Eldorado ins and outs : specifications of a data base management toolkit according to the functional model

87/11

**Eldorado ins and outs.
Specifications of a data base
management toolkit according to
the functional model.**

by

Pim Lemmens.

june 1987

COMPUTING SCIENCE NOTES

This is a series of notes of the Computing Science Section of the Department of Mathematics and Computing Science of Eindhoven University of Technology.
Since many of these notes are preliminary versions or may be published elsewhere, they have a limited distribution only and are not for review.
Copies of these notes are available from the author or the editor.

Eindhoven University of Technology
Department of Mathematics and Computing Science
P.O. Box 513
5600 MB EINDHOVEN
The Netherlands

editor: F.A.J. van Neerven

Eldorado ins and outs.

Specifications of a Data Base Management Toolkit according to the Functional Model.

by Pim Lemmens

Contents:

1. The functional model.

"At the end of the passageway, a small room is lit by a single candle standing on a wooden table. The candlelight reveals a murky room, but one used as a place of residence, judging by the furniture scattered about. Seated at the table is a spindly creature whose attention is focussed on a Glass Orb standing on a plinth. The creature is mumbling something at the Orb. Shapes and colours are swirling across its surface, but you cannot make out anything clearly...."

(from: Steve Jackson "The Seven Serpents", Adventure Gamebooks, Penguin Books, 1984)

An adventure game constitutes a world of its own, inhabited by weird creatures following strange laws. It may often be found inside a computer, where it may be entered by people striving to be "Grand Master of Adventure". An adventure computer contains many facts about the adventure world in such a way that they may easily be retrieved. In fact, it contains a data base system that has been filled with many items, like spindly sorceresses, Glass Orbs and strange rooms. It also contains relations among these items, like: a sorceress may use a Glass Orb to see you approach, so it won't be any use for you to try and sneak up to her.

In general we may say: a data base represents a model of some world composed of a collection of item representations (data base objects), that may have a value of some type, like names or numbers, and a collection of representations of relations among these objects. The collection of objects of the data base may be subdivided into a number of classes. In our adventure world we have actors (you, the sorceress), (material) objects (the Glass Orb, a magic sword), locations (the small room, a forest) and activities (move to some place, ask a question, pick up an object). Between these classes we have the relations: The Glass Orb (an object) is inside the small room (a location); the sorceress (an actor) consults (an activity) the Glass Orb. A relation links in a specific way two or more items, each from a specific class. Objects are linked to locations by the relation "is located at". Actors, activities and objects are linked by the relation "action performed by actor using object".

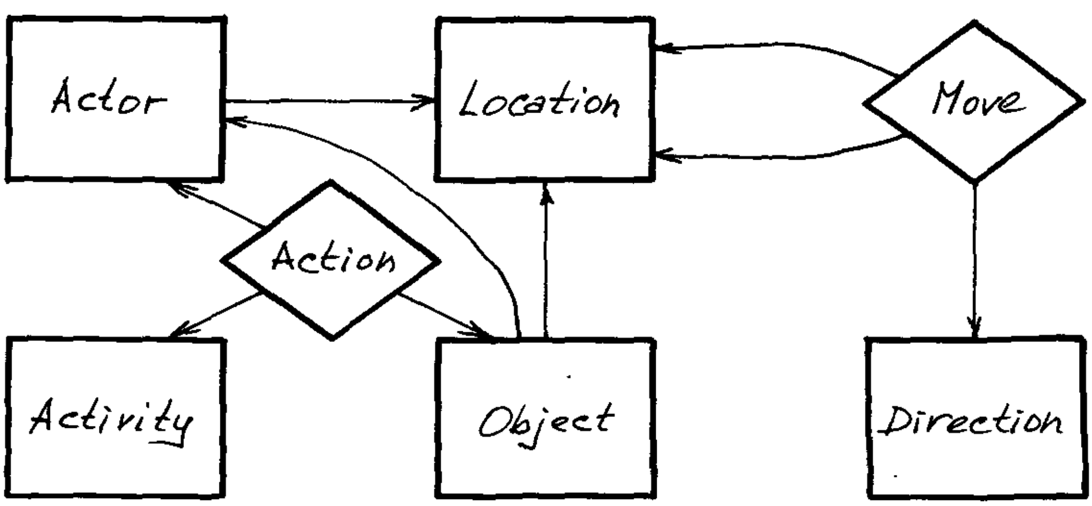
Many data base systems contain a large number of facts about their object system. In order to make these facts accessable the collection of facts is structured according to some data model. There are several different data modeling techniques, such as the relational model (Codd,) and the entity-relationship model (Chen,). We prefer the functional model, because of its combination of simplicity and flexibility.

The entities that play a role in a functional model are categories and functions. A class like "object", which contains such things as "Glass Orb", "Magic Sword", "Emerald Bracelet", will be called a category, if it contains each of these things only once (e.g. no two emerald bracelets in the object category). and all of its items may have the same pattern of relationships to items from other categories (or the same category). So a specific object is not necessarily involved in a specific relation, but if it is, it should be with items from the same categories as the other objects involved. In our world there may of course be two emerald bracelets, but these may be represented by an indication of their kind ("emerald bracelet") and two instantiations of this kind (the actual bracelets, e.g. indicated by the numbers 1 and 2), with kind and instances linked by a relation.

A function links two categories, respectively called the domain and the range category of the function. For each element from the domain category the function contains at most one pair <a, b> linking element a from the domain category to element b from the range category. Functions realise binary n-to-one relationships. But also ternary, or more generally, n-ary relationships and m-to-n relationships may be represented using functions. For that purpose we need so-called ghost categories, categories of items that don't have any value. The ternary relation "action performed by person using object", for example, may be realised using functions by introducing a category "action" that contains an item for every single action, and three functions linking this ghost category to "actor", "activity" and "object", respectively.

One kind of action is the "move" action: we may move from one location to another. The place where we arrive will be dependent upon the direction in which we left our previous location. So from some location we may reach more than one new location. On the other hand, a certain location may be reached from several other locations. Here we have a m : n relationship between "location" and "location" itself. Again a ghost category may be of help: A category "move" with an item for every from-to combination, together with two functions, one called "from" and one called "to", both linking "move" to "location".

Now we may draw a schema of the data base in which categories are represented by rectangles and functions by arrows between categories, pointing from the domain category to the range category. We have enhanced the schema described above by adding an extra category "direction" and a function from "move" to "direction". We further introduced functions indicating the location of actors and objects and the ownership of objects by actors.



Adventure Data Base

So a (functional) data base contains a (possibly large) set of data items that are uniquely identifiable - no item will occur twice in the data base. Many items have connections to other items, according to certain rules. An item may, or may not, be associated to a data object of a certain type. It may be retrieved from the data base on account of its data value or because of its connection to other items.

Items are grouped into categories. A category is a set of items of the same type, that is: associated with the same type of data object, and with the same pattern of connections to other items. Categories are linked by functions. A function is a set of connections among items. Potentially, all members of a category may be connected to items of a category that has a function link to that category.

More formally:

A data base state may be described by a 3-tuple:

<Obj, V, Link>

Where Obj = a set of indices
V = a (partial) function: indices -> values
Link = a ternary relation: labels * indices * indices

Each link is characterised (uniquely identifiable) by a label, a 'from' object and a 'to' object.

A data base skeleton according to the functional model is a 5-tuple:

DB = <Ci, Fi, D, R, T>

in which Ci and Fi are sets, indicating respectively: Category indices and function indices.
D and R are the domain function and the range function of the functions and T is a function that links each category name to a set of possible values, its type. This set may be empty.

$\forall$ y $\in$ Fi: D(y) $\in$ Ci AND R(y) $\in$ Ci

Some of the symbols used here and in the following sections are:

$\forall$ for the universal quantifier,
$\exists$ for the existential quantifier,
$\in$ for the set membership operator,
c for the subset operator,
i for the intersection operator,
u for the union operator,
AND, OR and XOR (= exclusive or) refer to the logical operators of these names.

A data base state in the functional model may be described by a pair of functions:

$$\langle c, f \rangle,$$

where c and f correspond to a partioning of Obj and Link, respectively.

$$\forall\, x \in Ci: c(x) \subseteq Obj$$
$$\quad \text{AND } \forall\, x1 \in Ci: \forall\, x2 \in Ci: x1 \neq x2 \Rightarrow c(x1) \cap c(x2) = \emptyset$$

f(y) is a projection of a selection of Link:

$$s = \{ \langle l, y \rangle \mid l \in labels \text{ AND } y \in Fi \} \text{ is a function}$$
$$\forall\, y \in Fi:$$
$$\quad \exists\, l \in labels: y = s(l)$$
$$\quad\quad \text{AND } f(y) = \{ \langle i1, i2 \rangle \mid \langle l, i1, i2 \rangle \in Link \text{ AND } i1 \in c(D(y)) \text{ AND } (i2 \in c(R(y)) \text{ OR } i2 = NIL) \}$$
$$\quad\quad \text{AND } f(y) \text{ is a function.}$$

Most applications need more than one data base. First there is the dictionary, or metadata base, that contains a description of the structure of the data base proper. Here we find the categories "category name", "type", "function name", and the functions "function domain", "function range", "category type", together with data about the way these are recorded in the data base.

Our adventure base may further contain, alongside the facts base described above, a rules base where we can find which actions are legal and which are not in a given situation. Many data base systems have such a 'rules base' in one form or another to hold the constraints to be observed by update operations.

And finally there is information about which commands to use for a given operation or which options are available in a given situation. This user manual or 'help' information may also be stored in a separate data base.

2. The ELDORADO System.

Eldorado (Ensemble of Loosely-connected Devices for Observation, Removal and Addition of Data base Objects) is a toolkit for the implementation of functional data bases. It manages data that are structured along the lines described above. For that purpose it offers a system of data structures and operations to be used in application programs or interpreters for DML-languages. Important extra features are ordering of data within a category and the possibility to add extensions to the data that may be retrieved by the system.

In the Eldorado system the values of the items within a category are ordered: There is a "greater than" / "smaller than" relation between any two values of items from a specific category. The ordering relation is type dependent. So if two values from one category each have equal counterparts in an other category of the same type, they will in both categories have the same ordering relation.

Apart from the typed data object, an item may also have an association to an untyped amount of data, the extension. This extension may be produced as a by-product of retrieval of the item from the data base.

EXT is a function,
with $\forall\ k \in c(x)$: EXT(k) = NIL OR EXT(k) $\in$ EF,
and EF is a set of untyped data.

The extension data may be of any kind: Text or graphics or even program code. Its use will be determined by the application and is of no concern to the DBMS.

The data structures and operations summed up below represent a choice from many possibilities. The considerations that led to this choice are mainly:

1. Do they agree with the way of thinking of the user? Don't they introduce concepts that are unknown to him or force him into an "unnatural" pattern of actions?

2. Are they of a sufficiently general nature? May all foreseeable applications be realised by them?

3. Are they realisable? Are they able to realise all kinds of actions that have to be performed in a reasonably efficient way, with regard to processor time, use of memory and necessary programming effort?

It is difficult, if not impossible, to fully satisfy all these requirements. The choice we made is primarily directed towards generality, and secondarily towards efficiency of execution. For this reason we chose collective operations as much as possible. Retrieve and update operations are performed setwise and not one item at a time.

The first consideration may be satisfied by adding some kind of user interface - an interactive program or an interpreter for some query/DML language that matches the user view of the data model. These programs should be constructed as a layer covering the ELDORADO system, using the operations and data structures described here. A proposal for an interactive user interface is given in chapter 5.

3. Temporary data structures

To allow full manipulation of its data, the Eldorado system offers the following structures in addition to the categories and functions from the data base:

- **Atoms**, which are the building blocks of the structures mentioned above. Atoms come in several possible types: integers, reals or strings, which are the types that are also used for the objects in the data base, booleans, to be used for expressions that evaluate to true or false, empty, denoting objects without a value, and refs.

A ref indicates an item in the data base. Every item will be uniquely identified by a ref. In fact, a category is a collection of refs, even the so-called ghost categories that do not contain any integer, real or string values. int and rl indicate, respectively, integer and real elements. wd elements are strings of some fixed length.

- **Sets**, which may be considered as temporary categories. A set may contain an extract of a specific category, or data to be added to it. A set may be empty or the type of its elements is either int, rl, wd or ref.

A set may contain refs, in which case it indicates a subset of some category, or it may be composed of integer, real or string values.

- **Tfunctions**, or temporary functions, to be used for extracts or updates of functions, among other things.

Functions and tfunctions may be considered as sets of pairs of refs that link two sets of items.

- **Tables**, which correspond to the relations of the relational model, to be used for the presentation of data from the data base in a surveyable form.

A table may be considered as a function that links a set of attribute names to a set of tfunctions. All tfunctions of a table should have the same domain.

Sets, tfunctions and tables are built from atoms or pairs of atoms and can not be used to construct more complex objects recursively. Things like sets of tfunctions or tables that have a complete set as an attribute value are not admitted by the system.

4. DBMS Operations.

Now that we have established the basic data structures of our system, we need a number of operations for the conversion of one structure to another, or one type to another or the transformation of certain values into others. In principle there are many possible choices of operation collections, but we want to concentrate on collective operations on categories and functions. So in stead of fetching one element from a category, processing it and then fetching the next one to repeat the processing on, we extract a whole set of candidate elements from the data base before processing them collectively. Addition, removal and retrieval of elements is performed setwise. Updating a function or a category means first building a set of elements to be updated and then do the update operation for the whole set.

Whereever possible, operations will be using refs instead of values of items. Only when, after a number of operations, the user needs the resulting values, they may be determined by a valuation operation.

Operations on atoms:

<u>Type(X)</u> = <u>int</u>

+, -, *, DIV, MOD	int * int -> int
=, ≠, <, ≤, >, ≥	int * int -> bool

<u>Type(X)</u> = <u>rl</u>

+, -, *, /	rl * rl -> rl
=, ≠, <, ≤, >, ≥	rl * rl -> bool

<u>Type(X)</u> = <u>wd</u>

=, ≠, <, ≤, >, ≥	wd * wd -> bool

<u>Type(X)</u> = <u>ref</u>

=, ≠	ref * ref -> bool
AtomVal	ref -> x: x ∈ {empty, int, rl, wd}
AtomExt	ref -> e ∈ EF

<u>Type(X)</u> = <u>bool</u>

NOT	bool -> bool
AND, OR	bool * bool -> bool

Sets:

CreaSet	type, cat of x -> set of type

Used for the creation of a new empty set in situations where the user needs to build a set by adding element by element. Several other operations also create a new set (e.g. Valuate or CatExtract, see below), although mostly not an empty one.

SetAdd, SetRetain, SetRemove set of x * set of x -> set of x

These are, respectively, the set union, intersection and difference operation.

SetExtract set of x -> x * set of x
Insert x * set of x -> set of x

These operations remove an atom from, c.q. add an atom to the set.

Valuate set of ref -> set of x
x ∈ {int, rl, wd, empty}

Valuate replaces the refs in a 'set of ref' by the values of the corresponding elements in the data base. Categories without data objects, so-called 'ghost' categories, produce an empty set.

Count set of x -> int
Min, Max set of x -> x
In all three cases: x ∈ {int, rl, wd}

Sum, Average, StdDev set of x -> rl
x ∈ {int, rl}

These are aggregate functions. They quantify over the whole set.

Categories:

CatMin, CatMax cat of x -> x

Furnish the smallest, c.q. largest element of a category, e.g. to be used as a lower, c.q. upper limit in the next operation:

CatExtract cat of x * x * x -> set of ref

CatExtract performs a range query. The atom parameters indicate, respectively, the lower and the upper limit of the elements of the set.

CatExpand cat of x * int
-> cat of x * set of ref

Adds a number of items to a category.

CatAdd cat of x * set of ref * set of x
-> cat of x
(x ∈ {empty, int, rl, wd})

Adds a set of values to a category (to existing items).

CatRemove cat of x * set of x -> cat of x * set of ref

Removes a set of values from a category and delivers the refs of the associated items.

CatReduce cat of x * set of ref -> cat of x

Removes a set of items from a category.

New values should be added to a category in two steps: first a number of items should be created and next these items should be given a value. By dropping the second step you may create a set of items within a category that don't have values, e.g. as an extension to a ghost category. The reverse, removal of items, takes as many steps as their addition.

Tfunctions:

CreaTfn cat of x * cat of x -> tfn

Analogous to CreaSet: The creation of an empty new temporary function that subsequently will be filled element by element (or will be kept empty, if necessary). Creation of tfns may also be done by other means, e.g. by FuncExtract (see below).

Compose tfn * tfn -> tfn

The composition of two functions can be fairly easily achieved for tfunctions, as opposed to permanent functions that reside on background storage.

TfnDom, TfnRange tfn -> set of ref

These operations deliver the domain set or the range set of the tfunction, respectively.

TfnAppl tfn * ref -> ref

Function application for one atom.

TfnInsert tfn * ref * ref -> tfn

This operation extends the tfunction by one element. It will among others be used to build a tfunction to be used for addition to a permanent function.

The domain and range of a tfunction are sets of ref and thus represent a subset of some category.

Functions:

Apply, InvApp   set of ref * func -> set of ref

These are, respectively, the function and the inverse function application.

FuncAdd, FuncRemove   func * tfn -> func

These are update operations. The main reason for not imposing the condition that a tfunction be a subset of a function is that tfunctions are used to extend functions (by FuncAdd).

FuncExtract   func * set of ref -> tfn

Produces a restriction of the function.

Tables:

CreaTable   -> table

Creates a new table after removing the current contents first, if necessary.

AddAttr   table * tfn * wd -> table

Adds a new column to a table.

Select   table * wd * ref -> x
$x \in$ {empty, int, rl, wd}

For sets and tfunctions we further have copy operations that create a duplicate of such a structure (CopySet and CopyTfn, respectively).

The set of operations mentioned here is not minimal: some of these operations may be replaced by a composition of other operations, e.g. Apply(f, x) is equivalent to Range(TfnExtract(f, x)).

As one application will require more than one data base (e.g. an expert system using a dictionary, a facts base, a rules base and help information), every operation that involves a category or function in fact has one parameter more than the ones mentioned: the relevant data base.

Precise specifications of all operations are shown in the appendix.

5. User Interface.

There are several kinds of users of a DBMS. First there is the data base designer, who uses the system to create a data base schema. This person will specify a set of categories and functions for the registration of the data that some institution needs and he or she will prescribe the constraints that the transactions on the DB should observe.

Next we have the applications designer. This person will specify certain queries to be done on the DB. He or she will also design a user interface that allows the end user to perform these queries without entering them explicitly into the system.

The third kind of user is the end user, who has a task to fulfill in the institution for which the data base has been created. He or she may be a desk employee in a travel agency who wants to make flight reservations for customers, or some such thing.

The first two kinds of users will have much the same requirements when using a user interface. They need a view of the data base schema and they will want to enter data into the system or receive data from the system, often straight from some category or function. The main difference between them is that users of the first kind will work on the dictionary, while the second kind of user will only use the primary DB. The third kind of user will only have to deal with the products of the application designer: Standard screens from which data should be read and into which data should be entered. This user will not be concerned with the structure of the data base or the kind of data it contains. Our main interest will be with the first two users.

Now we have a system of data structures and operations to manipulate them. The next thing to do is to create a means for a user to interact with the system. We want to have an interface that allows the user to realise the queries he has in mind on the system he uses. A user operates a computer by means of a keyboard, a mouse or some other input device. The system may respond through a display, using windows for presentation of groups of related data. The user interface will provide the link between the data structures and operations on one side and the input and output devices on the other.

Our main interest will be with the applications designer. Many of the tools he will have to use may easily be transfered to the DB designer.

As windows are often used to present (temporary) data, the obvious thing to do is to map sets, tfunctions and tables to windows. The mouse may be used to indicate windows to be used as an operand and to point at the operations, shown in some menu. Input data may be entered by keyboard.

While "toying around" with the data base the user will perform the actions needed for a query that should play a role in some application. Then it is up to the computer to convert the actions of the user, extended with certain commands, into a regular query statement. The difference between the computer generated statement and the user actions lies with the level of generalisation:

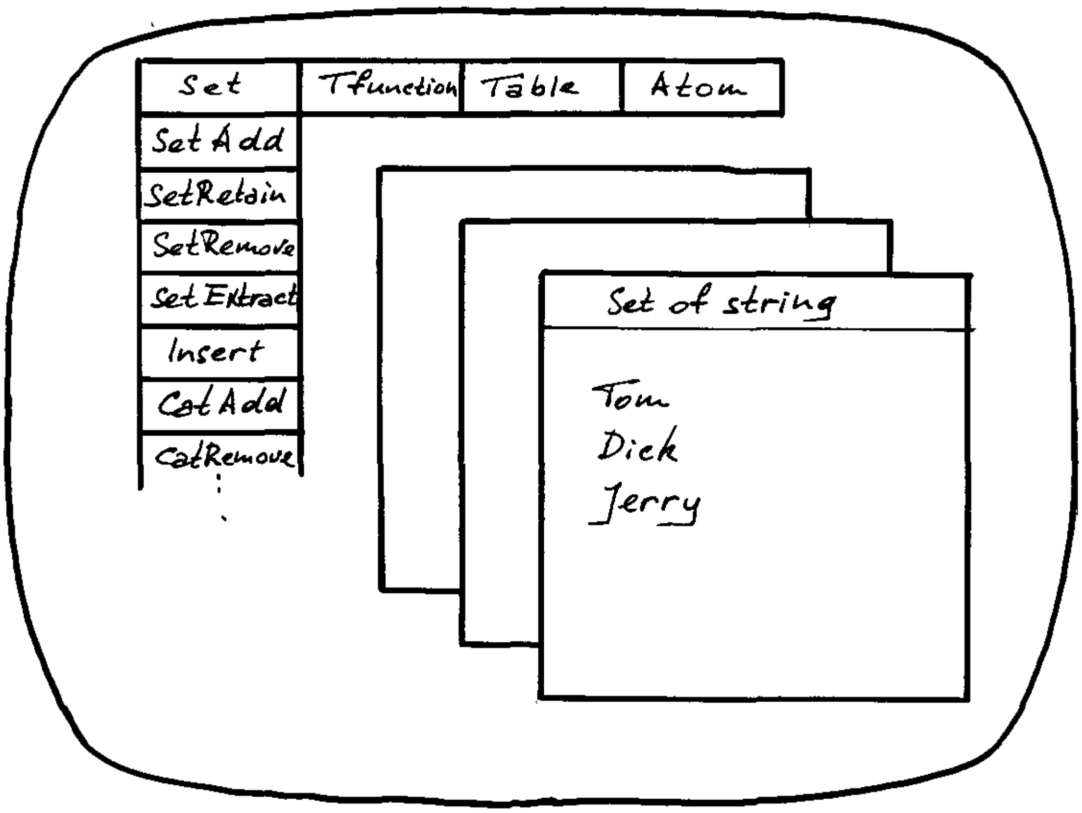
- The final statement will contain parameter indications where during the "playing" actual values were entered.

- Furthermore the final statement will contain things like the union of some (compound) operation over all elements of a set, where the defining actions were only concerned with one element of that set.

A window is associated with a 5-tuple <Or, Od, Cat, T, S>, where:

Or is an operator, one from the set of operators mentioned in section 4.
Od is a set of operands, $Od \in windows \cup parameters$
$Cat \in Ci \cup input$
S are the contents of the window.
T is the type of these contents.

A window will have two faces: On one side there will be data about the window, namely Or, Od, Cat, T and the number of elements. On the other side there will be the contents of the window. Refs will never be displayed, so in this case the contents face will not be presented. There will however be a command to display the valuation of these Refs.



User Interface for Applications Designer

6. Implementation.

An actual working system able to perform the operations described above on a given machine may be created using four layers of software on top of the file system. The operations and types described above constitute the top layer. For their operation they need a dictionary besides the data base proper. So, in the second layer from the top we have two DBs, each with their own categories and functions. Below this level we don't have categories and functions anymore, just items containing references, values and extensions. These items are identified by some number, the ref of the item. One level deeper, the items are dissolved into one or two records in different files, and the refs have been replaced by the adresses of these records. The lowest level consists of the file system.

Overview:

level 0: Standard file system.
level 1: records and adresses
level 2: items and refs
level 3: DBbuffer and Dictbuffer
level 4: Eldorado datastructures and operations.

Data and meta data.

The description of the skeleton of the DB, as shown in section 2, will reside in the dictionary, together with the labels and the connection between labels and functions plus the names assigned to the categories and functions. A function or category may have more than one name (synonyms).

The categories of the dictionary are:

Function name
Fi: Function id
Label (Link label of function, viz. section 2)
Category name
Ci: Category id
Type
NLabels

Type indicates a finite set of domains, among which the empty set {} ($\emptyset \in$ Type).
$\forall\, i \in Ci: T(i) \in Type$

Type doesn't change during the lifetime of the system. Which types there are and what is the ordering of their elements is determined beforehand.

NLabels is an integer category indicating the maximum number of forward or backward references an item from a specific category will have.

The functions are:

Function name -> Function id
Function id -> Label
Category name -> Category id
D: Function id -> Category id
R: Function id -> Category id
T: Category id -> Type
NF: Category id -> NLabels
NB: Category id -> NLabels

The NF and NB functions are for bookkeeping purposes. They will determine the maximum size of an item from the associated category in the data base.

This meta data base will be accessed every time the system needs information on the structure of the DB. It will be updated for addition or removal of categories and functions. For this purpose the same operations and data structures may be used as for the coresident data bases. However, an authorisation mechanism should be added to prevent unwanted schema modifications.

The meta data base is stored in the same files as the data base it describes. However, in order to limit the mutual interference between file accesses for different data bases, each data base has its own set of buffers for temporary copies of DB records. An application uses as many buffers as it needs data bases. So a straightforward functional facts base will need two buffer sets: One for the facts and one for the dictionary.

Items and refs.

All categories of an Eldorado data base, and possibly of various data bases, will be stored in the same files. It would be impractical to open a new file every time a new category is going to be accessed. So the files used will not correspond to the categories created. In fact, even the structure of the files used will not mirror the structure of the DB in terms of categories and functions.

At a certain level we are not aware of categories and functions in much the same way as, looking through a microscope, a plant is not seen to consist of leaves and stems, but of (more or less differently shaped) cells. In our case these cells correspond to the items of the data base. An item may have a value of some type and connections to other items. It may be found through its connections, by way of an other item, or it may be accessed on the basis of its value. For this purpose, there will be an index mechanism that, given a certain value from a specified category, allows us to locate the associated item.

So every item will eventually contain at least one of the following: a value and a number of refs of other items, where every ref is associated with a certain label at the item itself. If that would not be the case, the item could never be retrieved.

Files, Records and Adresses.

If we could sufficiently augment the magnification of our microscope, at a certain moment the cells would dissolve into molecules before our eyes. As we have seen, the "cells" of our system, the items, are built from various components. And these components will have their own inner structure. Different components will be stored in different files where each file will contain records of a different type.

First we have the index file. It consists of a tree of trees: a category tree that allows easy access to the different categories and a number of value trees. All trees are B-trees. The value trees are the leaves of the category tree. So if we need a specific value from a specific category, the system first searches for the category in the category tree. There it may find a value tree that possibly contains the value specified and provides us with the associated ref.

Next we have the reference file. Its records will not contain any values, just references. Their structure is:

<id, fr, br>,

where id is the ref of the item itself,

fr = <nf, fpl>,

with nf = the number of forward (functional) references and

fpl = A list of tuples <l, fp, lp>, with l ∈ labels, fp indicates the range value associated with the current element for the function concerned, and lp a reference to an other element from the same category with the same function value.

br = <nb, bpl>,

where nb = the number of backward references,

bpl = a list of tuples <l, bp> indicating an item from the inverse function associated with l.

An item may also have an entry in the objects file, where the values and extensions are stored. The elements of this file have the following structure:

<id, l, v, ext>,

where id is the ref of the element, l the total lenght (in bytes) of the associated data, v = <tp, cont> the value, composed of a type identifier (tp) and the representation (cont), which may also be used to locate the element by way of the index file (search argument). ext are free-format data of arbitrary length.

The position of items within the files may change as other items are added or removed and unused filespace is reclaimed. If we want to use their mutual connections we need to keep track of the items as they move. There is a special file for this purpose. It contains the locations of the reference and data parts of all the items:

<id, rloc, dloc>

and will be updated every time a location is changed. So, if the ref (id) of an item is known, it will allways be possible to locate its components (rloc and dloc).

The data in this storage scheme are intentionally made redundant, e.g. by storing the item identification and the lengths of the reference lists with every item. This is done for reasons of reliability and efficiency: If part of the stored data is damaged, much of it may be retrieved by inspecting the undamaged part. This redundancy further limits the number of disk accesses needed, especially to the dictionary.

If, for example, the locations file had been destroyed, it could be restored by going through the reference and object files, item by item, and noting the locations of the items and the identifications stored at these locations. Furthermore, the addition of a new function to a data base that already contains a large number of items, does not force us to change all the items of the domain and range categories, because of added labels, as each item has its own indication of the number of labels. Only those items that play a role in the new function need to be adapted.

The next lower level of the DBMS will be formed by the file system, that keeps a directory of file names and locations.

Temporary data structures.

Sets, tfunctions and tables will not reside on background memory. They will cease to exist when the program that uses them is terminated. These structures are represented in memory by an ordered list of elements, together with some associated data.

A set is characterised by the type of its elements and, if this type is ref, the category that holds them, together with the list of elements itself:

    S = <t, i, cont>,
where t ∈ {empty, int, rl, wd, ref},
    t = ref => i ∈ Ci,
    cont = NIL OR cont = <v, cont>.

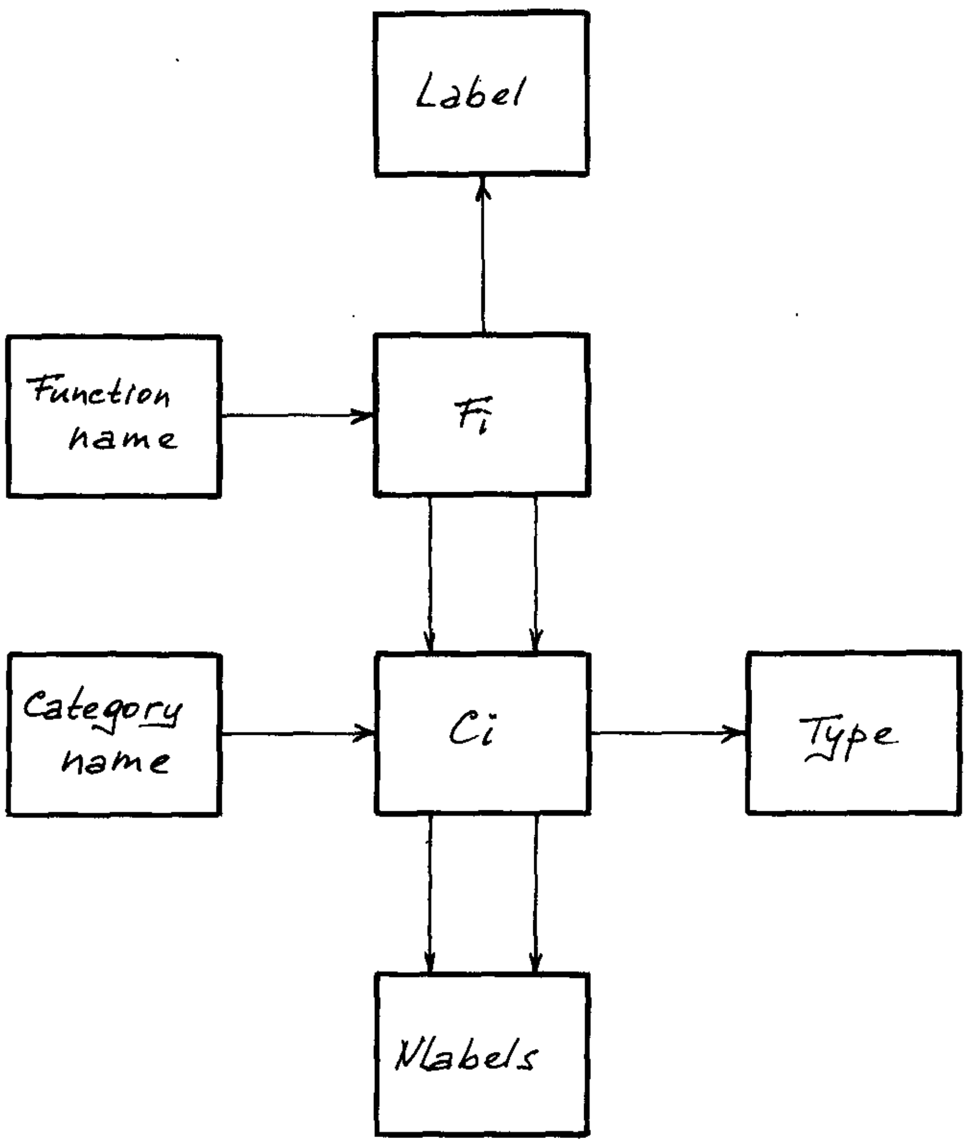
A tfunction will have a domain and a range category:

    TF = <cd, cr, cont>,
with cd ∈ Ci AND cr ∈ Ci,
    cont = NIL OR cont = <<rd, rr>, cont>
AND Type(rd) = Type(rr) = ref.

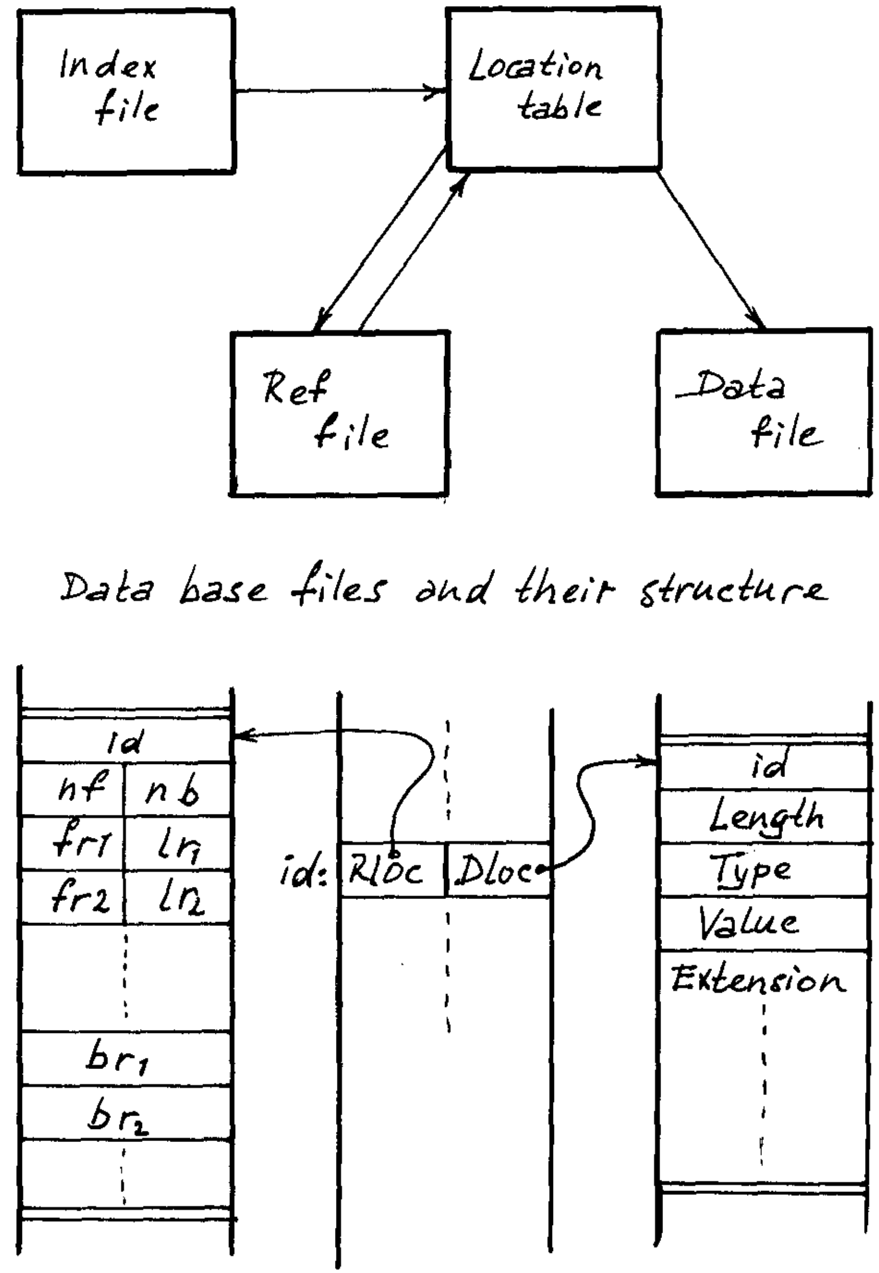
A table is a list of tfunctions:

    T = NIL OR T = <<a, f>, T>,
with Type(a) = wd AND type(f) = tfn.

More precise specifications of the layers described above may be found in the appendix.



ELDORADO dictionary



Data base files and their structure

Appendix A: Data structures and invariants.

Summary: The basic structures of Eldorado are: Atoms, Sets, Categories, Tfunctions, Functions and Tables. Atoms will be of one of the following types: empty, integer, real, string, ref or bool. Sets are either of type integer, real, string or ref, or they are empty. Tfunctions and functions may be considered as composed of refs, and tables are sets of named temporary functions. The following conditions hold:

A data base state may be described by a 3-tuple:

$$\langle Obj, V, Link \rangle$$

Where Obj = a set of indices
V = a (partial) function: indices -> values
Link = a ternary relation: labels * indices * indices

A data base skeleton according to the functional model is a 5-tuple:

$$DB = \langle Ci, Fi, D, R, T \rangle$$

in which Ci and Fi are sets, indicating respectively: Category indices and function indices.
D and R are the domain function and the range function of the functions and T is a function that links each category name to a set of possible values, its type. This set may be empty.

$\forall\ y \in Fi:\ D(y) \in Ci \text{ AND } R(y) \in Ci$

A data base state in the functional model may be described by a pair of functions:

$$\langle c, f \rangle,$$

where c and f correspond to a partioning of Obj and Link, respectively.

$\forall\ x \in Ci:\ c(x) \subseteq Obj$
$\quad \text{AND } \forall\ x1 \in Ci:\ \forall\ x2 \in Ci:\ x1 \neq x2 \Rightarrow c(x1) \cap c(x2) = \emptyset$

f(y) is a projection of a selection of Link:
$s = \{ \langle l, y \rangle \mid l \in labels \text{ AND } y \in Fi \}$ is a function
$\forall\ y \in Fi:$
$\quad \exists\ l \in labels:\ y = s(l)$
$\quad\quad \text{AND } f(y) = \{ \langle i1, i2 \rangle \mid \langle l, i1, i2 \rangle \in Link$
$\quad\quad\quad \text{AND } i1 \in c(D(y))$
$\quad\quad\quad \text{AND } (i2 \in c(R(y)) \text{ OR } i2 = NIL) \}$
$\quad\quad \text{AND } f(y)$ is a function.

```
∀ x ∈ Ci:
     V is a function: ∀ el ∈ c(x): V(el) ∈ T(x)
 AND K = { v|  v = V(el) }:
     ∀ k1 ∈ K: k1 = max(K) OR ∃ k2 ∈ K: k2 = succ(k1)
 AND ∀ k1 ∈ K: k1 = min(K) OR ∃ k2 ∈ K: k2 = pred(k1)
 AND ∀ k ∈ K, k ≠ min(K): succ(pred(k)) = k
 AND ∀ k ∈ K, k ≠ max(K): pred(succ(k)) = k.
```

Definition:

```
     x > y := x = succ(y) OR (∃ z: z > y AND x = succ(z)).
```

Now, within the Eldorado system, the following holds:

```
     ∃ y1, y2:
          (∃ x1, x2 ∈ c(C1): y1 ∈ c(C2) AND y2 ∈ c(C2)
                                   AND x1 = y1 AND x2 = y2
                                   AND x2 > x1)
          => y2 > y1.
```

The '<' relation may be defined analogously:

```
     x < y := x = pred(y) OR (∃ z: z < y AND x = pred(z)).

     EXT is a function,
          with ∀ k ∈ c(x): EXT(k) = NIL OR EXT(k) ∈ EF,
          and EF is a set of untyped data.
```

Temporary data structures:

Atom(X) => Type(X) ∈ {empty, bool, int, rl, wd, ref}

Type(X) = ref => ∃ i ∈ Ci: X ∈ c(i)

Set(X) => (∃ y ∈ { empty, int, rl, wd, ref }: Type(X) = set of y)

All elements of a set have the same type.

(Type(X) = set of ref) => ∃ i ∈ Ci: X ⊆ c(i)

Category(X) => (∃ y ∈ { empty, int, rl, wd }: Type(X) = cat of y)

Tfunction(X) => Type(X) = tfn

Of course, the following expression must hold:

```
Type(X) = tfn => ∃ i ∈ Ci: TfnDom(X) ⊆ c(i)
                    AND ∃ j ∈ Ci: TfnRange(X) ⊆ c(j)
```

Function(X) => Type(X) = func

Table(X) => Type(X) = table

All columns of a table should have the same domain, that is:

```
∀ T, Type(T) = Table:
     ∃ i ∈ Ci:
          ∃ S ⊆ c(i):
               (∀ <w, f> ∈ T: TfnDom(f) = S)
```

Storage structure.

The form in which the types mentioned are stored may be defined as follows, using the notation A.B as an indication of element B from tuple A:

```
Data base: DB = <index, references, objects>
     index = {< cat, v, id>|
                id ∈ c(cat) AND v = V(id) AND Type(v) ≠ empty}

     references = {<id, fr, br>| ∃ i ∈ Ci: id ∈ c(i)}
     fr = <nf, fpl>
     Type(nf) = int
     fpl = {<l, frefs>| 0 ≤ l < nf}
     frefs = <fp, lp>
     Type(fp) = Type(lp) = ref
          lp indicates an other item from the same category that
          has the same fp value for the same label l.
     br = <nb, bpl>
     bpl = {<l, bp>| 0 ≤ l < nb}
     Type(bp) = ref
          bp is a backward pointer indicating an element that has
          a forward pointer to the current item.

     objects =
          {<id, length, v, ext>|
           ∃ i ∈ Ci: id ∈ c(i) AND v = V(id) AND Type(v) ≠ empty}
     Type(length) = int
          length indicates the total length (in bytes) of the
          data (v and ext)
          (lenght = Length(Type(v)) + Length(ext))
     v = <tp, cont>
     tp ∈ {int, rl, wd}
     cont is the representation of a value of the type indicated
     ext is a byte string of arbitrary length.

Set: S = <t, cat, cont>
     t ∈ {empty, int, rl, wd, ref}
     t = ref => cat ∈ Ci
     cont = NIL OR cont = <v, cont>
     t = empty <=> S.cont = NIL
     (cont ≠ NIL AND cont.cont ≠ NIL) => cont.v < cont.cont.v

Tfunction: TF = <cd, cr, cont>
     cd ∈ Ci
     cr ∈ Ci
     cont = NIL OR cont = <p, cont>
     p = <rd, rr>
     Type(rd) = Type(rr) = ref
     (cont ≠ NIL AND cont.cont ≠ NIL)
                                  => cont.p.rd < cont.cont.p.rd
```

```
Table: T = NIL OR T = <a, T>
     a = <name, f>
     Type(name) = wd
     Type(f) = tfn
     (T ≠ NIL AND T.T ≠ NIL) => T.a.name < T.T.a.name
```

This definition only describes the implementation of the various structures. It is not used in the definition of the operations below.

Appendix B: Formal specification of Eldorado operations.

CreaSet(x, Cat, S)

Input parameters: x: datatype
Cat: cat of x

Input/output parameters: S: set of x

Preconditions:
$x \in \{ empty, int, rl, wd, ref \}$
$x = ref \Rightarrow Cat \neq NIL$

Postconditions:
$S = \{\}$

SetAdd(S1, S2)

Input parameters: S2: set of x

Input/output parameters: S1: set of x

Preconditions:
$Type(S1) = \text{set of ref} \Rightarrow \exists\, i \in Ci: S1 \subseteq c(i) \text{ AND } S2 \subseteq c(i)$
$S0 = S1$

Postconditions:
$S1 = S0 \cup S2$

SetRetain(S1, S2)

Input parameters: S2: set of x

Input/output parameters: S1: set of x

Preconditions:
$Type(S1) = \text{set of ref} \Rightarrow \exists\, i \in Ci: S1 \subseteq c(i) \text{ AND } S2 \subseteq c(i)$
$S0 = S1$

Postconditions:
$S1 = S0 \cap S2$

SetRemove(S1, S2)

Input parameters: S2: set of x

Input/output parameters: S1: set of x

Preconditions:
$Type(S1) = \text{set of ref} \Rightarrow \exists\, i \in Ci: S1 \subseteq c(i) \text{ AND } S2 \subseteq c(i)$
$S0 = S1$

Postconditions:
$S1 = S0 \setminus S2$

SetExtract(S, v)

Input/output parameters: S: set of x

Output parameters: v: x

Preconditions:
    S0 = S

Postconditions:
    v = min(S0)
    S = S0 \ {v}

Insert(S, v)

Input parameters: v: x

Input/output parameters: S: set of x

Preconditions:
    x ≠ ref
    S0 = S

Postconditions:
    Type(v) ≠ empty <=> S = S0 u {v}

Valuate(S1, S2)

Input parameters: S2: set of ref

Output parameters: S1: set of x

Preconditions:

Postconditions:
    S1 = { v| ∃ el ∈ S2: v = V(el) AND Type(v) ≠ empty }

CopySet(S1, S2)

Input parameters: S2: set of x

Output parameters: S1: set of x

Preconditions:

Postconditions:
    S1 = S2

CatMin(v, Cat)

Input parameters: Cat: cat of x

Output parameters: v: x

Preconditions:
$x \neq ref$

Postconditions:
$v = \underline{Min}\{ y \mid \exists\, el \in c(Cat) \text{ AND } y = V(el) \}$

CatMax(v, Cat)

Input parameters: Cat: cat of x

Output parameters: v: x

Preconditions:
$x \neq ref$

Postconditions:
$v = \underline{Max}\{ y \mid \exists\, el \in c(Cat) \text{ AND } y = V(el) \}$

CatExtract(S, Cat, v1, v2)

Input parameters: Cat: cat of x
v1, v2: x

Output parameters: S: set of ref

Preconditions:

Postconditions:
$S = \{ el \in c(Cat) \mid v1 \leq V(el) \leq v2 \}$

CatExpand(Cat, n, S)

Input parameters: n: integer

Input/outputparameters: Cat: cat of x

Output parameters: S: set of ref

Preconditions:
$So = c(Cat)$
$Sn \subseteq \{ r \mid \forall\, i \in Ci: r \notin c(i) \}$
$\#Sn = n$

Postconditions:
$c(Cat) = So \cup Sn$

CatAdd(Cat, S1, S2)

Input parameters: S1: set of ref
S2: set of x

Input/output parameters: Cat: cat of x

Preconditions:

$S1 \subseteq c(Cat)$

$x \neq ref$

Postconditions:

$\forall x \in S2: \exists el \in S1: x = V(el)$

CatRemove(Cat, S1, S2)

Input parameters: S1: set of x

Input/output parameters: Cat: cat of x

Output parameters: S2: set of ref

Preconditions:

$x \neq ref$

$S0 = \{ el \mid el \in c(Cat) \text{ AND } V(el) \in S1 \}$

Postconditions:

$S1 \not\subseteq \{ v \mid \exists el \in c(Cat): v = V(el) \}$

$S2 = S0$

CatReduce(Cat, S)

Input parameters: S: set of ref

Input/output parameters: Cat: cat of x

Preconditions:

$S \subseteq c(Cat)$

$\forall el \in S: Type(V(el)) = empty$
$\text{AND NOT } (\exists i \in Fi: \langle el, y\rangle \in f(i) \text{ OR } \langle x, el\rangle \in f(i))$

Postconditions:

$S \not\subseteq c(Cat)$

CreaTfn(C1, C2, TF)

Input parameters: C1, C2: cat of x

Input/output parameters: TF

Preconditions:

Postconditions:

$TF = \{\}$

TfnDom(F, S)

Input parameters: F: tfn

Output parameters: S: set of ref

Preconditions:

Postconditions:

$S = \{ x \mid \langle x, y \rangle \in F \}$

TfnRange(F, S)

Input parameters: F: tfn

Output parameters: S: set of ref

Preconditions:

Postconditions:

$S = \{ y \mid \langle x, y \rangle \in F \}$

Compose(F1, F2)

Input parameters: F2: tfn

Input/output parameters: F1: tfn

Preconditions:

$\exists\, i \in Ci$: TfnRange(F2) $\subseteq c(i)$ AND TfnDom(F1) $\subseteq c(i)$

$F0 = F1$

Postconditions:

$F1 = \{ \langle x, y \rangle \mid \exists\, z: (\langle x, z \rangle \in F0 \text{ AND } \langle z, y \rangle \in F2) \text{ OR } y = \text{NIL}) \}$

TfnAppl(F, el1, el2)

Input parameters: F: tfn
el1: ref

Output parameters: el2: ref

Preconditions:

Postconditions:

$\langle el1, el2 \rangle \in F$ OR $el2 = \text{NIL}$

TfnInsert(F, el1, el2)

Input parameters: el1, el2: ref

Input/output parameters: F:tfn

Preconditions:

$\exists\ i \in Ci:\ el1 \in c(i)\ \text{AND}\ TfnDom(F) \subseteq c(i)$

$\exists\ j \in Ci:\ el2 \in c(j)\ \text{AND}\ TfnRange(F) \subseteq c(j)$

$F0 = F \setminus \{ \langle el1, elr \rangle \mid elr \in TfnRange(F) \}$

Postconditions:

$F = F0 \cup \{\langle el1, el2 \rangle\}$

CopyTfn(F1, F2)

Input parameters: F2: tfn

Output parameters: F1: tfn

Preconditions:

Postconditions:

$F1 = F2$

Apply(S1, F, S2)

Input parameters: F: func
S2: set of ref

Output parameters: S1: set of ref

Preconditions:

Postconditions:

$S1 = \{ y \mid \langle x, y \rangle \in F\ \text{AND}\ x \in S2 \}$

InvAppl(S1, F, S2)

Input parameters: F: func
S2: set of ref

Output parameters: S1: set of ref

Preconditions:

Postconditions:

$S1 = \{ x \mid \langle x, y \rangle \in F\ \text{AND}\ y \in S2 \}$

FuncAdd(F, TF)

Input parameters: TF: tfn

Input/output parameters: F: func

Preconditions:

$TfnDom(TF) \subseteq c(D(F))$

$TfnRange(TF) \subseteq c(R(F))$

$F0 = F \setminus \{ \langle x, y \rangle \in F \mid x \in TfnDom(TF) \}$

Postconditions:

$F = F0 \cup TF$

FuncRemove(F, S)

Input parameters: S: set of ref

Input/output parameters: F: func

Preconditions:

$S \subseteq c(D(F))$

$F0 = F$

Postconditions:

$F = \{ \langle x, y \rangle \mid \langle x, y \rangle \in F0 \text{ AND NOT } (x \in S) \}$

FuncExtract(TF, F, S)

Input parameters: F: func
S: set of ref

Output parameters: TF: tfn

Preconditions:

Postconditions:

$TF = \{ \langle x, y \rangle \in F \mid x \in S \}$

CreaTable(T)

Input/output parameters: T: table

Preconditions:

Postconditions:

$T = \{\}$

AddAttr(T, TF, w)

Input parameters: TF: tfn
w: wd

Input/output parameters: T: table

Preconditions:
$\forall \langle x, f\rangle \in T: TfnDom(f) = TfnDom(TF)$
$\forall \langle wt, f\rangle \in T: wt \neq w$
$TO = T$

Postconditions:
$T = TO \cup \{\langle w, TF\rangle\}$

Select(v, T, w, r)

Input parameters: T: table
w: wd
r: ref

Output parameters: v: x

Preconditions:

Postconditions:
$x \in \{empty, int, rl, wd\}$
$(\exists f: \exists el: \langle w, f\rangle \in T \text{ AND } \langle r, el\rangle \in f \Rightarrow v = V(el))$
$\text{XOR } x = empty$

AtomVal(r, v)

Input parameters: r: ref

Output parameters: v: x

Preconditions:

Postconditions:
$x = empty \text{ OR } (v = V(r) \text{ AND } x \in \{int, rl, wd\})$

AtomExt(r, e)

Input parameters: r: ref

Output parameters: e: ^ext

Preconditions:

Postconditions:
$e = NIL \text{ OR } e\hat{} \in EF$

COMPUTING SCIENCE NOTES

In this series appeared :

No.	Author(s)	Title
85/01	R.H. Mak	The formal specification and derivation of CMOS-circuits
85/02	W.M.C.J. van Overveld	On arithmetic operations with M-out-of-N-codes
85/03	W.J.M. Lemmens	Use of a computer for evaluation of flow films
85/04	T. Verhoeff H.M.J.L. Schols	Delay insensitive directed trace structures satisfy the foam rubber wrapper postulate
86/01	R. Koymans	Specifying message passing and real-time systems
86/02	G.A. Bussing K.M. van Hee M. Voorhoeve	ELISA, A language for formal specifications of information systems
86/03	Rob Hoogerwoord	Some reflections on the implementation of trace structures
86/04	G.J. Houben J. Paredaens K.M. van Hee	The partition of an information system in several parallel systems
86/05	Jan L.G. Dietz Kees M. van Hee	A framework for the conceptual modeling of discrete dynamic systems
86/06	Tom Verhoeff	Nondeterminism and divergence created by concealment in CSP
86/07	R. Gerth L. Shira	On proving communication closedness of distributed layers

86/08	R. Koymans R.K. Shyamasundar W.P. de Roever R. Gerth S. Arun Kumar	Compositional semantics for real-time distributed computing (Inf.&Control 1987)
86/09	C. Huizing R. Gerth W.P. de Roever	Full abstraction of a real-time denotational semantics for an OCCAM-like language
86/10	J. Hooman	A compositional proof theory for real-time distributed message passing
86/11	W.P. de Roever	Questions to Robin Milner - A responder's commentary (IFIP86)
86/12	A. Boucher R. Gerth	A timed failures model for extended communicating processes
86/13	R. Gerth W.P. de Roever	Proving monitors revisited: a first step towards verifying object oriented systems (Fund. Informatica IX-4)
86/14	R. Koymans	Specifying passing systems requires extending temporal logic
87/01	R. Gerth	On the existence of sound and complete axiomatizations of the monitor concept
87/02	Simon J. Klaver Chris F.M. Verberne	Federatieve Databases
87/03	G.J. Houben J.Paredaens	A formal approach to distributed information systems
87/04	T.Verhoeff	Delay-insensitive codes - An overview

87/05	R.Kuiper	Enforcing non-determinism via linear time temporal logic specification.
87/06	R.Koymans	Temporele logica specificatie van message passing en real-time systemen (in Dutch).
87/07	R.Koymans	Specifying message passing and real-time systems with real-time temporal logic.
87/08	H.M.J.L. Schols	The maximum number of states after projection.
87/09	J. Kalisvaart L.R.A. Kessener W.J.M. Lemmens M.L.P. van Lierop F.J. Peters H.M.M. van de Wetering	Language extensions to study structures for raster graphics.
87/10	T.Verhoeff	Three families of maximally nondeterministic automata.
87/11	P.Lemmens	Eldorado ins and outs. Specifications of a data base management toolkit according to the functional model.