GUIDE to DATA STRUCTURES and ALGORITHMS for a DATABASE of CATEGORIES Version 1 March 1996

M. Fleming R. Gunther
R. Rosebrugh *
Department of Mathematics and Computer Science
Mount Allison University
Sackville, NB E0A 3C0, Canada

This manual describes the data structures used to store categories and functors, and the algorithms which implement the tools for manipulating categories. It accompanies the User Guide for the Categories Database.

This document is available as
`ftp://sun1.mta.ca/pub/papers/rosebrugh/catdsalg.{dvi,tex}`.
The user guide is
`ftp://sun1.mta.ca/pub/papers/rosebrugh/catuser.{dvi,tex}`.
ANSI C source and Unix (Sun Sparc1+) executables of the programs are in
`ftp://sun1.mta.ca/pub/sources/rosebrugh/unix/category.exe`
For DOS excutables see
`ftp://sun1.mta.ca/pub/sources/rosebrugh/DOS/category.exe`

*This work was done while the first two authors were supported by NSERC Canada Undergraduate Summer Research Awards

1 Data Structures

1.1 Storing Categories

```
struct   {
       int       lhs[MAXWORD]
       int       rhs[MAXWORD]
  }    relation;

  struct    {
       char      name[MAXWORD]
       char      obj[MAXARR]
       int       arr[MAXARR][2]
       char      arrow[MAXARR]
       int       numobj
       int       numarr
       int       num_rel
       relation  *rel_set
  }    category;
```

Categories are stored using the two C structures shown above. The name of the category is stored as an array of characters with maximum length defined by `MAXWORD`. Each object of the category is represented by a unique integer value. This is done by storing the names of the objects in an array of characters, called `obj`, the object names are assumed to be single characters. The subscripts of the array are the integers that represent the objects, i.e. if A is stored in the array with subscript 2, `obj[2]`$= A$, then the integer 2 represents object A. The maximum number of objects that can be stored is equal to `MAXARR`.

The names of arrows of the category are stored in exactly the same way as the objects. The name of the array which holds the arrows is `arrow`. The maximum number of arrows that can be stored is equal to `MAXARR`. The domains and codomains of the arrows are stored in a two-dimensional array of integers. The array for this is `arr[MAXARR][2]`. Suppose that the arrow represented by 1 had domain A (represented by 2) and codomain B (represented by 4). Then `arr[1][0]` $= 2$ and `arr[1][1]` $= 4$. This example illustrates how this array holds the domains and codomains of all of the arrows.

The equations for a category are stored in the `relation` structure which is a part of the category structure. The equations are stored using the integers that represent the arrows.

There are two arrays to do this, `lhs[MAXWORD]` and `rhs[MAXWORD]`. `lhs[0]` holds the left hand side of the first equation and `rhs[0]` holds the right hand side of the first equation. Suppose the second equation is $ab = 1$, where 1 is the identity. Let a be represented by 1 and b be represented by 2. When stored, all of the equations are terminated by -1. The identity is stored simply by -1. This second equation would be stored as follows:

```
rel_set[1].lhs[0] = 1          rel_set[1].rhs[0] = -1
rel_set[1].lhs[1] = 2
rel_set[1].lhs[2] = -1
```

Thus, all of the equations for a category are easily stored. There is no maximum number of equations.

The number of objects, number of arrows, and number of equations are also stored under the variable names `numobj`, `numarr`, `num_rel`.

1.2 Storing Functors

```
struct    {

      char       name[MAXWORD]
      int        obj[MAXARR]
      int        arr[MAXARR][MAXWORD]
  }    cat_functor;
```

The name of a functor is stored in an array of characters, with the maximum length of a name being equal to `MAXWORD`.

The image of an object under the functor is stored in the array of integers called `obj`. If the functor sends object A to object C where A is represented by 1 and C is represented by 3 then `obj[1]` $= 3$.

The image of each arrow of the category under the functor is stored in the two-dimensional array of integers called `arr`. What is stored in this array depends on whether the functor has codomain a finitely presented category or the category $\mathbf{set}_0$ of finite sets. If the functor's codomain is a finitely presented category, then in `arr`, the first subscript is the integer representing an arrow and the array corresponding to this subscript contains the image arrow

under the functor, which may be represented by a path. For instance, suppose arrow a (represented by 1), goes to the path cd (represented by 23) under the functor. Then `arr[1][0] = 2, arr[1][1] = 3`, and `arr[1][2] =` -1. If the functor has codomain the category **set**, then the first subscript is the integer representing an arrow and the array corresponding to this subscript contains the image function in **set**. For example, if a (represented by 1) has domain A and codomain B and A has image the object 2 in **set** and B has image the object 3 in **set** then if the image of a under the functor is the function given by $1 \mapsto 2$ and $2 \mapsto 3$ then this information would be stored as:

`arr[1][0] = 2, arr[1][1] = 3.`

1.3 Trees

```
struct    nodetype  {
  struct    nodetype  *down
  struct    nodetype  *across
  struct    nodetype  *tree
  datatype  data
  bool flag
  int  info
  }
```

This is a tree structure where each node of the tree can have any number of children. The first child of a node is pointed to by the down pointer. The second child is pointed to by the across pointer of the first child, the third by the across pointer of the second child, and so on. The across pointer of the last child of a node points back up to the node it is a child of. The flag variable of a node indicates whether or not it is the last child of its parent node. If the value of the flag is 1, then it is not the last child and its across pointer points to another child, if the value of the flag is 0, then this node is the last child of its parent and its across pointer points to its parent.

The tree pointer of a node determines whether or not the node is the last node of a path in the tree. If the tree pointer is `NULL` then it is not the last node of a path, otherwise it is the last node of a path and its tree pointer points to another tree which contains all of the objects associated with the path that it is the last node of. The `info` variable is used as a check to see whether an object has been checked already. If it equals 0, then the object has not been checked yet and if it equals 1 then the object has been checked. The `datatype` is

what is to be stored in each node of a tree. In our program it is just an integer. Each tree is created with a root node that has a value of -1 as its `data`.

1.4 Storing Left Kan Extensions

The tree structure described in 1.3 is also used to store Left Kan Extensions. The Left Kan information is stored in a series of "tables". Each table is stored as a pointer to a `nodetype`. Section 2.11 contains complete information on the purpose of these tables. With the exception of the epsilon-tables, the root node of the tree contains as its `data` the number of rows in the table. Each move "down" the tree is equivalent to moving down the left column of the table, while a move "across" the tree is equivalent to moving across a row of the table.

The coincidences described in Section 2.11 are simply stored in a list.

1.5 Storing Right Kan Extensions

```
struct    big_list  {
  struct    rkan_list *head
  struct    rkan_list *tail
  }

struct    rkan_list {
  struct    rkan_node *head
  struct    rkan_node *tail
  struct    eonodetype     *paths
  struct    rkan_list *next
  }

struct    rkan_node {
  struct    b_node    *head
  struct    b_node    *tail
  }

struct    b_list    {
  struct    b_node    *head
```

```
  struct    b_node    *tail
  }

struct    b_node    {
  struct    pointer_list   p_list
  struct    b_node    *next
  }

struct    pointer_list   {
  struct    pointer_node   *head
  struct    pointer_node   *tail
  }

struct    pointer_node   {
  struct    rkan_node *p
  struct    pointer_node   *next
  }
```

`Big_list` is a list of `rkan_lists`. There is an `rkan_list` for each object in the product tree. An `rkan_list` has a pointer to a tree which contains all of the paths out of the object in the product tree that the `rkan_list` is associated with. An `rkan_list` is a list of `rkan_nodes`, where each `rkan_node` contains an integer and a pointer to a `b_list`. Suppose (β, B) is an object in the product tree and $X(B) = 3$ in **set**. Then, the integers in the `rkan_nodes` of the `rkan_list` associated with that object would be 1, 2, and 3. A `b_list` is a list of `b_nodes`; there is a `b_node` for each `rkan_list` in the `Big_list` after the current `rkan_list`. Each `b_node` contains a pointer to a `p_list`, which is a list of `pointer_nodes`. A `pointer_node` contains `p`, a pointer to a `rkan_node`.

1.6 Lists of Categories and Functors

```
   Category list:
struct    {
     struct    category_node  *head
     struct    category_node  *tail
     }    category_list;
```

```
struct    category_node  {
     category  cat
     struct    category_node  *next
     int saved
     }
```

This structure is used to store a number of categories. Each node of the list contains a category and an integer variable named `saved`. If `saved` has a value of 0, then the category of that node has not been saved to disk, if `saved` has a value of 1 then the category has been saved to disk.

```
   Functor list:
struct    {
     struct functor_node *head
     struct functor_node *tail
     }    functor_list;

struct    functor_node   {
     cat_functor    f
     struct    functor_node   *next
     int  saved
     category  A
     category  B
     int  functor_type
     }
```

The `functor_list` structure is a list of `functor_node`s. The `functor_node`s contain all of the information for a functor. A node contains the functor (`f`), the domain category of the functor (`A`) and the codomain category (`B`). A node also contains an integer variable, `saved`, which is set to 0 if the functor has not been saved on disk and 1 if it has been saved on disk. The integer variable `functor_type` has a value of 0 if the functor is a functor between finitely presented categories. It has a value of 1 if the functor is a functor from a finitely presented category to the category of sets. If the functor goes to **set** then the category variable `B` contains nothing.

2 The Algorithms

2.1 Make Confluent

The functions in the file `confluen.c` are used to take a set of reductions, or equations directed here from left to right, in a category and add any necessary reductions to make the set confluent. That is to say that a path of generating arrows in the category will reduce to one and only one normal form, regardless of the manner in which it is reduced. This is accomplished primarily by the use of the overlap and subpath rules of the Knuth-Bendix algorithm. The main function, `make_confluent`, uses three other functions, `overlap`, `reduce_overlap` and `reduce_subpath`. The functions `reduce_overlap` and `reduce_subpath` use yet another function, `add_relation`.

`make_confluent`
The main operation of this function involves a double loop which looks at every pair of relations in the category. For each of these pairs, the following is done:

1. Check for any overlap between the left sides of the two relations. This is done by calling the function `overlap`, which returns the size of the overlap, or -1 if there is none.

2. If there is an overlap, call function `reduce_overlap` to add reductions to the category if this is necessary to make the set of reductions confluent.

3. Check to see if the left side of one of the relations is a subpath of the left side of the other relation. If so, call function `reduce_subpath` to add reductions to the category if necessary.

4. Once this has been done for all pairs of relations, the set of reductions is confluent.

`reduce_overlap`
This function accepts two reductions (call them $u_1 \longrightarrow u_2$ and $v_1 \longrightarrow v_2$), and checks to see if they satisfy the overlap rule of the Knuth- Bendix procedure. If not, it adds any necessary reductions to make the set of reductions confluent.

1. An overlap occurs when $u_1 = xy$ and $u_2 = yz$ for some paths x, y, z. That is, the end of one relation's left side is identical to the beginning of the other relation's left side. (In this example, the overlap is y).

2. Construct two paths $r_1 = xv_2$ and $r_2 = v_1z$.

3. Reduce each of these to their normal forms using the existing relations.

4. If these two reduced forms (w_1 and w_2, respectively) are not equal, then adding the relation $w_1 = w_2$ to the set of reductions will help to make it confluent. The relation is added by calling the function `add_relation`.

`reduce_subpath`
This function takes two reductions ($u_1 \longrightarrow u_2$ and $v_1 \longrightarrow v_2$), and checks to see if they satisfy the subpath rule of the Knuth-Bendix procedure. If not, it adds any necessary reductions to make the set of reductions confluent.

1. A subpath occurs when $u_1 = xu_2y$ for paths x and y.

2. Construct two paths $r_1 = v_1$ and $r_2 = xv_2$.

3. Reduce each of these to their normal forms using the existing relations.

4. If these two reduced forms (w_1 and w_2) are not equal, then the relation $w_1 = w_2$ is added to the set of reductions to make it confluent. Again, this is done by calling the function `add_relation`.

Once `reduce_overlap` has been called for all pairs of relations having an overlap, and `reduce_subpath` has been called for all pairs having a subpath, the set of reductions will be confluent and the process will be complete.

2.2 Get functor arrows and check relations

For functors from category **A** to category **B**, whether **B** is a finite category or the category of finite sets, we must check that all relations in **A** also hold in **B** under the functor. The process has two cases.

2.2.1 Functor to a finite category
The following algorithm makes use of two functions, namely `get_functor_arrows` and `check_functor_relations`.

1. For each arrow f in category **A**, the user is asked to enter the path in **B** that f is taken to by the functor.

2. Check to see that all arrows in the path entered are actually arrows in **B**.

3. Check that the path entered is composable, and that the path's domain and codomain are the images of f's domain and codomain under the functor.

4. For each relation in category **A**, find the paths that the left and right sides go to under the functor.

5. Reduce these paths. If the reduced forms are not equal for any relation, then the set of relations do not hold under the functor, and the functor is not valid.

6. If the reduced forms are equal for all relations in the category, then all relations hold and the functor is valid.

2.2.2 Functor to **set**

This algorithm also uses two functions, `get_sf_arrows` and `check_sf_relations`.

1. Loop through each arrow of the category A.

2. For each element of the arrow's domain, have the user enter which element in the codomain it is taken to by the **set** functor.

3. For example, consider the case where f is an arrow from object M to object N, and X is a functor from category A to **set**. If X(M) = [3] and X(N) = [4], then for arrow f, the user will be prompted for all 3 elements in the domain to enter an element in the codomain (a number from 1 to 4 in this case).

4. Loop through each relation in the category.

5. Get the domain of the left and right sides of the relation.

6. Call the function `function_value` (discussed below) to calculate the element of the codomain that each element of the domain is taken to under the functor by the paths on the left and right sides.

7. For example, suppose we are looking at a relation fhk = gm. The two paths fhk and gm must both have the same domain, say C, and suppose that X(C) = [2]. Then, we must check each element in [2], i.e.: we must check that X(fhk)(1) = X(gm)(1) and X(fhk)(2) = X(gm)(2).

8. If this equality does not hold in any case, then this relation does not hold under the functor, and the functor is not valid.

9. If all relations hold, then the functor is valid and it is stored in memory.

`function_value`
This function is best described using an example. Suppose once again that we are looking at the relation $fhk = gm$, and that we are checking it for the element 1. `function_value` will be called twice, once for fhk and once for gm.

Consider the case of fhk. `function_value` will first take $X(k)$ and apply it to 1. Then, $X(h)$ is applied to this element, and finally $X(f)$ is applied. The result returned will be an element of the codomain of fhk.

If the values returned for fhk and gm are equal, then this relation holds; if the values are different, the relation does not hold.

2.3 Check if an object is initial

The algorithm that we use to determine whether or not an object is initial uses two functions, `initial_object` and `check_initial`.

`initial_object`

1. Ask the user to enter the object to be tested.

2. Check that the object exists in the category.

3. Call `check_initial`. If there is an object in the category with more than one path into it from the test object, then `check_initial` will return false, and the object is not initial.

4. If `check_initial` returns true, then check that each object has one path into it from the test object. If there is an object with no path into it, then the test object is not initial.

5. If all objects are found to have exactly one path into them from the test object, then the test object is initial.

`check_initial`
This is a recursive procedure which makes use of the algorithm to go through a category, tracing all loop-free paths from the test object. For each path, the following is done:

1. If the path is the first path encountered with its codomain, then store that path in the array location reserved for that codomain.

2. If it is not the first path, then the following must be done:

   - If the codomain is the same as the test object, reduce the path and compare it to the identity arrow on that object. If the path equals the identity, continue on with the next path. If the path does not equal the identity, the test object has more than one path to itself and is therefore not initial. Display a message, and return false.

   - If the codomain is some object other than the test object, reduce the path and compare it to the path previously stored for that codomain. If they are equal, continue on with the next path. If the current path does not equal the stored path, then the test object has more than one path to this object and is therefore not initial. Display a message, and return false.

2.4 Make Product Tree

The product tree is built to contain all objects in the category B/F, for B an object of $\mathbf{B}$, that will be used in the product to build the right kan extension at B.

- The function `get_all_betas` is a recursive function which goes through a tree of βs (objects in the category B/F), and gets each β. One of the operations performed on each of these βs is adding it to the product tree using the tree function `search_and_add`. This is how the product tree is originally constructed.

- `eliminate_betas` is a recursive function which takes an arrow $\beta : B \longrightarrow F(A)$. It finds all paths $\alpha : A \longrightarrow A'$ and eliminates $F(\alpha)\beta$ from the `B_tree`. At the same time, it removes $F(\alpha)\beta$ from the product tree if it is there. This is accomplished by going to the last node of the path in the product tree, and setting the tree pointer to `NULL`.

- After all such paths have been eliminated, what is left in the product tree are all of the paths (objects in B/F) needed for the product to be used in the right kan extension.

2.5 Check for sum

The function `sum` is used to get user input about the sum to be tested. It then tests this sum by calling the function `is_sum`, which in turns calls three other functions: `check_oneone`, `check_onto` and `make_rhotree`.

`sum`

1. Prompt the user to enter the object of the sum.
2. Check that this is in fact an object in the category.
3. Prompt the user to enter α and β, the two paths into the object.
4. For each of these paths, check that each arrow in the path is an arrow in the category, check that the domains and codomains of consecutive arrows match up (i.e.: that the path is composable), and check that the path has the object of the sum as its codomain.
5. Prompt the user to enter the maximum number of visits to an object during the sum checking.
6. Call function `is_sum` to determine if the object and paths represent a sum in the category, and display an appropraite message.

`is_sum`

1. Loop through each object in the category.
2. If the current object is the test object, the identity is a γ (a path from the test object to the current object), so add α and β to the tree of $\gamma\alpha$s and $\gamma\beta$s, and add the identity to the tree of γs.
3. Call function `check_oneone`. If it returns false, then we do not have a sum, and `is_sum` can return false without any further testing.
4. If `check_oneone` returns true, then checking continues. Get the domain of the path β. Call function `make_rhotree` to construct the tree of paths from the domain of β to the current object (from the loop). If domain of β is the current object, then add the identity to the rho tree.

5. If the rho tree is not empty, then we must check onto. If the identity is not in the tree of $\gamma\alpha$s and $\gamma\beta$s, or if the subtree of the identity is not identical to the rho tree, then it is not onto, and `is_sum` can return false without any further checking.

6. Otherwise, call function `check_onto`. If it returns false, then we do not have a sum, and `is_sum` returns false; otherwise, `is_sum` returns true.

`check_oneone`

1. Use the algorithm to find all paths out of test object.

2. If the codomain of the path is the destination object, then add the reduced form of the path to the tree of γs.

3. Construct $\gamma\alpha$ and $\gamma\beta$ by appending α and β to the path, and reduce them.

4. If the pair $(\gamma\alpha, \gamma\beta)$ is already in the tree, then it is not one-one and return false. Otherwise, add $\gamma\alpha$ to the tree of $\gamma\alpha$s, and add the $\gamma\beta$ to the subtree of the $\gamma\alpha$, and continue with the next path.

5. If all paths are tested without finding that the sum is not one-one, then it is one-one, and return true.

`make_rhotree`
Creates a tree from one object to another by using the algorithm to go through a category to get every path out of an object. If a path has the desired codomain, adds it to the tree using the tree function `search_and_add`.

`check_onto`

1. Use the algorithm to go through a category and get all paths. For each path, do the following:

   - Reduce the path and search for it in the tree of $\gamma\alpha$s and $\gamma\beta$s. If it is not there, then it is not onto, and return false.

   - If the path is in the tree, then check that its subtree is identical to the rho tree. If not, return false; otherwise, continue with the next path.

2. If all paths are traversed without finding that the sum is not onto, then it is onto, and return true.

2.6 Algorithm to find all paths in a category:

This algorithm is used extensively throughout the program. It is a recursive algorithm that finds paths out of a given object. The recursive function has three main parameters passed into it. One keeps track of the current path, the other is the current domain, and the third is an array that keeps track of how many times each object has been visited during the current path. The user sets how many times an object can be visited in a path using the `Change maximum order of endomorphisms` option in the program's main menu. The first thing the function does is check to see that the object which is the current domain has not already been visited the maximum number of times in the current path. If it has already been visited the maximum number of times then backtrack to the previous arrow of the path(i.e. return to the function that called the current function), otherwise find all of the arrows out of the current domain and have a for loop which does the following each time through. It increments the number of times the object which is the current domain has been visited, adds the next arrow that is out of the current domain to the current path, finds the codomain of this arrow and calls the recursive function again passing in the codomain as the current domain. Then, right after the function has been called again the new arrow is removed from the current path and the number of times that the object which is the current domain has been visited is decremented. This ends the for loop. This algorithm allows us to recursively visit the the paths of a category. It is used throughout our program and often after each new path is found some kind of checking on this path will be done.

2.7 Algorithm to make the B/F tree:

This algorithm stores the objects and arrows of the category B/F in a tree. It uses the following functions. `make_beta_objects`, `go_through_betas`, `get_all_arrows`, `make_beta_arrows`. The functions `go_through_betas` and `get_all_arrows` are used in the function `make_beta_arrows`.

`make_beta_objects`

1. recursively trace through the category that B is an object of and find all of the paths out of B.

2. for each path out of B do the following: go through all the objects in domain the category of F and for each one of these objects, A, check to see if $F(A)$ equals the codomain of the loop free path. if it does then add the path to the B/F tree (if it is not already there) and set the tree pointer of the last node of this path to point to another tree where the A's that make up objects with this path will be stored.

Thus each object, $< \beta, A >$ is stored in the B/F tree which is now a tree of trees.

`make_beta_arrows`

1. recursively goes through the B/F tree and finds all of the objects in the tree. (an object is denoted by $< \beta, A >$)
2. for each $< \beta, A >$ the function `get_all_arrows` is called.

This function will find all of the arrows out of the object $< \beta, A >$ and store them in a tree. The tree pointer of the node which contains the A of $< \beta, A >$ will then be set to point to this tree of paths. Thus all of the arrows of the category B/F have been stored in the B/F tree which has now become a tree of trees of trees.

`get_all_arrows`

1. goes through the domain category of F and finds all of the paths out of the object A from $< \beta, A >$.
2. for each path the function `go_through_betas` is called.

`go_through_betas` does the following:

1. recursively goes through the B/F tree and finds all of the objects in the tree (an object is denoted by $< \beta', A' >$)
2. for each $< \beta', A' >$ it checks to see whether A' is the codomain of the path found in `get_all_arrows`. If A' is the codomain of this path then the function checks to see if $F(path)\beta = \beta'$. If this equation holds then we store the path in the tree pointed to by the tree pointer of the node of A in $< \beta, A >$. We also have the tree pointer of the last node of the path point to the node which contains A' of $< \beta, A' >$. In this way each path has a pointer to its codomain.

2.8 Algorithm to add to a tree:

This algorithm uses the functions, `search_and_add`, `search_level`, and `add_node` to add a path to a tree.

`search_and_add`
Call `search_level` to check if the current arrow of the path is in the present level of the

tree. If it is not, then this arrow and all the rest of the path is not in the tree so for each arrow left to add to the tree we call the function `add_node` and add the arrow to the tree.

`search_level`
Keep going across a level of the tree and return true if the arrow is found and false if at the end of the level and the arrow has not been found.

`add_node`

1. creates a new node and adds it to the beginning of a level (i.e. it is the node pointed to by the parent node of the level)

2. then we go through the level and sort it alphabetically. We do this by exchanging the information stored in each node.

2.9 Algorithm to create R(B) tree:

This algorithm uses the functions `make_big_list`, `assign_pointers`, `check_pointers`, `find_p1`, `find_p2`, `test_self`, `test_self2`, `display_RB_rho`, and `find_RofB`.

`make_big_list`: This is a recursive function that goes through the product tree and finds all of the objects stored in the product tree. For each object, it finds $X(A)$ for that object. It then makes a list for that object with the integers from 1 to n stored in the nodes of the list where $X(A) = n$. A pointer to the tree which contains all of the arrows from that object is also stored in the list structure. This list is added to the `big_list`.

`assign_pointers`:
This function goes through the `big_list` to access each `rkan_node`. For each of these `rkan_nodes`, it allocates space for a list of lists containing pointers to the `rkan_nodes` which follow it in the `big_list`.

`check_pointers`:
This function tests all pairs of objects in the `big_list`. For each pair of objects, it calls `find_p1` to find all paths out of the first object. For each path out of the first object, it calls `find_p2` to find all paths out of the second object with the same codomain as the path out of the first object. It then tests each pair of paths for all the pairs of `rkan_nodes`, one from each object. If $X(p_1)(\alpha) <> X(p_2)(\alpha')$, where p_1 and p_2 are the paths, α is an integer of an `rkan_node` of the first object and α' is an integer of an `rkan_node` of the second object, then remove the pointer from the `rkan_node` containing α to the `rkan_node` containing α'. The functions `test_self` and `test_self2` serve the same purpose as `find_p1` and `find_p2`,

except that they compare paths out of the **same** object which have a common codomain.

`display_RB_rho`
This is the first of two functions which goes through the `big_list` and finds all n-tuples (where n is the number of objects in the `big_list`) for which every element has a pointer to every other element that follows it in the n-tuple. This function displays R(B), for an object B in $\mathbf{B}$, along with information about the natural transformation ρ for each object A in $\mathbf{A}$where $B = FA$.

`find_RofB`:
This function is similar to `display_RB_rho`. However, instead of information about ρ, this function displays the action of the Right Kan extension R on all arrows out of the object B. The next section discusses in detail how these calculations are done.

2.10 Algorithm to calculate R(arrows):

A number of functions in the program are used in this procedure: `display_images`, `traverse_cod_tree`, `traverse_dom_tree`, and `alt_dom_tree`.

The process is best illustrated with an example. Suppose we are dealing with an object B in $\mathbf{B}$, and suppose that there is an arrow $f : B \to B'$ in $\mathbf{B}$.

`display_images`:
This function is called by `find_RofB` and is used to call the other functions.

`traverse_cod_tree`:
Consider an arrow $f : B \to B'$. This function picks out each object $\beta' : B' \to FA'$ in B'/F used to index the tuples in $R(B')$. It then calls `traverse_dom_tree`. If `traverse_dom_tree` is not successful, it calls `alt_dom_tree`.

`traverse_dom_tree`:
When called by `traverse_cod_tree`, this function tests to see if $\beta' f$ is equal to some β in B/F. If it is, then the β'th element of the image tuple in $R(B')$ is equal to the $beta$th element of the tuple in $R(B)$, and we are finished.

`alt_dom_tree`:
If `traverse_dom_tree` was unsuccessful, this function is called. $\beta' f$ is a path from B to FA', but it was not found in the B/F tree. This means that it was removed from the B/F tree because it had a path into it from some other object in B/F. This function looks for an object $\beta : B \to FC$ in B/F such that there is a path $\gamma : C \to A'$ such that $F(\gamma)\beta = \beta' f$. If this is found, then we define $s'_\beta = X(\gamma)(t_\beta)$ where s'_β is the β'th element of the image tuple

in $R(B')$ and t'_β is the βth element of the tuple in $R(B)$.

2.11 Left Kan Extensions

The algorithm used to compute Left Kan Extensions is the generalized Todd-Coxeter procedure. The following tables are stored as trees (see section 1.4):

(ϵ-tables) For all A in **A** there is a table to store information about the natural transformation $\epsilon : XA \to LFA$.

```
    ϵA


XA-->LFA
 1
 2
 :
 :
 n
```

(L-tables) These tables store the elements of $L(B)$ along with the action of L on each arrow in **B** that had B as domain.

```
    Lg  Lh
LB  LB´ LB"
 :
 :
 :
```

(relation-tables) For each relation $g_n.....g_1 = h_m.....h_1 : B \to B'$, there is a table of the form

```
  Lg1   Lgn      |   Lh1   Lhm
LB-->...-->LB´   |  LB-->...-->LB´
|
     ...         |        ...
```

(Naturality tables) For all generating arrows $f : A_1 \to A_2$ in **A**, there is a table of the following type, where $Ff = g_n.....g_1$, and `eA1, eA2` represent ϵ_{A1} and ϵ_{A_2}.

```
   Xf    eA2       |    eA1   Lg1    Lgn
XA1-->XA2-->LFA2  | XA1-->LFA1-->...-->LFA2
 1     5      1    |               ...
 2     3      2    |               ...
 :     :      :    |               ...
 :     :      :    |               ...
 n     2      n    |               ...
```

In the following algorithm, a *coincidence* is something that occurs in a relation or naturality table when the last entry in the left side of a row and the last entry in the right side of the same row are different elements. These elements are to be made equal. A *coincidence* can also occur in an L-table when two different elements occur in the same column of a pair of rows which have already been found to be a coincidence.

Filling in consequences means to use data from the ϵ-tables and L-tables to fill in the relation and naturality tables.

Defining a new element means, given an empty entry in a table in a column headed by LB, to fill the entry with the lowest integer not already defined in LB.

The general algorithm is:

```
   initialize tables
   while there are empty entries in a table
{
define new elements
fill in consequences
while there are coincidences
   {
   deal with coincidences
   fill in consequences
   }
}
```

Dealing with coincidences means, given a coincidence of the elements x and y under LB, to do the following:

1. Add any coincidences that arise as a result of this coincidence.

2. Replace y with x anywhere it appears under LB in any of the tables.

3. Remove the row headed by y from the L-table for LB and from any relations with LB as domain.

After everything is finished, the elements in each set $L(B)$ are renumbered so that they form a set of consecutive integers $1, 2, ..., n$.

The following are the main functions used in this section of the program:

`add_to_e_table`: adds an element to an ϵ-table.

`update_nat`: updates naturality tables after a change has been made to an ϵ-table.

`define_L`: defines new elements in an L-table.

`fill_conseq_rel, fill_conseq_nat`: fill relation and naturality tables with any information that is available in the ϵ-tables and L-tables.

`check_coinc`: checks a relation or naturality table for any coincidences that might occur. This function calls `add_coinc` to add any such coincidences to the list.

`deal_with_coinc`: eliminates coincidences using the algorithm discussed above. This function calls a number of others to accomplish this, including `update_e, update_L, check_coinc2, delete_row, delete_rel_row` and `update_coinc_list`.

`renumber`: performs the renumbering of $L(B)$'s.