EXTENDING FUNCTIONAL PROGRAMMING TOWARDS RELATIONS

Remi Legrand
LITP - CNRS UA 248, Université P. et M. Curie - Paris 6
2 place Jussieu, 75252 Paris Cedex 5, FRANCE
uucp : ....!mcvax!inria!litp!rl

Abstract : This article describes the relational programming paradigm. Because a function is a particular case of relation, we can consider the computation of points-to-set processes (relations) instead of points-to-point processes (functions). Relations are useful for parallel, non-deterministic or multi-valued algorithms. The first section presents the main features of the proposed language and it is shown how relations make programs more flexible and natural. Then, we present an efficient implementation of the language on a classical architecture.

1 - INTRODUCTION

In this section, the main principles of the relational programming concept are developed.

Functions and Relations : A computable function is a points-to-point process since it associates at most one value to its arguments. A n-ary function f can be defined by the n+1-ary relation R such that :

R = {(x1,x2,....xn,y) / y=f(x1,...xn)}

Relations are a generalization of functions in the sense that they associate a possibly infinite set of values to their arguments [Nguyen 85, Eilenberg 70]. For instance, the n+1-ary relation R(x1,..,xn,y) defines the points-to-set function f such that : f(x1,..,xn) = {y / R(x1,..,xn,y)}

Computable Relations : The automatic treatment of all mathematical relations is impossible so we must deal only with computable relations [Cutland 80]. The relations that we consider are defined by computable programs written in the language GREL given in the next section. The programs in GREL are constructed by combining smaller programs until we reach some primitives which are ordinary functions. Special operators are designed in order to turn several functions into a single relation with a finite number of results. And recursivity allows to construct relations with an infinite but nevertheless semi-computable set of results.

Relations and Continuations : The result of a relational program is a finite or infinite set of values. These values are returned one at a time and are used by the continuation of the relation which can be any kind of process, a function or an other relation. The overall program consisting of a relation followed by its continuation acts as a pipe where the relation provides input values and the continuation processes them. A special set of control operators allows to break the pipe or capture values.

Multisets of Results : The same value may occur many times in a relation. Because a value is transmitted to the continuation as soon as it is computed, no trace is kept of it and it is not possible to remove its further occurrences. Although this case is rare and issued from computation organization, it is better to talk about points-to-multiset relations than points-to-set relations. Moreover, the elements of these multisets are computed sequentially and we must consider that they are ordered by their times of appearance.

Control of Relations : Therefore, a relation applied to some arguments becomes a value generator which can be controlled and used with appropriate tools provided by the language. For instance, the programmer can choose between different strategies which determine the order of appearance of the values.

Running vs. Suspended : As in most functional languages where programs are objects, the result of applying a relation can be considered as an object of the language as soon as it is created and captured. The object relation can be treated as any other object and looks like a suspended process. For instance, we can duplicate it, kill it or activate it in order to get the next value. Thus a relation may have two states, running or suspended. A running relation partakes of the current process whereas a suspended relation may be temporarily runned as an oracle providing values.

Use of Relation : The usefulness of relations in the design of algorithms appears when programs are non deterministic, parallel or multi-valued. For instance we can construct a class of values by giving parallel inductive rules of construction which can be applied non-deterministicaly. The class of numbers $P(x,y) = \{ x^p * y^q , p,q \in N\}$ can be defined by :

$$P(x,y) = \{1\} \cup \{x * k, k \in P(x,y)\} \cup \{y * k, k \in P(x,y)\}$$

Relations can also be used for streams generation because a stream is a particular case of relation where any value is given by a function of its predecessor. Finally, it will be shown that relations are efficiently and flexibly usable in programs where functions have multiple results.

Host Language for Relations : Since early 60's, a lot of functional languages have been designed but none of them have been extended so as to support the computation of relations. The main reason seems to be the presence of variables since almost all these languages are lambda-languages. As a matter of fact, variables compel the implementation to do a very complex environment management which seems unrealistic in the general case of relations. That is why the variableless programming language GRAAL [Bellot 86a, Bellot 86b] has been chosen as the host language for the relational calculus.

Principle of Implementation : It is important to see that a relation will not only be a kind of function with multi-values. In order to accept all recursive relations with eventualy an infinite set of results, partial results must be immediately transmitted and not "put in wait". It is implemented by a *pipe-oriented* mecanism. Because all resulting values of a relation are transmitted to its continuation, the same computation occurs as many times as there are such values. That is to say that the unfoldings of these computations in the stack are done every time although they are roughly identical one to each other. The main principle of the implementation is to preserve the first unfolding in the stack so that it can be used without being constructed by further computations. This is realized through an original system of caches onto the reduction stack. With this method of implementation, the average loss of GREL compared to GRAAL is only 10% so that Relational GRAAL is still among the fastest applicative languages.

The plan of the article is the following. The first section is a brief introduction to the programming language GRAAL and its principles. The second section describes the new operators required in order to support the relational concept. The third section shows how relations can be used in the design of algorithms. Finally, the last section describes the implementation of relations as an extension to the GRAAL reduction machine described in [Bellot 86a].

2 - GRAAL, the Host Language

We present a short digest of the language GRAAL. A complete presentation of GRAAL and its issues is in [Bellot 86b], and its theorical support is given in [Bellot 87]. GRAAL is a functional programming language without variable. It is based on the notions of functional forms and uncurryfied combinators. The functions are polyadic. The application of a function **f** to the arguments **a1,...,an** is denoted **(f : a1...an)**. Applications are reduced using call-by-value. The notation E ➡ F stands for "E reduces (or evaluates) to F". The objects are numbers, symbols,lists or functional forms. Lists, used as data, are denoted by brackets instead of parenthesis. Examples : <>, < a b < 1 c 2 > >.

The primitive functions are issued from Lisp systems [Chailloux 84]. Semantic of functions is described by reduction rules. Examples :

car : < a . b > ➡ a	car : <> ➡ <>
cdr : < a . b > ➡ b	cdr : <> ➡ <>
cons : a b ➡ < a . b >	
null : <> ➡ true	null : a ➡ <> if a ≠ <>
add : 2 3 ➡ 5	sub1 : 5 ➡ 4
true : a1...an ➡ true	false : a1...an ➡ <>
eq : a b ➡ true if a = b	id : a ➡ a

Arguments of a function are implicitly numbered starting from one. If **k > 0**, **#k** applied to **n** arguments with **k ≤ n**, reduces to the argument whose rank is **k**. Example :

#3 : a b c d e f ➡ c #5 : a b c ➡ error

The reader in acquaintance with functional programming may complete by himself the set of primitive functions and their reduction rules.

More complex functions are built with functional forms which are combinations of functions. A functional form realizes a functional operation occurring frequently in programs. The primary syntax is **(opf p1 ...pn)** where **opf** is the name and **p1,p2,...pn** are the parameters. Despite of appearance, **(opf p1 ...pn)** is not a list. Functional behaviour of forms is described by reduction rules. The set of functional forms is not fixed. Only a few of them will be considered in this article.

Composition : the name of the composition form is **comp**. It accepts any number of parameters greater than two. Its reduction rule is :

$$\frac{gi : a1...an \Rightarrow bi\ ,\quad 1 \le i \le p}{(comp\ f\ g1\ ....gp) :\ a1...an \Rightarrow f : b1\ ...bp}$$

For sake of readibility, the syntactic analyser of GRAAL will recognize the following notations :

1) f o g stands for (comp f g)
2) {f g1...gn}stands for (comp f g1...gn)
3) (f g1...gn)stands for (comp f g1...gn) if **f** is not a combinator but is defined.

Conditional : the name of this form is **if** and it accepts three parameters. Its rules of reduction are:

$$\frac{p : a1 \ldots an \Rightarrow \langle\rangle}{(\text{if } p\ f\ g) : a1 \ldots an \Rightarrow g : a1 \ldots an} \qquad \frac{p : a1 \ldots an \Rightarrow x\ ,\ x \neq \langle\rangle}{(\text{if } p\ f\ g) : a1 \ldots an \Rightarrow f : a1 \ldots an}$$

Constant : in order to program a constant function, we must use the one-parameter form whose name is **cste**. Its reduction rule is : (cste c) : a1 ...an ➨ c
The syntactic analyser of GRAAL recognizes the notation **'c** for **(cste c)**.

Examples of defined functions : the definition of a function **f** is given by the evaluation of the expression **(de : f b)**, where **f** is a symbol (the name) and **b** is a function (the body). So, we have the examples :

```
(de : caddr   car o cdr o cdr)

(de : last  (if   null o cdr   car   last o cdr))

(de : append
      (if    null o #1
             #2
             {cons  car o #1  {append  cdr o #1  #2}} ))
```

The reader may find a lot of programming examples in [Bellot 86b] and the scheme of implementation is given in [Bellot 86a].

3 - RELATIONAL GRAAL

The relational GRAAL is also called GREL (acronym of Graal RELationnel). It is a development of GRAAL towards the relations. All the primitive functions and forms in GRAAL are still used in GREL. The evaluation of a function is almost the same. But, in GREL we add special forms and functions in order to built "non-functional" relations.

Notation : when the application of a relation **f** to the arguments **a1,...an** gives a multiset of results **b1,b2,...** , we denote the reduction (evaluation) by : (f : a1...an) ➨ b1 b2 b3...

3.1 - Union form

The most important form used for the construction of relations is **union**. No restrictions of use are made for this form. For instance, recursivity and an infinite number of results are allowed. In case of an infinite number of results, there are computed one at a time, and are given at once to the continuation of the relation. In a first step, we do not consider the order of the results given by the evaluation of a relation. It will be studied in the next section. The union form is defined by the reduction rule :

$$\frac{fi : a1...an \Rightarrow yi1.....yipi \quad , \quad 1 \leq i \leq n}{(\text{union } f1 \ldots fn) : a1...an \Rightarrow y11.... \ y1p1 \ y21....y2p2....ynpn}$$

Examples :

```
(de foo : (union        (union add1 sub1)
                        (union id '3 '4 )))
(foo : 45) ➨ 46 44 45 3 4

(de : integer (union    '0    add1 o integer))
(integer : ) ➨ 0 1 2 3 ....

(de : even {mul '2 integer})
(even : ) ➨ 0 2 4 6 ...
```

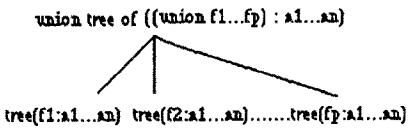
3.2 - Order of computations results

A relation gives the results in a specific order which is described as follows. The choice of the order is very important in case of infinite results or infinite loops. In a first step, we describe how to define the order of results in a relation. Then, we explain the variable strategies which order all the relational computations.

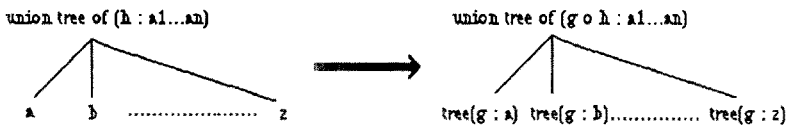
a) Union tree

The order of results is defined from the **union tree** joined to each relation applied to the arguments. Each leaf corresponds to a result of the application, and each node corresponds to a union. The union tree joined to the application **(f : a1...an)** is recursively defined by :

1) If f is a primitive function, then the tree is only reduced to the leaf labelled by the value obtained by the reduction of the application **(f : a1...an)**.
2) If **f = (union f1 ...fp)**, then the union tree of **(f : a1...an)** is a tree with **p** subtrees constitued of the union tree of all the **(fi : a1...an)** :



3) If **f = (cste x)**, then case 1 applies.
4) If **f = g o h**, then the union tree of **(f : a1...an)** is the union tree of **(h : a1...an)** in which each leaf **y** is replaced by the union tree of **(g : y)**. Example :

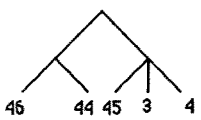


5) If **f = (if p q r)**, then the tree of **(f : a1...an)** is the union tree of **(p : a1...an)** in which each leaf **y** is replaced by :

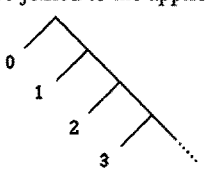
i) The union tree of **(q : a1...an)** if $y \neq <>$
ii) The union tree of **(r : a1...an)** else.

6) All other cases could be translated in one of the precedent cases, because all the considered forms may be only defined with the four forms union, cste, if and binary comp. The reader may also complete by himself the similar cases corresponding to the other forms.

Examples :
1) The union tree joined to the application **(foo : 45)** is :



2) The union tree joined to the application **(integer :)** is :



b) The control of relations

The order of results is characterized by the route (or strategy) chosen through the union tree. The reduction machine of the relational GRAAL contains a basic route used for all the relations. This route, called **limited-depth-first**, is complete (each result is find in a finite time) with no multi-computation of same partial results, which is not the case with the "Depth-First Iterative-Deepening" in [Korf 85].

Depth-first route : The rule bound to this strategy is to choose the subtree of a node before the other subtrees of last node encountered. The depth-first route of the union tree is the most efficient and easily implemented strategy. Nevertheless, it is not a complete strategy : some of the results are possibly not computed if there exists a infinite sub-tree. Therefore, we have used a **limited-depth-first** strategy which is complete and may be almost as efficient as depth-first strategy. This strategy is described as follows :

Limited-depth-first route : It is a depth first route whose depth is limited by a given value. At the beginning, the considered node is the root and we go over the tree with respect to the depth-first route, memorizing all the encountered nodes in which all the subtrees have not been go over. When we reach the maximal depth, we suspend and restart the route from the memorized node which is the nearest to the root. The route stops when there is no more memorized subtree.

It is a complete strategy because the maximal difference between two memorized nodes is lower than the maximal depth. Moreover, it is a efficient strategy when we choose a great maximal depth, because it will be almost a depth first strategy. The accessible parameter for the user is the maximal depth. In case of depth limited to one, it is a breadth first strategy. In case of infinite limit, it is a depth first strategy. Therefore limited depth strategy is a generalization of these two classical strategies.

So far, we know how to construct relation, that is to say that we are able to generate an infinite number of results using the union form. Now, we are faced to the problem of managing these results.

3.3 - Management of results

In some cases, we do not want to find or compute all the results of a relation but only a few of them. Thus, we define a lot of functions or forms which stop or do not execute the computations of unwanted results.

a) The cut

According to the portion of unwanted results, we use different "cut" as for example :

1) The function **culdesac** only cuts the current subtree of the union tree. Example :

((union '1 '2 o culdesac '3) :) ➨ 1 3

2) The function **stop** cuts all the subtrees of the union tree joined to the relation. Example :

((union '1 '2 o stop '3) :) ➨ 1

4) The form **exit** included in the form **tag** (with the same joined function) only cuts the subtrees appeared in the form **tag**.The second parameter of the form **exit** gives the last result of the form **tag**. Example :

((union '1 (tag end (union '2 '3 o (exit end '4) '5)) '6) ➨ 1 2 4 6

b) The first form

In order to obtain only the **n** first results of a relation, we use the **first** form. The reduction rules are :

$$\frac{f : a1...ap \rightarrow r1 ...rq\ ,\ q \leq n}{(\text{first } n\ f) : a1...ap \rightarrow r1 ...rq} \qquad \frac{f : a1...ap \rightarrow r1 ...rq...\ ,\ n \leq q}{(\text{first } n\ f) : a1...ap \rightarrow r1 ...rn}$$

Example : ((first 10 integer) :) ➨ 0 1 2 3 4 5 6 7 8 9

c) The set form

Results of a relation do not make a set because the same value may occur many times. With the set form, we built a set of values instead of a multiset. The form **set** is a filter which captures the results already computed and memorized in a list. This form is defined by the reduction rule :

$$\frac{(f : a1 \ldots an) \Rightarrow r1\ r2 \ldots rp \quad , \quad \bigcup_{1 \le i \le p} \{ri\} = \{s1, s2, \ldots sq\}}{(\text{set } f) : a1 \ldots an \Rightarrow s1\ s2 \ldots sq}$$

3.4 - Processes

The computation of results requires particular mecanisms which look like the control of processes in a parallel language. So, we can easily use the relations as processes without the need of a modification of the GREL reduction (evaluation) machine. A relation, seen as a process is an object, which can be handle with appropriate functions. A process is self modified each time it is activated, in order to give a new result of the relation at each new activation. Processes are useful to control the stream of computations results. They may be include in a relation or in an other process.

Creation : A relation is constructed as a process with the **create** function. Its reduction rule is :

create : f x1...xn ➡ (process f x1...xn)

The form (process f x1...xn) is a GREL object, which is interpreted as a suspended process, corresponding to the application of the relation **f** to **x1....xn.**

Activation : The activation of this process is given by the function **next.** This function computes the next result of the relation.

1) If **(f : x1...xn) ➡ y1 y2 ...** , then we have **(next : (process f x1...xn)) ➡ y1**, and the process self modifies to a process **(process g z1...zm)** such that we have the reduction

(g: z1...zn) ➡ y2 y3 ...

2) If **(f : x1...xn)** has no result, then **(next : (process f x1...xn)) ➡ error.**

With this form, we can compute all the results of a relation by repeating the function **next.** Example :

(de : foo (union next foo))

(foo o create : f x1 ...xn) ➡ y1 y2 ....

Peek : It is not possible to know if there exists other results without activating the process. But we can know if no more activations of the process may be done. The **peek** function applied to a process reduces to **true** if other activations are possible, else reduces to **nil.**

kill : In order to relieve the memory and the stack management, the **kill** function applied to a process deletes it, and frees memory space.

Duplicate : A process may be duplicated in order to compute the same results of a relation several times. Since GREL is a pure relational language, no partial memory copying is required. The **duplicate** function, applied to a process, does not copy the process but only keeps a continuation address.

4 - EXAMPLES

Because GREL is an extension of GRAAL, all functional problems may be very efficiently written in GREL. But, this language is particularly interesting when many results are required. In that case, it shows its full efficiency and flexibility. We give three examples of such programs.

4.1 - The powers of two and three

We want to compute all the numbers of the set $P(2,3) = \{ 2^p * 3^q , p,q \in N\}$. In order to have a constructive definition of P(2,3), we transform the definition in :

$$P(2,3) = \{1\} \cup \{2 * k, k \in P(2,3)\} \cup \{3 * k, k \in P(2,3)\}$$

So, the definition of the relation **pui23** which gives all the powers of **2** and **3** is :

```
(de : pui23
     (union     '1
                {mul '2  pui23}
                {mul '3  pui23} ))
```

In accordance with the strategy, the reduction of **((set pui23) :)** gives powers of two and three in different orders :

1) If the limited depth is one
   ((set pui23):) ➨ 1 2 3 4 6 9 8 12.....
2) If limited depth is three
   ((set pui23) :) ➨ 1 2 4 8 16 12 24.....
3) If limited depth is infinite (i.e. a very large number)
   ((set pui23) :) ➨ 1 2 4 8 16 32 ....

But, we can also compute the subset of P(2,3) given by :

$$P^{n,m}(2,3) = \{ 2^p * 3^q , \ 0 \le p \le n , 0 \le q \le m \}$$

We define the relation **pui23l** computing the elements of this set. The two arguments of **pui23l** are respectively the number of power of **2** and **3** allowed.

```
(de : pui23l
     (union     '1
                (if    {eq '0  #1}
                       culdesac
                       {mul  '2 {pui23l  sub1 o #1  #2}} )
                (if    {eq '0  #2}
                       culdesac
                       {mul  '3 {pui23l #1  sub1 o #2}} )  ))
```

With a limited depth equal to one, we have the reductions :

((set pui23l) : 2 1) ➨ 1 2 3 4 6 12
((set pui23l) : 3 2) ➨ 1 2 3 4 6 9 8 12 18 24 36 72

4.2 - Subsets management

The subsets of a set : All the subsets are easily given by the relation (set are implemented as lists) :

```
(de : subsets
    (if   null
          nil
          (union      {cons  car  subsets o cdr}
                      subsets o cdr )))
```

(subsets : < 1 2 3 >) ➨ <1 2 3> <1 2> <1 3> <1> <2 3> <2> <3> <>

The partitions : The partitions of a set are defined by the relation **partition** :

```
(de : partition
    (if   null
          nil
          {partition1  car  partition o cdr}
```

The reduction rule of **partition1** is :

(partition1 : x < l1 l2 ....>) ➨ < <x . l1> l2...> <l1 <x . l2> ...> .....

The relation **partition1** is defined by :

```
(de : partition1
    (if   null o #2
          {cons       {cons  #1  nil}  nil}
          (union      {cons       {cons  #1  car o #2}
                                  cdr o #2}
                      {cons       car o #2
                                  {partition1  #1  cdr o #2}  })))
```

Remark : In case of an infinite limited depth, these two relations need a space in the stack and in the memory proportional to the number of elements in the set. In case of a pure functional programming of these two relations, the space is exponential in the number of elements in the set.

4.3 - Same fringe

With the relations used as processes, we are able to solve the **same fringe** problem defined in [Durieux 81]. The required algorithm must compare the leaves of two trees. The bad method is to linearize the two trees before to compare them. The difference may appear at the first leaf, and the two leaves lists would have been useless. The good solution which we describe here, is to compare the leaves streams of the two trees.

Representation of a tree : We only consider the binary trees with nodes labelled by a symbol. So, a tree is a atom (i.e. a leaf) or a list of three elements which are the symbol of the node and the two subtrees. Examples : <n1 <n2 a b> c> c <n d e>

The tree fringe : The fringe of a tree is the ordered list of its leaves. We define the relation **fringe** which computes the leaves. If the relation is evaluated with a infinite limited depth, then the order of results will be correct for the relation :

```
(de : fringe
     (if   atomp
           id
           (union      fringe o cadr
                       fringe o caddr)))
(fringe : <n1 <n2 a b> <n3 <n4 c d> e>>) ➨ a b c d e
```

Sentinel add : In order to mark the end of the fringe, we add two sentinels :

```
(de : fringe_s     (union fringe 'end1 (exit end2 true) ))
```

Same fringe relation : The relation **same_fringe** reduces to **true** if its two arguments have the same fringe, else it reduces to <>.

```
(de : same_fringe
     (tag  end2       {same_fringe1   {create 'fringe_s #1}
                                      {create 'fringe_s #2} }))
(de : same_fringe1
     (if   {eq  next o #1  next o #2}
           {same_fringe1 #1  #2}
           nil ))
(same_fringe : <n1 <n2 a b> <n3 <n4 c d> e>> <n a  <n b <n c <n d e>>>>) ➨ true
(same_fringe : <n1 a b > <n  b  <n c >) ➨ <>
```

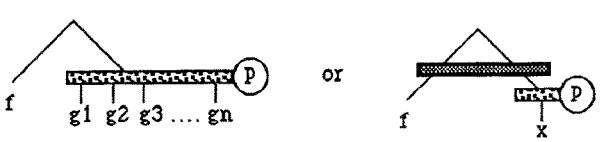
5 - IMPLEMENTATION

The GREL system has been implemented on a VAX 11/780 at the LITP (Laboratoire d'Informatique Théorique et Programmation). It is based on a execution mecanism called **graph reduction machine** [Turner 79]. This machine runs on a classical Von Neumann machine. It is a extension of the reduction machine of GRAAL, which is described in [Bellot 86b]. We explain the GREL machine and the dynamic representation of graphs.

5.1 - GREL machine

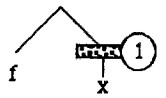
The GREL machine reduces graphs describing expressions of the language, until the obtention of an irreducible expression, which is the result. We suppose that relations are monoadic in order to simplify the writing. So, we have to describe the graphs bound to GREL expressions, their reduction, and the strategy of reduction used.

Graph : Since a relation has a class of results, the graph considers classes of element corresponding to GREL expressions. A graph is represented by :



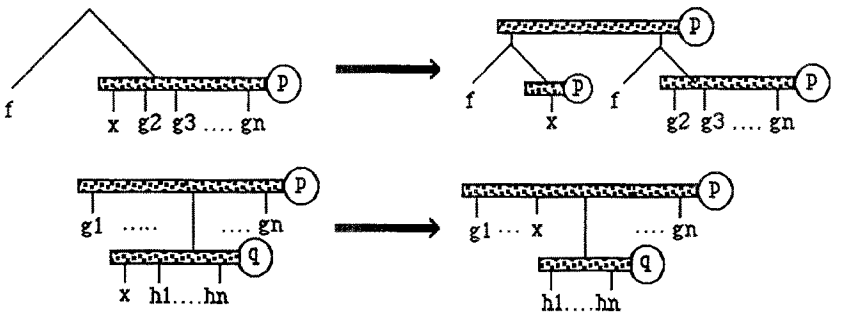
with **g1,...gn** graphs, **f** a relation, **x** a GREL object, and **p** an integer called the current depth, which will indicate when the limited depth is reached. In the first graph, **f** will have to be applied to the reduction of each graph **gi**. In the second graph, **f** will have to be applied to **x**. The grey bar indicates that the reduction have to be delayed. Moreover, this second graph (issued from the form union) is called "delayed node".

Initially, the graph representing the expression (f : x) is :

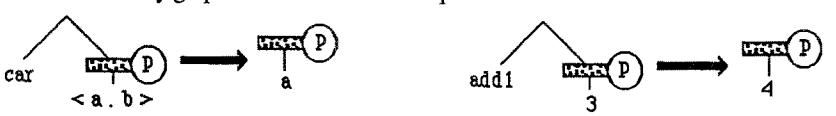


Reduction rules : They transform the graphs describing an expression from the initial representing until an irreducible graph. The reduction rules are classified in four groups.

1) Management results rules :

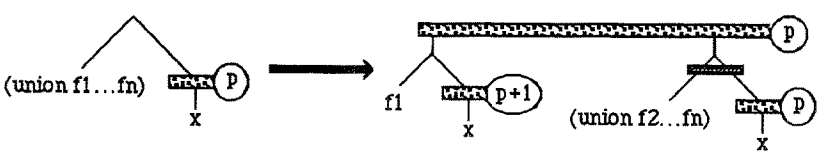


2) Primitive rules : Each primitive function behaviour is described with a reduction rule. These rules are intuitive but are not really graph reduction rules. Example :



3) Functional forms rules : We bind to each form a reduction rule. For instance the binary composition is bound to the rule :

4) Union rules : The form union is bind to a rule which is applicable only if **p** is smaller than the limited depth. This rule is :



If **p** is equal to the limited depth, the reduction rule is :

Strategy of reduction : The call-by-value and the limited-depth-first route indicate in which order the reductions are done. Graphs Reductions are deterministic (at each step, only one reduction is allowed). The reductions must correspond to the call-by-value used in GREL. Thus, the reductions previously described are only allowed when **x** is a GREL object and not a graph. The limited-depth-first strategy corresponds to the two principles (the first must be tried before the second) :

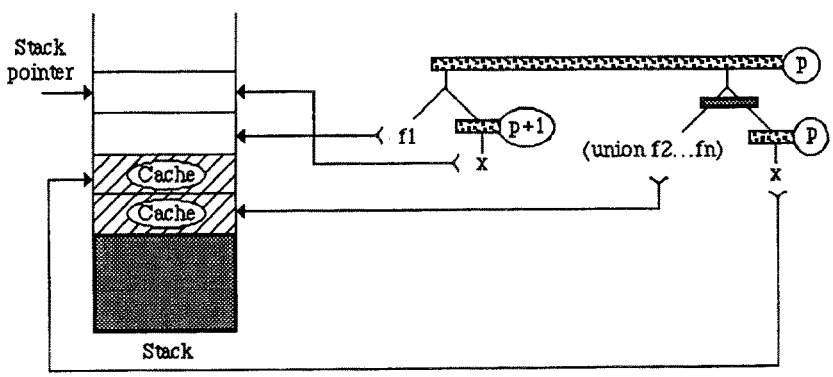
1 - Between all the delayed nodes appeared in the graph, since the last reduction with the rule applied in next case (or since the beginning), we select the left most and depth most node in the graph. Thus, if the graph is irreducible, we apply the rule to this node:

2 - If the graph is still irreducible, then we apply to the delayed node, which contains the smallest **p** in all the graph, the rule :

5.2 - Dynamic representation of graphs

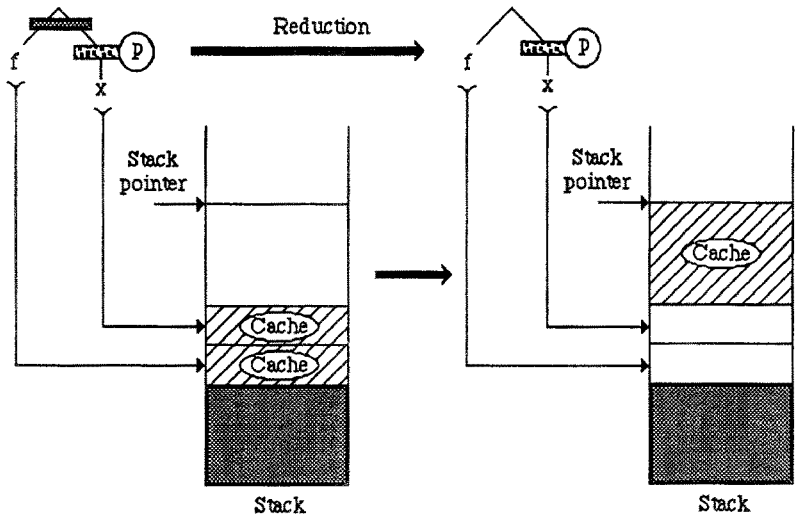
This section describes how the graphs are represented in the memory. We do not entirely examine it, because it is a generalization of the GRAAL system which is explained in [Bellot 86b]. We just specify what is modified. In this way, there are two new developments to do : the memorization of delayed nodes, and the activation of delayed nodes. These two notions appear in the stack management.

Memorization : Only the form union creates delayed nodes in the reduction graph. The delayed nodes are memorized in the stack, but hiden to the rest of the stack. The correspondence between the graph and the stack is :



Remark : The creation of a delayed node is very simple : we just have to push in the stack the address of the form **(union f2...fn)** and the argument **x**. This is why GREL is implemented in a functional language and not in an applicative language. The applicative languages, as the Lisp system, require a **safeguard-restoration** mecanism to variables management. This is really unefficient, whereas the functional languages do not need it.

Activation : The caches are activated when the second principle is used. The correspondence between graphs, reductions and the stack is :



6 - CONCLUSIONS

This work describes a relational programming language without variable, issued from GRAAL system. This language is as efficient as a functional language and gives more flexibility.

The appearance of relations in programming almost preserves the same efficiency that the functional language, when they are implemented in a language without variable. This is another element on account of functional language better than applicative language.

As only functional languages have been developed until now, almost all the classical examples are functional. But, the class of "multi-results" problems is as important as the class of functional problems. In point of fact, the "multi-results" problems are implemented in functional languages, using side effects. They can not be efficiently written in a pure functional language, like the algorithm giving all the partitions of a set. So, relations avoid inappropriate programming, and allow more flexibility in programs writing.

One of the most important applications of Relational GRAAL is given by the programming in logic. The logic languages are declarative one. Relation appears as the procedure corresponding to the translation of a declarative Horn clause used in logic. Thus, relation is nearest computation than clauses, and consequently is more efficient. Therefore, logic languages can be naturally implemented in a relational language. Moreover, We have realized [Legrand 87] a programming language in logic from GREL . This kind of approach gives an easy and efficient implementation, because the bactracking used in logic programming is already implemented in the Relational Language. The limited-depth-first is even a generalization of the PROLOG strategy, and the relations as processes allow implementation of coroutines and predicate "freeze".

The programmers used to functional or applicative languages do not have to try to adapt when they use the relational programming. Relations give flexibility and efficiency to the functional languages without compensations.

Acknowledgement : I am thankful to P. Bellot and V. Jay for their generous contribution to this paper. I also thank O. Danvy, A. Belkhir, D. Sarni and C.T. Lieu for interesting discussions. This work has been supported by the Gréco de Programmation (Bordeaux) under project SPLA.

References

[Bellot 86a] P. Bellot, Graal : a functional programming system with uncurryfied combinators and its reduction machine, European Symposium on Programming, (ESOP 86), LNCS 213, Saarbrücken, mars 1986

[Bellot 86b] P. Bellot, Sur les sentiers du Graal, étude, conception et réalisation d'un langage de programmation sans variable, Thèse d'état, Rapport LITP 86-62, Paris, octobre 1986

[Bellot 87] P. Bellot, V.Jay, A theory for Natural Modelisation and Implementation of Functions with Variable Arity, to appear in LNCS, Portland, septembre 1987

[Chailloux 84] J. Chailloux, M. Devin, J.M. Hullot, LE_LISP, a portable and efficient LISP System, Conference Record of the 1984 ACM Symposium on LISP and functional Programming, p113-123, Austin, Texas, 1984

[Cutland 80] N.J. Cutland, An introduction to recursive function theory,

[Durieux 81] J.L Durieux, Sémantique des liaisons nom-valeur : application à l'implémentation des lambda-langages, Thèse d'état, Université Paul Sabatier, Toulouse, 1981

[Eilenberg 70] S. Eilenberg, C.C. Elgot, Recursiveness, Academic press, New york, 1970

[Korf 85] R.E. Korf, Depth-First Iterative-Deepening: An Optimal Admissible Tree Search, Artificial Intelligence, Vol 27, p 97-109, 1985

[Legrand 87a] R. Legrand, Le calcul relationnel au service de l'implantation d'un langage de programmation en logique, Séminaire de programmation en logique, CNET Lannion, Ed M.Dincbas, p 333-346, Trégastel, 1987

[Legrand 87b] R. Legrand, Calcul Relationnel et Programmation en Logique, Thèse de l'Université Paris VI, 1987

[Turner 79] D.A. Turner, Another Implementation Technic for applicative Language, Software Practice and Experience, Vol. 9, 1979

[Nguyen 85] T.T. Nguyen, Algebraic theory of predicate transformers for relational programming, Research Report **No** RR 85-12, Louvain, 1985