second LHS graph will be tried. This is successful, as the variable node ‘WR’ will match with any of the ‘ward res’ instances, and none of them have a ‘person’ node attached by a ‘winner’ arc. The graph matched is that containing the node ‘_wr3’ (the later section on graph matching explains the method used for determining the matched subgraph) and ‘WR’ is then instantiated to this value. The comparison with the second RHS graph indicates that a ‘FindWinner’ application node connected to ‘_wr3’ by an ‘in’ function arc is created. The result of this rewrite can be seen in figure 2.7.

There are now 2 application nodes in the graph, of which ‘FindWinner’ is the newest, and so will be applied in the next step. ‘Winner’ will not be applied again until ‘FindWinner’ and any subsequent application nodes have disappeared.

The first rewrite of ‘FindWinner’ takes any ‘person’ connected via a ‘candidate res’ arc to the specified ‘ward res’ instance without a ‘winner’ arc and attaches such an arc to it. When all the relevant people have a ‘winner’ arc attached the first rewrite will no longer be applied and the second rewrite will be used. This removes the ‘FindWinner’ application node and adds a ‘GetBest’ application node.

The ‘GetBest’ transformation will remove all these recently created ‘winner’ arcs, except the one for the person with the highest vote. It does this by first copying two of the vote nodes of people with ‘winner’ arcs (deleting those two ‘winner’ arcs as well) and uses the builtin transformation ‘Greatest’ to delete the least of these duplicate nodes. The second rewrite takes the remaining duplicate node and finds the original that matches, reattaching the ‘winner’ arc for the relevant person. This continues until neither of the first two rewrites will match (ie. there is only one winner arc left), leaving the third rewrite to delete the application node.

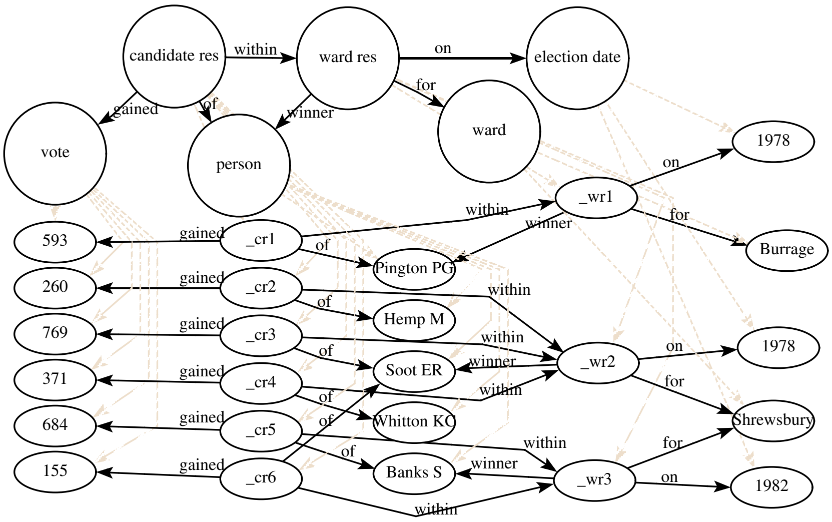


**Figure 2.8: The Application Window After the Execution halts on Step 26**

‘Winner’ is left as the only application node in the graph. It takes the next ‘ward res’ instance that has not been dealt with and repeats the above process. This continues until the second ‘Winner’ rewrite cannot be used, meaning the third will match, deleting the application node and finishing execution to leave just the updated database (figure 2.8).

We note that the above program deals nicely with a partially updated database. If new ‘ward res’ instances were added after the program had been run once, the program could be run again to find the new winners without adversely affecting the updates from the first run.

## 2.10: Attractor nodes

Attractor nodes modify the rewriting process. If only one of the source or destination node of an arc is deleted, then that end of the arc is attached to the attractor node (if one exists). Attractor nodes have a shaded appearance. They become normal nodes when they appear in the application graph. Attractor nodes can only be in the RHS graph, and at most one is allowed.

## 2.11: Built-in transformations

Some hard coded transformations exist, such as the transformations which expect two instance nodes connected to the application node by the arcs ‘arg1’ and ‘arg2’. They have an attractor instance as their result instance node. These include: ‘Add’, ‘Multiply’, ‘Minus’, ‘Divide’, ‘Least’ and ‘Greatest’. Others do similar calculations, but upon all the instances of a set, including: ‘AddSet’, ‘MultiplySet’, ‘LeastSet’, ‘GreatestSet’, ‘CountSet’ and ‘CopySet’. These produce a result calculated from the instances of the set node. There are some built in transformations that simply alter the display features of the primitives they act on. For example ‘Highlight’ causes its argument node to become outlined in bold. Any arcs between two highlighted nodes becomes highlighted.

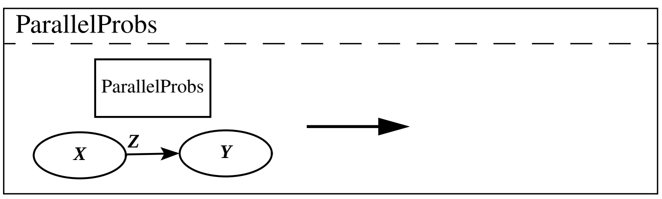


**Figure 3.1: The Transformation ‘ParallelProbs’**

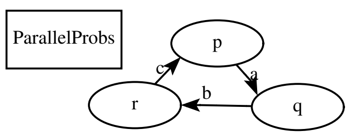


**Figure 3.2**

## 3: The graph matching method

Graph matching was introduced briefly in the previous section. Here we discuss the method Spider uses in more detail. Previous rewriting systems have taken different approaches to graph rewriting. In GOOD [10] all matching subgraphs are rewritten. If this was used in Spider then the only rewrite in the transformation of figure 3.1 (that deletes two connected nodes and the application node) would cause a conflict when used in the application graph in figure 3.2, as the LHS graph could match in three overlapping ways. A conflict could be avoided if only one subgraph was rewritten. PROGRES [9] takes this approach, arbitrarily selecting the rewritten subgraph out of the candidate matches. This unfortunately causes non-deterministic behaviour. A variation on this idea is that of Loyall and Kaplan [7], who rewrite the largest subset of non conflict causing subgraphs, this would also result in non-determinism if implemented in Spider. In the example of figures 3.1 and 3.2 both the last two methods would leave a single node in the graph, but it is not defined which one it would be.

Spider rewrites a single subgraph to gain the advantages of increasing the number of classes of rewrites allowed. We solve the problem of maintaining a deterministic programming language by sorting the candidate subgraphs. This sorting first considers the subgraphs themselves, and then the surrounding areas of the application graph. The result is to differentiate subgraphs, except where they are entirely equivalent (which means that rewriting any one of equivalent subgraphs results in the same application graph). In most cases, comparing the surrounding subgraph will be unnecessary, as there will be differences in the two candidate subgraphs. These are ordered by comparing the highest ordered nodes first, an intuitive method that makes it easy for the user to decide which subgraph will be chosen.

Although the above paragraph gives the users perception of the method we use, it is equivalent to say that we sort both the application graph and LHS graph and use a backtracking search of the application graph, starting with the highest valued primitives, for each node in the LHS graph in turn. This node matching must ensure both that the node characteristics are the same, and that the arcs which are connected to previously matched nodes can be satisfactorily matched.

Taking the application graph sorting view improves the implementation efficiency of the system, as the highest ordered subgraph is always the first matched. this means only one subgraph has to be found in the application graph, rather than a large number, in the case where the LHS graph contains mostly variable primitives. This large number would then have to be sorted at a large processing cost.

Efficiency can be maximised if the sorting algorithm orders more specific primitives (for instance: constants; and nodes with a large number of arc connections) higher, so that they will be matched first. This should reduce the amount of backtracking performed, and would not be possible if we were to take a number of equal subgraphs and attempt to sort them by their surrounding graph.

After the initial sort of the application graph, changes to it due to graph rewriting tend to leave the majority of primitives unchanged, so that the graph sort after a transformation step has only to integrate the unsorted new primitives, rather than performing a sort of the entire graph.

### 3.1: Graph sorting algorithm

The graph is represented by a list of nodes and a list of arcs. The sorting will change the order of the primitives in each list and add a position value to each primitive. This is its position in the list, with position 1 being the highest valued. If two primitives are not separable, then they have the same position. The sorting algorithm uses a variety of comparison functions, which take either two nodes or two arcs and return less than, equal or greater than. Sorting is based on the principle of constant refinement of the position of the nodes and arcs, so that if the position of a pair of primitives has already been discovered to be different, then they will not be compared again at a later stage. An outline of the algorithm is as follows:

```
BEGIN
Sort nodes by characteristic
Sort arc by characteristic
Repeat until no more changes in position:
        Sort arcs by connection
        Sort nodes by connecting arcs
END
```

First the nodes and arcs are sorted by characteristic. Here they are given with the most significant first. The order of the node characteristics is: positive higher than negative; constant higher than variable; application higher than instance higher than set; node names by the C string comparison function (strcmp), which is defined so that the string “b” is valued higher than the string “aa”, which in turn is valued higher than the string “a”; finally duplicate identifiers also by strcmp. The order of the arc characteristics is: positive higher that negative; constant higher than variable; instance higher than function; finally function arc names by the C function strcmp.

The nodes are then sorted according to their neighbouring nodes. This requires a series of iterations to propagate any difference through the graph. To this end a loop is performed, which sorts the arcs by their connecting nodes and then the nodes by the position and number of arcs connected, until there are no more improvements in the ordering of the graph.

The arc connection sort in the loop compares first the position of the source nodes of the arcs, and then if no difference is found the destination nodes are compared.

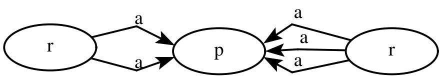


**Figure 3.3**

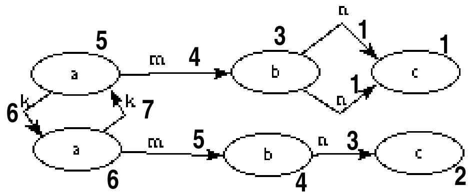


**Figure 3.4**

The "sort nodes by connecting arcs" is by the first difference found in the position values of the connecting arcs, with the arcs paired by taking then in order of position. Otherwise, if a node has more connecting arcs, then it is ordered higher. The algorithm must take account of the possibility of multiple duplicate arcs between duplicate nodes. This is resolved by ordering higher the node with the largest number of duplicate arcs to the same node. Figure 3.3 shows a graph with this type of duplicate arc. In this case the node ordering is first the rightmost one labelled 'r', then the leftmost 'r' node, and finally the node labelled 'p'.

The complexity of this algorithm is not much greater than a standard sorting algorithm, as the routines in the middle sort only the elements which have previously not been changed. These sorts are effectively on 'pockets' of primitives that a previous comparison has not been able to distinguish.

Figure 3.4 shows an example application graph with each primitives position, to illustrate some of the features of sorting. We note that the two 'a' nodes are inseparable unless we consider all the graph that surrounds them. The only difference in this graph is that the top 'c' and 'b' nodes have more 'n' arcs connected, so they have a higher position. Differences can then be found with some of the 'n' arcs (but the two at the top are entirely equivalent and therefore have the same position). Eventually the algorithm propagates the differences through to the 'a' nodes, which means the 'k' arcs can be separated (the one leaving the highest valued node has a higher value). The ordering is then as shown by the numbers next to the primitives.

Applying this method to the example of figures 3.1 and 3.2 shows us that the highest valued instance node in the LHS graph, 'Y' will match with the highest valued instance node in the application graph, 'r'. It follows that the connecting arc 'Z' matches with 'b' and the node 'X' must match with 'q'. The resultant graph is simply the node 'p'.

## 4: Associative queries

Associative queries were introduced by Ayres and King [14] in the Hydra system. The principle is that the information about the relationships between data in a database can be as useful as the data itself. An example given is that of a criminal investigation database, where "pathways" between suspects can be of major interest, as also can the objects closely linked to a particular individual (in the "vicinity" of the individual). In order to allow associative queries of this sort Hydra has special commands to return the functions and instances that make up the associations. This has knock on effects throughout the language. (For instance dealing with the problem of functions as data elements). Associations in graphs are also used in semantic networks [15].

In this section we show that associative queries can be written in Spider without the need for any extensions to the language.

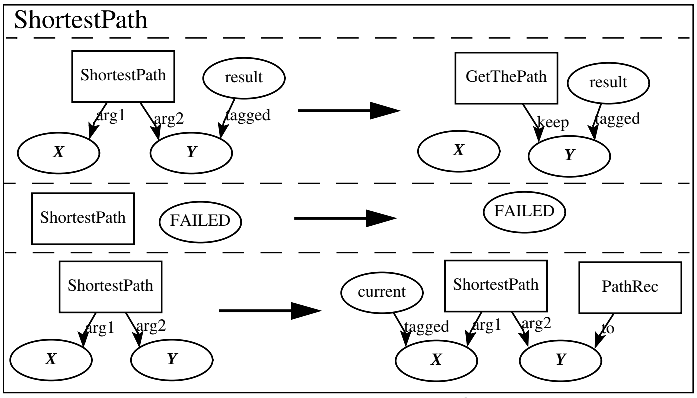


**Figure 4.1: The transformation 'ShortestPath'**

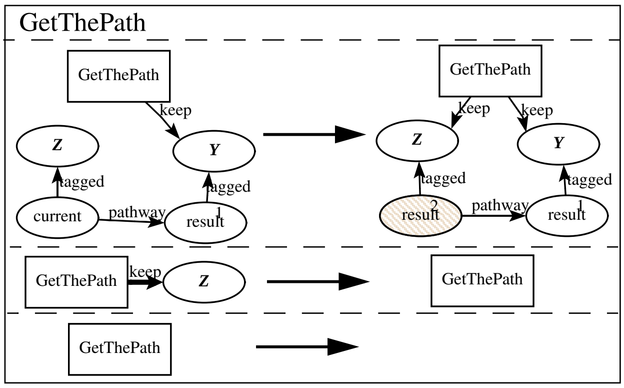


**Figure 4.3: The transformation 'GetThePath'**

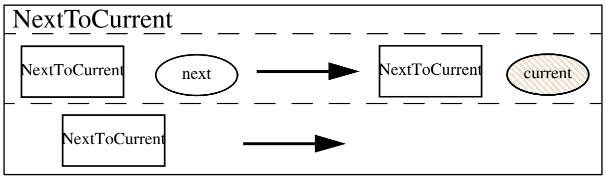


**Figure 4.4: The transformation 'NextToCurrent'**

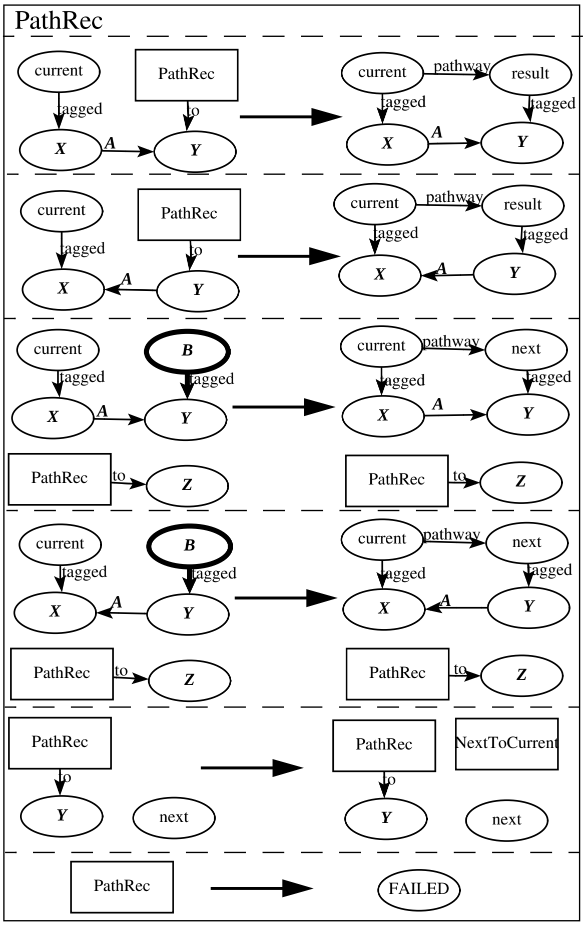


**Figure 4.2: The transformation 'PathRec'**

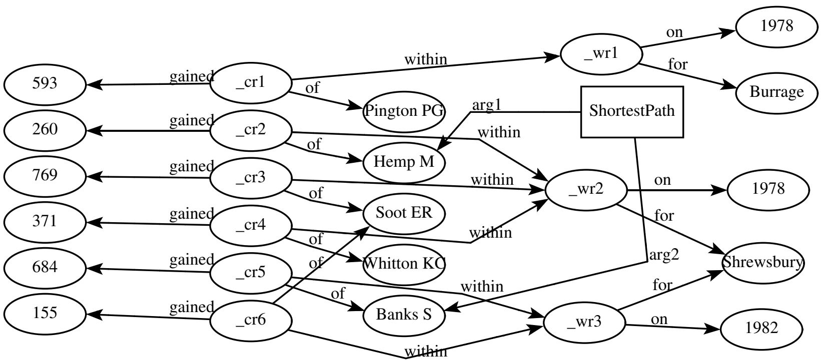


**Figure 4.5: The Application Graph at the Start of 'ShortestPath'**
The set nodes have been hidden in this picture.

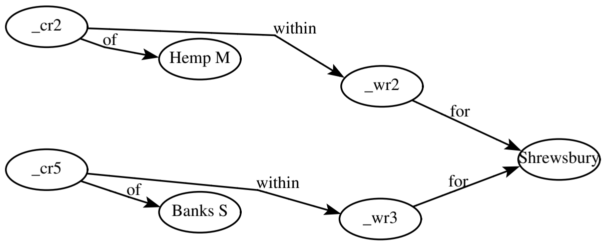


**Figure 4.6: The Application Graph with the result of 'ShortestPath'**

### 4.1: Shortest Path Queries

Figures 4.1 to 4.4 show the transformations that make up the 'ShortestPath' query. This will find a connection between two nodes by a shortest path, or add a 'FAILED' instance node to the graph if there is no such path (ie. the nodes are in unconnected subgraphs). Figure 4.5 shows how it might be used with the Election database (as with all the application graphs in this section, the set nodes are hidden from view) to find a connection between the two people 'Hemp M' and 'Banks S'. These two nodes have been connected to the 'ShortestPath' application node by the arcs 'arg1' and 'arg2'. Figure 4.6 shows the application graph after execution has been completed. We can see that they have both stood for election in the 'Shrewsbury' ward. Note that we have chosen, for ease of comprehension, to present the results by deleting all nodes not in the path.

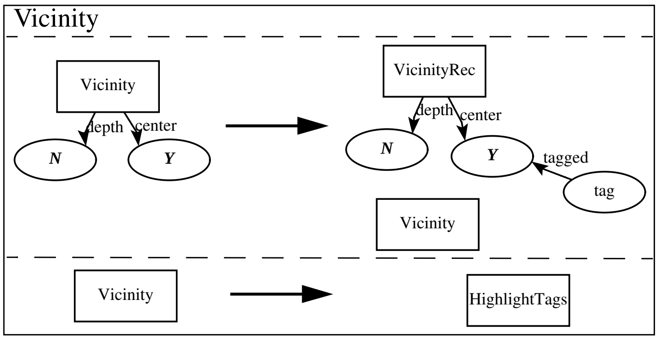


**Figure 4.7: The transformation 'Vicinity'**

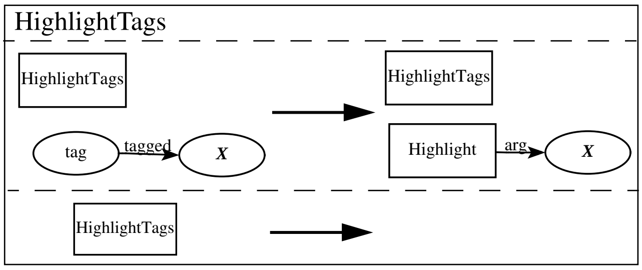


**Figure 4.9: The transformation 'HighlightTags'**

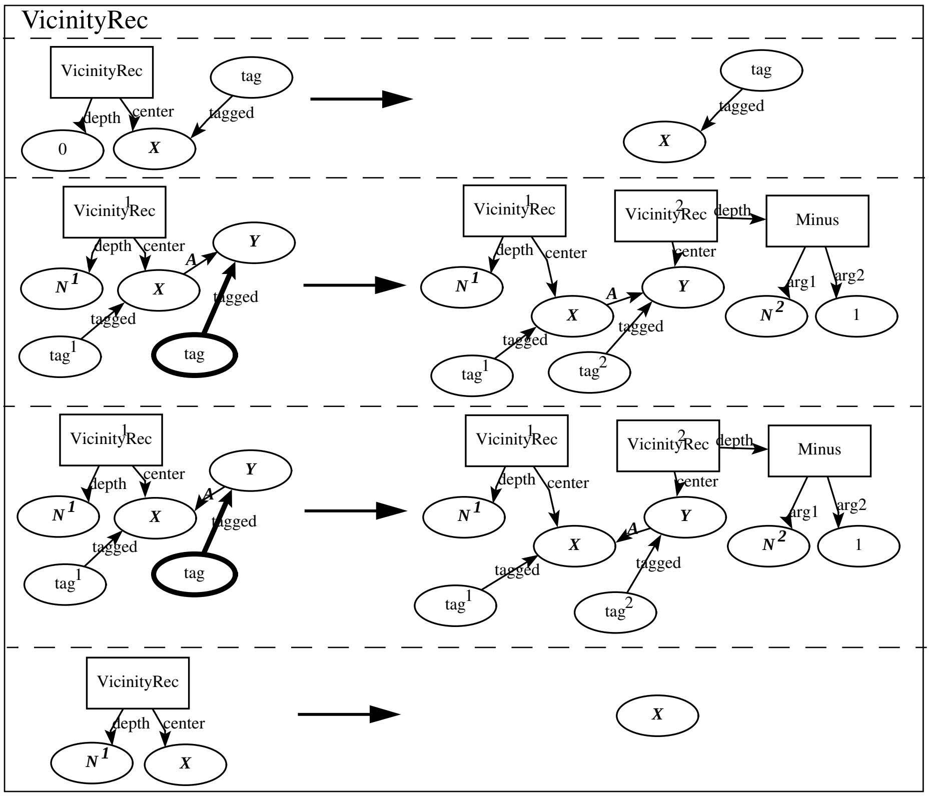


**Figure 4.8: The transformation 'VicinityRec'**

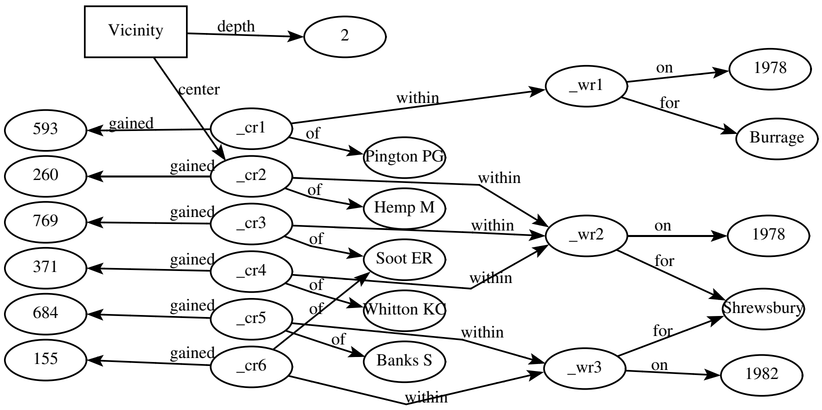


**Figure 4.10: The Application Graph at the Start of 'Vicinity'**

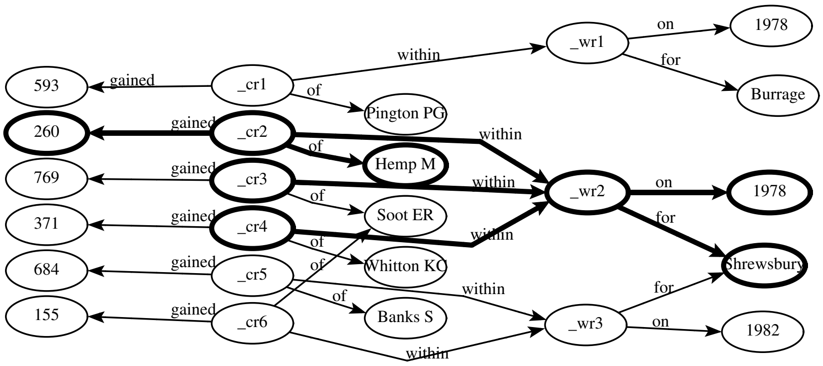


**Figure 4.11: The Result of 'Vicinity'**

### 4.2: Vicinity Queries

A vicinity query finds the instances close to the chosen center. The user decides how deep to make the search. Figures 4.7, 4.8 and 4.9 show the transformations for this query. Figure 4.10 shows how it might be used to find all the instances within 2 arcs of the candidature '_cr2'. The result is shown in figure 4.11. Note that this query makes use of the built in transformation 'Highlight' to emphasise the resultant instances. The primitives in the highlighted subgraph appear with thick borders, but are not to be confused with negative primitives that are also drawn with thick borders, but cannot appear in an application graph. This is an alternative method of showing the result to that used in the path query.

## 5: Conclusions and further work

In this paper we have given an informal overview of Spider, including its syntax, semantics and user interface. For illustration purposes we performed a relatively complex calculation on an example database. We have also shown how Spider enables programmers to perform queries not normally available in database programming languages.

The paradigm introduced in this paper differs greatly from the ones currently used by database programmers. Although we believe that it could form the basis of a practical system, hard empirical evidence of Spiders suitability as a useful programming language is an area of future work.

Other interesting avenues of research include looking at useful primitives that might be added to the system. Here we have presented a syntactically simple system. Making it more complex might add substantially to its semantics. Among candidates for inclusion are LHS 'fold' nodes (that may match with several different nodes) and LHS 'once only' primitives that will match at most one time with the same application graph primitive, perhaps reducing the need for tagging used nodes. A method of encapsulating subgraphs into a single node might be useful. One approach is Hypernodes [16], where the subgraphs have only one arc connecting to the outside graph. A more general approach, where arbitrary subgraphs are encapsulated, is more difficult, but possible.


## Acknowledgements

The authors are grateful to the EPSRC and IBM UK Laboratories, Hursley for financial support, and to Professor G.C.H. Sharman of IBM for most helpful discussions.


## References


1. Petre M. & Green T.R.G. (1992) Learning to Read Graphics: Some Evidence that 'Seeing' an Information Display is an Acquired Skill. *Journal of Visual Languages and Computing*, (1992) 3. pp. 25-47.
2 Angelaccio M., Catarci T. & Santucci G. (1990) QDB*: A Graphical Query Language with Recursion. *IEEE Transactions on Software Engineering*, 16, 10. pp. 1150-1163.
3. Consens M.P., Cruz I.F. & Mendelzon A.O. (1992) Visualizing Queries and Querying Visualizations. *SIGMOD RECORD* 21, 1, March 1992. pp. 39-46.
4. Cordy J.R. & Graham T.C.N. (1992) GVL: Visual Specification of Graphical Output. *Journal of Visual Languages and Computing*, (1992) 3. pp. 25-47.
5. Mohan L. & Kashyap R.L. (1993) A Visual Query Language for Graphical Interaction With Schema-Intensive Databases. *IEEE Transactions on Knowledge and Data Engineering*, 5, 5. pp. 843-858.
6. Nagl M. (1983) A Tutorial and Bibliographical Survey on Graph Grammars. *Proceedings 2nd International Workshop on Graph Grammars and Their Application to Computer Science. LNCS* 153. Springer-Verlag. pp. 70-126.
7. Glauert J.R., Kennaway J.R. & Sleep M.R. (1991) Dactl: An Experimental Graph Rewriting Language. *Proceedings 4th International Workshop on Graph Grammars and Their Application to Computer Science. LNCS* 532. Springer-Verlag. pp. 378-395.
8. Loyall J.P. & Kaplan S.M. (1992) Visual Concurrent Programming with $\Delta$ Grammars. *Journal of Visual Languages and Computing* (1992) 3. pp. 107-133.
9. Schurr A. (1994) Rapid Programming with Graph Rewrite Rules. *Proceedings USENIX Symposium on Very High Level Languages (VHLL), Santa Fe*, October 1994. pp. 83-100.
10. Gyssens M., Paredaens J., Van den Bussche J. & Van Gucht D. (1994) A Graph-Oriented Object Database Model. *IEEE Transactions on Knowledge and Data Engineering*, 6, 4. pp. 572-586.
11. Frost R.A. (1982) Binary Relational Storage Structures. *The Computer Journal*, 25, 3. pp. 358-367.
12. Southerden S.B. (1992) Information Technology: Support for Law Enforcement Investigations and Intelligence. *ICL Technical Journal* November 1992. pp. 302-315.
13. Goldman K.J., Goldman S.A., Kanellakis P.C. & Zdonik S.B. (1986) ISIS: Interface for a Semantic Information System. *Proceedings ACM SIGMOD International Conference on Data Engineering* 1986. pp. 151-164.
14. Ayres R. & King P.J.H. (1994) Extending the Semantic Power of Functional Database Query Languages with Associational Features. *Congres INFORSID* 1994, Aix-en-Provence. pp. 301-320.
(Also appears in: *Ingénierie des systèmes d'information* 3, 2-3/1995. pp. 441-463.)
15. Quillian M.R. (1967) Word Concepts: A Theory and Simulation of Some Basic Semantic Capabilities. *Behavioural Science* 12, 1967. pp. 410-430.
16. Poulovassilis A. & Levene M. (1994) A Nested-Graph Model for the Representation and Manipulation of Complex Objects. *ACM Transactions on Information Systems* 12, 1, 1994. pp. 35-68.


## Appendix A: Formal definition

This section gives a formal treatment of the graphs and structures introduced in a more informal way in section 2. This allows us to define the graph matching and rewriting method used in the Spider system. We also use this formal definition to specify restrictions on the graphs.

### A.1: Definition - Spider Graph

We define a Spider Graph to be a tuple, $\boldsymbol{G} = (\boldsymbol{N}, \boldsymbol{A}, \boldsymbol{L_N}, \boldsymbol{L_A})$, where:

• $\boldsymbol{N}$ is the set of nodes in the graph;

• $\boldsymbol{A}$ is the set of arcs;

• $L_N = (Q_N, I_N, T_N, C_N, V_N)$ is a tuple containing functions which describe the nodes. The node characteristics can be described in a single function instead of this tuple, but for convenience the functions are kept separate;
• $L_A = (Q_A, T_A, V_A, S_A, D_A)$ is a tuple containing functions which describe the arcs. As with the similar node tuple the functions describing the arc characteristics are kept separate;
• $Q_N: N \rightarrow N_{name}$ is a function that associates nodes with labels. $N_{name}$ is the set of possible node labels;
• $I_N: N \rightarrow N_{dup}$ is a function that associates nodes with duplicate identifiers. $N_{dup}$ is the set of possible duplicate identifiers;
• $T_N: N \rightarrow \{Set, Instance, Application\}$, $C_N: N \rightarrow \{Attractor, Nonattractor\}$ and $V_N: N \rightarrow \{Variable, Constant\}$ each describes a characteristic of the nodes. The elements of the codomains are atomic values;
• $Q_A: A \rightarrow A_{name}$ is a function that associates arcs with labels. $A_{name}$ is the set of possible arc labels;
• $T_A: A \rightarrow \{Function, Instance\}$ and $V_A: A \rightarrow \{Variable, Constant\}$ each describes a characteristic of arcs. The elements of the codomains are atomic values;
• $S_A: A \rightarrow N$ and $D_A: A \rightarrow N$ are the functions which give the source and destination nodes of the arcs, respectively.

Empty node labels, arc labels and duplicate identifiers are given the value $nil$.

The following restrictions hold for every Spider Graph:
• $\forall n \in N, Q_N(n) \neq nil$, that is, nodes cannot have empty labels;
• $\forall a \in A, T_A(a) = Function \Rightarrow Q_A(a) \neq nil$, that is, function arcs cannot have empty labels;
• $\forall a \in A, T_A(a) = Instance \Rightarrow Q_A(a) = nil$, that is, instance arcs must have empty labels.
• $\forall a \in A, T_A(a) = Instance \Rightarrow V_A(a) = Constant$, that is, instance arcs must be constant.

## A.2: Definition - Application Graph

An Application Graph is a Spider Graph where the following restrictions apply:
• $\forall n \in N, I_N(n) = nil$;
• $\forall n \in N, V_N(n) = Constant$;
• $\forall n \in N, C_N(n) = Nonattractor$;
• $\forall a \in A, V_A(a) = Constant$.

## A.3: Definition - LHS Graph

A LHS Graph is a Spider Graph where the following four sets are defined: $N_{pos}$, $A_{pos}$, $N_{neg}$ and $A_{neg}$, where $N_{pos}$ and $N_{neg}$ partition $N$, and similarly $A_{pos}$ and $A_{neg}$ partition $A$. (A partition of a set is a collection of non-intersecting subsets whose union is the set.) Informally, these sets divide the graph into positive and negative primitives.

The following restrictions also hold:
• $\forall a \in A, S_A(a) \in N_{neg} \vee D_A(a) \in N_{neg} \Rightarrow a \in A_{neg}$, that is, an arc connected to a negative node must be a negative arc;
• $\forall n1, n2 \in N, T_N(n1) = Application \wedge T_N(n2) = Application \Rightarrow n1 = n2$, that is, at most one Application node is allowed in the graph;
• $\forall n \in N_{neg}, T_N(n) \neq Application$, that is, the application node is positive;
• $\forall n1, n2 \in N, n1 \neq n2 \wedge Q_N(n1) = Q_N(n2) \Rightarrow I_N(n1) \neq I_N(n2)$, that is, if two or more node labels are the same, then their duplicate identifiers must be different.
• $\forall n \in N, C_N(n) = Nonattractor$.

### A.4: Definition - RHS Graph

A RHS Graph is a Spider Graph with the following restrictions:

• $\forall \boldsymbol{n1}, \boldsymbol{n2} \in N, \boldsymbol{C}_N(\boldsymbol{n1}) = \boldsymbol{Attractor} \wedge \boldsymbol{C}_N(\boldsymbol{n2}) = \boldsymbol{Attractor} \Rightarrow \boldsymbol{n1} = \boldsymbol{n2}$, that is, at most one Attractor node is allowed in the graph;

• $\forall \boldsymbol{n1}, \boldsymbol{n2} \in N, \boldsymbol{n1} \neq \boldsymbol{n2} \wedge \boldsymbol{Q}_N(\boldsymbol{n1}) = \boldsymbol{Q}_N(\boldsymbol{n2}) \Rightarrow \boldsymbol{I}_N(\boldsymbol{n1}) \neq \boldsymbol{I}_N(\boldsymbol{n2})$, that is, if two or more node labels are the same, then their duplicate identifiers must be different;

### A.5: Definition - Transformation Definition and Program

A Transformation Definition, $\boldsymbol{K}$ is a label $\boldsymbol{l}_K \in \boldsymbol{K}_{name}$ and a sequence of pairs $(\boldsymbol{G}_1^{LHS}, \boldsymbol{G}_1^{RHS})\ldots(\boldsymbol{G}_r^{LHS}, \boldsymbol{G}_r^{RHS})$ where $\boldsymbol{K}_{name}$ is the set of possible Transformation Definition labels. Each $\boldsymbol{G}_i^{LHS}$ is a LHS Graph and each $\boldsymbol{G}_i^{RHS}$ is a RHS Graph for $\boldsymbol{i} = \boldsymbol{1}\ldots\boldsymbol{r}$. The following must be also true:

[In the formulae that follow in the rest of this section we do not distinguish the functions by their particular node and arc sets where it is obvious, so that $\boldsymbol{T}_N(\boldsymbol{n})$ represents $\boldsymbol{T}_{N_i^{LHS}}(\boldsymbol{n})$ in the formula immediately below.]

• $\forall \boldsymbol{n} \in N_i^{LHS}, \boldsymbol{T}_N(\boldsymbol{n}) = \boldsymbol{Application} \Rightarrow \boldsymbol{Q}_N(\boldsymbol{n}) = \boldsymbol{l}_K$ where $\boldsymbol{i} = \boldsymbol{1}\ldots\boldsymbol{r}$, that is, an application node in any left hand side graph must have the same label as the Transformation Definition.

A Program, $\boldsymbol{SP}$ consists of a set of Transformation Definitions.
A.6: Definition - Graph Submatch

Given 2 node sets $N^{LHS}$ and $N^{App}$, and 2 arc sets $A^{LHS}$ and $A^{App}$, where $N^{LHS}$ and $A^{LHS}$ are subsets of node and arc sets of the same LHS graph, and the sets $N^{App}$ and $A^{App}$ are the nodes and arcs of an Application Graph, then a Graph Submatch is a pair of injective functions $(\boldsymbol{M}_N, \boldsymbol{M}_A)$ where $\boldsymbol{M}_N\colon N^{LHS} \rightarrow N^{App}$ and $\boldsymbol{M}_A\colon A^{LHS} \rightarrow A^{App}$ (an injective function is one in which two different members of the domain cannot be mapped to the same member of the codomain). Informally, it is two functions mapping each given primitive of the LHS Graph to a unique primitive of the Application Graph. This definition will be used later to define a Graph Match. The following restrictions also apply:

• $\forall \boldsymbol{n} \in N^{LHS}, \boldsymbol{T}_N(\boldsymbol{n}) = \boldsymbol{T}_N(\boldsymbol{M}_N(\boldsymbol{n}))$, that is a node must map to a node of the same type;

• $\forall \boldsymbol{n} \in N^{LHS}, \boldsymbol{V}_N(\boldsymbol{n}) = \boldsymbol{Constant} \Rightarrow \boldsymbol{Q}_N(\boldsymbol{n}) = \boldsymbol{Q}_N(\boldsymbol{M}_N(\boldsymbol{n}))$, that is, a constant node must map to a node of the same label;

• $\forall \boldsymbol{n1}, \boldsymbol{n2} \in N^{LHS}, (\boldsymbol{V}_N(\boldsymbol{n1}) = \boldsymbol{Variable}) \wedge (\boldsymbol{V}_N(\boldsymbol{n2}) = \boldsymbol{Variable}) \wedge (\boldsymbol{Q}_N(\boldsymbol{n1}) = \boldsymbol{Q}_N(\boldsymbol{n2}))$
$\Rightarrow \boldsymbol{Q}_N(\boldsymbol{M}_N(\boldsymbol{n1})) = \boldsymbol{Q}_N(\boldsymbol{M}_N(\boldsymbol{n2}))$
That is variable nodes of the same label must map to nodes of the same label;

• $\forall \boldsymbol{a} \in A^{LHS}, \boldsymbol{T}_A(\boldsymbol{a}) = \boldsymbol{T}_A(\boldsymbol{M}_A(\boldsymbol{a}))$, that is, an arc must map to an arc of the same type;

• $\forall \boldsymbol{a} \in A^{LHS}, \boldsymbol{V}_A(\boldsymbol{a}) = \boldsymbol{Constant} \Rightarrow \boldsymbol{Q}_A(\boldsymbol{a}) = \boldsymbol{Q}_A(\boldsymbol{M}_A(\boldsymbol{a}))$, that is, constant arcs must map to an arc of the same label;

• $\forall \boldsymbol{a1}, \boldsymbol{a2} \in A^{LHS}, (\boldsymbol{V}_A(\boldsymbol{a1}) = \boldsymbol{Variable}) \wedge (\boldsymbol{V}_A(\boldsymbol{a2}) = \boldsymbol{Variable}) \wedge (\boldsymbol{Q}_A(\boldsymbol{a1}) = \boldsymbol{Q}_A(\boldsymbol{a2}))$
$\Rightarrow \boldsymbol{Q}_A(\boldsymbol{M}_A(\boldsymbol{a1})) = \boldsymbol{Q}_A(\boldsymbol{M}_A(\boldsymbol{a2}))$
That is, variable arcs of the same label must map to arcs of the same label;

•$\forall a \in A^{LHS}, M_N(S_A(a)) = S_A(M_A(a))$, that is the node that matches with the source of an arc must be the same as the source of the matched arc;

•$\forall a \in A^{LHS}, M_N(D_A(a)) = D_A(M_A(a))$, that is the node that matches with the destination of an arc must be the same as the destination of the matched arc.

### A.7: Definition - Graph Match

Let $G^{App}$ be an Application Graph and $G^{LHS}$ be a LHS Graph. Then we define a Graph Match to be a Graph Submatch, $(M_N, M_A)$ of $N^{LHS}_{pos}$, $N^{App}$, $A^{LHS}_{pos}$ and $A^{App}$, where there is no Graph Submatch $(W_N, W_A)$ of $N^{LHS}$, $N^{App}$, $A^{LHS}$ and $A^{App}$ such that:

•$\forall n \in N^{LHS}_{pos}, W_N(n) = M_N(n)$, that is positive nodes map to the same Application Graph node in both Submatches.

•$\forall a \in A^{LHS}_{pos}, W_A(a) = M_A(a)$, that is positive arcs map to the same Application Graph arc in both Submatches.

Informally, a Graph Match is a mapping of the positive primitives of the LHS Graph to primitives of the Application Graph as long as there is not also a mapping of all the LHS negative primitives of the LHS Graph to unmapped primitives in the Application Graph.

We note that for simplicity we do not define which subgraph of the application graph is to be matched.

### A.8: Definition - Rewrite Rule

A Rewrite Rule $E(G^{LHS}, G^{RHS})$, where $G^{LHS}$ is a LHS Graph and $G^{RHS}$ is a RHS Graph, is a tuple, $(B_N, B_A)$ ,where $B_N: N^{LHS} \to N^{RHS}$ and $B_A: A^{LHS} \to A^{RHS}$ are both injective partial functions. (Note that a partial function is not strictly a function at all, as it maps from a subset of the domain to a subset of the codomain.) These will be useful as they indicate that a node is deleted if $n^L \in N^{LHS}$ and $B_N(n^L)$ is not defined, similarly an arc is deleted if $a^L \in A^{LHS}$ and $B_A(a^L)$ is not defined. Also a node is created if $n^R \in N^{RHS}$ and $B^{-1}_N(n^R)$ is not defined, and an arc is created if $a^R \in A^{RHS}$ and $B^{-1}_A(a^R)$ is not defined.

We define $B_N$ as follows:

$$\begin{aligned}
&\bullet \forall n^L \in N^{LHS}_{pos}, \forall n^R \in N^{RHS}, T_N(n^R) = T_N(n^L) \wedge V_N(n^R) = V_N(n^L) \\
&\quad \wedge Q_N(n^R) = Q_N(n^L) \wedge I_N(n^R) = I_N(n^L) \\
&\quad \Rightarrow B_N(n^L) = n^R
\end{aligned}$$

That is $B_N$ exists for a LHS node if the node is positive and it is equal in all characteristics to a RHS node.

Let $J$ be a function from the set of pairs, $A^{LHS}_{pos} \times A^{RHS}$ to a boolean value, where $J(a^L, a^R)$ is true if:

$$\begin{aligned}
&\bullet T_A(a^L) = T_A(a^R) \wedge V_A(a^L) = V_A(a^R) \wedge Q_A(a^L) = Q_A(a^R) \\
&\quad \wedge B_N(S_A(a^L)) = S_A(a^R) \wedge B_N(D_A(a^L)) = D_A(a^R)
\end{aligned}$$

That is the characteristics, source and destination are equal. We note that if $B_N$ does not exist for the source or destination of the arc, then the value of $J$ for that arc is false.

$B_A$ is a one-to-one function between $Y^{LHS}$ and $Y^{RHS}$, where $Y^{LHS}$ and $Y^{RHS}$ are the largest possible subsets of $A^{LHS}$

and $A^{RHS}$ that can be defined by:

- $\forall a^L \in Y^{LHS}, \forall a^R \in Y^{RHS}, J(a^L, a^R) \Rightarrow B_A(a^L) = a^R$

$B_A$ is a partial function where each possible arc in $A^{LHS}$ is mapped to an arc in $A^{RHS}$. This definition is necessary because duplicate arcs between the same nodes can exist. If less duplicates exist in one graph, then an injective partial function is preserved.

### A.9: Definition - Labelling Functions

We define two Labelling Functions, $H_N{:}\, N^{LHS} \cup N^{RHS} \to N_{Name}$ and $H_A{:}\, A^{LHS} \cup A^{RHS} \to A_{Name}$ of $(M_N, M_A)$ and $(B_N, B_A)$ where $(M_N, M_A)$ is the Graph Match of $G^{App}$ and $G^{LHS}$, and $(B_N, B_A)$ is $E(G^{LHS}, G^{RHS})$. These are for finding the labels of the new and matched primitives, defined as follows:

[We note that the functions $Q_N$, $Q_A$, $V_N$ and $V_A$ are assumed to be the appropriate ones for the primitive.]

- $\forall n \in N^{LHS} \cup N^{RHS}, V_N(n) = \textit{Constant} \Rightarrow H_N(n) = Q_N(n)$, that is, constant nodes keep their label.
- $\forall n \in N^{LHS}, V_N(n) = \textit{Variable} \wedge B_N(n) \textit{is undefined} \Rightarrow H_N(n) = \textit{nil}$, that is, deleted variable nodes are given no label.
- $\forall n \in N^{LHS}, V_N(n) = \textit{Variable} \wedge B_N(n) \textit{is defined} \Rightarrow H_N(n) = Q_N(M_N(n))$, that is, variable nodes kept in the graph are given their instantiated label.
- $\forall n \in N^{RHS}, V_N(n) = \textit{Variable} \wedge B_N^{-1}(n) \textit{is undefined}$
  $\wedge (\exists p \in N^{LHS}, V_N(p) = \textit{Variable} \wedge T_N(p) = T_N(n) \wedge Q_N(p) = Q_N(n)) \Rightarrow H_N(n) = Q_N(M_N(p))$
  That is, new variable nodes that are instantiated are given the instantiated label.
- $\forall n \in N^{RHS}, V_N(n) = \textit{Variable} \wedge B_N^{-1}(n) \textit{is undefined}$
  $\wedge \neg(\exists p \in N^{LHS}, V_N(p) = \textit{Variable} \wedge T_N(p) = T_N(n) \wedge Q_N(p) = Q_N(n))$
  $\Rightarrow H_N(n) \textit{is a unique value}$
  That is, new variable nodes that are not instantiated are given a unique label.

- $\forall a \in A^{LHS} \cup A^{RHS}, V_A(a) = \textit{Constant} \Rightarrow H_A(a) = Q_A(a)$, that is, constant arcs keep their label.
- $\forall a \in A^{LHS}, V_A(a) = \textit{Variable} \wedge B_A(a) \textit{is undefined} \Rightarrow H_A(a) = \textit{nil}$, that is, deleted variable arcs are given no label.
- $\forall a \in A^{LHS}, V_A(a) = \textit{Variable} \wedge B_A(a) \textit{is defined} \Rightarrow H_A(a) = Q_A(M_A(a))$, that is, variable arcs kept in the graph are given their instantiated label.
- $\forall a \in A^{RHS}, V_A(a) = \textit{Variable} \wedge B_A^{-1}(a) \textit{is undefined}$
  $\wedge (\exists b \in A^{LHS}, V_A(b) = \textit{Variable} \wedge Q_A(b) = Q_A(a)) \Rightarrow H_A(a) = Q_A(M_A(b))$
- That is, new variable arcs that are instantiated are given the instantiated label.
- $\forall a \in A^{RHS}, V_A(a) = \textit{Variable} \wedge B_A^{-1}(a) \textit{is undefined}$
  $\wedge \neg(\exists b \in A^{LHS}, V_A(b) = \textit{Variable} \wedge Q_A(b) = Q_A(a)) \Rightarrow H_A(a) \textit{is a unique value}$
  That is, new variable arcs that are not instantiated are given a unique label.

These two functions give the label of the primitive when it appears in the application graph, with a constant primitive retaining the same label as the corresponding primitive in the LHS or RHS graph. A primitive in an application graph that is an instantiated variable gets the label from the matched application graph primitive of the variable. If it has not been instantiated, then the label is an undefined unique new label.

## A.10: Definition - Graph Rewrite

We are now in a position to formally define what a Graph Rewrite is. A Graph Rewrite, $\boldsymbol{P}(\boldsymbol{G}^{App}, \boldsymbol{G}^{LHS}, \boldsymbol{G}^{RHS})$ where $\boldsymbol{G}^{App}$ is an Application Graph, $\boldsymbol{G}^{LHS}$ is a LHS Graph and $\boldsymbol{G}^{RHS}$ is a RHS Graph, is a new Application Graph, $\boldsymbol{G}^{New} = (\boldsymbol{N}^{New}, \boldsymbol{A}^{New}, \boldsymbol{L}_N^{New}, \boldsymbol{L}_A^{New})$. It is derived from the Application Graph using a Graph Match of the LHS Graph and a Rewrite Rule derived from the LHS and RHS Graphs:

If there is no Graph Match of $\boldsymbol{G}^{App}$ and $\boldsymbol{G}^{LHS}$ then $\boldsymbol{G}^{New} = \boldsymbol{G}^{App}$. Otherwise:

Let $(\boldsymbol{M}_N, \boldsymbol{M}_A)$ be a Graph Match of $\boldsymbol{G}^{App}$ and $\boldsymbol{G}^{LHS}$.

Let $(\boldsymbol{B}_N, \boldsymbol{B}_A)$ be $\boldsymbol{E}(\boldsymbol{G}^{LHS}, \boldsymbol{G}^{RHS})$, the Rewrite Rule formed from $\boldsymbol{G}^{LHS}$ and $\boldsymbol{G}^{RHS}$.

Let $\boldsymbol{H}_N$ and $\boldsymbol{H}_A$ be the Labelling Functions of $(\boldsymbol{M}_N, \boldsymbol{M}_A)$ and $(\boldsymbol{B}_N, \boldsymbol{B}_A)$.

[We note that the definition of an application graph requires nodes to be constant, nonattractors with a ***nil*** duplicate identifier and requires arcs to be constant.]

Unmatched primitives:

- $\forall \boldsymbol{n} \in \boldsymbol{N}^{App}, \boldsymbol{M}_N^{-1}(\boldsymbol{n})\, \textbf{\textit{is undefined}} \wedge \neg(\boldsymbol{T}_N(\boldsymbol{n}) = \textbf{\textit{Instance}} \wedge$
  $(\exists \boldsymbol{a} \in \boldsymbol{A}^{App}, \exists \boldsymbol{n}^{Set} \in \boldsymbol{N}^{App}, \boldsymbol{T}_N(\boldsymbol{n}^{Set}) = \textbf{\textit{Set}} \wedge \boldsymbol{T}_A(\boldsymbol{a}) = \textbf{\textit{Instance}} \wedge \boldsymbol{S}_A(\boldsymbol{a}) = \boldsymbol{n}^{Set} \wedge$
  $\boldsymbol{D}_A(\boldsymbol{a}) = \boldsymbol{n} \wedge \boldsymbol{n}^{Set} \notin \boldsymbol{N}^{New}))$
  $\Rightarrow \boldsymbol{n} \in \boldsymbol{N}^{New} \wedge \boldsymbol{Q}_N^{New}(\boldsymbol{n}) = \boldsymbol{Q}_N^{App}(\boldsymbol{n}) \wedge \boldsymbol{T}_N^{New}(\boldsymbol{n}) = \boldsymbol{T}_N^{App}(\boldsymbol{n})$

That is, unmatched Application nodes (that are not instance nodes with their set node deleted) are members of the New Application Graph, with the same label and type.

- $\forall \boldsymbol{a} \in \boldsymbol{A}^{App}, \boldsymbol{M}_A^{-1}(\boldsymbol{a})\, \textbf{\textit{is undefined}} \wedge \boldsymbol{S}_A(\boldsymbol{a}) \in \boldsymbol{N}^{New} \wedge \boldsymbol{D}_A(\boldsymbol{a}) \in \boldsymbol{N}^{New}$
  $\Rightarrow \boldsymbol{a} \in \boldsymbol{A}^{New} \wedge \boldsymbol{Q}_A^{New}(\boldsymbol{a}) = \boldsymbol{Q}_A^{App}(\boldsymbol{a}) \wedge \boldsymbol{T}_A^{New}(\boldsymbol{a}) = \boldsymbol{T}_A^{App}(\boldsymbol{a})$
  $\wedge \boldsymbol{S}_A^{New}(\boldsymbol{a}) = \boldsymbol{S}_A^{App}(\boldsymbol{a}) \wedge \boldsymbol{D}_A^{New}(\boldsymbol{a}) = \boldsymbol{D}_A^{App}(\boldsymbol{a})$

That is, unmatched App arcs are members of the New Application Graph if both their source and destination nodes are in the new graph.

Matched but unchanged primitives:

- $\forall \boldsymbol{n} \in \boldsymbol{N}^{App}, \boldsymbol{M}_N^{-1}(\boldsymbol{n})\, \textbf{\textit{is defined}} \wedge \boldsymbol{B}_N(\boldsymbol{M}_N^{-1}(\boldsymbol{n}))\, \textbf{\textit{is defined}}$
  $\Rightarrow \boldsymbol{n} \in \boldsymbol{N}^{New} \wedge \boldsymbol{Q}_N^{New}(\boldsymbol{n}) = \boldsymbol{Q}_N^{App}(\boldsymbol{n}) \wedge \boldsymbol{T}_N^{New}(\boldsymbol{n}) = \boldsymbol{T}_N^{App}(\boldsymbol{n})$

That is, for each matched but not deleted node in the Application graph, put it in the new graph.

- $\forall \boldsymbol{a} \in \boldsymbol{A}^{App}, \boldsymbol{M}_A^{-1}(\boldsymbol{a})\, \textbf{\textit{is defined}} \wedge \boldsymbol{B}_A(\boldsymbol{M}_A^{-1}(\boldsymbol{a}))\, \textbf{\textit{is defined}} \wedge \boldsymbol{S}_A(\boldsymbol{a}) \in \boldsymbol{N}^{New}$
  $\wedge \boldsymbol{D}_A(\boldsymbol{a}) \in \boldsymbol{N}^{New} \Rightarrow \boldsymbol{a} \in \boldsymbol{A}^{New} \wedge \boldsymbol{Q}_A^{New}(\boldsymbol{a}) = \boldsymbol{Q}_A^{App}(\boldsymbol{a}) \wedge$
  $\boldsymbol{T}_A^{New}(\boldsymbol{a}) = \boldsymbol{T}_A^{App}(\boldsymbol{a}) \wedge \boldsymbol{S}_A^{New}(\boldsymbol{a}) = \boldsymbol{S}_A^{App}(\boldsymbol{a}) \wedge \boldsymbol{D}_A^{New}(\boldsymbol{a}) = \boldsymbol{D}_A^{App}(\boldsymbol{a})$

That is, for each matched but not deleted arc in the Application graph that has its source and destination in the new graph, put it in the new graph.

New primitives:

- $\forall \boldsymbol{n} \in \boldsymbol{N}^{RHS}, \boldsymbol{B}_N^{-1}(\boldsymbol{n})\, \textbf{\textit{is undefined}} \Rightarrow \boldsymbol{n} \in \boldsymbol{N}^{New} \wedge \boldsymbol{Q}_N^{New}(\boldsymbol{n}) = \boldsymbol{H}_N(\boldsymbol{n}) \wedge \boldsymbol{T}_N^{New}(\boldsymbol{n}) = \boldsymbol{T}_N^{RHS}(\boldsymbol{n})$, that is, for each new node in the RHS graph, put it in the new graph.

•$\forall a \in A^{RHS}, B_A^{-1}(a) \textit{is undefined} \Rightarrow a \in A^{New} \wedge Q_A^{New}(a) = H_A(a) \wedge T_A^{New}(a) = T_A^{RHS}(a)$
$\wedge S_A^{New}(a) = S_A^{RHS}(a) \wedge D_A^{New}(a) = D_A^{RHS}(a)$

That is, for each new arc in the RHS graph, put it in the new graph.

Attracted arcs:

If $\exists n_{att} \in N^{RHS}, C_N(n_{att}) = \textit{Attractor}$

•$\forall a \in A^{App}, M_A^{-1}(a) \textit{is undefined} \wedge S_A(a) \notin N^{New} \wedge D_A(a) \in N^{New}$
$\Rightarrow \exists b \in A^{New}, Q_A(b) = Q_A(a) \wedge T_A(b) = T_A(a) \wedge S_A(b) = n_{Att} \wedge D_A(b) = D_A(a)$

•$\forall a \in A^{App}, M_A^{-1}(a) \textit{is undefined} \wedge S_A(a) \notin N^{New} \wedge D_A(a) \in N^{New}$
$\Rightarrow \exists b \in A^{New}, Q_A(b) = Q_A(a) \wedge T_A(b) = T_A(a) \wedge S_A(b) = S_A(a) \wedge D_A(b) = n_{Att}$

Informally an attracted arc is an Application arc that has not been matched and with either a source or destination node deleted. It has the same characteristics, and one of its end nodes is the existing one in the application graph, however the unconnected end is attached to the attractor node. If there is no attractor node then these arcs are not created.

### A.11: Definition - Node Application

If we have an Application Graph, $G^{App}$, a node $n^{App} \in N^{App}$ where $T_N(n^{App}) = \textit{Application}$, and a Transformation Definition $K$ consisting of $l_K$ and a sequence of pairs $(G_1^{LHS}, G_1^{RHS})...(G_r^{LHS}, G_r^{RHS})$. A Node Application is an Application Graph $G^{App'}$ defined as:

•$Q_N(n^{App}) \neq l_K \Rightarrow G^{App'} = G^{App}$, that is, if the node label and Transformation Definition label are different, then the new graph is the old graph.

•$Q_N(n^{App}) = l_K \Rightarrow G^{App'} = P(G^{App}, G_m^{LHS}, G_m^{RHS})$, where $(G_m^{LHS}, G_m^{RHS})$ is the first pair in the Transformation Definition sequence, where there is a Graph Match, $(M_N, M_A)$ of $G_m^{LHS}$ and $G^{App}$.

Informally, a Node Application is an Application Graph that is formed the rewrite of a LHS & RHS pair where the LHS Graph is the first one in the sequence that matches.

### A.12: Definition - Transformation Step

If we have an Application Graph, $G^{App}$, with the set of most recently created nodes $\{n_1...n_q\}$ (created at the same time), where $\forall i = 1...q, T(n_i) = \textit{Application} \wedge n_i \in N^{App}$, and a Program, $SP = \{K1...Ks\}$. Then a Transformation Step, $G^{Step}$, is the Application Graph formed, from the parallel Node Application of those nodes most recently created, by the following method.

Let $\{G_1^{App}, ..., G_q^{App}\}$ be the set of Node Applications formed from $G^{App}$, $n_i$, and either the Transformation Definition $Kj$ where $l_{Kj} = Q(n_i)$, or $K1$ if there is no such transformation label.

Then $\{G_1^{Sub}, ..., G_q^{Sub}\}$ is the set of graphs where:

[Let $(M_{N_v}, M_{A_v})$ be the graph match of $G^{App}$ and the LHS graph $G_{Kv}^{LHS}$, of $Kv$]

•$\forall i = 1...q, \forall n \in N_i^{App}, M_{N_i}^{-1}(n) \textit{is defined} \vee n \notin N^{App} \Rightarrow n \in N_i^{Sub}$, ie the matched and new node subset.

•$\forall i = 1...q, \forall a \in A_i^{App}, M_{A_i}^{-1}(a) \textit{is defined} \vee a \notin A^{App} \Rightarrow a \in A_i^{Sub}$, ie the matched and new arc subset.

Let $(N^{Rest}, A^{Rest})$ be a pair containing the nodes and arcs not matched by any LHS graph, defined by:

• $\forall n \in N^{App}, \forall j = 1...q, n \in N_j^{App} \wedge M_{N_j}^{-1}(n) \textit{ is undefined} \Rightarrow n \in N^{Rest}$, that is, the application nodes that are in all the rewritten graphs (ie. they are not deleted by being an instance node with a deleted set node) and have no match.

• $\forall a \in A^{App}, \forall j = 1...q, a \in A_j^{App} \wedge M_{A_j}^{-1}(a) \textit{ is undefined} \Rightarrow a \in A^{Rest}$, that is, the application arcs that are in all the rewritten graphs (ie. they are not deleted by disconnected source or destination) and have no match.

A Transformation Step cannot be defined for the Application Graph and Program if an Application Conflict occurs. An Application Conflict occurs if, for any $v, w$, $\exists n \in N^{App}, n \notin N_v^{App} \wedge (\exists s \in N_{Kw}^{LHS}, n = M_{N_j}(s))$ or $\exists a \in A^{App}, a \notin A_v^{App} \wedge (\exists b \in A_{Kw}^{LHS}, a = M_{Aj}(b))$. That is, an Application Conflict occurs if there are any primitives deleted in one Node Application and matched in another.

We can now define the Transformation Step, $G^{Step} = (N^{Step}, A^{Step}, L_N^{Step}, L_A^{Step})$:

• $N^{Step} = N^{Rest} \cup N_1^{Sub} \cup N_2^{Sub} \cup ... N_q^{Sub}$

• $A^{Step} = A^{Rest} \cup A_1^{Sub} \cup A_2^{Sub} \cup ... A_q^{Sub}$

•Let $Q_N^{'} = Q_N^{App} \cup Q_{N1}^{App} \cup Q_{N2}^{App} \cup ... Q_{Nq}^{App}$, then $\forall n \in N^{Step}, (n, Q_N^{'}(n)) \in Q_N^{Step}$

•Let $T_N^{'} = T_N^{App} \cup T_{N1}^{App} \cup T_{N2}^{App} \cup ... T_{Nq}^{App}$, then $\forall n \in N^{Step}, (n, T_N^{'}(n)) \in T_N^{Step}$

•Let $Q_A^{'} = Q_A^{App} \cup Q_{A1}^{App} \cup Q_{A2}^{App} \cup ... Q_{Aq}^{App}$, then $\forall a \in A^{Step}, (a, Q_A^{'}(a)) \in Q_A^{Step}$

•Let $T_A^{'} = T_A^{App} \cup T_{A1}^{App} \cup T_{A2}^{App} \cup ... T_{Aq}^{App}$, then $\forall a \in A^{Step}, (a, T_A^{'}(a)) \in T_A^{Step}$

•Let $S_A^{'} = S_A^{App} \cup S_{A1}^{App} \cup S_{A2}^{App} \cup ... S_{Aq}^{App}$, then $\forall a \in A^{Step}, (a, S_A^{'}(a)) \in S_A^{Step}$

•Let $D_A^{'} = D_A^{App} \cup D_{A1}^{App} \cup D_{A2}^{App} \cup ... D_{Aq}^{App}$, then $\forall a \in A^{Step}, (a, D_A^{'}(a)) \in D_A^{Step}$

Informally, a Transformation Step is formed from all the rewrites of the newest nodes in the Application Graph, as long as none of the rewrites deletes a primitive matched in another rewrite.

## Appendix B: Computational completeness

In this appendix we give an implementation of the lambda calculus in order to demonstrate that Spider is computationally complete.

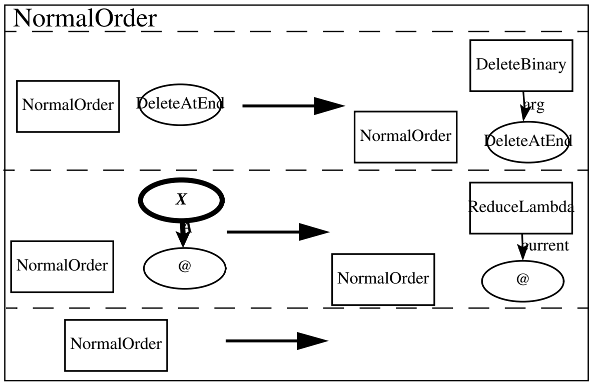


**Figure B.1: The transformation ‘NormalOrder’**

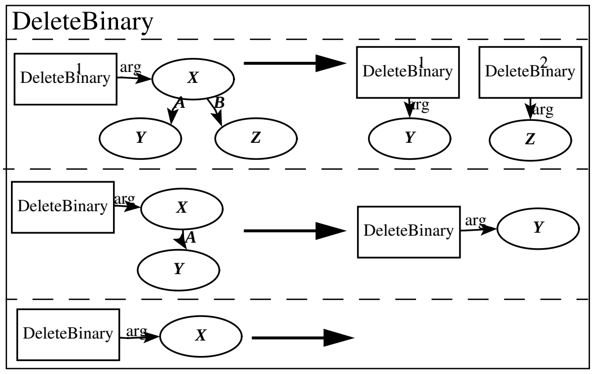


**Figure B.2: The transformation ‘DeleteBinary’**

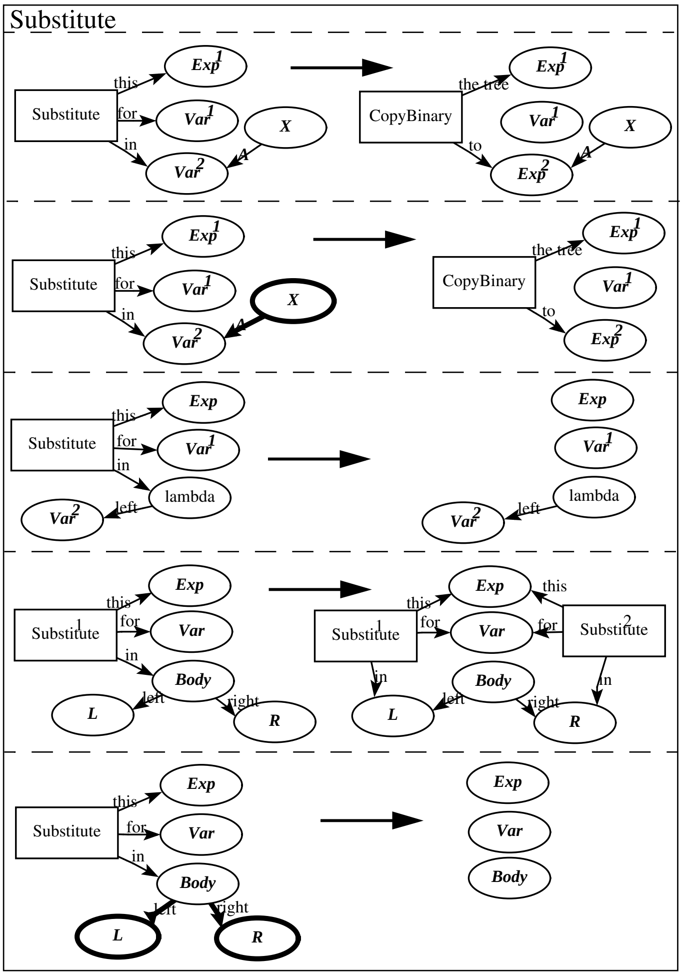


**Figure B.3: The transformation ‘Substitute’**

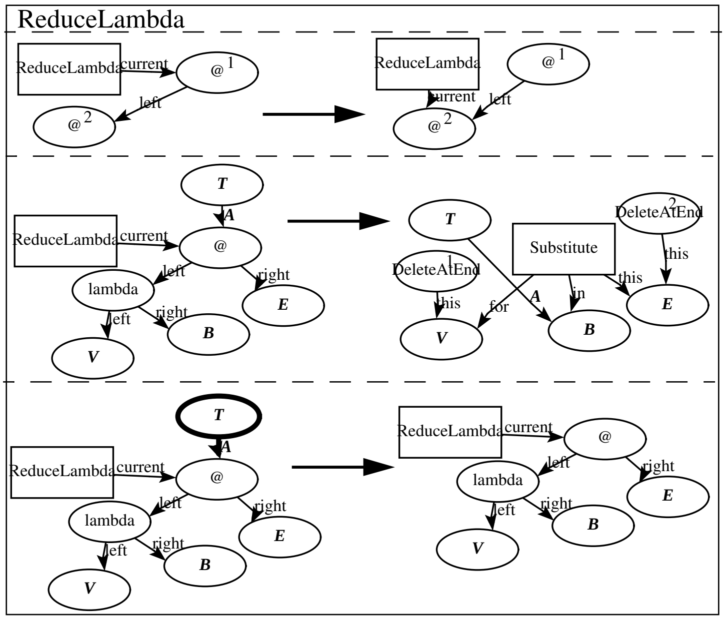


Figure B.4: The transformation 'ReduceLambda'

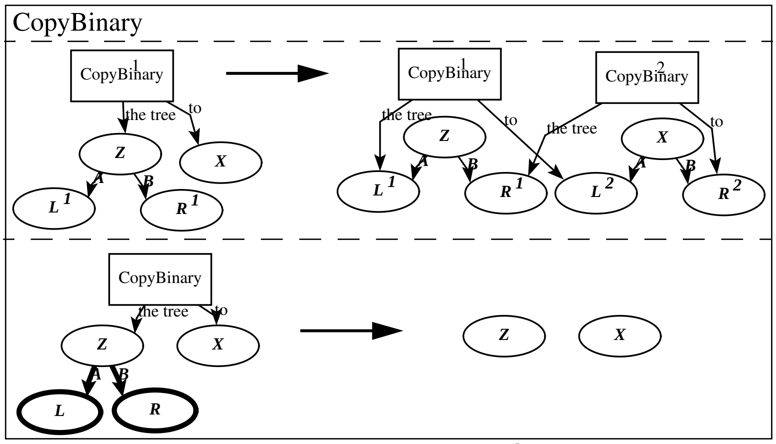


Figure B.5: The transformation 'CopyBinary'

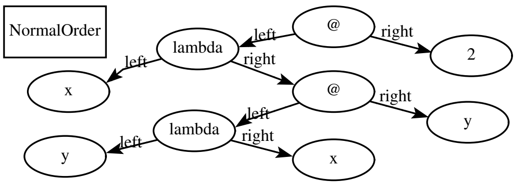

Figure B.6: The Application Graph with an example of using 'NormalOrder'

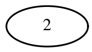

Figure B.7: The Resultant Application Graph

Figures B.1 to B.5 give the program to calculate the normal order reduction of a lambda calculus expression. A simple example expression is given in the application graph of figure B.6. with the result (a single node) in figure B.7. This program uses a pure version of the lambda calculus with constants, but it is easily extendable to include builtin functions such as 'if' and 'add' by adding appropriate reduction rewrites to the 'NormalOrder' transformation.

The implementation of this program takes less effort than would be required in a traditional textual language. This is due to two main features: the ability to make explicit changes in the lambda tree; and the rewriting nature of the lambda calculus reduction. The specifics of the reduction, such as requiring no substitutions of variables in lambda subexpressions with the same variable on the left, make the program less simple. Another major difference is the requirement to copy whole subtrees, which requires a Spider transformation to achieve.